Autor: Martin Vogel



# Inhalt

Autor: Martin Vogel

# PROGRAMMIEREN IN VISUAL BASIC UND ASSEMBLER

## Vorwort

Ein Einstieg in eine Programmiersprache wird oft von Realschulen oder Gymnasien angeboten. Doch welche Anwendung soll erstellt werden, die noch nicht perfekt in irgendeiner Form vorliegt. Das Internet ist voll von kostenloser Freeware.

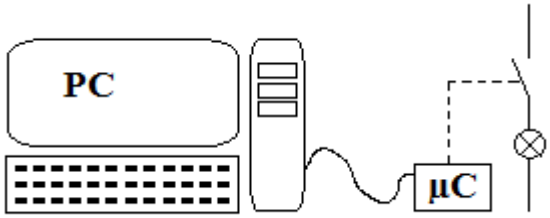
Programmieren ohne Ziel ist wie „Essen gehen“ ohne Hunger. Sehr schnell ist die Begeisterung verflogen und nur wenige haben nach der ersten Euphorie noch Lust, sich mit einer Programmiersprache zu befassen. Dennoch gibt es interessante Bereiche, die noch nicht mit fertiger Software abgedeckt sind. So hat wohl jeder, der ein Interesse an PC und Elektronik hat, mit dem Gedanken gespielt, kleine Steueraufgaben per PC durchzuführen.

Die Aufgabe, mit einem PC eine analoge Eisenbahnanlage aufzupeppen oder die alte Carrerabahn wieder vom Boden zu holen, um mit Freunden Spaß zu haben, könnten Vorgaben für interessante Projekte sein. Aber auch eine Alarmanlage für die eigenen vier Wände oder verspielte Effekte mit künstlerischem Ansatz werden durch Einsatz von Controllerbausteinen ein leicht erreichbares Ziel. Vorausgesetzt, man weiß um die Grundlagen von Elektrotechnik und versteht die Struktur von Programmen.

Um mit dieser Lektüre arbeiten zu können, werden wesentlichen Grundlagen erörtert. Auf tiefgreifendes Fachwissen in der Elektrotechnik kann allerdings nicht eingegangen werden. Dafür wird mit Schaltungen und Experimenten ohne unnötigen theoretischen Ballast eine Basis erarbeitet.

Ein PC ist Aufgrund seines Aufbaus nicht dafür geeignet, irgendwelche Steuerungsaufgaben zu erledigen, obwohl das Prinzip dem des Microcontrollers gleicht. Er ist der typische Aktenverwalter und Analyst, wenn es um Auswertung von Daten geht. Dafür ist er Konzipiert. Daten erfassen, Schriftstücke verwalten, Internetverbindungen zu schaffen und viele weitere Funktionen, auf die wir heute nicht mehr verzichten wollen. Er ist sozusagen der Manager, der Aufgaben beschreibt und vergibt. Aber so einfach mal das Licht anschalten, dafür ist er von Hause aus nicht ausgerüstet. Das ist auch eigentlich nicht die Aufgabe eines Managers, sondern die eines Handwerkers. Eine einfache Umsetzung solcher Aufgaben ist in der Regel nur durch geeignete Hardware gegeben. Sie sollte mit den vorhandenen Schnittstellen kommunizieren können

und die gewünschte Arbeit umsetzen. So betrachtet ist der PC der Boss und der µC sein Arbeiter. Es gibt viele Wege zu diesem Ziel, doch ist oft ein nicht geringer Kostenfaktor damit verbunden. Manchmal reicht aber eine kleine Bastellösung und Spaß macht es obendrein, wenn sich der Erfolg einstellt. Es ist lediglich erforderlich, eine einfache Elektronik zu finden, die über die Schnittstellen des PC ansprechbar ist. Der Mikrocontroller ist eine komplexe Einheit, die für unsere Zwecke bestens geeignet ist.




*Das PC-Controller System*

Die Arbeit mit Mikrocontroller ist für den Hobbyelektroniker besonders durch die einfache Verwirklichung elektronischer Spielereien interessant. Der Aufbau von Schaltungen ist mit wenigen Bauteilen übersichtlich und stellt keine großen Anforderungen. Ein Controller und ein wenig Anpassung für Relais, Schnittstellen oder Anzeigen. Alles mit geringem Aufwand realisierbar. Die Komplexität der Schaltung wird mit Software erreicht. Ob Drehzahlmesser, Uhr, Alarmanlage, Messdaten oder Rundenzähler, alles ist im Kern mit einem einzigen IC aufzubauen. Eine Grundschaltung für viele verschiedene Projekte. Je nach Bedarf ergänzen Pegelwandler oder Treiberbausteine die Schaltung. Daher wird der Schwerpunkt bei der Programmierung liegen. In einem gut dokumentierten Beispielprogramm wird von der Entstehung schrittweise bis zur fertigen Anwendung jede genutzte Funktionalität des Controllers beschrieben.

Programmieren in Visual Basic (Microsoft) und Assembler (AVR-Studio)

Autor: Martin Vogel

# Teil 1 PC-Programm in Visual Basic


*Der Boss*

Ob ein Visual Basic Programm für den PC oder ein Assembler für den Controller, ohne das Wissen um die Struktur von Programmen wird man auch mit viel Zeiteinsatz schnell an Grenzen gelangen. So kann auch dieser durchaus spannende Weg in die Elektronik ohne Anleitung schnell ein Ende voller Frust nehmen. Obwohl die fertigen Programme beiliegen, rate ich davon ab, diese so zu installieren. Der richtige Weg ist mitzugehen und der „Bauanleitung“ zu folgen. Gerade weil das Programm praktisch noch mal Schritt für Schritt entwickelt wird, sitzt das Verständnis tief und es wird auch sofort erkannt, welche der eingebundenen Programmteile für eigene Ideen nutzbar sind.

Die Vorlage ist ein kleines Tool, das die Werte aus dem Variablenbereich des Controllers zur Laufzeit auf dem Monitor darstellt und liefert beim Erstellen einen interessanten Einblick in Programmstrukturen.

Das Tool ist vielseitig nützlich und nicht nur für eine einzige Aufgabe erstellt. Alle Programme in einem Controller, die mit den erforderlichen Programmteilen versehen sind, können damit auf ordnungsgemäße Bearbeitung kontrolliert werden. Es ist der Blick auf die „inneren Werte“, der ohne dieses Programm nicht oder nur begrenzt möglich ist.

Der gut dokumentierte Aufbau bringt Licht in die sonst so diffuse Vorstellung einer Programmierung. Schrittweise wird mit Screenshots und Skizzen an die Aufgabenstellung herangeführt. Die Programmroutinen sind mit Rücksicht auf Lesbarkeit einfach gehalten und weitestgehend kommentiert. Aufgaben werden sehr weit herunter gebrochen, um in kleinen Unterprogrammen gut verständlich abgebildet zu werden. Daher ist es auch wichtig, den Programmcode selbst zu schreiben, um auch mit Fehlern konfrontiert zu werden und ein Gefühl für Programmzeilen zu bekommen. Der mitgelieferte Programmcode sollte nur im äußersten

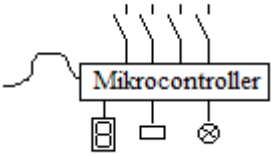
Notfall in kleinen Abschnitten in den Programmaufbau eingefügt werden. Sicherlich ist es mühsam, alles noch einmal zu schreiben, aber das Wissen um die einzelnen Funktionen wird wesentlich tiefer sitzen. So wird auch eine eigene, abgewandelte Anwendung zum Erfolg. Der Grundbaustein ist die bidirektionale Kommunikation, die es für andere Aufgaben abzuleiten gilt, z.B. Messdaten- und Signalerfassung. Als Ziel sind Alarmanlagen, Steuerung einer kompletten Modelleisenbahnanlage mit einem PC, Verwaltung und Organisation von Rennen mit einer Carrerabahn einschließlich, Zeiterfassung und Rennablaufsteuerung denkbar. Mit jeder selbst gestellten Aufgabe müssen Probleme gemeistert und Hürden genommen werden. Die Ansprüche wachsen und sicherlich wird auch manchmal ein Hilferuf in die Fachforen erfolgen. Mit dem hier erworbenen Wissen allerdings sind die Fragen gezielt und kompetente Antworten zu erwarten.

Bei der Arbeit mit diesen Anleitungen wird auch klar, das nicht die Programmiersprache entscheidend ist, sondern wie man selbst in der Lage ist, eine Aufgabe zu beschreiben. Die Sprache selbst ist Fleißarbeit, vergleichbar mit Vokabeln lernen. Für ein Grundgerüst unumgänglich. Die Perfektion aber kommt erst bei der steten Anwendung. So wie sich eine Sprache entwickelt, wird auch das Fachchinesisch der Experten seine Geheimnisse preisgeben. Das hier vorgestellte und entwickelte Programm ist lediglich der Anfang. Es ist „Open Source“ und Erweiterungen nach eigenen Wünschen steht nichts im Weg.

Programmieren in Visual Basic (Microsoft) und Assembler (AVR-Studio)

Autor: Martin Vogel

Teil 2 befasst sich mit dem Mikrocontroller und der Programmierung in Assembler




*Der Arbeiter*

Mit überschaubaren Experimenten vermittelt dieser Teil das erforderliche Wissen zum Mikrocontroller und dessen Programmierung in Assembler. Dabei wurde auf einfache Strukturen geachtet, die auch von blutigen Anfängern verstanden und angewendet werden können. Wer hier professionelle Raffinesse erwartet, wird sicherlich enttäuscht. Eher liegt der Schwerpunkt bei der Vermittlung binären Denkens. Logische Verknüpfungen von Registerinhalten bilden den Schwerpunkt. Geleitete Experimente zu den Schaltungsvorschlägen runden diesen Abschnitt ab. .

Ein paar Grundlagen der Elektrotechnik werden ebenfalls erklärt, soweit dies für den Aufbau einer Schaltung erforderlich ist. Tiefgreifendes Wissen darf allerdings hier nicht erwartet werden. Die Elektrotechnik und Elektronik sind nicht mit ein paar Seiten in einem Buch abgehandelt. Werden bei eigenen Projekten später diesbezüglich Bereiche tangiert, die einer weiteren Erklärung bedürfen, so sind auch hier im Internet genügend Informationsquellen zu finden. Speziell ist die Suche nach Datenblättern der verwendeten Bauteile zu empfehlen.

In diesem Buch liegt der Schwerpunkt auf kleinen verständlichen Programmböcken in Assembler, die den Einstieg erleichtern. Professionalität wird daraus wachsen, wenn diese „Bauanleitung für Programme“ Inhalte für eigene Ideen vermittelt und die Wege zur Umsetzung erkannt sind. Assembler ist wie keine andere Programmiersprache so nah an der Hardware des Controllers. Wenn nicht gerade umfangreiche mathematische Aufgaben zu bewältigen sind, ist auch Assembler für einen Einstieg gut geeignet. Die Aussage das Assembler die Grundlage für eine andere Programmiersprache wäre, ist

aber falsch. Jede Programmiersprache hat ihren eigenen Aufbau. Lediglich den Weg zum Ziel haben alle gemeinsam. Eine Programmstruktur ist, vielleicht mit geringen Änderungen, auf alle Sprachen adaptierbar.

Wer verstehen will, wie ein µC arbeitet ist mit Assembler gut beraten. Werden zusätzlich auch ein paar Regeln eingehalten, ist diese Programmiersprache gar nicht so kompliziert, wie immer behauptet wird. Sicherlich, eine Programmentwicklung erfordert gegenüber Hochsprachen etwas mehr Zeit, aber es gibt auch Vorteile, die Assembler durchaus in der Anwendung rechtfertigen. Wie wir bereits in VB erkannt haben, gibt die objektorientierte Programmierung mit der Ereignisbearbeitung Strukturen vor. Assembler ist diesbezüglich völlig zwanglos und es ist leicht möglich, einen unübersichtlichen Befehlsablauf zu erstellen. Die Arbeit mit VB sollte aber schon ein paar Wege aufgezeigt haben, wie dies zu verhindern ist. Auch in einem Controller ist eine Ereigniserfassung und somit auch die Bearbeitung mit Jobflags durchaus sinnvoll. Das Ergebnis wird sich in übersichtlichen und gut anpassbaren Bausteinen darstellen.

Das notwendige Know -how rund um die Hardware rundet das Thema ab. Dazu wird eine Schaltung vorgeschlagen, die auf einem Steckbrett stückweise im Experiment erarbeitet wird. Ziel ist es, durch diese Experimente Wege zu eigenen Projekten zu finden. Die Übungen mit dem Mikrocontroller fördern intensiv das Verständnis um die internen Funktionen. Langweilige Theorie bleibt auf ein Minimum begrenzt. Bei den Experimenten wird immer wieder auf das VB-Programm **OPEN_EYE** aus Teil 1 hingewiesen, damit die korrekte Bearbeitung der Programmierung eines Controllers nachvollzogen und kontrolliert werden kann.

Autor: Martin Vogel

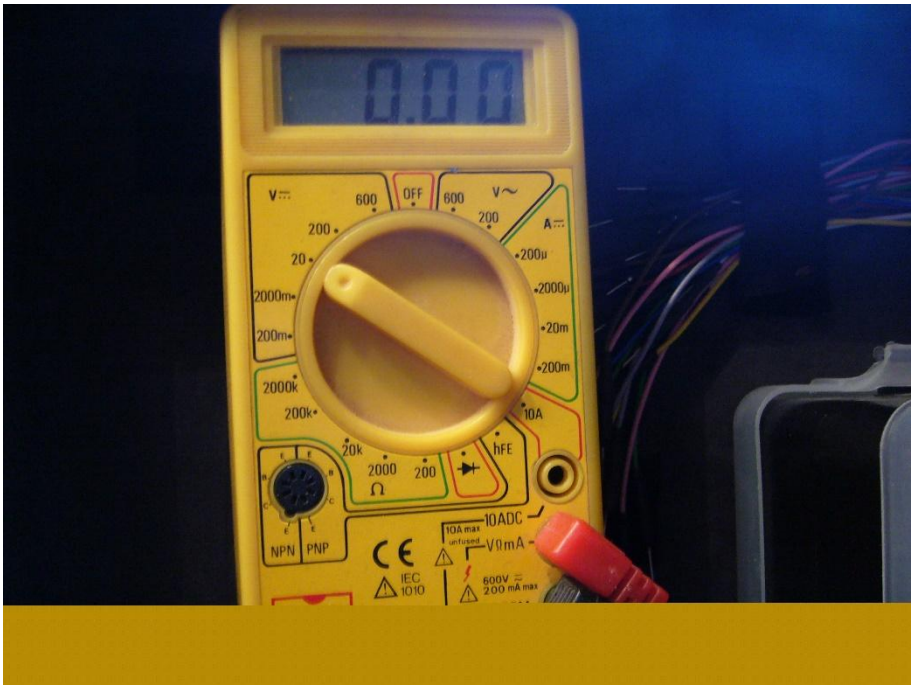
## Das Equipment

Verschaffen wir uns zuerst einmal einen Überblick über das Notwendige. Ohne eine Ausrüstung bleibt eben alles nur Theorie. Daher beginne ich mit dem Werkzeug.

Es muss ja nicht eine Lötstation für 200 € sein, für die wenige Arbeit reicht auch eine preiswerte aus. Nicht unbedingt zu empfehlen sind Lötkolben. Das heiße Stück sollte schon in eine ordentliche Ablage gesteckt werden können. Dazu gehört Elektroniklot. Es ist zwar teuer, aber mit Lötwasser und –paste versaut man mehr, als das es nutzt. Die Experimente werden im Allgemeinen auf einem Steckbrett durchgeführt. Trotzdem ist hier und da schon mal ein Adapter anzufertigen, oder eine Lochraster-Platine zu bestücken.

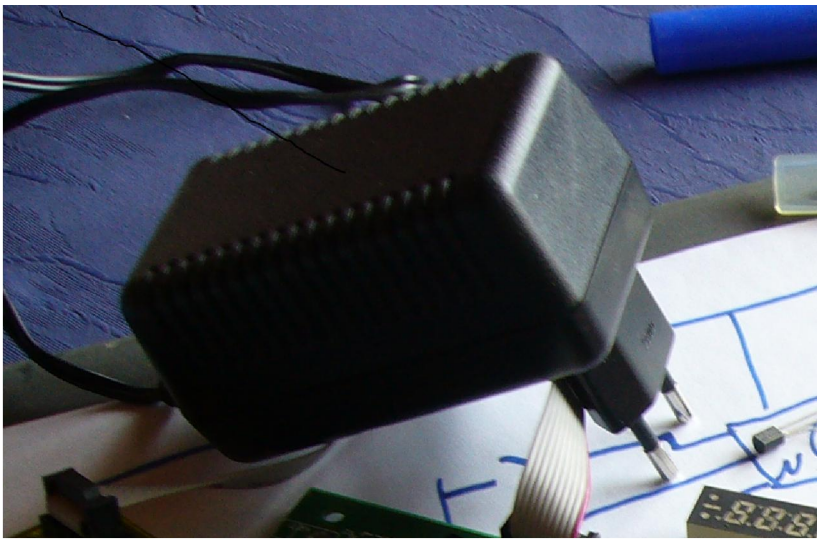

*Lötstation*

Ein kleiner Seitenschneider, Spitzzange sowie ein Schraubendrehersatz gehört ebenfalls zur Werkzeugausrüstung. Am *kleinen* Arbeitsplatz sind auch häufig Messungen erforderlich. Ich empfehle mindestens ein gutes Vielfachmessgerät. Die Preisklasse ab 40 € liefert schon ausreichende Qualität für den Bastler. Es schadet aber nicht, wenn zusätzlich ein paar Billiggeräte angeschafft werden. Diese liegen z.B. im Versandhandel unter 10 €, ja teilweise bei Sonderangeboten sogar unter 5 €. Sie können für Messungen von Kleinspannungen hinzugezogen werden, wenn es nicht so genau sein muss. Ein kleiner Vorteil: Wer sein Equipment in einen Alukoffer bauen möchte, kann diese kleinen Schätzeisen leicht mit Klett band im Deckel befestigen. In hohen Spannungsbereichen sollten sie allerdings nicht eingesetzt werden.



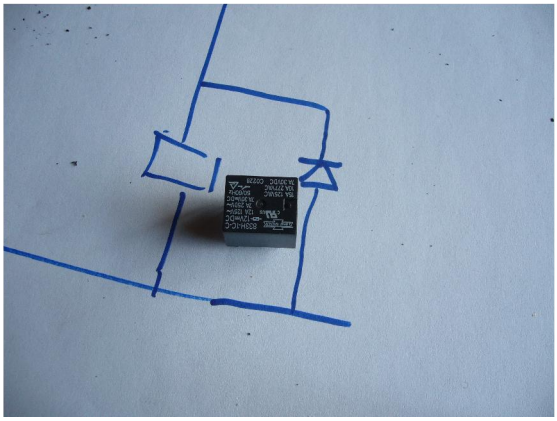

*Kleine Helfer Multimeter*

Für alle Arbeiten mit Elektronik benötigt es eine Spannung. Es reicht aus, ein Steckernetzteil 12 V einzusetzen. Da wir nur das Evaluationsboard und evtl. ein paar LED auf dem Steckbrett betreiben wollen, reichen 1 A Netzteile völlig aus.



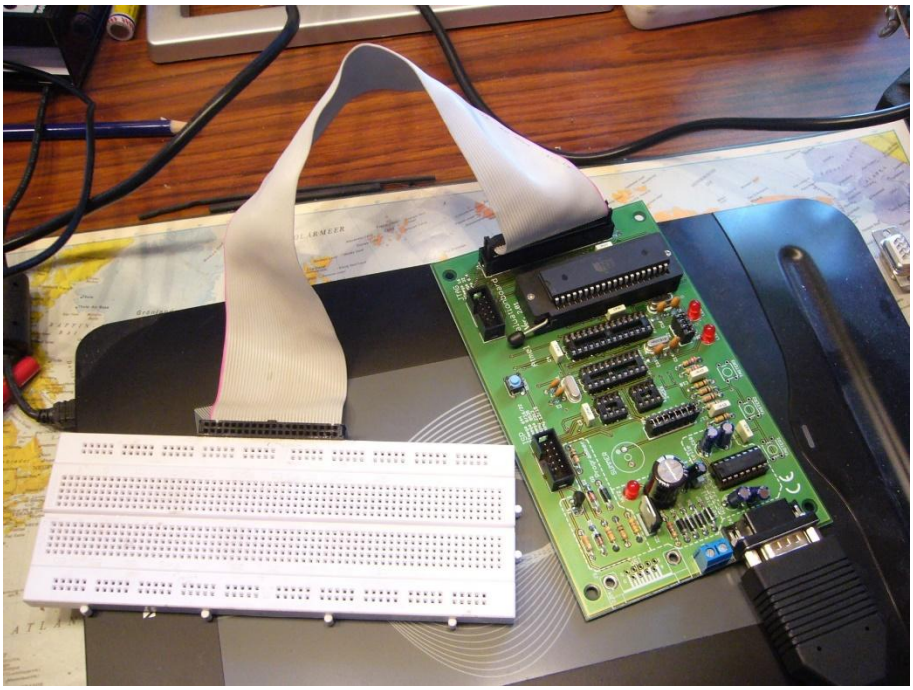
*Steckernetzteil*

Auch 12 V Relais lässt sich damit über Transistorstufen oder Treiberbausteine ansteuern.



*Relais*

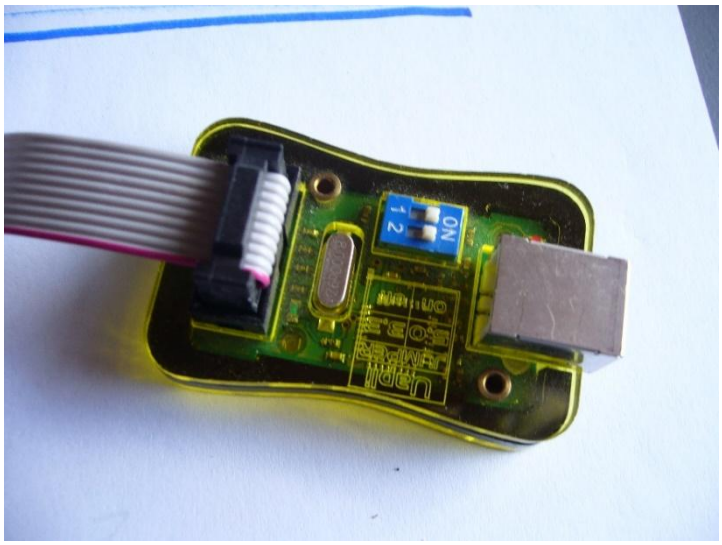
Ein Evaluationsboard, ein USB-ISP Stick und ein Steckbrett runden die Hardware ab. Die Spannungsversorgung von 5V ist beim ATMEL Evaluations-Board V2.0.1 integriert, so dass es nur eine Spannung ab 9 V benötigt. Damit ist ein 12 V Steckernetzteil ideal. Gleichspannung ist zwar nicht unbedingt erforderlich, aber die Eingangsdioden beim Evaluationsboard dienen gleich dem Verpolschutz und die Spannung kann so auch ohne Aufwand für Relais und andere Bauteile genommen werden.



*Atmel Evaluationsboard mit Steckbrett*

Bei diesem Board kann mit einer seriellen RS 232 ein Controller beschrieben werden, allerdings nicht vom AVR Studio. Mit AVR Studio werden die Programme erstellt und übersetzt, aber der Schreibvorgang läuft über ein anderes Programm. Z.B. PonyProg, welches auch kostenlos im Netz zum Download angeboten wird. Allerdings ist das ein sehr

langsamer Prozess und auch nicht ganzproblemlos. Besser ist ein USB ISP Programmierstick. Das Board hat dafür einen 10 pol. Wannenstecker.




*ISP-USB Programmiergerät*

Für die Experimente braucht es noch ein wenig Zubehör. Dies sollte man sich in aller Ruhe überlegen, wenn die erste Bestellung oder der Einkaufszettel für den Elektronikladen erstellt wird. Ein kleines Sortiment an Transistoren und Dioden, Widerstände mit den Werten 100 Ohm, 270 Ohm, 330 Ohm, 680 Ohm, 1 KOhm, 5 KOhm und 10 KOhm (mind.30 Stk/Wert) sowie Kondensatoren mit den Werten 22pF, 33pF, 33nF, 100nF, 1µF, 10 µF und evtl. 100µF mit Nennspannung 16V. Auch hiervon mindestens 10 Stk/Wert.

Dazu ein paar LED, vielleicht ein paar Gabellichtschranken sowie Optokoppler und ein paar 7-Segmentanzeigen.

Ein kleines Schalter- und Tastersortiment mit ausgesuchten Typen, keine bunt gemischten Sortimente. Und schließlich das wichtigste: die Controller.

Bei einer Bestellung sollte man schon ein paar von der gewünschten Sorte mitliefern lassen. In diesem Buch wird allerdings nur der Atmega8 zur Sprache kommen. Er liefert ausreichend Potenzial für viele verschiedene Projekte. Allerdings ist der Einsatz eines Atmega16 auch kein Problem. Die Beispiele sind fast 1: 1 übertragbar.

Zusammenfassung

Lötkolben
Elektronik-Seitenschneider,
Spitz-oder Telefonzange,
Schraubendreher,
Multimeter,
Steckernetzteil 12 V, 1 A
Experimentier-Steckbrett
PC mit Software Microsoft Visual Studio 2008
Visual Basic 2008
Microsoft SQL Server 2008
AVR-Studio 4.18
PonyProg
Atmel Evaluationsboard oder anderer Programmer
div. elektronische Bauteile für die Experimente
Optional
USB-ISP- Programmierstick ( AVR MKII od. ähnl.)
USB-RS232 Converter

## Wichtige Hinweise

Die erforderliche Software „Visual Basic“ von Microsoft sowie „AVR Studio“ von Atmel ist nicht Bestandteil dieses Buches.

Die Programme sind in „Visual Basic 2008“ und „AVR Studio 4“entwickelt. Beide Programme sind kostenlos im Internet zu erhalten. Auch wenn mittlerweile die Programme um Generationen weiter sind, braucht niemand Bedenken zu haben, auf ein altes Pferd gestiegen zu sein. Dieses Buch schult nicht eine Sprache, es soll die Vorgehensweise beim Programmieren aufzeigen. Und das ist nicht abhängig vom Ausgabestand einer Sprache. Auch liest man immer öfter, der Atmega8 sei ein alter längst verstaubter Controller. Mag ja sein, aber er ist günstig zu bekommen und bis man sich soweit entwickelt hat, das die neuen Eigenschaften benötigt werden, nun ja, da vergehen ein paar Tage, Wochen oder gar Jahre. Bis dahin sind die jetzt gelobten „modernen“ Controller vermutlich auch schon wieder aus dem Rennen. Die Erfahrung, die mit dem Atmega8 aber bereits vorhanden ist, wird den Umstieg auf einen zeitgemäßen µC keinesfalls erschweren..

Autor: Martin Vogel

**Texteinfärbung:**
Kommentartexte und auskommentierte Befehle sind grün markiert

Quellen:
Datenblätter der Controller
Datenblätter von eingesetzten elektronischen Bauteilen

Internetseiten und Fachforen:
www.mikrocontroller.net
www.avr-praxis.de
www.roboternetz.de
www.msdn.mikrosoft.com

Lieferanten:
Elektronik Versandhandel

Die Liste der benötigten Bauteile ist für jedes Projekt unterschiedlich. Die Angaben hier sind deshalb nicht abschließend vollständig und auch nur grobe Richtwerte.

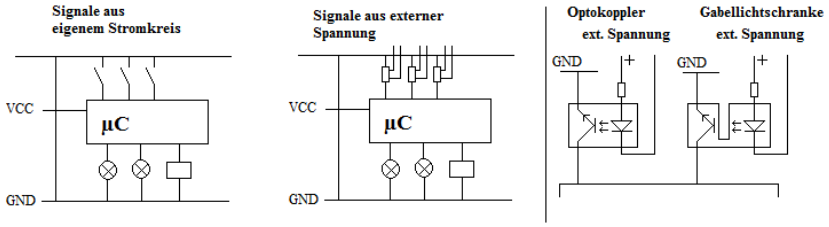
Bauteile :
Mikrocontroller ATMega8, ATMega88, ATMega16
Treiberbaustein ULN 2803
Kommunikation Pegelwandler MAX232
Universaldioden 1N4001 ( ca. 50 Stk)
PNP-Transistoren (bspw.BC 328) (mind. 10 Stk)
NPN-Transistoren (bspw.BC 338) ( mind. 10 Stk)
LED verschiedene Farben (jeweils 10-20 Stk)
Anzeigen 7 Segment
Relais 12 V
Taster
Gabellichtschranken
Kondensatoren 22 pF, 33 nF, 100 nF, 1 µF, 10 µF
Widerstände: 270 Ohm, 330 Ohm, 470 Ohm, 680 Ohm, 1 KOhm, 5 KOhm, 10 KOhm
( je 100 Stk pro Wert )
Schaltdraht
Buchsen, Stecker
Leitungsmaterial

Weitere nach Bedarf

Optokoppler sind zu empfehlen, wenn Signale in einen Controller geführt werden sollen, die aus einer anderen Spannungsquelle kommen. So ist eine galvanische Trennung von Stromkreisen realisierbar.

Das Prinzip ist das gleiche wie das der Gabellichtschranken.

Vorteil: Der Controller kann nicht durch die Signale zerstört werden, da er mit dem Stromkreis nicht in Berührung kommt. Entsteht auf der Signalseite eine Überspannung, so leidet schlimmstenfalls der Optokoppler. Da im Buch nicht weiter darauf eingegangen wird, möchte ich das Prinzip hier kurz erläutern.




*Optokoppler*

Im linken Bereich ist erkennbar, das die Signaleingänge auf dem Spannungspotential des MC liegen. Durch Zwischenschalten eines Optokoppler bleibt zwar das Spannungspotential des Controllers am Eingang, aber das auslösende Signal schaltet lediglich eine LED an. Egal was auf dem Stromkreis des Signales geschieht, der Controller bekommt nur die Auswirkung durch das Licht mit. Von Spannungsspitzen bleibt der Eingang des Controllers unberührt. Im linken Bildteil ist der Innenaufbau eines Optokoppler sowie einer Gabellichtschranke dargestellt. Bei der Gabellichtschranke besteht im Gegensatz zum Optokoppler die Möglichkeit, ein Signal durch Unterbrechung des Lichtstrahles zu bilden. Die Spannung liegt dabei immer an.

# Teil 1

## 1.1 PC-Programme in Visual Basic

### *1.1.1 Vorstellung Projekt*

Das kleine Tool Open_Eye, welches die Variablenwerte aus einem Controller im PC zur Laufzeit sichtbar macht, gibt es bereits. Ich habe es vor einiger Zeit in Delphi entwickelt und in einem Forum für Mikrocontroller der Öffentlichkeit zugänglich gemacht. Nun habe ich weitere Anforderungen an dieses Programm gestellt, und nahm es zum Anlass, einen Umstieg auf Visual Basic durchzuführen.

Visual Basic ist Delphi sehr ähnlich und besitzt fast die gleichen Mechanismen. Der Umstieg ist begründet in der preiswerten Verfügbarkeit von Visual Basic. Für den privaten Gebrauch sind kostenlose Downloads älterer Versionen im Internet angeboten.

Dies sind die Anforderungen, die das neue Programm erfüllen soll:

Es soll eine Kopie aus einem Assemblerlisting mit dem Bereich der Variablendeklaration in VB zerlegt und in Variablen und evtl. Bits aufgegliedert werden.

Es soll einen Trigger geben, der im Programm des Controllers benutzt wird und die Daten einer definierten Programmbearbeitung liefert.

Die Einstelldaten zum Projekt sollen gespeichert werden.

Es sollen auch Werte in Integer (16Bit) oder LongInt (32Bit) sowie Word oder DWord angezeigt werden.

Die Ausgabe soll in Bitfeld, ASCII, Hex, Integer formatiert werden können

Einzelne Bits sollen ihrem Kommentar zugeordnet werden.

Parametrieren der Schnittstelle

Option zum Wechseln der Programmiersprache für den Filter (BASCOM, C)

Das sind schon eine Menge Aufgaben. Dazu soll natürlich diese Software völlig unkompliziert bedient werden. Ich bin da zuversichtlich, das alle Kriterien umgesetzt werden können, bis auf den Punkt „Option“. Hier sind die Experten gefragt, die mit diesen Sprachen Controller programmieren. Da der Source-Code mit dem kompletten, gut kommentierten Listing vorliegt, sollte es ein Anreiz sein, den Filter für die Option nachzubauen und die Ausgabeelemente entsprechend zu erweitern. Das Buch beschränkt sich lediglich auf den Assembler.

Der schrittweise Aufbau des Programms vertieft die Kenntnisse in der Kommunikation zwischen PC und µC. Somit ist ein Datenaustausch über die serielle Verbindung in beiden Richtungen auch für eigene Ansprüche kein Problem mehr.

Zur Ablage der Projektdaten kommen Datenbankelemente zum Einsatz. Ein nicht uninteressanter Bereich, auch wenn dies für diese Aufgabe vielleicht etwas übertrieben ist.

### *1.1.2 Das kleine Grundwissen zum Programmieren*

An diese Stelle möchte ich ein paar immer wiederkehrende Begriffe klären, die für jede Programmiersprache von Bedeutung sind. So ist es sowohl in Basic als auch in Assembler erforderlich, Variable zu definieren. Eigentlich sind es Speicherbereiche und damit der Prozessor weiß, wo er sie findet und wie er sie behandeln muss, werden durch die Deklaration die Adressen erfasst..

**In Basic mit Formatangabe:**

```
DIM ein_Wert as Integer
Dim Werte_Feld(20) as Integer
```

Damit wird ein Speicherplatz für einen Zahlenwert sowie einer Tabelle mit 20 Feldern einer mit Integerformat bestimmten Größe festgelegt. Integer ist bei Visual Basic mit 32 Bit (vier Byte) definiert.

Doch was bedeuten die verschiedenen Formatangaben? Nun, ein Computer, ob PC oder µC, selbst ein Taschenrechner, kennt nur Ziffern „0“ und „1“. Dies ist darin begründet, das es für die Elektronik auch nur die zwei elektrischen Zustände „Aus“ und „Ein“ gibt. Wenn ich zwei spannungsführende Leitungen habe, dann gibt es schon vier verschiedene Zustände:

| | Zustand 1 | Zustand 2 | Zustand 3 | Zustand 4 |
|---|---|---|---|---|
| Leitung 1 | 0 | 1 | 0 | 1 |
| Leitung 2 | 0 | 0 | 1 | 1 |

Hat man weitere Leitungen, erhöht sich auch die möglichen Zustände in der Kombination für jedes weitere Bit um das Doppelte. Es ist hier nun auch erkennbar, das die kleinste Informationseinheit, eine Leitung, nur zwei Werte liefern kann. Diese kleinste Informationseinheit nennt man Bit. Vier Bit in einer Gruppe werden Nibble genannt und schon etwas bekannter ist die Gruppe mit acht Bit: das Byte. So hat ein Datenträger mit zwei Gigabyte 2 * 8 Milliarden Speicherzellen. Wir haben uns längst an diese Angabe gewöhnt, doch wenn man einmal die Zahl in ihrer Bedeutung erfasst, ist es schon gigantisch, was auf so einen kleinen

Datenträger wie einen USB-Stick oder gar auf die SD-Speicherkarten abgelegt werden kann.

Im weiteren Verlauf wird aber eine Speicherzelle als Byte mit 8 Bit betrachtet.

Es gibt noch eine weitere Größe in der Datentechnik, das Format Word. In einem Word sind zwei Byte enthalten, einzeln betrachtet in Lowbyte und Highbyte. Somit ist ein Word eine Gruppe von 16 Bit.

Ältere Computer hatten gerade einmal einen Adressbereich von 16 Bit, das entsprach 65536 Adressen, um Speicherzellen anzusprechen. Mittlerweile bewegen sich die Adressräume im 32 Bit, bzw. im 64 Bitbereich, entsprechend 4.294.967.296 bzw. 18.446.744.073.709.551.616 Adressen, eine unvorstellbar große Zahl. Doch wir brauchen die Zahlengröße nicht wirklich direkt zu erfassen und unser Programm wird auch nur bis zum Doppelwort, einer 32 Bitzahl auflösen. Allerdings ist die Darstellung dann in Hexadezimaler Form angebracht.

Hoppla, Hexadezimal, was ist das denn?

Nun, die Computerwelt hat in den Anfängen nach einer Möglichkeit gesucht, übersichtliche Darstellung der Bytes zu bekommen. Die Programmdarstellung lagen als Maschinencode in den Speicherzellen. Das tun sie heute auch noch, aber mittlerweile interessiert das nicht mehr. Um nun die Programmcodes in den Speicherzellen zu lesen, hatte die dezimale Schreibweise keine Chance, da die Zahlen mal einstellig, ein andermal 3 stellig waren. Die Darstellung in Bits lieferte da schon exakt immer 8 Stellen, doch die Fehlerquote beim Lesen war erheblich. Nicht lange, und es flimmerte vor den Augen. Wenn man nun ein Byte in zwei Teile (Nibble) zerlegt, dann kann jedes Nibble die Werte von 0 bis 15 annehmen. Also brauchte man sechszehn verschiedene Ziffern. Die Ziffern von 0 bis 9 konnte man beibehalten, und so wurde einfach mit den ersten Buchstaben im Alphabet „A“ bis „F“ fortgesetzt. Somit hat ein Nibble den Wertebereich von 0 bis F. Nun ließen sich die Bytes schön darstellen und mathematisch betrachtet, ist lediglich eine Zahl entstanden, die die Basis 16 statt 10 oder 2 hat. Merken wir uns einfach:

Dualzahl:Basis 2
Dezimalzahl:Basis 10
Hexadezimalzahl:Basis 16

### *1.1.3 Das kleine Wissen über Zahlen*

Unser Verständnis zu Zahlen bewegt sich im Dezimalsystem. Das sind Zahlen mit der Basis 10. Jeder weiß, mit einer Dezimalzahl umzugehen, aber ist auch jedem bewusst, das der Zahlenbereich einer Stelle eigentlich nur von 0 bis 9 geht?

Klar, wir lesen dreihundert vier und achtzig (384). Im Kopf nachgerechnet 300 + 4 +80…

Man kann auch sagen 3 mal 100 +8 mal 10 +1 mal 4. Nun gehen wir noch einen Schritt weiter, denn ich sagte, unser Dezimalsystem hat die Basis 10. Somit sind die Einerstellen mit 10^0 zu multiplizieren.

Nicht so ganz klar, aber wenn 10 ^1 = 10 ist und 10 ^2 =100 so ist es vielleicht logisch, wenn 10^0= 1 ist. Und tatsächlich, jede Zahl ^ 0 ist 1. Ihr könnt gern euren Taschenrechner damit beauftragen…

Zahl eingeben x^y drücken und dann eine 0 - Ergebnis =1

Nun ist es nicht mehr so schwierig die Zahl 384 stellenweise zu betrachten

4 mal 10^0 + 8 mal 10^1 + 3mal 10^2 bzw. da wir ja von links nach rechts lesen entsprechend andersrum. Dennoch, alle Zahlen bauen entgegen der Leserichtung von rechts nach links auf. Auch ein Computer geht diesen Weg. Allerdings kennt er nur Ziffern 0 und 1. Die Information wird mit Bit bezeichnet. Wird ein Wert größer 1 muss auf die nächste Stelle zugegriffen werden, wie beim Dezimalsystem, wenn der Wert einer Zahl größer 9 wird.

Da wir in Assembler mit Dezimal- Hex- und Binärzahlen arbeiten, gleich hier mal eine kleine Erklärung dazu:

Ein Byte enthält 8 Bit. Das ist in der Computerwelt so festgelegt. Ein Bit kann wie bereits erwähnt 2 Werte einnehmen, eine 0 und eine 1. Demnach ist die größte darstellbare Zahl 255.

Begründung:
Die größte binäre Zahl in einem Byte ist 11111111
Demnach mathematisch
1*2^0 +1+2^1+1*2^2+1*2^3+1*2^4+1*2^5+1*2^6+1*2^7

also dezimal 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 =255

Kommen wir zu einer anderen Darstellung, die in der Programmierung oft verwendet wird, die Hexadezimalzahl.

Ein Byte besteht aus einer zweistelligen Hexadezimalzahl. Um diese darzustellen, teilen wir ein Byte in 2 Nibble, das sind zusammengefasste 4 Bit mit einem Maximalwert von „1111“ und „1111“. Dies entspricht einem Wert von 15. Um aus Zahlen >9 eine Ziffer darzustellen, greift man auf das Alphabet zurück und fügt die Hex. Ziffern „A“ - „F“ hinzu. Somit entspricht Dezimal 255 dem Binärwert 11111111 und dem Hexadezimalwert FF.

Also 255 (dezimal) = 1111 1111 (binär) = FF (hexadezimal).

Die Zahlenbasis einer Hexzahl ist die 16

Somit ist die erste Stelle mit x*16^0, die zweite mit y*16^1, die dritte mit z*16^2 usw.

Auch hier gilt: 16^0 =1

Auf unsere 255 bezogen

F(15)*16^0 + F(15)*16^1

15*1 + 15*16 = 15 + 240 = 255

Mit diesen Zahlen lässt sich natürlich auch rechnen. Während wir uns mit dem Dezimalsystem herumschlagen und Ziffern mit den Werten von 0 – 9 betrachten müssen, macht es sich der Computer einfach: er rechnet nur mit 0 und 1 je Stelle. Zugegeben, für uns erscheint es viel schwerer, eine binäre Zahl zu addieren oder gar zu multiplizieren. Der Computer macht es genau wie wir, er erkennt einen Stellenüberlauf und nimmt die nächste Stelle.

```
So ist für uns                 für einen Controller

  13                              00001101
+ 78                            + 01001110
------------                  ----------------------
= 91                              = 01011011
```

Im Dezimalsystem ergibt das Ergebnis einer Addition zweier Zahlen größer 9 einen Übertrag. So haben wir es mal in den ersten Schuljahren gelernt.

```
        7
       +8
     -------
     = 15

Ausgesprochen ist     7 plus 8 gleich 5 und 1 Übertrag.
```

Gleiches Prinzip im Binärsystem. Die größte Zahl ist 1.

---

```
        1
       +1
      -----
      = 10
Ausgesprochen ist     1 plus 1 gleich 0 plus 1 Übertrag
```

Ist das Ergebnis einer Addition im Binärsystem größer 1, dann habe ich auch einen Übertrag auf die nächste Stelle.

Es sei noch vermerkt, das ein Controller nur 2 Bytezahlen addieren kann, also Zahlen einer Liste werden somit Schrittweise addiert.

```
Dezimal:                          Binär

    6                1.Schritt              r16   = 00000110
   +18                                    +r17  = 00010010
   +23          Ergebnis im 1. Register   -------------------------
  +112                Ergebnis in          r16  = 00011000
 --------             2.Schritt           +r17  = 00010111
 =  159         Ergebnis im 1. Register   -------------------------
                      Ergebnis in           r16  = 00101111
                      3.Schritt            +r17 =  01110000
                Ergebnis im 1. Register   -------------------------
                      Ergebnis in           r16  = 10011111
```

Aber ich will euch nun nicht weiter mit Theorie quälen und auch nicht weiter Irre machen. Daher wende ich mich nun einigen Programmierbegriffen zu. Das Verständnis wird sich mit der Anwendung schon weiter vertiefen.

### *1.1.4 Unterprogramme - Prozeduren und Funktionen*

Ein wichtiger Bestandteil von Programmen ist die „Sub-Routine“ oder zu Deutsch: Unterprogramm. In Basic wird sie Sub oder Function genannt. Für den Compiler sind dies Adressmarken, die auf eine Adresse im Speicher zeigen. Dort wird dann ein Programmteil bearbeitet, wenn er vom Hauptprogramm aufgerufen wird. Ist die Arbeit erledigt, wird hinter dem Aufruf mit der Arbeit im Hauptprogramm fortgesetzt. In Basic sind die „Sub“ und „Function“ sehr komfortabel. Im Aufruf ist es möglich, Parameter zu übergeben und damit die Routinen in sich zu kapseln. Dies bedeutet, es werden in einer Sub oder Function ganz selten Variablen gebraucht, die als global definiert im ganzen Programm gültig sind. Aber dazu kommen wir noch. Hier die typischen Aufrufe von Sub oder Function in Basic:

```
Public Sub Set_Values(ByVal Byte_Cnt As Integer)

Public Function Is_Ok(ByVal Byte_Cnt As Integer)as Boolean
```

Das Wort Public bedeutet, dass alle Komponenten vollen Zugriff auf diese Routine haben. Hier ist auch ersichtlich, dass eine „Sub“ eine Routine ohne Rückgabewert ist.

Bei der Function ist die Rückgabe „wahr“ oder „Falsch“. Die Rückgabewerte können auch unter anderem als Zahl (As Integer) oder Text (As String) erfolgen.

Das Ende einer Subroutine wird mit End Sub oder End Function definiert. Der in einer Function berechnete Wert wird mit Return <Wert> zurückgeliefert.

Bei unserer Arbeit werden wir fast ausschließlich mit diesen Programmteilen arbeiten, denn Visual Basic arbeitet mit Objekten und deren Ereignissen. Wer noch Basic aus alten Zeiten kennt, wird hier vergeblich die Programmstruktur vermissen. Dennoch ist viel Programmierarbeit erforderlich, um die Objekte mit ihrer Aufgabe zu versorgen. So soll ein Button, welches im Programm installiert wird, eine Aufgabe erledigen, wenn man es angeklickt. Die Subroutine für das Ereignis erstellt das Objekt selbst, wenn es angewählt wird. Es entsteht lediglich ein Rahmen mit Sub <Name> und End Sub. Wenn wir unser Programm schreiben, gibt es auch oft Programmteile, die noch nicht programmiert werden können oder wo wir noch nicht genau wissen, wie

eine Aufgabe zu lösen ist. Dann setzen wir ebenfalls Subroutinen ein, die lediglich mit Namen und dem Ende definiert sind. So kann diese Programmierung der Aufgabe zu einem späteren Zeitpunkt angegangen werden. Das Programm selbst bleibt aber compilierbar und somit können andere Programmteile erst einmal getestet werden. Diese Technik werden wir zu gegebener Zeit ausgiebig anwenden. Beginnen wir nun mit einem Programm in Visual Basic, welches uns die Brücke zu einem Mikrocontroller liefert.

### *1.1.5 Visual Basic 2008.*

VB, in Worten „Visual Basic“ ist eine objekt- und ereignisorientierte Programmierung. Vergesst diesbezüglich einfach das Wort „Programmschleife“. Stellt euch einen Lego-Kasten mit vielen verschiedenen Bausteinen vor. Jeder Baustein für sich ist ein Objekt. Steckt man diese Bausteine zusammen, ergibt es ein Gebäude oder etwas anderes. Einige Bausteine haben auch Funktionen. So gibt es Antriebe, Räder, Getriebe und Gelenke. Um nun ein Gebilde zu erstellen, erfordert es eine Fläche, auf der das Geschaffene dann seinem Zweck zugeführt wird.

Nun können auf einer Fläche weitere Flächen sein, bebaut. Verschiebt man diese Flächen, werden die darauf befindlichen Gebäude mit verschoben. Aus Objekten sind neue Objekte entstanden. Doch damit ist das Szenario längst nicht am Ende. Wem Lego nix sagt, dann halt ein Meer. Schiffe schwimmen darauf, auf den Schiffen laufen Kellner mit Tabletts voller Getränke…

Visual Basic kann man sich ebenfalls so vorstellen. Das Programm bietet in einer Toolbox Steuerelemente an, die sich auf die Arbeitsfläche ziehen lassen. Einige davon sind in der Lage, ebenfalls Steuerelemente aufzunehmen, analog zum Tablett mit den Gläsern…

Dadurch erreicht man, dass mit wenigen Befehlen ganze Gruppen ihre Eigenschaft verändern. Nehmen wir ein ***Panel*** und packen da ein paar Steuerelemente drauf. Jedes Steuerelement besitzt die Eigenschaft *Visible* Setzt man diese auf **False**, wird das Steuerelement zur Laufzeit nicht angezeigt. Will ich mehrere ausblenden, müssen alle die Eigenschaft *Visible* auf **False** gesetzt bekommen. Wird die Eigenschaft des ***Panels*** *Visible* auf **False** gesetzt, so ist nicht nur das ***Panel***, sondern auch alles darauf befindliche nicht mehr sichtbar. Es gibt viele weitere Vorteile, die ich mir auch in meinem Programm zunutze machen werde.

Ein weiterer Vorteil von solchen Objekten ist ihre Eigenschaft, auf Ereignisse zu reagieren. Nehmen wir eine ***Textbox***. Sie enthält einen kompletten Texteditor, löschen, markieren, Groß- und Kleinschrift, Schriftarten und vieles mehr. Nun möchte ich nur die Eingabe von Ziffern erlauben. Zum Beispiel, weil der Wert in der Textbox in ein Zahlenformat gewandelt werden muss. Es ist das Ereignis des Tastendruckes, welches ein Zeichen in die ***Textbox*** einträgt.

Dieses Ereignis *KeyPress* ruft eine Routine *-.Keypress* auf. In dieser Routine prüfe ich nun den Code der Taste und wenn nicht in „0“-„9“ dann lösche ich einfach den Inhalt der zugehörigen Variable. Zum Verständnis:

Um den Tastencode zu löschen darf da nicht „0“ hineingeschrieben werden, das ist ja auch ein Code. Es wird einfach ein Leerstring, also zwei Gänsefüßchen ohne was dazwischen zugewiesen.

“das ist ein String“

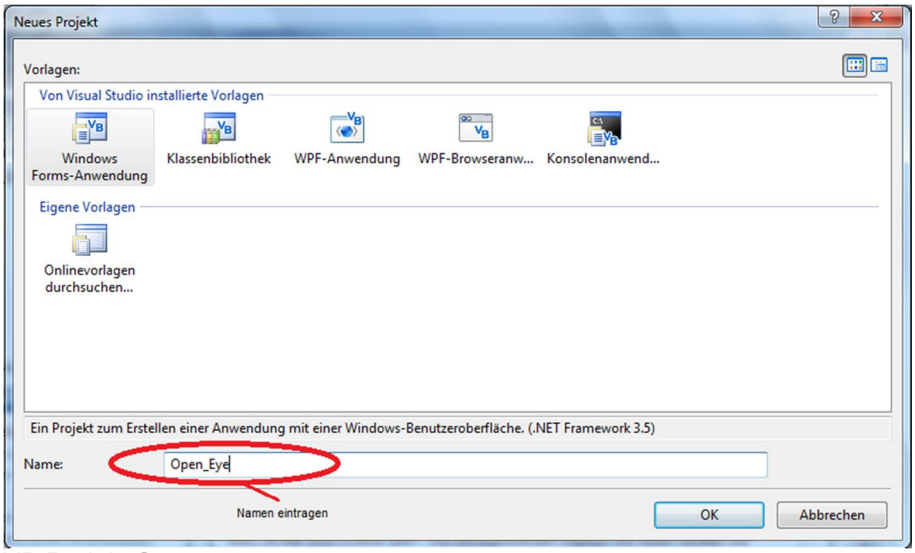
““ -> das ist ein leerer String

Eine entsprechende Anwendung habe ich auch in meinem Programm, daher gehe ich hier nicht näher darauf ein. Objekte lassen sich aber auch zur Laufzeit erzeugen. In meinem Fall ist es eine ***Textbox***, die den Wert einer zugehörigen Variable im Controller sichtbar macht. Da ich zur Entwicklungszeit gar nicht weiß, wie viele Variablen ich darstellen will, kann ich erst bei Benutzung des fertigen Programms, also zur Laufzeit die Anzahl der ***TextBoxen*** ermitteln. Dass ich da ein wenig tricksen muss, um ein gewünschtes Ergebnis zu bekommen, werde ich dann zu gegebener Zeit genauer erklären. Im Moment soll es auch erst einmal genug um die objektorientierte Programmierung sein. Ich denke, auch wenn noch viele Fragen offen sind, ist ein erster Einblick schon mal gegeben. Aber beginnen wir mal ganz am Anfang.

### *1.1.6 Installation von Visual Basic*

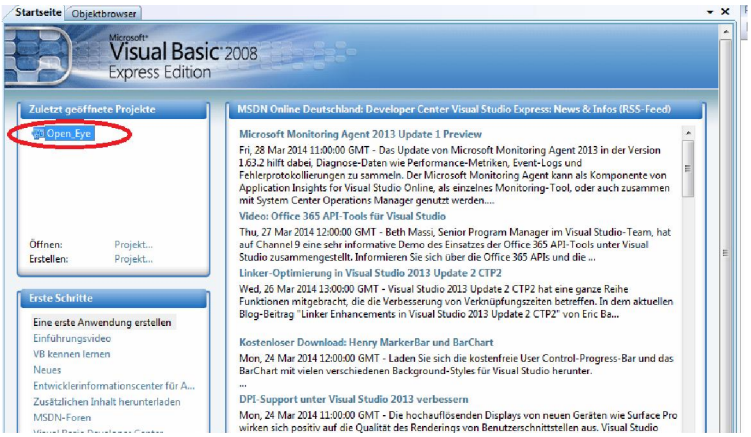
Nachdem Visual-Basic von Netz geladen ist, wird die Installation nach Vorgabe durchgeführt. Dies sollte keinerlei Probleme bereiten. Danach steht VB als Anwenderprogramm zur Verfügung.

Nun, der erste Start. VB verlangt ein vorhandenes oder ein neues Projekt und da wir ja noch keins haben, erstellen wir ein Neues. Ich denke, das folgende Bild sagt mehr als tausend Worte. Ich hab auch nicht vor, eine Bedienungsanleitung für VB zu schreiben, daher fällt dieser Part auch relativ kurz aus und beschränkt sich auf ein paar Screenshots.




*VB Projekt Start*

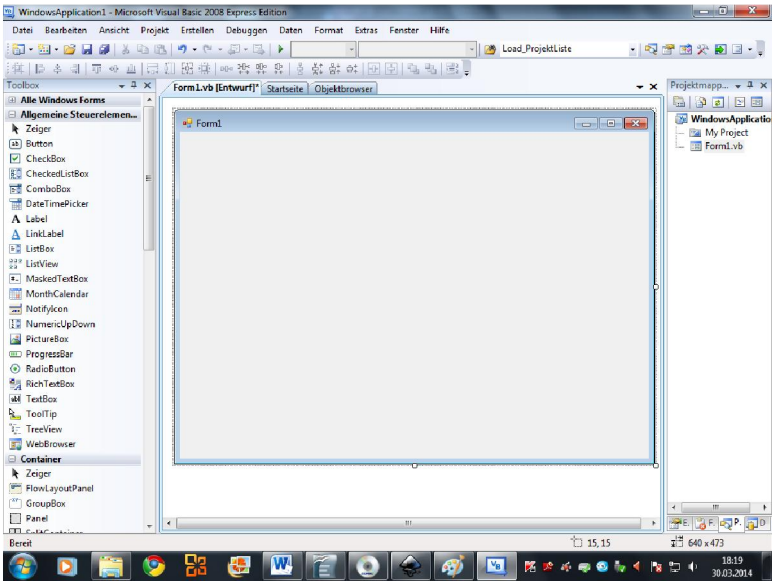
Zu einem späteren Zeitpunkt kann dieses Projekt direkt über „zuletzt geöffnete Projekte“ geladen werden. Diese Möglichkeit wird in der Startseite von VB auf der linken Seite angeboten.




*Start mit Projektliste*

## Arbeiten mit Visual Basic

Beim Start erhält man eine Arbeitsfläche. Sie ist ein Objekt und von Hause aus schon mit ein paar Funktionen zur grundsätzlichen Bedienung ausgestattet



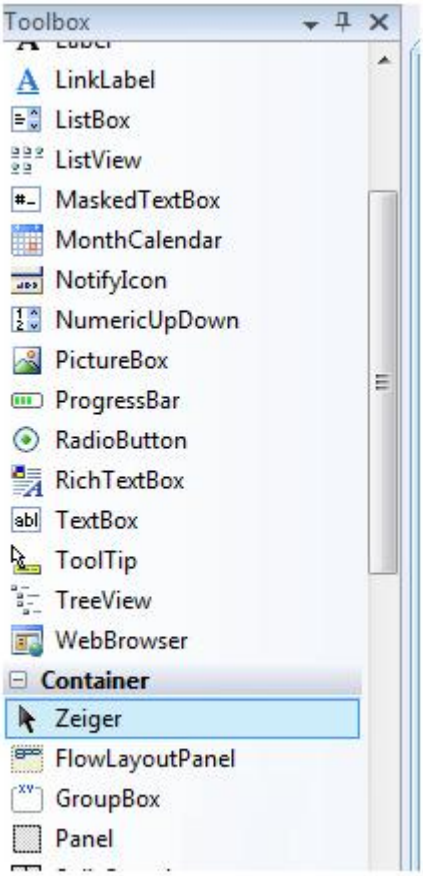

*VB erste Ansicht Projekt*

Das Bild zeigt die Form, so wird diese Arbeitsfläche genannt, eines neuen Projektes. Die Form wird auf die erwünschte Größe gezogen, damit alle erforderlichen Objekte untergebracht werden können. Eine leere Form lässt sich schon als Programm starten. Es erscheint einfach nur ein weißes Fenster, welches minimiert oder maximiert werden kann. Mit der Maus lässt sich dieses Formular auch hin und herschieben oder in der Größe verändern. Wohlgemerkt, wir haben nicht eine einzige Zeile Code geschrieben und doch funktioniert unser erstes Programm schon mit einer komfortablen Bedienung. Aber anfangen können wir mit einem solchen Programm nichts. Da muss noch mehr eingebaut werden. Nehmen wir uns nun die (denkt dabei an Lego!) Bausteine vor.

Es sollte euch nicht entgangen sein, dass unter der Menüzeile eine Icon-Zeile ist. Mittlerweile ist es üblich, Menübefehle in Icons abzubilden und durch Anklicken der Icons die Menübefehle aufzurufen. Suchen wir zuerst mal die Toolbox. Sie ist vergleichbar mit den Legokasten, in dem sich die Bausteine befinden.


*VB Iconleiste*

Zu Toolbox könnte der Begriff *Werkzeugkasten* passen und zum Begriff Werkzeug der Hammer. Somit suchen wir ein Icon, in dem irgendein Bild eines Werkzeuges zu erkennen ist. Wenn ich dieses Icon dann anklicke öffnet sich der Werkzeugkasten und die verfügbaren Objekte werden in einer Auswahlliste angeboten.




## Toolbox

Sie ist auf dem Bild der ersten Ansicht vom Projekt auf der linken Seite zu sehen. Diese Objekte kann ich nun auf mein Arbeitsblatt, das Formular einbauen. Dazu wird ein Objekt durch anklicken ausgewählt und ein Klick auf das Formular fügt dieses Objekt an der gewählten Stelle ein. Nun kann es mit der Maus an die richtige Stelle gezogen und mit den Richtungstasten exakt positioniert werden. Zu gegebener Zeit wird auf die Icons in der Toolbox noch einmal hingewiesen.

Da könnte man eigentlich auch schon loslegen und ehrlich gesagt, hab ich im ersten Anlauf auch getan. Aber es ist nicht klug und um nun die folgenden Schritte zu verstehen, muss man sich ein wenig mit den Eigenheiten von Objekten befassen.

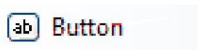
Um es verständlich zu machen, erinnere ich noch mal in die Legobox.

Auf meinem Arbeitsbereich habe ich eine große Legoplatte. Auf diese Platte lege ich mehrere kleinere Platten und auf diese kleineren Platten baue ich Häuser, Getriebe oder andere Bauwerke auf. Nehme ich nun die kleineren Platten, so nehme ich automatisch auch das darauf befindliche Gebilde mit. Gleiches gilt für die nächst größeren Platten, die natürlich auch die darauf befindlichen Gruppen mit bewegt. Diese Struktur ist in ähnlicher Art in den objektorientierten Programmierungen vorhanden. Die Geschichte mit der Sichtbarkeit von Objekten hatte ich ja bereits angesprochen. Um sie zu nutzen, müssen die Gruppierungen erkannt werden.

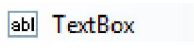
Es ist durchaus kein Fehler, Objekte einfach auf das Formular zu packen, doch sinnvoller ist es, Gruppen zusammenzufassen. So ist durch einen einzigen Befehl: „Mach dich unsichtbar“ gleich eine komplette Gruppe verschwunden. Oder es wird ein anderes Panel in den Vordergrund geschoben, sozusagen eine andere Seite auf geblättert, ohne gleich ein anderes Formular zu erzeugen. Aha werden nun einige sagen, ist ja einfach.

Da leg ich doch einfach auf meine Form ein paar ***Panel***-Objekte und platziere darauf Schalter und Textfelder.

*ICON Panel*



*ICON Button*



*ICON TextBox*

Ja, genau so geht es im Prinzip.

Die Kunst ist nun das Herausfinden der richtigen Objekte und das Füllen mit der richtigen Befehlsfolge, welches jedes Objekt für sich, und das ist wichtig, abzuarbeiten hat.

An dieser Stelle mal wieder ein einfaches Beispiel:

Stellt euch eine Firma vor, in der mehrere Mitarbeiter werkeln. Aufträge werden vergeben, vom Kollegen, vom Vorgesetzten, vom Kunden, alles möglich. Es ist immer einer, der dann speziell diesen Auftrag abarbeitet. Mal ganz allein, mal mit Kollegen. Jeder ist aber dabei eigenständig.

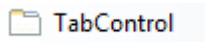
Zurück zu unseren Mitarbeitern, den Objekten. Haben verschiedene gleiche Aufgaben, werden Subroutinen von den Objekten aufgerufen. Nun muss aber erst ein Ereignis eintreffen, welches eine Bearbeitung

anfordert. Wie bereits geschrieben, die Geschichte mit der ***Textbox***, welche nur Ziffern aufnehmen darf. So kann es durchaus sein, das es mehrere ***Textboxen*** gibt, die diese Beschränkung erhalten sollen. Da greift man auf eine Funktion zurück, die das Tastaturzeichen prüft und zurückliefert, entweder mit einer gültigen Ziffer oder aber leer. Wenn wir die leere Form einmal betrachten, dann überlegen wir in der Regel, wie nun die Oberfläche gestaltet werden soll. Dass so eine Form nicht unendlich groß ist und die Bildschirmgrenzen nicht sprengen sollte, bedeutet schon eine gründliche Planung. Alle erforderlichen Objekte werden nicht auf die Form passen, oder doch? Vielleicht ist es ja möglich, nur die erforderlichen Objekte für die einzelnen Schritte darzustellen, während die anderen nicht sichtbar im Hintergrund liegen?

Eignen sich dafür die ***Panel*** oder gibt es andere Lösungen.

Manchmal muss man etwas primitiv denken, um zur Lösung zu kommen. Ein Ordner hat z. B. Tabulatoren, die einzelne Schriftstücke in Gruppen zusammen fügt.

Suchen wir nun in der Toolbox einen Begriff, der auf Tabulatoren hinweist und wir werden fündig bei ***TabControl***. Das Icon zeigt auch, das es sich um ein geeignetes Objekt handelt.




*ICON TabControl*

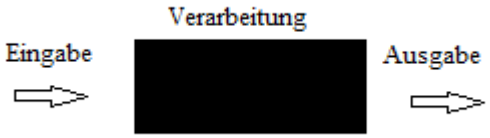
Doch lasst uns den Einstieg etwas ruhiger angehen und erst ein paar kleine Übungen durchführen. Sie sollen die bisher angesprochenen Befehle festigen und Programmabschnitte in der Ausführung verdeutlichen.

Es ist sicherlich verwirrend, wenn man das erste Mal mit Programmzeilen arbeitet. Die Beschreibung mag noch so gut sein, der Aha-Effekt stellt sich erst ein, wenn die Zeilen selbst geschrieben und in den empfohlenen Abschnitten auch getestet werden. Doch die Vorlage ist ja nur für diesen einen Zweck gültig. Wie baut man sich eigene Programmstrukturen? Gibt es Regeln oder ein Schema, welche allgemeine Hilfe beim Aufbau von Programmen liefern?

Die Antwort : Jain

Programmieren ist wie Bücher schreiben. Jeder Autor hat da so seinen eigenen Stil. Aber ein paar nützliche Tipps kann ich wohl geben.

Die Vorgehensweise beim Programmieren erkläre ich gern mit einer Blackbox, einem geschlossenen Kasten, dessen Inhalt erst einmal unbekannt ist. Er hat zwei Öffnungen, eine, um etwas hinein zu geben und eine für etwas, das ich haben will.




*Prinzip Blackbox*

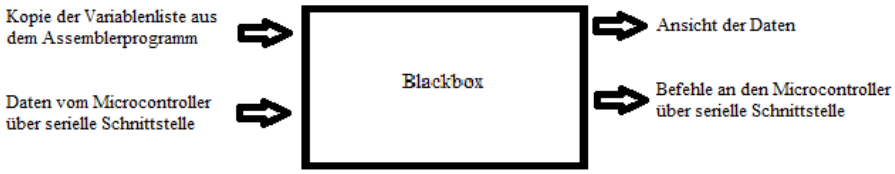
So arbeitet auch das Programm Open_Eye:

Es wird eine Kopie der Variablendeklaration aus dem Assemblerlisting hinein gegeben und heraus möchte ich Information über die Inhalte der dort definierten Speicherzellen erhalten.

Will man nun wissen, wie dieser Kasten funktioniert, muss man die Lösung im Inneren suchen.

Neugierig, wie man nun so von Hause aus ist, öffnet man einen Deckel und findet weitere schwarze Kisten die nach und nach geöffnet ihre Geheimnisse preisgeben.

Ich gehe ähnlich vor, indem ich einfach schwarze Kästen erzeuge und nach und nach deren Deckel öffne und in jede eine Aufgabe hinein fülle. Die Öffnungen für die Eingaben werden mit Öffnungen von Ausgaben anderer Blackboxen verbunden. Die Kisten bekommen Namen, die ihre Aufgabe, die sie erfüllen sollen, beschreiben. Irgendwann funktioniert das Ganze im Zusammenspiel wie es soll. Um damit arbeiten zu können muss ich wissen, welche Information für Eingaben vorhanden sind und welche Ergebnisse gebraucht werden.




*Aufgabe mit Blackbox beschreiben*

Solch eine Aufgabe in einem Kasten ist vielleicht mit einer Skizze darzustellen. Alles, was eine Aufgabe zu lösen hilft, kann angewendet werden. Je kleiner so ein Kasten, eine Aufgabe ist, umso einfacher ist es, sie zu beschreiben und zu programmieren.

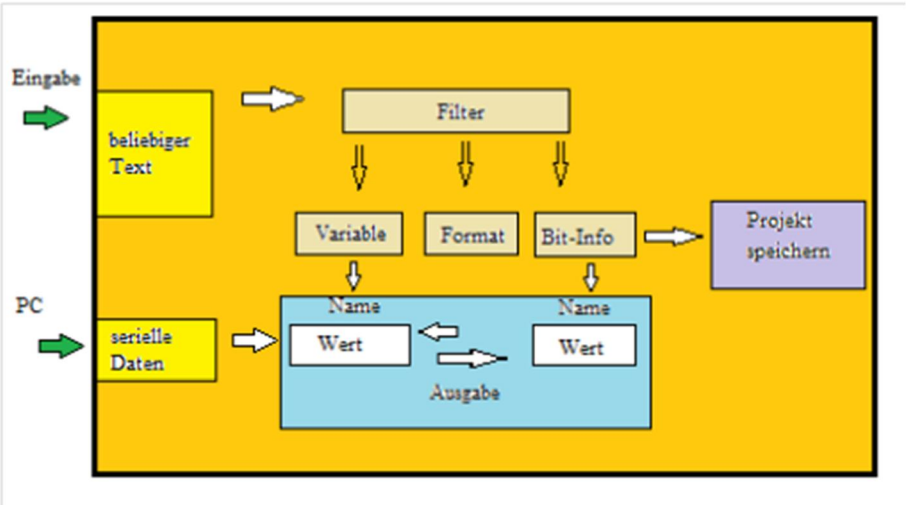
Versucht zum Beispiel einmal eine Handbewegung für eine Tätigkeit bis ins kleinste Detail zu zerlegen und die einzelnen Abläufe zu beschreiben. So wird bewusst, wie komplex wir denken.

Der Griff zu einem Apfel, kaum jemand denkt darüber nach, das die Muskeln der Finger der Hand erst öffnen und die Armmuskeln den Arm von der Schulter her über den Ellenbogen in Richtung Apfel ausstrecken müssen. Erst, wenn der Apfel erreicht ist, können die Fingermuskeln die Krümmung der Finger veranlassen. Das ist noch viel zu komplex beschrieben, der Programmierer muss da viel tiefer eintauchen. Er muss mit analytischem Denken eine Aufgabe in viele kleine Aufgaben zu zerlegen, bis er die Schritte einfach mit ein paar Worten beschreiben kann.

Das Programm ist zwar vorgegeben, doch versucht an einigen Stellen einfach mal selbst einen Lösungsansatz. Bis auf wenige Unterprogramme sind die Routinen klein gehalten und sollten auch leicht verstanden werden.

### *1.1.7 Ein Programm skizzieren*

Ein Programm in seinem Aufbau zu skizzieren ist sehr hilfreich. Das muss nicht perfekt jede kleine Teilaufgabe widerspiegeln. Es reicht eine einfache Struktur auf ein Blatt Papier zu bringen. Schon beim Erstellen solcher Vorlagen erkennt man weitere Teilaufgaben, die in dieser Art erst einmal mit aufgenommen werden können. Bei einem Blatt Papier heißt es „in die Tonne“, wenn der Ansatz nicht stimmt. Warum nicht das Werkzeug nutzen, was letztlich vor uns steht, den PC. Mit PowerPoint oder ähnlichen Programmen, ja sogar mit Paint, wenn auch etwas mühsamer, lassen sich solche Skizzen erstellen und auch bei Bedarf anpassen. So hat man die Vorlage immer vor Augen und später einmal eine gute Dokumentation.




*Programmblockbild*

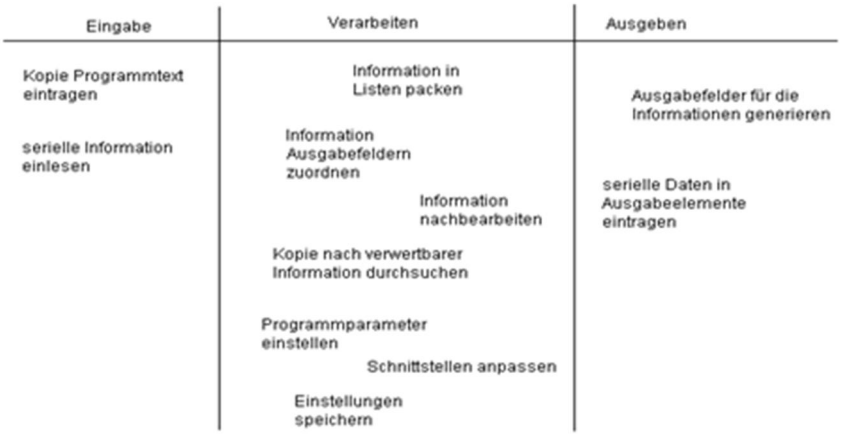
Dieses Bild zeigt noch ziemlich komplex, wie unser Programm arbeiten soll. Da ist das Feld beliebiger Text der in einem Filter zerlegt wird und die Informationen liefert **Variable**, **Format** und eventuell **Bitinfo**. Die ermittelte Information wird für spätere Verwendung gespeichert. Nun soll aus den seriellen Daten der Wert zur Ansicht gebracht werden, und das passend zum Format.

Ein solches Bild zeigt, wie diese Aufgabe gestartet werden muss. Ohne die Information **Variable** und **Format** kann ich empfangen was ich will, es wird mir nichts nützen. Auch ein Provisorium ist kaum machbar. Aber der

Filter liefert diese Voraussetzung und da er sowieso erforderlich ist, wird auch damit sinnvoller Weise begonnen.

Im Weiteren ist auch erkennbar, welche Module das Programm erfordert. Da schreiben wir doch einfach mal auf, was uns dazu so alles einfällt. Völlig wirr und ungezwungen. Ein wenig grob sortiert vielleicht:

was gehört in welche Kategorie – Eingabe – Verarbeitung – Ausgabe?
Dies lässt sich doch leicht zuordnen. Schauen wir uns nun einmal die skizzierten Schritte an.



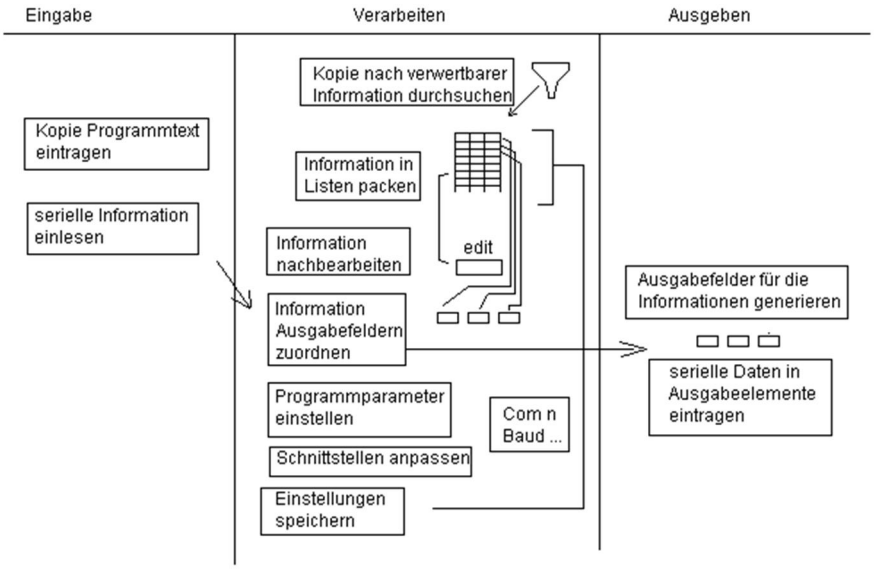

*EVA Grundprinzip*

So ist klar, ein Filter zum Start. Sind die Informationen herausgefiltert, werden sie zur Bearbeitung verfügbar gemacht und auch abgespeichert. Die Nachbearbeitung ist dann sozusagen der zweite Schritt.

Nun kann die Ausgabe generiert werden, da für alle Variablen die erforderliche und gewünschte Information bereit liegt.

Es fehlt nun noch die serielle Verbindung zum Controller und einer Möglichkeit, diese für jedes Projekt anzupassen. Das wäre dann der vierte und letzte Schritt.

Je detaillierter eine Skizze vom Programm und den Aufgaben ist umso einfacher lassen sich die Zusammenhänge und damit die Folge der Bearbeitung erkennen. Sortieren wir nun zuerst einmal unseren *Gedankensturm* und formen dann ein weiteres Bild.




*EVA Prinzip beschreiben*

Auf diese Art beginnt man, sein Programm zu definieren. Es ist nicht gleich perfekt, nicht vollkommen und einiges wird sich in der Ausführung hinterher anders darstellen. Aber wir bekommen ein Bild von dem Programm. Es ist nicht mehr so abstrakt.

### *1.1.8 Eine kleine Vorschau auf Variablendeklarationen und Befehle*

Für die Arbeit mit Visual Basic ist es erforderlich, etwas über Formate von Variablen zu wissen. Bevor diese benutzt werden können, müssen sie dem Programm bekannt sein. Dafür gibt es die **Dim-Anweisung:**

Dim Ganze_Zahl as Integer '(1234)

Dim Komma_Zahl as Real '(12,34)

Dim Satz as String '(„1234“ aber auch „ein beliebiger Text“)

Diese Liste ist noch lange nicht vollständig, doch für das Grundverständnis sollte es vorerst ausreichen.

**Bedingte Wiederholungen oder Schleifen:**

While <Bedingung>

….

End While

Wird solange durchlaufen, bis Bedingung nicht mehr erfüllt ist

Beachte: diese Schleife prüft am Anfang, ob eine Bedingung erfüllt ist.

For <zähler> = <Anfang> to <Ende>  '(optional Step <Weite>)

…..

Next <Zähler>

Diese Schleife hat eine feste Anzahl von Durchgängen, die durch die Werte in <*Anfang*> und <*Ende*> definiert ist. Die Zählvariable wird bei

jedem Schleifendurchlauf mit der Anweisung *Next* um 1 erhöht. Man kann aber auch optional die Schrittweise mit *Step* <*Weite*> vorgeben.

**Umwandlungen von Variablentypen:**

Das Umsetzen von Zahlen in einen String und umgekehrt.

Str( Zahl) 'ergibt einen String

Val(String) 'ergibt eine Zahl aus einer Stringzahl

Also aus Str(123) wird „123“ und aus Val(„123“) wird 123.

**Kopierfunktion für Teile aus einem String:**

Die Funktion *Mid(String, Anfang, Länge)* liefert einen String beginnend mit dem Zeichen (Anfang)

und der Anzahl Zeichen ( Länge)

Mid(„Mein String“, 1, 4) 'ergibt „Mein“

Mid(„Mein String“,6,5) 'ergibt „String“

**Erfassen der Länge eines Strings:**

*Laenge=Len(Buffer_Str)* liefert die Anzahl der Zeichen innerhalb eines Strings.

**Erfassen einer Position eines bestimmten Zeichens:**

InStr(Buffer_Str,“:“)

liefert die Position des ersten gefundenen Zeichens in der Stringvariablen *Buffer_Str*. Diese Funktion sucht auch Teile einer Zeichenkette und liefert die zuerst gefundene Funktion.

*InStr(„Mein String“, „ein“)* liefert im Ergebnis eine 2

**Vergleiche:**

```
If <Bedingung> then

….

End if
```

Die Bearbeitung im If-Teil erfolgt nur bei erfüllter Bedingung.

```
If 3<>2 then        ‘wird ausgeführt, weil wahr

If 3=2 then         ‘wird nicht ausgeführt, weil falsch
```

Auch diese Angabe der Befehle ist nur ein geringer Teil der möglichen Befehle. Beim Aufbau des Programms werden noch einige dazu kommen. Weitere Information zu Befehlen und Funktionsaufrufen liefert die Hilfe.

## Zu Bedingung:

Eine Bedingung kann wahr oder falsch sein. Müssen mehrere Bedingungen beachtet werden, ist die boolesche Algebra anzuwenden. Wie in der Mathematik sind hier Klammern zu setzen, um eine Reihenfolge einzuhalten. Hier mal ein paar Zeilen dazu:

```
Dim Bedingung as Boolean

Bedingung = 3<7
```

Das Ergebnis kann nur richtig sein, denn 3 ist kleiner 7. Also ist die Bedingung **wahr**. Das ist ja noch einfach, aber nun kommt es dicke…

```
If ( ( InStr(Buffer_Str,„:“)>0) and (InStr(Buffer_Str ,„:“)< InStr(Buffer,(Chr(10))))) then
    CopyStr=Mid(Buffer_Str,1, Instr(Buffer_Str„:“)-1)
end if
```

Das lässt sich besser lesen, wenn man hier Variablen deklariert und die Position von „:“ und Chr(10) im String vorher berechnet. Einzelne Bedingungen werden in Klammern zusammengefasst.

```
X= InStr(Buffer_Str,„:“)          ‘x enthält die Position vom ersten Doppelpunkt
Y=InStr(Buffer_Str,Chr(10))   ‘y enthält die Position vom ersten Zeilenende
If (x>0) and (x <y) then
     CopyStr=Mid(Buffer_Str,1, x-1)
    ‘bedeutet, es muss ein Doppelpunkt vorhanden und die Position vor einem Zeilenende
sein.
End if
```

Noch ein Hinweis zu dem Befehlssatz. Beachten wir ein paar Grundregeln.

Einige Befehle befinden sich Sprachraum von Basic. Andere sind in den Objekten enthalten. Wie bereits erwähnt setzen sich diese aus dem Objektnamen, einem Punkt und der Eigenschaft, der Methode oder einem eingebundenen Objekt, welches wiederum Eigenschaften besitzt, zusammen.

Es ist nicht immer gleich zu erkennen, ob dieser Befehl aus einem Objekt oder aus VB kommt. Allerdings gibt es die Taste F1, die uns zum aktuellen Befehl Hinweise und Informationen liefert. Auch wird schon beim Editieren erkannt, ob es sich um einen Bestandteil eines Befehls oder Objektes handelt und entsprechende Online – Hilfe geboten. Trotzdem sucht man halt immer wieder nach Begriffen, die bestimmte Befehle beinhalten. Manchmal ist es da hilfreich, einfach mit einem Buchstaben zu beginnen und abzuwarten, wie weit die Hilfe da mitgeht oder was sie vorschlägt.

### *1.1.9 Kommentarzeilen*

Und weil wir auch schon mitten in der Programmierung sind, möchte ich euch die Nutzung von Kommentarzeilen ans Herz legen.

Ein Programm wird einerseits über das Einrücken von Programmabschnitten lesbar, aber man kann mittels Hochkommata auch Zeilen einfügen, die Unterprogramme oder die Aufgabe einer Programmzeile beschreiben

```
,-----------------------------------------
,*   hier steht nun die Erklärung     *
,*   zur Subroutine                   *
,-----------------------------------------
```

Manchmal möchte man eine einzelne Zeile kommentieren

```
CopyStr=Mid(Buffer_Str,1, x-1) ‘ Ein Stück aus der Variablen Buffer herauskopieren
```

Oder vielleicht auch mal eine Programmzeile ausblenden

```
‘CopyStr=Mid(Buffer_Str,1, x-1)
```

Die grüne Einfärbung markiert Text, der vom Compiler ignoriert wird. Das kann bei der Fehlersuche sehr hilfreich sein. Ist das Programm mit vielen kleinen Subroutinen aufgebaut, so bewirkt das Kommentarzeichen vor dem Aufruf einer Subroutine, dass diese nicht mehr aufgerufen wird. Das Programm wird aber nicht großartig verändert und nach dem Test ist dieses Kommentarzeichen einfach wieder zu entfernen, um die Funktion oder Subroutine wieder einzubinden.

Bei umfangreichen Programmen könnte es sein, das das Auffinden dieser Zeilen nach einer Arbeitsunterbrechung schwierig und umständlich ist. Aber was hindert uns, nicht nur das Kommentarzeichen, sondern noch ein

Sonderzeichen und vielleicht einen Hinweistext abzulegen. Die Suche nach Kommentarzeichen und dem Sonderzeichen findet dann die entsprechend ausgeblendeten Befehle und so kann die Arbeit auch gezielt wieder aufgenommen werden.

Hier mal ein Beispiel anhand eines Code-Abschnitts.

```
TB_TriggerWert.Text = "0"
Load_Projekte()                    ' Zugriff auf Datenbank zum Laden der Projektliste
If CB_Projekte.Items.Count > 0 Then
   Load_Akt_Projekt(TB_Projekt.Text)  ' Wenn Projekte vorhanden sind, wird das erste
Projekt geladen
End If

TB_TriggerWert.Text = "0"
'!-Subroutine noch nicht erstellt -! Load_Projekte()
' Zugriff auf Datenbank zum Laden der Projektliste
If CB_Projekte.Items.Count > 0 Then
   '!-Subroutine noch nicht erstellt -! Load_Akt_Projekt(TB_Projekt.Text)
   ' Wenn Projekte vorhanden sind, wird das erste Projekt geladen
End If
```

Der erste Codeabschnitt ist das fertige Programm und die Aufrufe **Load_Projekte** sowie **Load_Akt_Projekt(TB_Projekt.Text)** werden ausgeführt. Wenn beim Programmaufbau aber die Subroutinen noch nicht erstellt sind und das Programm trotzdem mit der bisherigen Programmierung getestet werden soll, würden diese Zeilen eine Fehlermeldung liefern. Da sie aber nicht ausgeführt werden, weil der Compiler an diesen Stellen keinen Code, sondern Kommentare sieht, bleibt der übrige Code lauffähig. Die ausgeblendeten Befehle aber sind in der Suche einfach mit dem Suchbegriff **'!-** wiederzufinden.

### *1.1.10 Die Struktur von Quellcode*

An dieser Stelle möchte ich noch ein paar Worte zur Lesbarkeit von Quellcodes sagen. Soweit es möglich ist, werden Befehle in einer Zeile gehalten. Dies gelingt nicht immer, wie es im Anhang beim Quellcode sichtbar wird.

Ein wenig entgegenkommend sind die Bereiche If … end if.

Folgende Schreibweise ist durchaus gängig:

```
If (CR_Pos = 0) And (Len(Buffer_Str) > 5) Then: CR_Pos = Len(Buffer_Str)‘ letzter Eintrag
```

Wie leicht zu erkennen ist, zeichnet sich der **If**-Zweig überhaupt nicht ab. Dagegen ist die folgende Schreibweise wesentlich übersichtlicher und die Zeilen sind auch viel kürzer

```
If (CR_Pos = 0) And (Len(Buffer_Str) > 5) Then
    CR_Pos = Len(Buffer_Str) ‘ letzter Eintrag
End If
```

Erklärung:

**Len(<String>)** liefert einen Integerwert der Anzahl Zeichen im String.

Beispiel:

`Len(„Dies ist ein Text“)` liefert die Integerzahl **17.**

Mit der *Zeile* `CR_Pos = Len(Buffer_Str)` wird der Wert der Variable **CR_Pos** zugewiesen

Durch das Einrücken des Textes zwischen **If**, **While**, **For** und dem zugehörigen **End** kommt der Code jedes Mal ein Stück weiter nach rechts. Trotzdem bleibt der Code selbst bei umfangreichen Programmen lesbar. Das ausführbare Compilat wird deshalb nicht größer oder langsamer.

Eine weitere Möglichkeit zur Verbesserung der Lesbarkeit sind Kommentarzeilen. Niemand bestimmt, dass Kommentare nur am Ende von Befehlszeilen angehängt werden dürfen. Ein Kommentarblock vor einer Programmroutine mit Hinweisen auf Funktion erlaubt auch noch nach einer langen Pause wieder den Einstieg und verrät dem Programmierer, was er damals mit diesem Programmteil bezweckt hat. Auch werden deutlich Programmbereiche getrennt und das Listing übersichtlicher. Die Projektprogramme sind mit vielen solchen Kommentarbereichen ausgestattet.

So kann eine komplizierte Funktion mit einem Kommentarblock noch etwas näher erklärt werden.

```
‘************** Eine komplizierte Funktion ***************
‘*                                                       *
'*   Hier steht nun die Erklärung, die verwendeten Variablen *
‘*    oder die Bedeutung von Signalbits.                 *
‘*                                                       *
‘*******************************************************
```

Es gibt Leute, die behaupten, Basic sei eine erklärende Sprache und jedwede Kommentierung sei überflüssig. Nun, diese Leute haben vermutlich noch nie in einem Programm eine Anpassung gemacht oder Fehler gesucht. Es ist schon wichtig, manche Programmschritte etwas ausführlicher zu kommentieren.

Ein Kommentar wie dieser

```
CopyStr=Mid(Buffer_Str,1, x-1) ‘ Ein Stück aus der Variablen Buffer herauskopieren
```

für solch eine Befehlszeile ist Blödsinn, denn **CopyStr** sagt genau das aus. Aber beispielsweise

```
'Kopiert Variablennamen aus Assembler- Quellcode
```

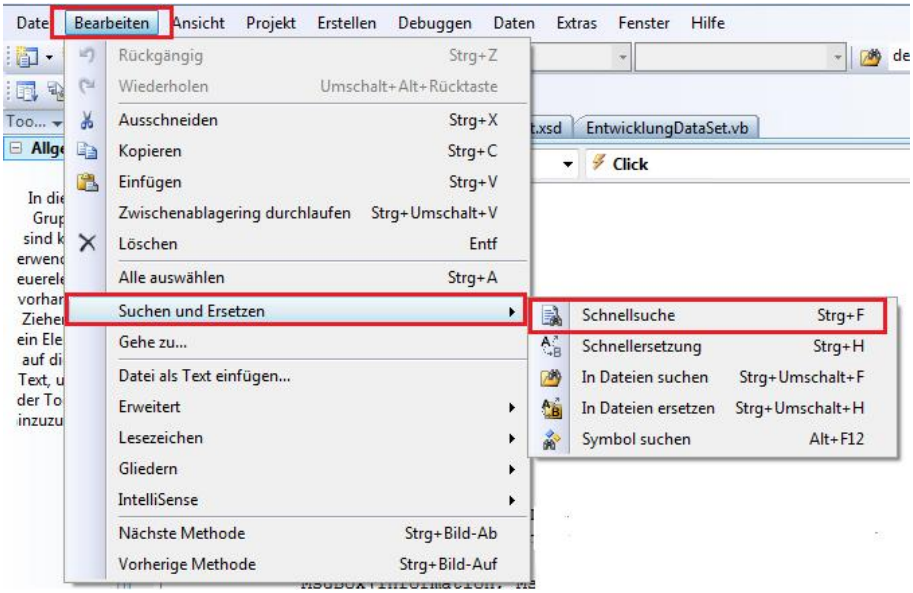
wäre ein hilfreicher Kommentar.

Sicherlich, ich hab da an einigen Stellen mit den Kommentaren stark übertrieben, aber es ist ja nicht für mich, sondern für euch. Wer die Programmteile verstanden hat, der braucht nicht jeden Kommentar übernehmen.

### *1.1.11 Quellcode durchsuchen*

Im Verlauf der Programmerstellung wird unser Programm ziemlich umfangreich. Dann eine Funktion oder Subroutine zu suchen wird zum Geduldsspiel, wenn man nicht so richtig weiß, welche Möglichkeiten das Programm bietet. Einmal ist schon die Struktur angesprochen worden. Das ist schon hilfreich, wenn es Anhaltspunkte gibt, wo das Gesuchte zu finden ist.

Aber es gibt auch eine Suchfunktion. In der Menüleiste unter Bearbeiten finden wir den Eintrag Suchen und Ersetzen. Die Schnellsuche werden wir sicherlich öfter benutzen, wenn unser Programm an Umfang zugelegt hat.



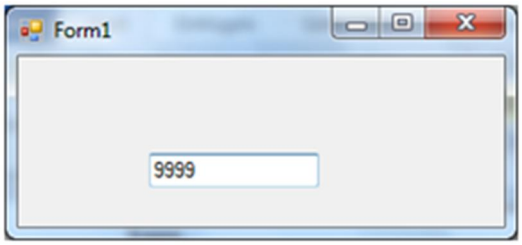

*Suchfunktion*

### *1.1.12 Aufgabe: Ein Zeichen vor der Eingabe „filtern“*

Ich habe hier mal einen kleinen Auszug als Beispiel, wie so eine Aufgabe gelöst werden kann.

Dazu ziehen wir eine ***Textbox*** auf das noch leere Formular.

*ICON TextBox*




*Übung mit Textbox*

Das *KeyPress*-Ereignis der ***Textbox*** wird aufgerufen, bevor ein Zeichen über die Tastatur im Textfeld eingetragen wird. Wir nutzen diese Funktion, um den **Key** prüfen und ihn eventuell verändern.

Das *KeyPress*-Ereignis der ***Textbox*** ruft diese Funktion auf:

```
Private Sub TB_Integerzahl_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TB_Read_Puffer.KeyPress
        e.KeyChar= Is_Ziffer_Code (e.KeyChar)
        ……
end sub
```

*e.Keychar* ist das Zeichen, welches von weiteren hier unsichtbar im Hintergrund arbeitenden Routinen an das Textfeld übergeben wird. Ich fange es sozusagen nur ab und prüfe es in einer weiteren Routine. Diese liefert mir den korrigierten Inhalt von *e.KeyChar* zurück und trägt ihn in *e.KeyChar* zur Ausgabe ein. Sub-Routinen, die einen Wert zurückliefern, bezeichnet man als Funktionen.

```
Public Function Is_Ziffer_Code(ByVal Key as Char) as Char
Dim Is_IntKey as Char
```

```
        Is_IntKey=Key
        If (Is_IntKey < “0”) or (Is_IntKey >“9“) then
            Is_IntKey =““
      End if
     Return Is_IntKey
End Function
```

Hier stellt sich dem aufmerksamen Leser klar die Frage: Warum muss ich die Funktion separat aufrufen. Wäre es nicht möglich gewesen, den Code auch in die Ereignisroutine zu schreiben?

Selbstverständlich, aber es ist vermutlich nicht nur ein einziges numerisches Eingabefeld vorhanden.

Damit nun nicht der Code x-mal geschrieben werden muss, wird nur die eigene

```
Function Is_Ziffer_Code(e.KeyChar)
```

aufgerufen. Das macht eine Programmierung einfacher. Und was an dieser Stelle besonders wichtig ist: Wird eine Korrektur erforderlich, weil zu einem späteren Zeitpunkt noch die Tabulatoren berücksichtigt werden sollen oder ein Komma für Realzahlen gebraucht wird, dann genügt die Änderung nur an einer einzigen Stelle, in der

```
Function Is_Ziffer_Code(e.KeyChar).
```

Ab dann haben alle Eingabefelder sofort das neue Verhalten.

Eine weitere Besonderheit sollte auch gleich erwähnt werden. In der Funktion ist eine Variable deklariert. Dies bedeutet, sie ist nur innerhalb dieser *Function* gültig, nirgends anders im Programm.

Eine solche Variable nennt man auch **Lokalvariable**, die im Gegensatz zur **Globalvariablen**

mehrfach im Programm in unterschiedlichen **Sub** oder **Function** deklariert werden kann. Durch Verwendung solcher Lokalvariablen wird ein Programm übersichtlicher und die Unterprogramme erhalten eine Unabhängigkeit vom Programm. Das schafft auch die Möglichkeit, diese Routinen in anderen Programmen wieder zu verwenden.

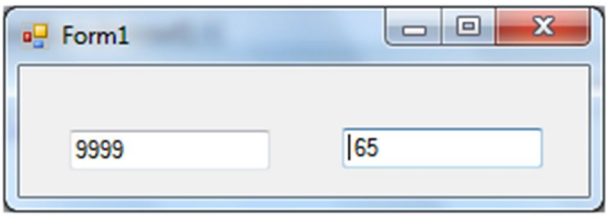
### *1.1.13 Funktionen erweitern*

Nicht immer ist eine so einfache Prüfung ausreichend und weitere Begrenzungen, z. B. die maximal zulässigen Stellen dieser Zahl, müssen hinzugefügt werden. Dazu wird die Funktion ***Len*** herangezogen, die bereits in Visual Basic vorgegeben ist. Sie liefert die Länge des Textes in einem String.

```
If Len(TB_IntegerZahl.Text)>3 then
      e.KeyChar=""
end if
```

Mit dieser Anweisung bekomme ich maximal *9999* in das Textfeld. Und nun geht auch nichts mehr. Auch kein zurück löschen oder irgendeine Cursor-Bewegung. Schuld ist *e.KeyChar=""*. Auch Steuerzeichen werden damit plattgemacht. Will man diese weiterhin behalten, müssen diese in der If –Klausel berücksichtigt werden. Nun hat nicht jeder die ASCII-Tabelle im Kopf. Welche Zeichen sind nun zur Steuerung innerhalb des Textes?

Auch dabei hilft ein kleiner Trick mit dem Ereignis *KeyPress*.




*Übung mit zwei TextBoxen*

Man nehme eine ***Textbox*** und programmiere ein KeyPress-Ereignis, welches einfach den ASCII-Code in das Textfeld schreibt

```
Private Sub Textbox2_KeyPress(ByVal sender As System.Object, ByVal e As System.Windows.Forms.KeyPressEventArgs) Handles TB_Read_Puffer.KeyPress
        Textbox2.Text=Str(ASC(e.KeyChar))
        e.KeyChar=""
end sub
```

Diesmal gibt es keinen Umweg über eine eigene Funktion. Warum auch. Das Textfeld wird ja später sowieso entfernt, da ich nur den Wert aus *e.KeyChar* brauche. So braucht die ***Textbox*** auch keinen besonderen Namen, da dies nur temporär zur Hilfe eingebaut ist und mit dem Programm nix zu tun hat. Es wird später entfernt. Auch interessiert nach Erfassung des ASCII-Codes der Inhalt von *e.KeyChar* nicht mehr, ja, er darf auch gar nicht mehr vorhanden sein, denn dann würde ich den zugewiesenen Text für die ***Textbox*** nachträglich verändern. Deshalb wird mit *e.KeyChar=""* der Inhalt gelöscht.

## Erklärung:

*ASC(<Char>)* liefert den ASCII-Code. Das Ergebnis ist eine Integerzahl

Beispiel: ASC(„A“) ergibt die Zahl 65

*STR(<Zahl>)* ergibt einen Zahlenstring

Beispiel: STR(65) ergibt „65“

CHR(65) ergibt ein ASCII-Zeichen

Beispiel: CHR(65) ergibt „A“

Da einer ***Textbox*** Text zugewiesen werden soll, erfolgt hier eine zweifache Wandlung. Innen wird der ASCII-Code ermittelt, der im zweiten Schritt in einen Text gewandelt wird, um ihn dann der ***Textbox*** zuzuweisen. Das gestartete Programm wird nun den Code jeder gedrückten Taste anzeigen. Nun ist es mir auch möglich, die sonst unsichtbaren Steuerzeichen mit in den Filter meiner Funktion zu übernehmen.

```
Public Function Is_Ziffer_Code(ByVal Key as Char) as Char
Dim Is_IntKey as Char
   Is_IntKey=Key
   If ((Is_IntKey < “0”) or (Is_IntKey >“9“)) and (Is_IntKey <>Chr(8) ) then
      Is_IntKey =""
   End if
   Return Is_IntKey
End Sub
```

Hinweis:
Man kann auch auf die Wandlung von Zahl nach String verzichten, da Textboxen auch Zahlen direkt aufnehmen können. Es gibt aber Bedingungen, wo eine Wandlung sinnvoll ist. In diesem Programm werden den Ausgabeobjekten Textboxen, Listboxen und Tabellen grundsätzlich Strings zugewiesen.

Die Vorgehensweise ist relativ einfach. Da bereits ein Bereich zugelassen ist, bzw. das verlassen des Bereichs die Variable *Is_IntKey* löscht, dürfen weitere zugelassene Bedingungen auch nicht zutreffen. Es ist hilfreich, wenn man sich erinnert, wie in der Mathematik mit Klammern gearbeitet wird. Für eine logische Verknüpfung trifft dies auch zu. Daher wird zuerst einmal die *Oder*-Verknüpfung eingeklammert und dann mit ***And*** eine weitere Bedingung angehängt. Dieses *und ungleich* erweitert die Bedingung für das Löschen der Variablen *Is_IntKey*.

Diese Art der Logik ist etwas verwirrend, da hier Ziel ist, den Inhalt von *Is_IntKey* zu löschen *und nicht* zu erhalten. Daher wird auch mit jedem weiteren zugelassenen Zeichen immer mit *und ungleich* gearbeitet.

**Erklärung:**

***Or*** zu Deutsch *Oder* bedeutet zulassen einer Alternative

***And*** zu Deutsch *Und* setzt beide Bedingungen voraus

Innerhalb von Basic wird mit Ergebnissen **true** (wahr) oder **False** (falsch) gearbeitet. Die Anweisung

```
If (((Is_IntKey < “0”) or (Is_IntKey >“9“)) And (Is_IntKey <>Chr(8) ) then
```

ist in etwa so zu übersetzen:

```
wenn (<Bedingung 1> oder <Bedingung 2>) und <Bedingung 3
```

Ersetzt man die ersten zwei Bedingungen der Klammer mit

```
Bedingung 4 =<Bedingung 1> oder <Bedingung 2>
```

bleibt

```
Wenn <Bedingung 4> und <Bedingung 3>
```

Vielleicht wird es so besser deutlich

```
If ( (Is_IntKey < “0”) or (Is_IntKey >“9“)) and (Is_IntKey <>Chr(8) ) then
wahr wenn Wert<0 wahr wenn Wert >9 wahr wenn nicht CHR(8)
```

Die erfüllte Bedingung löscht nun den Inhalt der Information für das Textfeld. Mit anderen Worten:

Befindet sich diese Information im Zahlenbereich oder ist sie ein Steuerzeichen, bleibt die Information erhalten, da keine Bearbeitung innerhalb der **If**- Abfrage erfolgt. Dabei gilt zu beachten, wir wollen bei erfüllter Bedingung den Inhalt von *Is_Int_Key* entfernen. Daher die *Und*-Verknüpfung.

### *1.1.14 Erkennen einer Änderung im Textfeld*

Es gibt weitere Ereignisse von ***TextBoxen***, die von Interesse sind. So wird auch das Ereignis *TextChanged* genutzt, um komfortabel auf eine Texteingabe zu reagieren. Doch dieses Ereignis ist mit Vorsicht zu genießen und kann unerwartetes Programmverhalten hervorrufen. Manchmal vergisst man, dass dieses Ereignis bei jeder Änderung des Textes in der ***Textbox*** eintritt. So auch beim Start oder Füllen von Tabellen, unabhängig, ob überhaupt die Voraussetzung des beabsichtigten Vorgehen bei *TextChanged* gegeben sind. Das bedeutet unkontrollierte und auch unerwünschte Aufrufe von Programmteilen. Wenn dann plötzlich nicht vorhandene Einträge aus Comboboxen oder ***Listboxen*** aufgerufen werden, führt das zu einem Laufzeitfehler. Das ist nicht immer lustig.

Man muss schon überlegen, welche Ereignisse Sinn machen und welche Auswirkung sie haben. Sollen die Ereignisse bei bestimmten Programmbearbeitungen ausgeschlossen werden, zum Beispiel bei einem Neustart, sucht man sich entsprechende Bedingungen.

So wird ein Fortschrittsanzeiger beim Laden von Daten angezeigt. Er ist nur während eines Ladezyklus sichtbar. Diese Sichtbarkeit genügt für eine Bedingung. Ist die ***ProgressBar*** sichtbar, wird das Ereignis nicht bearbeitet.

Die folgenden Seiten beschreiben schrittweise den Aufbau und die verschiedenen Stufen der Entwicklung. Das Endergebnis weicht von diesen Vorlagen ab, da erst mit fortschreitendem Stand viele notwendigen Änderungen und Erweiterungen erkennbar werden. Auch wird offenbar, dass die Bedienung plötzlich eine ganz andere Voraussetzung erfüllen muss. Natürlich kann ich diese fehlerhaften Ansätze verschweigen und nur das *perfekte* Ergebnis vorlegen. Doch genau dies wird ein Anfänger immer wieder erleben. Selbst ausgefuchste Programmierer finden immer wieder ein Manko im Design. Darum gehört diese Erfahrung auch in diese Anleitung. Es hilft später bei eigenen Projekten, einige der Fehler zu vermeiden, obwohl die Ansätze schlüssig scheinen. Einige Änderungen sind ohne Aufwand durchführbar, andere können ein ganzes Programmkonzept kippen. Daher ist eine gute Vorarbeit und Planung unerlässlich.

Die Design und Denkfehler zeigen aber auch, wie wichtig eine gut überlegte Programmstruktur ist und warum es Sinn macht, viele Unterprogramme zu schreiben und nicht alles in einen Programmblock zu

quetschen. So sind Objekte schnell ausgetauscht oder mit einem einfachen Funktionsaufruf den Bedingungen angepasst. Bin ich mir nicht sicher, reicht es, ein Steuerelement einfach unsichtbar zu machen, neue Steuerelemente hinzuzufügen und anschließend die neu erstellte Funktion zu prüfen.

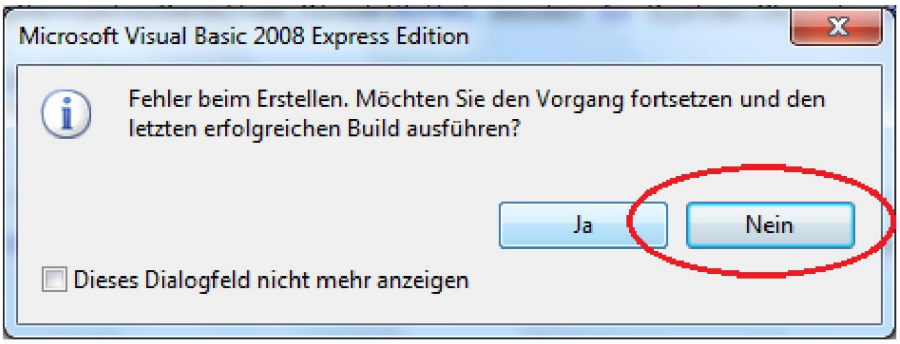
Bereinigt wird dann das Programm, wenn das Ergebnis zufriedenstellend ist.

Vorerst sollten diese Informationen für einen Grundstock in der Programmierung ausreichen. Es ist bei Weitem noch nicht mal ein Bruchteil angerissen, welche Möglichkeit eine Programmiersprache bietet, dennoch sollten wir mit den bisher vermittelten Anweisungen mit unserem Projekt starten.

## 1.2 Das Visual Basic Programm

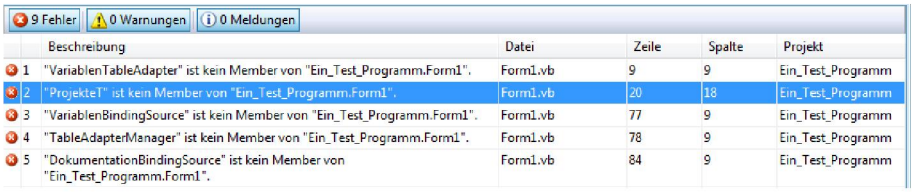
### *1.2.1 Behandeln von Fehlermeldungen des Compilers*

Im Laufe dieses Projektes wird es öfter vorkommen, das der Compiler eine Fehlermeldung liefert. Ignoriert den Fehler nicht, sondern seht euch die Zeile an, die den Fehler verursacht hat.




*Compilerfehler*

Nach dem Schließen wird die Fehlerliste gezeigt.



9 Fehler | 0 Warnungen | 0 Meldungen

| | Beschreibung | Datei | Zeile | Spalte | Projekt |
|---|---|---|---|---|---|
| 1 | "VariablenTableAdapter" ist kein Member von "Ein_Test_Programm.Form1". | Form1.vb | 9 | 9 | Ein_Test_Programm |
| 2 | "ProjekteT" ist kein Member von "Ein_Test_Programm.Form1". | Form1.vb | 20 | 18 | Ein_Test_Programm |
| 3 | "VariablenBindingSource" ist kein Member von "Ein_Test_Programm.Form1". | Form1.vb | 77 | 9 | Ein_Test_Programm |
| 4 | "TableAdapterManager" ist kein Member von "Ein_Test_Programm.Form1". | Form1.vb | 78 | 9 | Ein_Test_Programm |
| 5 | "DokumentationBindingSource" ist kein Member von "Ein_Test_Programm.Form1". | Form1.vb | 84 | 9 | Ein_Test_Programm |

*Fehlerliste*

Ein Doppelklick in eine Zeile führt an die fehlerhafte Programmstelle.

```
Public Sub Datenladen()
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Projekte As Integer
    Anzahl = Me.ProjekteT
    Anzahl = TeilelisteTableAdapter.Get_Teil.Rows.Count
    TextBox1.Text = Val(Anzahl)
```

*Programfehlerstelle*

Man kann Fehler auch daran erkennen, dass der Text blau unterstrichen ist.

```
Private Sub DokumentationBindingNavigatorSaveItem_Click(ByVal sender As Syst
    Me.Validate()
    Me.DokumentationBindingSource.EndEdit()
    Me.TableAdapterManager.UpdateAll(Me.EntwicklungDataSet)

End Sub
```

*Markierungen fehlerhafter Programmzeilen.*

Im ersten Fall ist eine Zeile nicht vollständig. Entweder man ergänzt hier den Text oder aber man entfernt ihn, um solche Fragmente gar nicht erst in den Code zu bekommen.

Im zweiten Fall ist es schon schwieriger, den Fehler zu erkennen. Nun, wir sind noch nicht soweit, diesen Befehl zu verstehen. So etwas kann passieren, wenn munter drauflos kopiert oder etwas vom Programm gelöscht wird. Hier ist die Komponente DokumentationBindingSource nicht vorhanden. Wenn man nicht sicher ist, ob diese Zeile nicht doch noch benötigt wird, kann man durch **Auskommentieren** diesen Fehler unterdrücken.

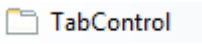
```
Private Sub DokumentationBindingNavigatorSaveItem_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs)
    Me.Validate()
    ' Me.DokumentationBindingSource.EndEdit()
    ' Me.TableAdapterManager.UpdateAll(Me.EntwicklungDataSet)
End Sub
```

### 1.2.2 *Der Entwurf*

Zurück zum Projekt. Die Anwendung bekommt erst einmal vier verschiedene Programmteile.

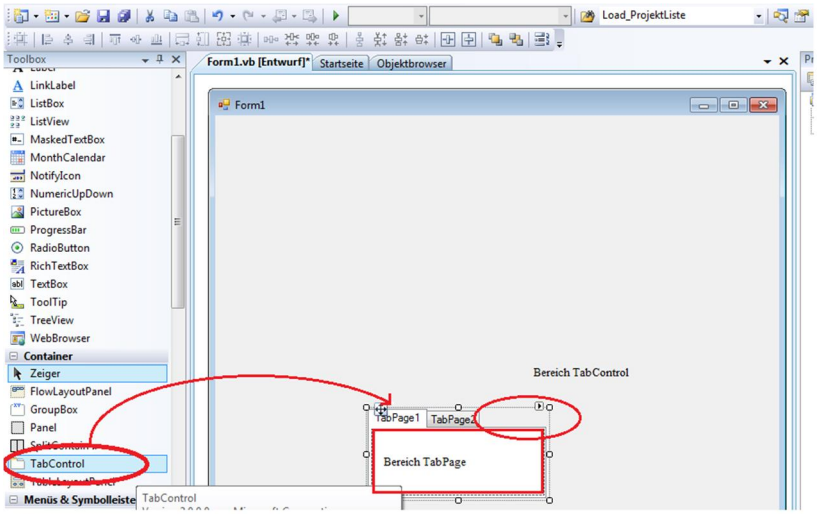
Der erste Teil beinhaltet einen Filter, um aus einem kopierten Codeabschnitt die Variablen heraus zu fummeln. Klar, man könnte da selbst Hand anlegen, doch dafür ist ja ein Computerprogramm gedacht, das Leben zu erleichtern.

Hier eignet sich ein ***TabControl**-Objekt*.




*ICON TabControl*

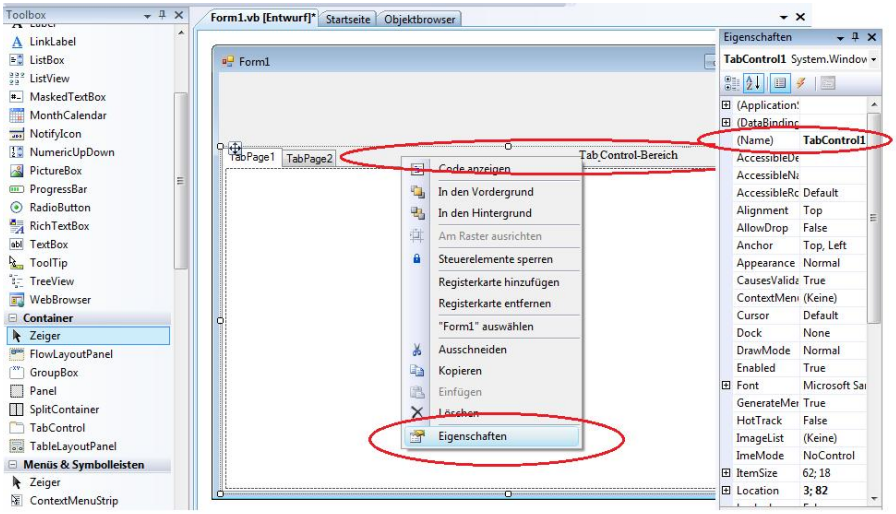
Es besitzt Tabulatoren und funktioniert wie eine Registerablage. Das Objekt beinhaltet schon alle zum Bedienen der Tabulatoren notwendigen Funktionen. Es befindet sich in der Toolbox und kann dort angeklickt und auf der Arbeitsfläche mit einem weiteren Klick installiert werden.




*Erstes Objekt Tab-Control*

Da dieses ***TabControl*** alle Programmteile abbilden wird, kann es der Arbeitsfläche voll angepasst werden. Lediglich im oberen Bereich werden später noch Objekte installiert und darum wird dort ein wenig Platz gelassen.

Ein ***TabControl***-Objekt hat von Hause aus zwei verschiedene Beerührungspunkte. Einmal das TabControl-Objekt selbst im oberen Bereich und einmal das eingebettete TabPage_Objekt in der Mitte. Um an die Eigenschaften des TabControl Objektes zu gelangen, klicken wir mit der rechten Maustaste in den oberen Bereich neben den Tabulatoren. Nun haben wir die Möglichkeit im aufgeblendeten Eigenschafts-Editor Änderungen vorzunehmen. Beginnen wir zuerst mit der Namensgebung.

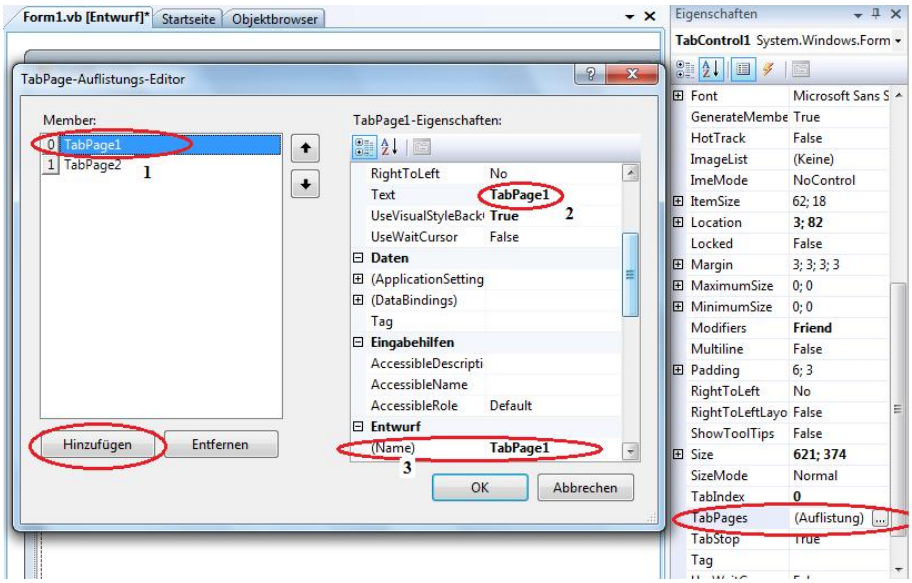



*Eigenschaft Name Tab-Control*

Ich hab mir angewöhnt, grundsätzlich Namen nach Verwendung zu vergeben. Nichts ist schlimmer, als ein Objekt im Programm ansprechen zu wollen und den Namen erst mal lange suchen zu müssen. Die Defaultwerte sind da nicht sehr hilfreich und deshalb opfere ich die Zeit. Ein ***TabControl*** bekommt einen Präfix **TC_** für *TabControl*, einem Trennzeichen und den Namen der Funktion. In diesem Fall **TC_Auswahl**, denn es soll mir die verschiedenen Programmebenen anbieten.

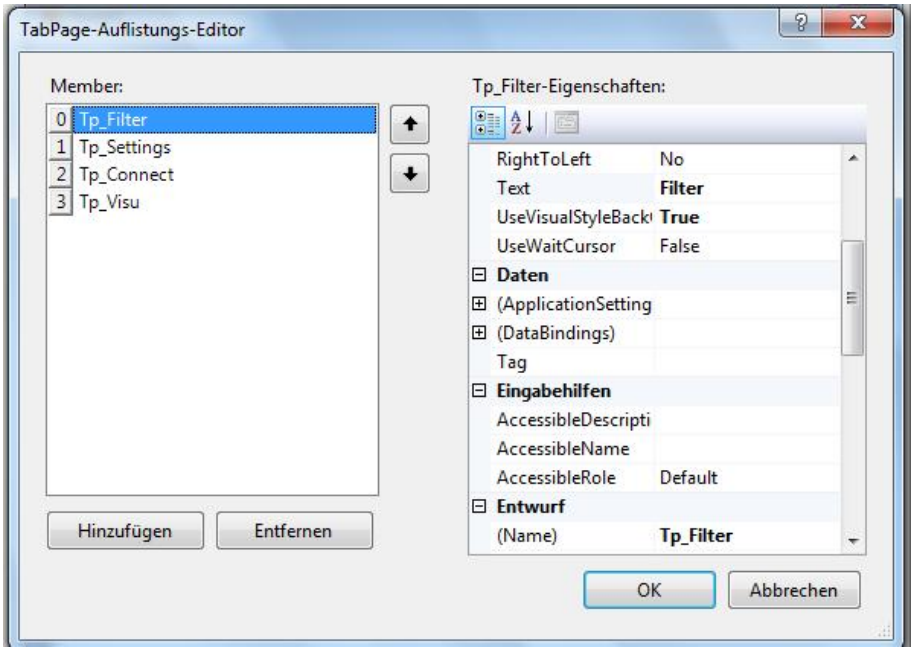
Lediglich bei ***Labels***, also Überschriften, lasse ich die Defaultwerte stehen, wenn es im Programm nicht erforderlich ist, den Text zu ändern.

Einem solchen ***TabControl*** können nun weitere Seiten, die TabPages, zugefügt werden. Es entsteht ein Tabulatorobjekt mit mehreren Registerkarten. Ab sofort muss beachtet werden, das ***TabControl*** nur im oberen Bereich bei den Tabulatoren erreicht wird. Darunter ist die zur Registerkarte gültige ***TabPage***. Eine TabPage hinzufügen geht nur im TabControl, also oben. Einer ***TabPage*** einen Namen und eine Überschrift geben, wird im Eigenschaftseditor der ***TabPage*** durchgeführt. Um diesen zu öffnen, in die Mitte der ***TabPage*** mit rechter Maustaste klicken. Es ist aber auch direkt im Eigenschaftsfenster von ***TabControl*** die Möglichkeit, die Seiten einzurichten. Dazu wird nach der Eigenschaft *TabPages* Auflistung gesucht und die Auflistung angeklickt.



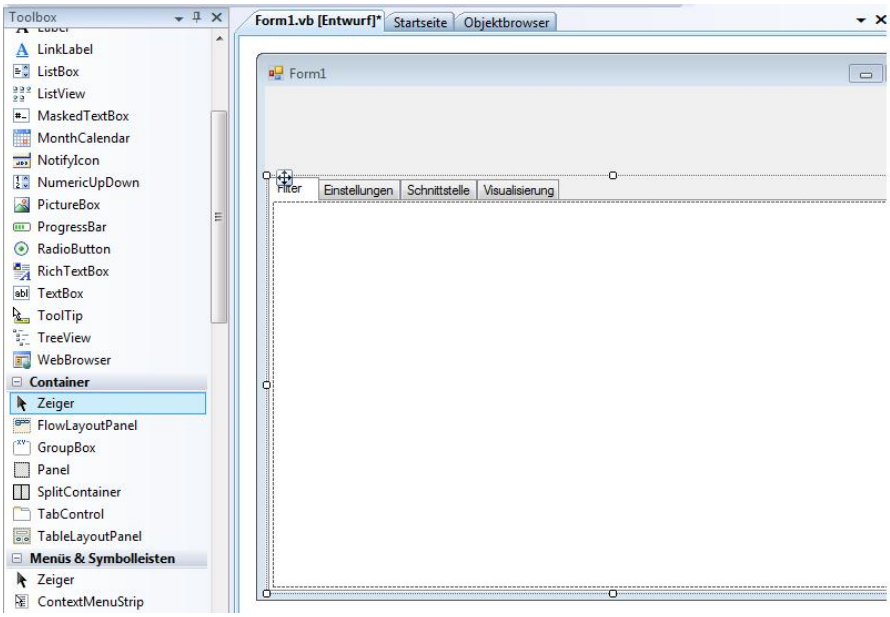

*TabPage einrichten*

Es öffnet sich ein Fenster, in dem die beiden vorhandenen Seiten bereits eingetragen sind. Hier ist wichtig zu beachten, dass der unter Member gelistete Name (1) der Name der ***TabPage*** ist. Die Bezeichnung neben Text auf der rechten Seite (2) ist die Beschriftung des Tabulators. Um den Namen der Seite zu ändern, muss dies im Feld (Name) **TabPage1** (3) erfolgen. Richten wir nun zuerst unsere 4 Seiten ein




*Seiten einrichten*

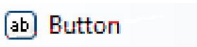
Mit dem Button Hinzufügen werden die erforderlichen Seiten generiert und parametriert Schließlich erhalten wir die gewünschte Ansicht.

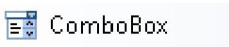



*Ansicht TabControl Objekt*

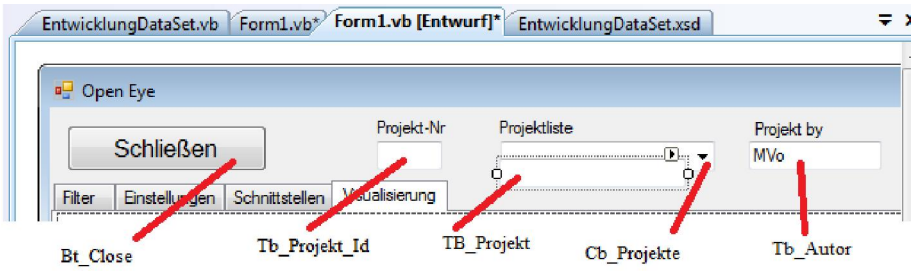
Es ist aber immer nur die ausgewählte Seite zu sehen. Die Umschaltung auf eine andere Seite erfolgt durch Anklicken des entsprechenden Tabulators. Noch ist nichts auf den Seiten eingebracht, und deshalb wird sich die Umschaltung nicht zeigen.

Setzen wir noch einen ***Button*** Schließen und eine ***Combobox*** sowie 3 ***TextBoxen*** wie im folgenden Screenshot gezeigt, in unsere Form

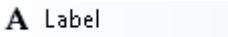



*ICON Button* *ICON Combobox*



Die Beschriftungen sind ***Label***, die keinen gesonderten Namen bekommen.

*ICON Label* 


Die ***Textbox*** *Tb_Projekt* hat eine besondere Aufgabe. Sie verhindert, dass direkt in die ***Combobox*** Eingaben erfolgen können. Im Moment ist sie noch versetzt, in der Anwendung aber liegt die ***Textbox*** direkt über der ***Combobox***. Der Grund dafür wird im weiteren Verlauf ersichtlich. Zu diesem Zeitpunkt genügt es, wenn die Objekte mit der entsprechenden Eigenschaft benannt werden. Diese Objekte sind, egal welche Seite aufgeschlagen ist, immer sichtbar und auch erreichbar. Es sind Informationen und Aktionen, die jederzeit zur Verfügung stehen müssen. Daher werden diese Bausteine unseres Programms auch nicht in den ***TabControl*** eingebracht, sondern außerhalb gesetzt.

Es ist auch gleich eine Gelegenheit, einen kleinen Test durchzuführen.

Mit einem Doppelklick auf den ***Button*** gelangen wir in den Code-Bereich und dort gleich in das Ereignis *Click* des ***Buttons***. Die Ereignisroutine wird vom Rahmen her aufgebaut. Alles, was nun noch getan werden muss, ist die Aktion, die mit dem *ButtonClick* durchgeführt werden soll, zu programmieren. Dieser ***Button*** soll die Applikation beenden, also die Form schließen. In Neudeutsch *Close*.

Schreiben wir diesen Befehl einfach mal in die Subroutine

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button1.Click
    Close()
End Sub
```

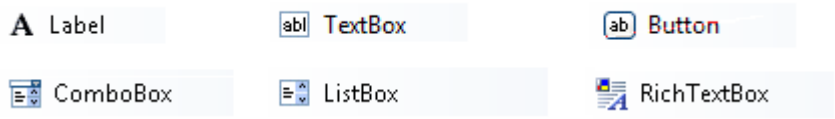
Nun starten wir das Programm. Es sollte ohne Fehler übersetzt werden und starten. Nun betätigen wir den ***Button***. Das Programm wird beendet. Nun wird man fragen, wozu ein Schließen-Button? Die Form selbst hat doch eine Schließfunktion. Sicherlich könnte das tatsächlich ausreichen, aber es ist einfach eleganter.

Was uns dieser Test zeigt ist, dass nun alles, was wir programmieren, seinen Auslöser in einem Ereignis findet. Es sind gar nicht ellenlange

Listings zu erstellen, um eine Aktion durchzuführen. Gut, ein paar umfangreiche Bearbeitungsschritte werden schon erforderlich sein, aber der Großteil der Programmierung in unserer Anwendung sind kleine Ereignisbehandlungen.

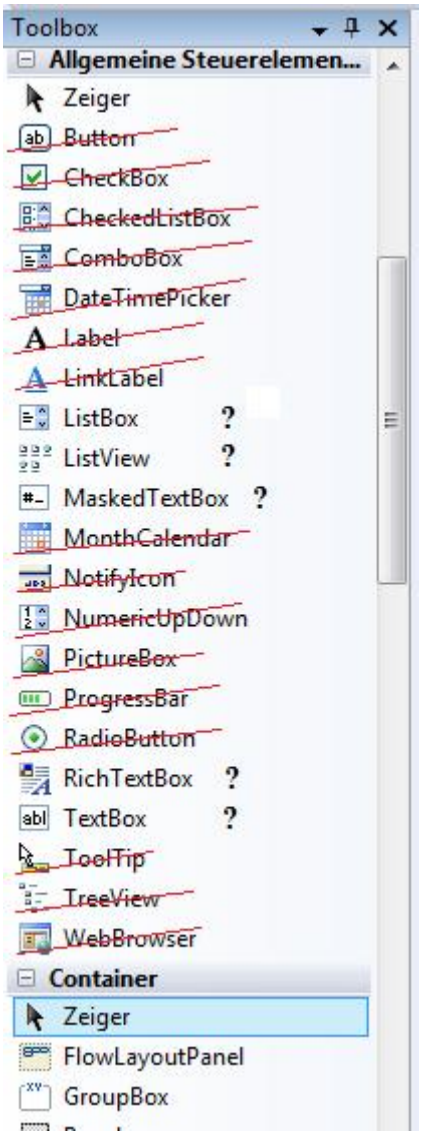
### *1.2.3 Einrichten Seite Filter*

Gehen wir nun die erste richtige Herausforderung an. In diesem Abschnitt werden die Objekte der folgenden Liste abgehandelt:



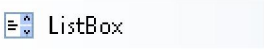

*Objekte Seite Filter*

Die erste Aufgabe ist es, einen Textfilter zu bauen, der aus einem Assemblerlisting die Variablen und zugehörigen Formate herauslöst. Der Weg ist auch klar, der Deklarationsbereich der Variablen wird kopiert und muss nun in eine Ablage in unserem Programm eingefügt werden. Wer noch nie mit Visual Basic gearbeitet hat, wird sich nun die Frage stellen, welches Tool dafür geeignet sein könnte, das in der **Toolbox** verfügbar ist. Nun, werfen wir einen Blick darauf und streichen erst mal alles, was von vornherein allein schon durch seinen Namen ungeeignet erscheint. Wo wir nicht sicher sind, setzen wir gedanklich ein Fragezeichen. Der Screenshot verdeutlicht diese Vorgehensweise.

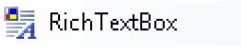



*Toolbox Auswahl Objekt*

Versuchen wir nun unser Glück, indem wir das erste denkbar mögliche Objekt, in diesem Fall die ***Listbox***, auf die erste Seite des Tabulatorobjektes einfügen und das Programm starten.

*ICON ListBox* 

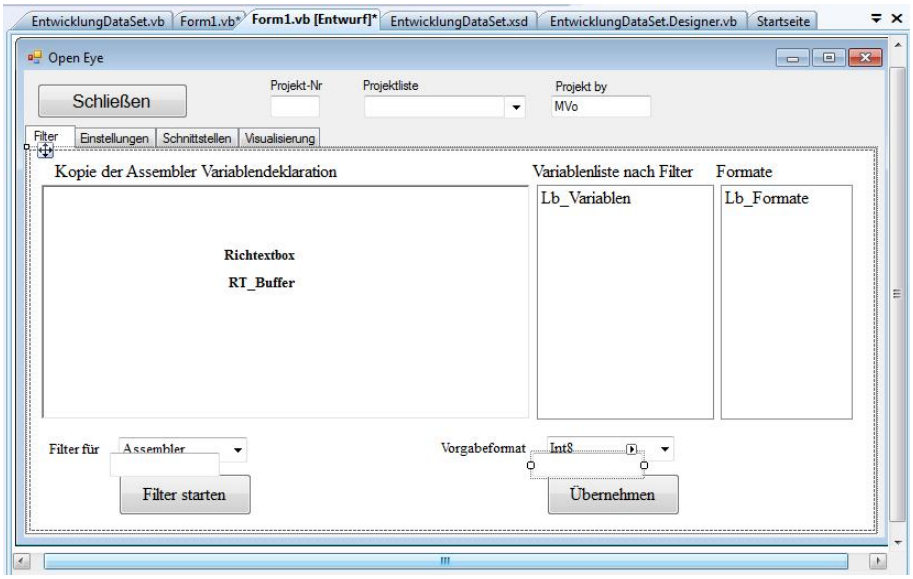

Nun versuchen wir einen beliebigen, Text, etwa eine Seite, aus einer anderen Datei in dieses Objekt zu kopieren. Geht nicht! Also ungeeignet. Ein gleiches Ergebnis bringt das Objekt ***ListView***. Bei dem Versuch mit dem Objekt ***MaskedTextBox*** sieht es schon besser aus. Aber das Feld bleibt so klein, das nicht der gesamte Text zu sehen ist. Das wär uns ja egal, wenn trotzdem der gesamte Text enthalten wäre, doch bei der Prüfung mit dem Cursor stellen wir fest, dass der Text nicht wirklich komplett ist. Enthält der Text ein Zeilenende Zeichen, wird kein weiterer Text übernommen. Unser Listing hat aber nach jeder Programmzeile ein Zeilenende Zeichen und deshalb ist auch dieses Objekt genau wie die ***Textbox*** ungeeignet. Bleibt das Objekt ***Richtextbox***.

*ICON RichTextBox* 


Der Test mit diesem Objekt sieht vielversprechend aus. Auch wenn nicht der gesamte Text sichtbar ist, kann er aber unter Nutzung der eingefügten **Scrollbalken** sichtbar gemacht werden. Dieses Objekt ist also geeignet, kopierten Text zu übernehmen. Nun sollen die herausgefilterten Worte, nichts anderes sind die Variablennamen im Assemblerlisting zur weiteren Verwendung auch gespeichert werden. Da erscheint die ***Listbox*** geeignet, denn der Name sagt es bereits, es ist eine Box für eine Liste und die Variablen sollen ja gelistet werden. Hier sparen wir uns den Test und vertrauen auf den Objektnamen. Eine weitere Liste ist für die Variablenformate gedacht. Wie diese Information gewonnen werden kann beschreibe ich bei der Filterprogrammierung. Gestalten wir erst einmal die Seite Filter und fügen die benötigten Objekte ein.

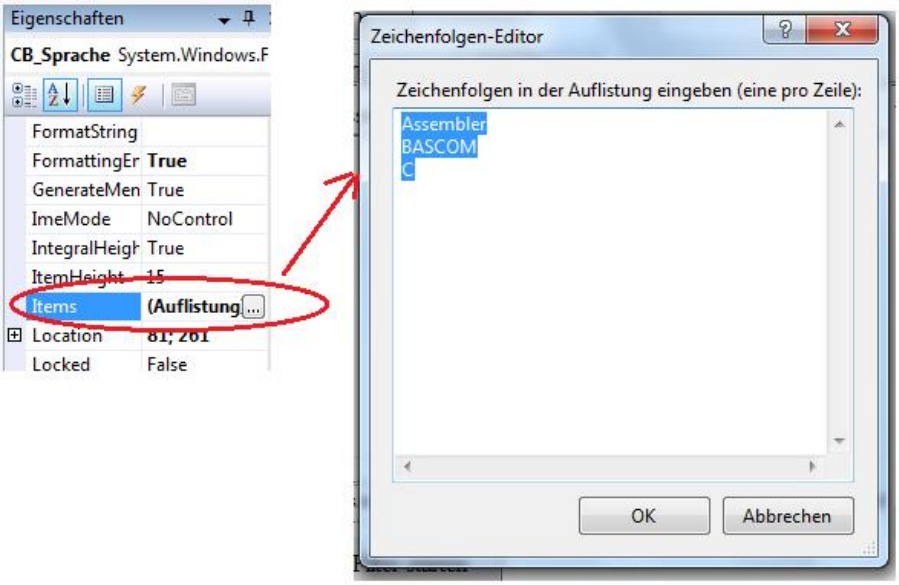
Eine Richtextbox, sie bekommt den Namen Rt_Buffer
Eine Listbox mit dem Namen LB_Variablen
Eine Listbox für die Formate, demnach auch der Name LB_Formate
Eine Combobox für die Sprachauswahl mit Namen Cb_Sprache
Eine darüber liegende Textbox mit Namen TB_Sprache
Für die Formatvoreinstellung eine Combobox CB_DefaultFormat
Ebenfalls eine darüber gelegte Textbox TB_DefaultFormat
Ein Button, um den Filter zu starten BT_Filter
Ein Button für die abschließende Übernahme BT_Uebernahme

Zusätzlich ***Labels***, um die nicht ersichtliche Funktion der Objekte zu beschreiben. Schließlich sollte sich die folgende Ansicht ergeben




*Aufbau Seite Filter*

Die ***TextBoxen*** **TB_Sprache** und **TB_DefaultFormat** sollen direkt über der zugehörigen ***Combobox*** liegen und für den Zugriff mit der Eigenschaft *Enabled* = **False** für den Zugriff gesperrt sein. Das verhindert unerwünschte Eingaben in das Textfeld der ***Combobox***. Für einen Anfänger ist es nicht so einfach, nur das Textfeld einer ***Combobox*** zu sperren, die Bedienung der Liste aber zu erhalten. Die darüber liegende ***Textbox*** erledigt das mit einem einfachen Trick. Die Combobox bleibt aktiv, aber das Textfeld ist durch die ***Textbox*** verdeckt. Rufen wir nun die Eigenschaft der ***Combobox*** **Cb_Sprache** auf, indem wir mit der rechten Maustaste die ***Combobox*** anklicken. Unter der Eigenschaft Name wird **Cb_Sprache** eingetragen. Über die Eigenschaft *Items* werden entsprechende Vorgaben eingetragen, die in der ***Combobox*** zur Auswahl stehen sollen.




*Itemliste Combobox Sprachauswahl*

Obwohl wir nur einen Filter für Assembler aufbauen, fügen wir diese Option für C und BASCOM ein. Der Screenshot zeigt, wie eine *Itemliste* einer ***Combobox*** gefüllt werden kann.

Die ***Combobox*** CB_DefaultFormat bekommt die Items

```
Int8
Byte
ASCII
Hex8
```

Dieses Format wird vergeben, wenn kein Format aus dem Assemblerlisting abgeleitet werden kann.

Mit diesen Objekten ist die Seite Filter fertig aufgebaut. Starten wir einmal das Programm und testen verschiedene Bedienungen. Wir können nun

die verschiedenen Seiten aufrufen. Lediglich die „Filter“-Seite zeigt einen Inhalt, die beiden anderen sind ja noch leer.

Wir können die ***Comboboxen*** Cb_Sprache und CB_DefaultFormat öffnen, allerdings noch keinen ausgewählten Eintrag sichtbar machen.

Wie aber schön zu erkennen ist, funktionieren die Komponenten schon und das ohne auch nur eine einzige Anweisung zu schreiben.

Um die Funktion der einzelnen Komponenten noch etwas besser kennen zu lernen, folgen nun ein paar einfache Experimente, die zuerst einmal die ***Button*** „BT_Filter“ sowie die ***Listbox*** „LB_Variablen“ nutzen.

### *1.2.3.1 Experimente mit dem Button Filter und einer Listbox*

In der Entwurfsansicht wird mit einem Doppelklick auf das ***Button*** Filter der Basic-Editor aufgerufen und ein Gerüst der Ereignisroutine *BT_Filter_click* erzeugt.

```
Private Sub BT_Filter_click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles BT_Filter. Click

End Sub
```

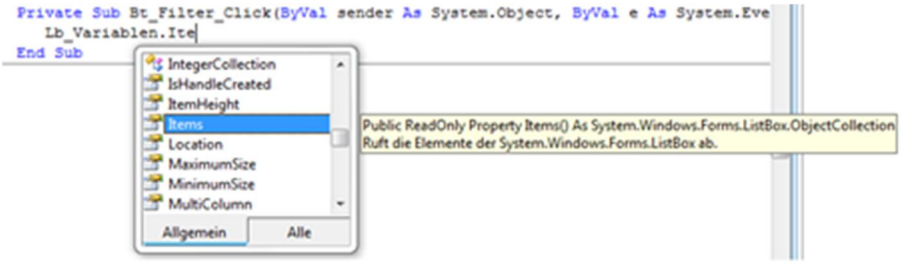
Die Parameterliste im Prozedurkopf ist in diesem Fall für den Programmierer nicht relevant. Uns interessiert aber, wie nun diese **Sub** genutzt werden kann.

Schreibt in dieses Gerüst einmal den Befehl

```
LB_Variablen.Items.Add(„Mein Eintrag“)
```

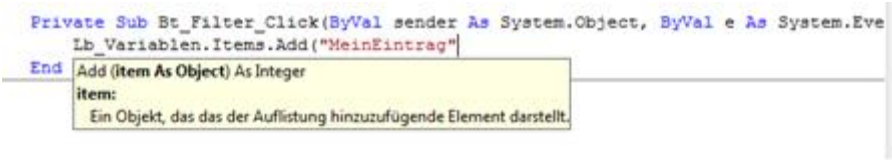
Wenn nun das Programm gestartet und der ***Button*** betätigt wird, erscheint ein neuer Eintrag in der ***Listbox***. Eine einfache Zuweisung wie bei einer Textbox mit einem **=** ist nicht möglich, weil ja eine Liste mit Einträgen verwaltet wird. Die Einträge nennt man *Items* und **Add** erinnert an hinzufügen.

Die möglichen Eigenschaften eines Objektes werden bei der Eingabe schon vom Editor vorgeschlagen. Gebt einmal nur den Variablennamen und einen Punkt ein. Es sollte sich nun ein Fenster öffnen mit einer Liste der möglichen eingebundener Funktionen und Parameter öffnen. Sucht nun den Eintrag *Items* oder gebt *Items* langsam Zeichen für Zeichen ein. Die Liste wird automatisch zu diesem Begriff navigiert.




*Onlinehilfe bei Programmerstellung*

Mit einem weiteren Punkt werden nun die Eigenschaften der Item-Objekte in der Liste dargestellt. *Add* dürfte durch die alphabetische Sortierung ziemlich am Anfang stehen. Mit *Add* wird nun auch angezeigt, dass ein Objekt als *Item* erwartet wird. Ein String ist hier ein solches Objekt.




*Onlinehilfe Parametereintrag*

So unterstützt VB mit einem Hilfetext, wenn mal die gewünschte Funktion oder der Syntax nicht bekannt ist. Oft gibt die Auflistung hier die entsprechende Auskunft. Starten wir nun das Programm und testen das Ergebnis durch anklicken des Filter-Button. Bei jedem Klick wird ein Eintrag in die Variablenliste hinzugefügt.

Aus diesem Abschnitt sollte das Prinzip der Ereignissteuerung deutlich werden. Das Ereignis Click eines Objektes ruft eine Methode auf, die nun in sich gekapselt einen kleinen Job erledigt. Danach ist wieder Ruhe.

Für weitere Experimente werden Variablen erforderlich. Diese müssen dem Programm bekannt gemacht werden. Der Programmierer unterscheidet in der Regel zwischen **globalen** und **lokalen** Variablen. Dazu sollte es eine kleine Erklärung geben, denn es ist wichtig, den Unterschied zu kennen.

Kurz:

Global ist eine Variable, wenn sie am Programmanfang steht und in keiner Subroutine vereinbart wird. Sie ist überall im Programm lesbar, aber auch veränderbar.

Eine lokale Variable wird in einer Sub deklariert und ist auch nur in diesem Unterprogramm gültig. Für außen ablaufende Programmteile existiert diese Variable nicht

### *1.2.3.2 Experiment Lokale Variable*

Noch ist unser Beispiel aktuell und ich möchte nun die Information mitgeben, der wievielte Eintrag gegeben ist. Dazu muss ich eine Variable dimensionieren. Der Befehl dazu lautet DIM gefolgt von einem Variablennamen (z.B.) **X** und dem Format **Integer**

In der deutschen Sprache würde man den Befehl *dimensioniere X als Integer* formulieren. In Basic ist dies ähnlich, allerdings in der englischen Sprache: **Dim X as Integer**

Diese Zeile setzen wir direkt unter den Prozedurkopf vor unsere Zeile mit dem Befehl für den Eintrag in die Listbox. Damit die Variable nicht einfach irgendeinen zufälligen Wert hat, wird sie noch vorbesetzt, z.B. mit **X=0**

Bei jedem Tastendruck soll nun um 1 erhöht werden. Damit diese Zahl zu einem String, also Text angehängt werden kann, ist eine Wandlung von Zahl zu Text erforderlich. Dies erledigt die Funktion **Str(Zahl)**. Sie liefert den String der Zahl zurück. Damit hätten wir nun grob alles, um dem Text **Mein Eintrag** noch eine Nummer anzuhängen.

Nun die geänderte Befehlsfolge im Click-Ereignis des Button:

```
Dim X as Integer
 X=0
 X=X+1
 LB_Variablen.Items.Add(„Mein Eintrag “+Str(X))
```

Testet nun das Ergebnis. Es wird immer nur **Mein Eintrag 1** in der ***Listbox*** ausgegeben. Nun, die Variable X wird ja auch bei jedem Aufruf der Routine erneut auf 0 gesetzt. Ah ja, ist klar, da steht ja **X=0**. Das muss da weg, dann geht’s.

Auch diese Ansatz ist falsch, denn X kann nun alles Mögliche sein, ist ja nicht definiert. So ist jedenfalls keine Lösung in Sicht. Der Grund: die Variable ist lokal definiert. Sie ist außerhalb dieses

Ereignisses nicht veränderbar, ja, sie existiert nicht einmal. Bei jedem Aufruf des Ereignisses wird sie temporär neu erzeugt und ist danach auch nicht mehr gültig. Was hier benötigt wird, ist eine Variable, die über die Ereignisroutine hinweg Gültigkeit behält. Sie muss **global** sein. Das heißt, sie muss außerhalb der Ereignisroutine am Anfang vom Programm deklariert.

```
Public Class Form1
'*********************************************
'*Globalvariablen*
'*********************************************
Public X as Integer
....
```

und auch außerhalb 0 gesetzt werden. Zum Beispiel in der Startroutine

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
  X=0
End Sub
```

Diese findet sich in der Ereignisliste von ***Form1*** wieder oder wird mit einem Doppelklick in die Form erzeugt. Erst dann funktioniert auch der Zähler.

Verändert einmal das Programm entsprechend und löscht die Deklaration **Dim X as Integer** innerhalb der Routine **BT_Filter_Click** sowie die Zuweisung **X=0**

Ein neuer Test bringt nun den gewünschten Erfolg. Aber, die Verwendung globaler Variablen sollte sparsam behandelt werden. Schnell hat man vergessen, wo sie zuletzt ihren Wert bekommen haben und ob dieser überhaupt verändert werden darf. Im weiteren Verlauf wird ausführlich erklärt, wie mit Hilfe von Schnittstellen auf globale Variablen fast ganz verzichtet werden kann.

### *1.2.3.3 Objekteigenschaften nutzen*

An dieser Stelle kommen die Eigenschaften von den Objekten zum Tragen. Eine ***Listbox*** sollte schon wissen, wie viele Einträge bereits eingetragen sind. Dies kann man abfragen. Die Anzahl der Einträge in der ***Listbox*** wird in *Listbox.Items.Count* mitgeführt. Damit ist die Nummer des neuen Eintrags zu berechnen. Die Änderungen in globaler Deklaration von X und die Vorbesetzung in der Startroutine nehmen wir wieder zurück und bewegen uns nun nur noch innerhalb der Ereignisroutine *BT_Filter_Click*. Die Folgenden Zeilen werden dort eingefügt.

```
Dim X as Integer
  X = LB_Variablen.Items.Count
  X = X+1
  LB_Variablen.Items.Add(„Mein Eintrag “+Str(X))
```

Nun ist die Variable **X** als lokale Variable sinnvoll eingesetzt, denn nur an dieser Stelle wird sie benötigt. Mein neuer Eintrag soll ja auch um 1 höher sein, wie die aktuelle Anzahl der Einträge. Dadurch, dass ein Objekt selbst über seine Eigenschaften Auskunft geben kann, werden globale Variablen fast überflüssig. Allerdings gibt es einen Pferdefuß:

Sobald Einträge in der Liste gelöscht werden, könnten doppelte Nummern vergeben werden. Daher taugt dieses Beispiel nur zur Demonstration. In der Praxis gibt es andere Möglichkeiten. Startet mal das Programm und testet die Funktion.

Nun bereitet es nicht gerade ein Vergnügen, dauernd auf das ***Button*** zu klicken. Kann ein einzelner Click nicht eine ganze Reihe von Ergebnissen bringen? Natürlich, aber erst einmal etwas Neues.

Vielleicht müssen mehrere Ergebnisse irgendeiner Bearbeitung in einer Feldvariablen abgelegt werden. Zum Beispiel eine Zahlenreihe. Eine solche Feldvariable ist ein *Array*. Die Deklaration dazu lautet:

*Dim Werte_Array(7) as Integer*

Damit besitzt dieser Programmbereich, auch diese Variable ist lokal und nur innerhalb der aufgerufenen Sub-Routine gültig, 8 einzelne Variablen:

```
Werte_Array(0)
Werte_Array(1)
Werte_Array(2)
Werte_Array(3)
Werte_Array(4)
Werte_Array(5)
Werte_Array(6)
Werte_Array(7)
```

Der Zugriff kann nun auch über einen variablen Wert zwischen 0 und 7 erfolgen. Da eine Variable vor Verwendung einen definierten Wert benötigt, wird auch ein Array initialisiert und mit Werten beschrieben. Bei einem derart kleinen Array wäre es fast noch zu vertreten, jeden Wert direkt zuzuweisen, eleganter ist dies mit einer Schleife. Da ein Endwert bekannt ist, kann hier eine feste Schleife eingesetzt werden. Dazu ist ein Schleifenzähler erforderlich, eine Variable, die Werte vom Start bis Endwert annehmen kann. Hier reicht ein einzelner Buchstabe und es ist durchaus geläufig **i** zu nehmen.

*Dim i as Integer.*

Die folgenden drei Zeilen füllen das Array mit 0

```
For i = 0 to 7
Werte_Array(i)=0
Next i
```

Wenn statt 0 der Schleifenzähler zugewiesen wird, enthalten die Speicherzellen Werte von 0 bis 7

```
For i = 0 to 7
Werte_Array(i)=i
Next i
```

### *1.2.3.4 Die Ausführung sichtbar machen*

Damit die Auswirkung sichtbar wird, soll nun die ***Listbox*** „LB_Variablen“ die Inhalte vom Array anzeigen. Dazu werden die Zahlenwerte in Text gewandelt. Der Befehl zur Wandlung wurde bereits beim Zähler der einzelnen Klicks auf den Button benutzt. Der gesamte Vorgang soll nun wie folgt ausgeführt werden:

Zuerst wird das Array gefüllt und anschließend der Inhalt in die ***Listbox*** eingetragen. Damit wir uns wieder zurechtfinden, zeige ich die ganze Sub-Routine.

```
Private Sub BT_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles BT_Filter_Click
Dim Werte_Array(7) as Integer
Dim I as Integer
  LB_Variablen.Items.Clear 'Listbox Einträge löschen
  For I = 0 to 7
     Werte_Array(i)=i 'Versuch mit anderen Einträgen
  Next I
  For I = 0 to 7
    LB_Variablen.Items.Add(Str(Werte_Array(i)))
  Next I
End Sub
```

Erklärung:

Der Integerwert aus dem Array wird in einen Zahlenstring gewandelt, da die ***Listbox*** Strings aufnehmen soll. Dies erledigt die Function *Str(<Zahl>).*

Die umgekehrte Funktion, aus einem String mit Ziffern eine Zahl zu konvertieren, ist die Function *Val(<String>)*

Die Ereignisbehandlung von Button-Objekten sollte nun klar sein.

Bisher waren die Aufgaben anspruchslos. Wie sieht es aber aus, wenn in der ***RichTextBox*** ein paar Zeilen eingegeben werden. Nun, starten wir ein weiteres Experiment.

### *1.2.3.5 Zerlegen von Strings*

In diesem Abschnitt wird erklärt, wie ein Text aus dieser Beschreibung in die ***RichTextBox*** des Visual Basic Programms kopiert und nach Worten zerlegt wird. Hier ist es nicht möglich, mit einer festen Schleife zu arbeiten. Der kopierte Text kann ja beliebig groß sein.

Die Lösung ist eine *While-Schleife*. Diese wird nur durchlaufen, wenn eine angegebene Bedingung Wahr ist. Das Werte_Array nützt auch nicht viel, aber zwei Stringvariablen können da nützlich sein. In eine werden wir den Inhalt der ***RichTextBox*** kopieren und in die anderen das ermittelte Wort. Die Zählvariable I wird mit der Variablen *Leer_Pos* ersetzt.

Die Routine beginnt mit der Vorbesetzung der Variablen und Löschung eventuell vorhandener Einträge in der ***Listbox*** *LB_Variablen*.

```
Private Sub BT_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles BT_Filter_Click
Dim Work_Buf As String
Dim Wort As String
Dim Leer_Pos As Integer
  Work_Buf = RT_Buffer.Text     'Text aus der RichTextBox in Stringvariable kopieren
  Wort = ""
  LB_Variablen.Items.Clear()    'Liste leeren
  While Len(Work_Buf) > 0       'Länge der Stringvariable abfragen und prüfen
    Leer_Pos = InStr(Work_Buf, " ") 'Position nächstes Leerzeichen erfassen
    If Leer_Pos > 0 Then          ' wenn Leerzeichen gefunden dann ->
      Wort = Mid(Work_Buf, 1, Leer_Pos – 1) 'Wort aus Stringvariable kopieren
      Work_Buf = Mid(Work_Buf, Leer_Pos + 1, Len(Work_Buf) – Leer_Pos)
      ' Wort aus Stringvariable löschen
    Else
      Wort = Work_Buf               ' sonst letztes Wort übernehmen
      Work_Buf = ""                 ' und Stringvariable leeren (Länge =0)
    End If
    If Wort <> " " Then             ' Wenn Wort kein Leerzeichen
      LB_Variablen.Items.Add(Wort)          ' dann Eintrag in Liste
    End If
  End While
End Sub
```

Zu diesem kleinen Listing eine Beschreibung:

Die benötigten Variablen werden lokal deklariert, da sie auch nur in dieser Routine zur Bearbeitung der Aufgabe benötigt werden. Bevor die Schleife

aufgerufen werden kann, muss der Arbeitsstring besetzt und die Variable *Wort* auf einen Defaultwert gesetzt werden. Schließlich ist es auch wichtig, eine gelöschte Liste vorzufinden. Beim ersten Aufruf ist sie zwar leer, aber es ist ja durchaus denkbar, dass der Button ein weiteres Mal betätigt wird.

In der Schleife wird die Position des ersten Leerzeichens bestimmt. Gibt es ein Leerzeichen, das heißt, die Variable *Leer_Pos* ist >0 , wird der Text davor aus dem Arbeitsstring in die Variable *Wort* kopiert und anschließend im Arbeitsstring gelöscht. Gibt es keines mehr, ist möglicherweise noch ein Wort vorhanden, daher kopieren wir den Rest vom Arbeitsstring in die Variable Wort.

Nach dieser Auswertung wird geprüft, ob die Variable *Wort* nicht leer ist und unter dieser Bedingung in die Liste eingetragen. Das Ergebnis sollte nun eine Liste mit allen Worten aus dem Textstück in der ***RichTextBox*** sein. Dies können wir prüfen, indem wir einen Textabschnitt in die ***Richtextbox*** des gestarteten Programms übertragen und den ***Button*** Filter betätigen.

### *1.2.3.6 Ein zweites Experiment*

In einem weiteren Experiment werden wir nun Sätze aus unserer ***RichTextBox*** herauslösen. Dazu werfen wir noch einmal einen Blick auf das voran gegangene Experiment. Dort haben wir die Position eines Leerzeichens erfasst und dadurch ein Wort aus dem Text gefischt. Wenn nun die Position eines Punktes ein Satzende signalisiert, genügt es doch, einfach nur statt einem Leerzeichen einen Punkt zu suchen. Der Rest dürfte doch genauso funktionieren. Auch unter diesen Bedingungen wird der Text im Arbeitsstring um den vorher kopierten Bereich gelöscht werden.

Ändern wir also nur die Zeile

`Leer_Pos = InStr(Work_Buf, " ")` in `Leer_Pos = InStr(Work_Buf, ".")`

Starten wir nun das Programm und sehen uns das Ergebnis an. Dazu kopieren wir diesmal einen großen Text aus irgendeiner Datei in die ***RichTextBox***.

Nicht immer ist es so einfach, langsam zur Lösung einer Aufgabe zu kommen, doch hier wird klar, welche Mechanismen für einen Filter erforderlich werden, der aus einem Assembler-Quellcode die Variablennamen liefert. Betrachten wir dazu einmal eine Variablendeklaration in Assembler.

*Variablenname: .Byte 1 ; dies ist eine Assemblervariable*

Am Zeilenanfang steht der Variablenname und ist mit einem Doppelpunkt abgeschlossen. Es folgt ein Punkt und das Schlüsselwort Byte mit einer Zahl. Schließlich ist hinter einem Semikolon ein Kommentar, der vom Compiler ignoriert wird, dem Listing aber aussagekräftige Informationen liefert. Bei den verwendeten Variablen ist eine Beschreibung des Formats durchaus sinnvoll und so können wir auch bei Einhaltung gewisser Strukturen in Open_Eye das Format aus dem kopierten Text herauslösen.

### *1.2.3.7 Codebeispiel Assembler*

```
.DSeg                                   ; Ablageort dynamischer Speicher
Trigger_In:         .Byte 1             ; Byte Triggerbits für Programmkontrolle
Trigger_Out:        .Byte 1             ; Byte Rückmeldung des bearbeiteten Triggers
Old_In:             .Byte 1             ; Byte Zuletzt eingelesene Signale
                                        ; Bit 0 = Taster 1
                                        ; Bit 1 = Taster 2
                                        ; Bit 2 = Taster 3
New_In:             .Byte 1             ; Byte neu eingelesene Signale
                                        ; Bit 0 = Taster 1
                                        ; Bit 1 = Taster 2
                                        ; Bit 2 = Taster 3
In_To_High:         .Byte 1             ; Byte Ereignis steigende Flanke
                                        ; Bit 0 = Taster 1
                                        ; Bit 1 = Taster 2
                                        ; Bit 2 = Taster 3
In_To_Low:          .Byte 1             ; Byte Ereignis fallende Flanke
                                        ; Bit 0 = Taster 1
                                        ; Bit 1 = Taster 2
                                        ; Bit 2 = Taster 3
In_Debounce:        .Byte 1             ; Byte Eingänge entprellen
                                        ; Bit 0 = Taster 1
                                        ; Bit 1 = Taster 2
                                        ; Bit 2 = Taster 3
Simply_Cnt:         .Byte 1             ; Int8 einfacher Zähler
Deb_Time:           .Byte 1             ; Int8 Entprellzeit
Befehl:             .Byte 1             ; ASCII
Port_Out:           .Byte 1             ; Byte
                                        ; Bit 0 = Relais 1
                                        ; Bit 1 = LED 1
                                        ; Bit 2 = LED 2
                                        ; Bit 3 = LED 3
Buffer:             .Byte 20            ; Int8 Empfangspuffer RS 232
```

Diesen Abschnitt verwenden wir ab jetzt für die weiteren Experimente beim Aufbau von Open_Eye. Es ist sinnvoll, diesen kleinen Abschnitt in eine einfache Textdatei zu schreiben, z.B. mit dem Editor aus dem Zubehör des Betriebssystems. In weiteren Verlauf wird immer wieder dieser Textblock benötigt und so ist er schnell verfügbar und mit C&P einfach in die ***RichTextBox*** kopiert.

### *1.2.3.8 Experimente mit Assembler-Variablen*

Der nächste Schritt soll uns nun die Variablennamen aus dem Assemblerlisting liefern. Kopieren wir diesen Text einmal in die ***RichTextBox*** und starten das Programm, ohne eine Änderung. Na ja, das Ergebnis ist nicht wirklich brauchbar. Da müssen wir wieder etwas ändern.

Beginnen wir zuerst einmal damit, diesen Text in einzelne Zeilen zu zerlegen. Der Punkt kann dafür nicht benutzt werden, da er eine Compilerdirektive anzeigt. z.B. **.DSeg** oder **.Byte**

Aber jede Zeile wird mit einem *neue Zeile* in Neudeutsch *New Line* oder *Line Feed* kurz *LF* beendet. Und das lässt sich doch sicherlich auswerten. Allerdings brauchen wir dafür den ASCII Code, denn es gibt kein direkt verwendbares Zeichen. Ein paar Seiten zuvor habe ich erklärt, wie die Ereignisroutine *KeyPress* einer ***Textbox*** arbeitet. Nutzen wir es einfach aus. Wir installieren eine ***Textbox*** an beliebigem Platz wo sie nicht stört und programmieren die Ereignismethode *Keypress*.

```
Private Sub TextBox2_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TextBox2.KeyPress
    Dim Chr_Nr As String
    Chr_Nr = Asc(e.KeyChar)
    e.KeyChar = ""
    TextBox2.Text = Str(Chr_Nr)

  End Sub
```

Nach einem Start des Programms setzen wir mit der Maus den Focus in diese ***Textbox*** und mit *Enter* wird die Zahl 13 ausgegeben. Der Befehl *e.KeyChar =““* ist erforderlich, da sonst der Inhalt der ***Textbox*** wieder überschrieben wird. Wir erhalten aber auch den ASCII-Code anderer Tasten.

Nun wissen wir, das Zeichen mit der ASCII-Nummer 13 enthält das Zeilenende- Zeichen. Für das Zeichen im String braucht es aber wieder ein Char, also muss die 13 in ein Zeichen gewandelt werden, welches im Text dem ASCII Code 13 entspricht. Das erledigt für uns die Function CHR(<Integerzahl>), ebenfalls ein Bestandteil von Visual Basic. Ändern wir also die entsprechende Zeile im Klick-Ereignis des Filter ***Button***

`Leer_Pos = InStr(Work_Buf, ".")` in `Leer_Pos = InStr(Work_Buf, CHR(13))`

Nun, wie ein Test mit der geänderten Abfrage zeigt, wird dieses Chr(13) nicht gefunden. Das liegt daran, das nicht *Line Feed* angezeigt wird,

sondern *Carriage Return* zu Deutsch *Wagenrücklauf*. Der Begriff stammt noch aus der Zeit der Schreibmaschinen. Dort wurde mit einem Hebel die Druckwalze mit dem Papier zurückgeschoben und je nach Kraftaufwand auch um eine Zeile weiter gedreht. Und dieses *Line Feed*, das Weiterdrehen der Druckwalze wird benötigt. Mit unserem Test in der ***Textbox*** werden wir diesen Code nicht erhalten, deshalb ist ein Blick in die ASCII-Tabelle erforderlich, der das *LF* mit dem Code 10 angibt. Ab und zu muss man halt auch mal im Internet nach so etwas suchen. Steuerzeichen haben da so ihre Tücken. Mit *Chr(10)* lassen sich nun einzelne Zeilen aus dem Text lösen.

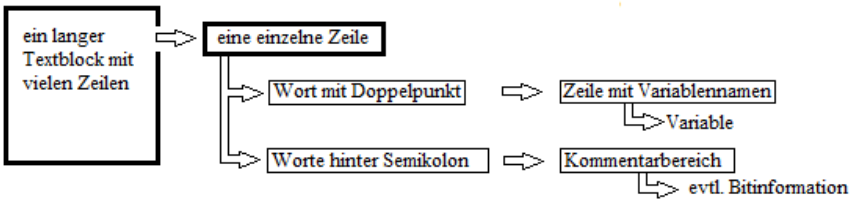
Die Änderung der Zeile

`Leer_Pos = InStr(Work_Buf, CHR(13))` in `Leer_Pos = InStr(Work_Buf, Chr(10))`

liefert nun das gewünschte Ergebnis.

Nun haben wir zwar alle Zeilen in der Variablenliste, aber eigentlich ist die Aufgabe, die Variablen, und nur die Variablen herauszufiltern.

Dazu muss die vorgestellte Routine ein wenig geändert und erweitert werden. Wir werden uns nun zum besseren Verständnis mit kleinen Schritten unserem Ziel nähern. Geht man die Aufgabe zu komplex an, erscheint die Funktion unerhört kompliziert. Betrachtet man einzelne Schritte, wird klar, wie einfach letztendlich der Aufbau doch wird. Dazu zeichnen wir uns erst einmal eine kleine Skizze. Skizzen helfen die Abläufe zu verstehen. Deshalb sollten wir immer wieder an schwierigen Stellen diese zu Hilfe nehmen.

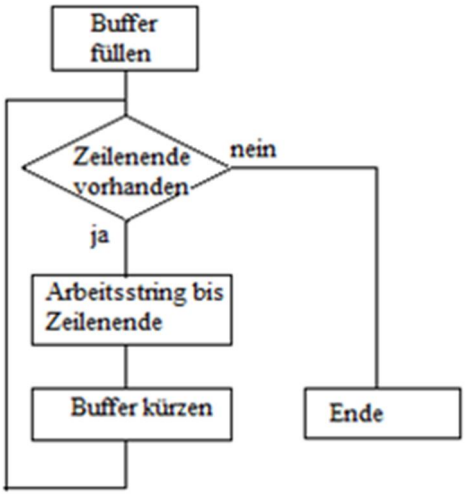



*Aufgabe Textblock zerlegen*

Zuerst ist ein Textblock unbekannter Größe vorhanden. Daraus wird eine einzelne Zeile herausgelöst. Ist in dieser Zeile ein Doppelpunkt vor einem Semikolon, enthält diese Zeile auch eine Variable. Ist ein Semikolon vorhanden, existiert ein Kommentarbereich mit evtl. weiteren Informationen.

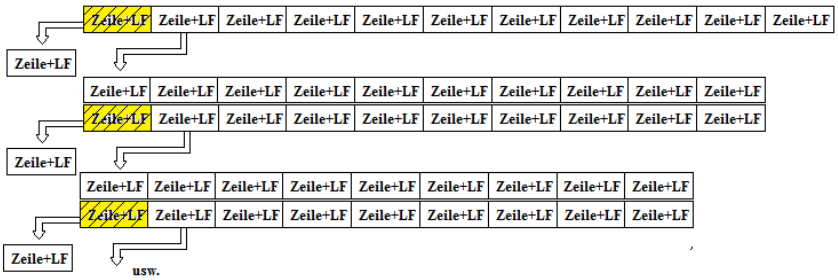
### *1.2.3.9 Eine Arbeitszeile benutzen*

Die vorangegangenen Experimente haben gezeigt, wie ein großer Text in viele kleine Texte oder besser in einzelne Zeilen zu zerlegen ist. Wenn man aus einem Assemblerlisting eine Kopie von ca. 25 gut dokumentierten Variablen zur Bearbeitung in die ***RichTextBox*** kopiert, dann schaut das ein wenig wirr aus. Keine Angst, die Übungen haben gezeigt, wie es funktioniert. Und wenn der Ansatz richtig ist, ist es dem Filter letztendlich egal, ob 10, 20 oder 439 Variablen zu finden sind. Die beigestellt Skizze zeigt eine ganz einfache Programmstruktur von diesem Vorgang.



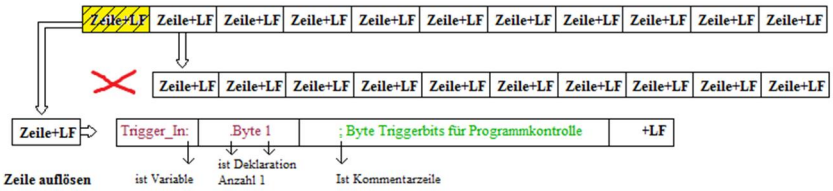

*PAP Textblock in Zeilen zerlegen*

Solch eine Skizze ist schnell auf ein Blatt Papier gezeichnet. Das Lösen der Aufgabe wird so in einem Ablaufplan sichtbar gemacht. Der erste Entwurf ist ein reiner *Brainstorming*, völlig ohne Bezug auf das eigentliche Programm. Man stellt sich einfach vor, ein langer Satz ist auf einen Papierstreifen geschrieben. Nun sucht man vom Anfang her ein festgelegtes Zeichen. Dort schneidet man den langen Papierstreifen ab. Der linke Teil ist der Arbeitsteil, der Rest wird weiter betrachtet, wenn der Arbeitsteil ausgewertet ist.




*Filter Zeilen*

Auf das Assemblerlisting bezogen würden wir Zeile für Zeile vom Blatt abschneiden und die Streifen einzeln nach Informationen durchsuchen.




*Filter Variable*

Diese skizzierte Vorgehensweise macht den möglichen Algorithmus deutlich. Bei jedem Durchlauf wird nun der Pufferstring um die herausgelöste Zeile gekürzt und irgendwann ist dann der Pufferstring leer. Ist diese Arbeitsweise deutlich, sind wir auch in der Lage, diese Aufgabe mit ein paar Anweisungen vom PC ausführen zu lassen.

### *1.2.4 Der Filter, das richtige Programm*

Das Zerlegen von Texten sollte soweit verstanden sein, dass nun die Programmierung des Filters für die Variablennamen begonnen werden kann.

So eine Variable hat ja folgenden Aufbau:

Name – Format - Kommentar, beginnend mit dem verwendeten Format

```
-----------------------------------------------------------------------------------------
MyVar:        .Byte 1          ; Byte, eine Variablenstruktur <Zeilenende LF>
Next_Var:     .Byte 1          ; Int8, weitere Variablen <Zeilenende LF>
Ctrl_Var:     .Byte 1          ; Byte, Variable mit Kontrollflags <Zeilenende LF>
                                ; Bit 0 = Start <Zeilenende LF>
                                ; Bit 1 = Stopp <Zeilenende LF>
                                ; Bit 2 = Time_Flag <Zeilenende LF>
                                ; etc.
Var_Array:    .Byte 10          ; Byte, ein 10 Byte großes Variablenfeld <Zeilenende LF>
```

Hier kommt die bereits in AVR Studio oder im Codebeispiel eingegebene Variablenliste zur Verwendung.

Zuerst lösen wir eine Zeile aus dem Text. Das Ergebnis wird diesmal nicht in die Listbox eingetragen, sondern in eine Variable *Zeile*. Dann suchen wir den Doppelpunkt in dieser Zeile. Dafür benötigt es ein paar lokale Variablen und es macht auch Sinn, die Namen den neuen Bedingungen anzupassen.

```
Dim Work_Buf as String     'bleibt für die Kopie aus der RichTextBox
Dim Zeile as String        'neu für die herauskopierte Zeile
Dim Var_Name as String     'neu für den ermittelten Variablennamen
Dim LF_Pos as Integer      'neu für Position von Zeilenendzeichen
Dim DP_Pos as Integer      'neu für Position von Doppelpunkt
```

Wenn nun statt wie bisher mit *Leer_Pos* mit *LF_Pos* gearbeitet und statt *Wort* die Variable *Zeile* benutzt wird, haben wir die Grundlage für unsere Aufgabe.

```
Private Sub BT_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles BT_Filter_Click
  Dim Work_Buf As String
```

```
  Dim Zeile As String
  Dim LF_Pos As Integer
  Dim DP_Pos as Integer
  Dim Var_Name as String
  Work_Buf = RT_Buffer.Text                    'Kopie aus RichTextBox zur Bearbeitung
  Zeile = ""                                    'mit Leerstring vorbesetzen
  Var_Name=""                                   'mit Leerstring vorbesetzen
  Format="Integer8"                            'Defaultwert für Format setzen
  LB_Variablen.Items.Clear()
  While Len(Work_Buf) > 0
    LF_Pos = InStr(Work_Buf, Chr(10))
    If LF_Pos > 0 Then
      Zeile = Mid(Work_Buf, 1, LF_Pos – 1)          ' Zeile aus Pufferstring herauslösen
      Work_Buf = Mid(Work_Buf, LF_Pos + 1, Len(Work_Buf) – LF_Pos) ' Puffer kürzen
    Else
      Zeile = Work_Buf                              ' letzte Zeile
      Work_Buf = ""                                 ' Puffer leeren
    End If
    If Zeile <> " " Then
              'ab hier kommt nun ein neues Programm
    End If
  End While
End Sub
```

### *1.2.4.1 Untersuchen einer Zeile auf Information*

In einer solchen Zeile steht nicht immer eine Variable. Und so beginnen wir mit der Suche nach dem Doppelpunkt in der Zeile. Um nun nicht immer wieder das gesamte Programm aufzulisten, zeige ich hier nur noch den Bereich, der die Informationen in den Zeilen abfragt und herausholt.

Die wichtigste Information ist : gibt es einen Variablennamen? Die Antwort gibt der Doppelpunkt. Er steht immer direkt hinter einer Variablen. Die folgenden Programmzeilen erledigen diesen Job und setzen den Variablennamen in die so definierte Variable. Anschließend fügen wir den Namen in die Liste ein.

Es ist dieser Bereich, der jetzt neu beschrieben wird

```
If Zeile <> " " Then                                    'alt
  DP_Pos=InStr(Zeile,":")                   'ab hier kommt nun ein neues Programm
  If DP_Pos >0 then
    Var_Name = Mid(Zeile, 1, DP_Pos-1)   ' Variablennamen herauslösen
    LB_Variablen.Items.Add(VarName)      ' und in die Liste eintragen
  End If
End If                                                  'alt
```

Nach der Änderung in Visual Basic stehen nur die Variablennamen in der Liste. Schön, das war ja einfach. Nur, es gibt da einen kleinen Haken. Die Zeile

```
Buffer: .Byte 20 ; Integer8, Empfangspuffer RS 232
```

belegt im Controller 20 Speicherstellen. Wenn eine solche Variablendefinition mitten in der Variablenliste steht, passen die Namen nicht mehr zu den physikalischen Speicherstellen. Unser Programm soll dieses berücksichtigen. Diesmal ist der Punkt mit dem Wort Byte dahinter zu finden. Auch ist das nur in der Zeile mit dem Variablennamen von Bedeutung. Daher kommt die Erweiterung in den Bereich

```
  If DP_Pos >0 then
        ....
  end
```

Bevor das zusätzliche Programm erstellt wird, sehen wir uns die Zeile einmal an. Die Position .Byte zu finden ist kein Problem, aber die Anzahl der Bytes kann kleiner 10, aber auch größer 9 sein. Wir müssen sogar mit

einer 3 stelligen Zahl rechnen, die hinter .Byte stehen kann. Betrachten wir die Zeile einmal genau.

Angenommen, die Position von .Byte wird mit dem Wert 23 angegeben, dann beginnt die Zahlenangabe 6 Stellen weiter. Es gilt nun eine allgemein gültige Struktur für ein Programm zu finden, welche eine 1 genauso herausfiltert wie 341. Ein wenig hilft die Zeile selbst dabei, denn in einer Zeile stehen keine Zahlen sondern Zeichen. Und wir können über eine Abfrage herausbekommen, ob die Zeichen 0 bis 9 an einer bestimmten Stelle stehen.


*Zeichen Positionsberechnung*

Die erste abzufragende Stelle ist also die Position von .Byte+6. Da erwarten wir auf jeden Fall eine Ziffer. Trotzdem muss die Abfrage auf gültigen Bereich erfolgen, nur so kann in einer Schleife gearbeitet werden.

Um diese Ziffernfolge nun aus der Zeile herauszuholen, nehmen wir einen Hilfsstring, der zuerst einmal geleert werden muss. Wird eine Ziffer erkannt, wird sie mit einer String Addition an den Hilfsstring angefügt.

### *1.2.4.2 Variablendimension erfassen*

Erweitern wir zuerst die Variablendeklaration um eine Integer für die Positionsangabe .Byte und eine Stringvariable, um die Ziffernfolge aufzunehmen.

```
Dim Byte_Pos as Integer
Dim Byte_Cnt as String
```

In unserem Programm befinden wir uns an dieser Stelle

```
    If DP_Pos >0 then
      Var_Name = Mid(Zeile, 1, DP_Pos-1)   ' hier geht es nun weiter

      LB_Variablen.Items.Add(VarName)
    End If
```

Der Variablenname muss zuerst erfasst werden, das ist so ok. Aber für den Eintrag in die Liste ist es noch zu früh.

Hier setzt die weitere Bearbeitung und Auswertung der Zeile mit der Erfassung der ersten Ziffer hinter **.Byte** ein. Zuerst wird die Variable *Byte_Cnt* gelöscht, und da es eine Stringvariable ist, wird ihr mit *Byte_Cnt=“*“ ein Leerstring zugewiesen

Es folgt die Positionsberechnung von **.Byte**

```
      Byte_Pos=InStr(Zeile,“.Byte“)
```

Auch hier wird nur weiter gearbeitet, wenn Byte_Pos größer 0 ist, da sonst keine weitere Bearbeitung erfolgen kann.

```
      If Byte_Pos>0 then
```

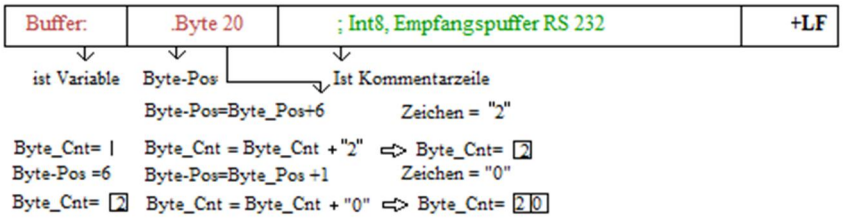
Im ersten Schritt wird die Variable Byte_Pos auf die Position der ersten Ziffer gestellt

```
        Byte_Pos= Byte_Pos+6
```

Nun folgt eine While Schleife, die nur durchlaufen wird, wenn das Zeichen an *Byte_Pos* nicht außerhalb von 0 – 9 liegt.

```
While (Mid(Zeile, Byte_Pos, 1) >= "0") And (Mid(Zeile, Byte_Pos, 1) <= "9")
  Byte_Cnt = Byte_Cnt + Mid(Zeile, Byte_Pos, 1)          ' String Addition
  Byte_Pos = Byte_Pos + 1                                ' Zeiger nachführen
End While
```

Auch hier eine kleine Skizze:




*Variablenarray zusammenstellen*

Nun steht in der Stringvariablen ein Zahlenwert, der dem Wert hinter .Byte entsprechen sollte. Steht nur eine 1 in der Variablen, gut, dann kann der Variablenname unverändert in die Variablenliste. Steht aber eine andere Zahl darin, so müssen nun mehrere Variablennamen mit gleichen Namen und einem zusätzlichen Index versehen in die Variablenliste eingefügt werden. Nur so stimmt später der übermittelte Datenstrom mit den hier aufgeführten Zellennamen überein. Zuerst also die Abfrage nach dem Inhalt von Byte_Cnt

```
If Byte_Cnt="1" then
  LB_Variablen.Items.Add(Var_Name)
```

Wenn diese Bedingung nicht erfüllt ist, dann ist ein Arrayaufbau erforderlich. Dies geschieht im *Else* -Zweig der *If*-Abfrage. Eine feste *For-Next*-Schleife baut das Array auf, da die Grenzen durch den Inhalt von *Byte_Cnt* festgelegt sind. Der Variablenname wird dann in der Schleife mit einer Klammer und der laufenden Nummer, die vom Schleifenzähler abgeleitet wird, ergänzt.

```
else
  For I = 0 To Val(Byte_Cnt)-1
  LB_Variablen.Items.Add(Var_Name+"[„+Str(i)+"]"
Next I
```

```
    end If
```

Die der konvertierte Wert von *Byte_Cnt* muss um 1 verringert werden, da der Index bei 0 beginnt.

Hier noch einmal zur besseren Übersicht, die gesamte Routine

```
Private Sub BT_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles BT_Filter_Click
  Dim Work_Buf As String
  Dim Zeile As String
  Dim Var_Name as String
  Dim Byte_Cnt as String
  Dim Format as String
  Dim I as Integer
  Dim LF_Pos As Integer
  Dim DP_Pos as Integer
  Dim Byte_Pos as Integer
  Work_Buf = RT_Buffer.Text
  Zeile = ""                                  'mit Leerstring vorbesetzen
  Var_Name=""                                 'mit Leerstring vorbesetzen
  Format="Integer8"                           'Defaultwert für Format setzen
  LB_Variablen.Items.Clear()
  While Len(Work_Buf) > 0
    LF_Pos = InStr(Work_Buf, Chr(10))
    If LF_Pos > 0 Then
      Zeile = Mid(Work_Buf, 1, LF_Pos - 1)
      Work_Buf = Mid(Work_Buf, LF_Pos + 1, Len(Work_Buf) - LF_Pos)
    Else
      Zeile = Work_Buf
      Work_Buf = ""
    End If
    If Zeile <> " " Then
      DP_Pos=InStr(Zeile,":")                 'ab hier kommt nun ein neues Programm
      If DP_Pos >0 then
        Var_Name = Mid(Zeile, 1, DP_Pos-1)
 ' hier muss noch die Auswertung für Format erfolgen

        Byte_Cnt=""
        Byte_Pos=InStr(Zeile,".Byte")
        If Byte_Pos>0 then
          Byte_Pos=Byte_Pos+6
          While (Mid(Zeile, Byte_Pos, 1) >= "0") And (Mid(Zeile, Byte_Pos, 1) <= "9")
            Byte_Cnt = Byte_Cnt + Mid(Zeile, Byte_Pos, 1) ' String addition
            Byte_Pos = Byte_Pos + 1 ' Zeiger nachführen
          End While
          If Byte_Cnt="1" then
            LB_Variablen.Items.Add(Var_Name)
          else
            For I = 0 To Val(Byte_Cnt)-1
              LB_Variablen.Items.Add(Var_Name+"[„+Str(i)+"]")
            Next I
          end If
        end If
      end If
    End If
  End While
End Sub
```

### *1.2.4.3 Die Formatangabe ermitteln*

Das Ergebnis kann sich schon einmal sehen lassen. Aber es ist noch lange nicht perfekt. So fehlt uns zur Darstellung der Werte noch eine wichtige Information. Welche Aufgabe hat die Assembler-Variable. Soll sie irgendetwas zählen, dann muss die Darstellung dezimal erfolgen. Sind es Bitmuster, sollte ein Bitfeld ausgegeben werden. Deshalb geben wir der Variablen in ihrer Deklarationszeile auch noch ihr Format an. Das gehört zwar nicht zum Assembler, kann aber im Kommentar abgelegt werden. So ist es ganz einfach möglich, eine Auswertung durchzuführen. Die Stelle habe ich bereits markiert. Nun, das Format soll zu jeder Variablen passen und da auch Variablenarrays erzeugt werden, muss der Formateintrag auch mit dem Eintrag des Variablennamens erfolgen.

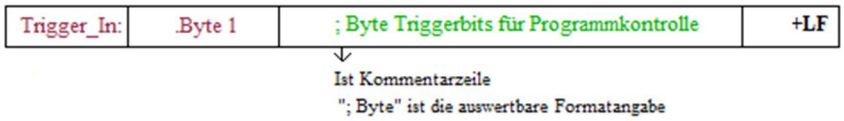
Die Gestaltung der Information zum Format kann ganz individuell eingebaut werden. Wichtig ist: es muss eine eindeutige Erkennbarkeit vorliegen. Sinnvollerweise nutzt man das Semikolon, welches in Assembler Kommentar vom Code trennt, und prüft, ob dahinter ein Formatschlüsselwort Byte, Int8, Int16 oder ASCII eingetragen ist. Wie nun so ein Stringinhalt gefunden und zugeordnet werden kann wissen wir bereits. Aber Byte ist ja bereits vorhanden. Nun, das erste Byte für die Dimensionierung hatte in der Abfrage noch einen Punkt. Das zweite Byte für die Format-Information bekommt ja das Semikolon vorangestellt und auch mit in die Funktion *InStr()* übergeben. Man muss nur beachten, ob hinter dem Semikolon ein Leerzeichen vorgesehen ist, oder steht die Information direkt hinter dem Semikolon. Ich lasse gern ein Leerzeichen dazwischen, also muss es mit in die Abfrage:

```
Format_Str="Int8"
If InStr(Zeile, „; Byte")>0 then Format_Str ="Byte"          ' Darstellung Binär
If InStr(Zeile, „; ASCII")>0 then Format_Str ="ASCII"        ' Darstellung Ascii-Zeichen
If InStr(Zeile, „; Hex")>0 then Format_Str ="Hex"            ' Darstellung 8 Bit Hexadezimal
If InStr(Zeile, „; Int16")>0 then Format_Str ="Int16"        ' Darstellung 16 Bit Dezimal
If InStr(Zeile, „; Int32")>0 then Format_Str ="Int32"        ' Darstellung 32 Bit Hexadezimal
If InStr(Zeile, „; Word")>0 then Format_Str ="Word"          ' Darstellung 16 Bit Hexadezimal
If InStr(Zeile, „; DWord")>0 then Format_Str ="Dword"        ' Darstellung 32 Bit Hexadezimal
```

Natürlich ist auch eine Erfassung beider Möglichkeiten einfach umzusetzen:

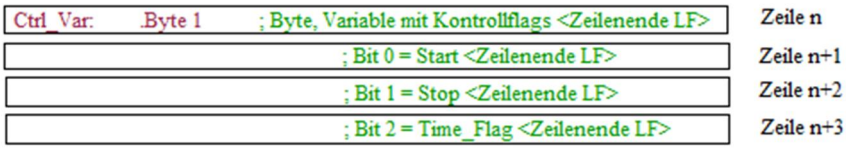
```
If ((InStr(Zeile, „; Byte")>0) or (InStr(Zeile, „;Byte")>0)) then Format_Str ="Byte"
If ((InStr(Zeile, „; ASCII")>0) or (InStr(Zeile, „;ASCII")>0)) then Format_Str ="ASCII"
```

```
If ((InStr(Zeile, „; Hex“)>0) or (InStr(Zeile, „;Hex“)>0)) then Format_Str =“Hex“
If ((InStr(Zeile, „; Int16“)>0) or (InStr(Zeile, „;Int16“)>0)) then Format_Str =“Int16“
If ((InStr(Zeile, „; Int32“)>0) or (InStr(Zeile, „;Int32“)>0)) then Format_Str =“Int32“
If ((InStr(Zeile, „; Word“)>0) or (InStr(Zeile, „;Word“)>0)) then Format_Str =“Word“
If ((InStr(Zeile, „; Dword“)>0) or (InStr(Zeile, „;DWord“)>0)) then Format_Str =“Dword“
```




*Format bestimmen*

Ein Blick auf die folgende Skizze zeigt, dass die Formatangabe auch nur in der Zeile mit dem Variablennamen Sinn macht.




*Zeilen mit und ohne Formatangabe*

Nun kann Format_Str auch mit in die LB_Format-***Listbox*** parallel zum Variablennamen eingetragen werden. Eine nicht ausgewertete Zeile wird automatisch mit dem Format „Int8“ gesetzt.

```
LB_Variablen.Items.Add(Var_Name)
LB_Formate.Items.Add(Format_Str)
```

und im anderen Zweig ebenfalls

```
LB_Variablen.Items.Add(Var_Name+“[„+Str(i)+“]“)
LB_Formate.Items.Add(Format_Str)
```

### *1.2.4.4 Erfassung einzelner Bits*

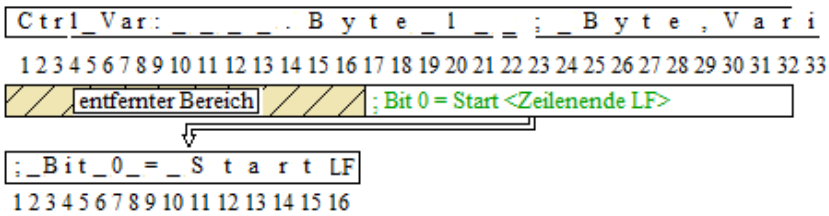
Nun bleibt noch eines zu tun, um diesen Filter perfekt zu machen. Die Auswertung eventuell zugeordneter Bits. Betrachten wir dazu einmal einen kleinen Ausschnitt vom Codebeispiel Assembler

```
In_To_Low: .Byte 1        ; Byte Ereignis fallende Flanke
                           ; Bit 0 = Taster 1
                           ; Bit 1 = Taster 2
                           ; Bit 2 = Taster 3
```

Der Zeile mit dem Variablennamen haben wir nun alle relevanten Informationen entnommen. Klar ist, eine Zuordnung einzelner Bits wird nicht in einer Zeile durchzuführen sein. Der Editor ließe das zwar zu, aber die Darstellung wäre kein bisschen übersichtlich.

So wie im Ausschnitt aber stellen sich die einzelnen Bits mit ihrer Bedeutung in den Kommentarzeilen schön übersichtlich dar. Bisher wurden Zeilen ohne Variablennamen einfach ignoriert. Dies wird sich jetzt ändern. Wir werden, wenn die Variable als Byte formatiert ist, die folgenden Zeilen auch auf den Inhalt **; Bit** prüften und Informationen über einzelne Bits herauslösen. Das Leerzeichen hinter Bit definiert klar die Suche nach dem Schlüsselwort Bit, weil sonst eventuell auch Bitmaske oder Bitmuster gefunden werden. Alternativ wäre auch der Begriff Bit1 oder Bit7 eindeutig. Aber dies würde eine erheblich aufwändigere Auswertung bedeuten. Da sind die Zahlen 0 bis 7 und das Leerzeichen. Das sind schon 8 Möglichkeiten, ein Bit zu definieren. Steht erst einmal ein Leerzeichen ist klar, dass es eine Bitzuordnung gibt. Die Nummer kann nur einstellig sein und daher ist das Zeichen auch an einer definierten Stelle relativ zur gefundenen Stelle des Suchbegriffs **; Bit** . Hier sollte aber festgelegt sein, ob hinter dem Semikolon ein Leerzeichen eingefügt wird. Das ändert die relative Position der Bitnummer.

Wieder malen wir uns ein Bild, um die Struktur zu verstehen



*Bitbeschreibung herausfiltern*

Der Aufbau zeigt deutlich den relativen Bezug der Bitnummer zum Semikolon. Der Text mit der Beschreibung der Bitfunktion hat ebenfalls einen Festen relativen Abstand zum Semikolon, aber auch zum = Zeichen. Mit diesen Informationen ist er auch aus dem String herauszulösen. So beginnt er an der Stelle 11 und endet an 16. Der Teil, der nun herauskopiert werden muss hat die Länge 16 – 11, also 5

Der Kopierbefehl

```
Mid(String,Anfang, Laenge)
```

braucht genau diese Werte. Also definieren wir

```
Anfang = 11
Laenge = Len(String)-Anfang
```

Und dann haben wir die Zusatzinformation zur Bitfunktion.

### *1.2.4.5 Liste mit Einzelbit ergänzen*

Nun sind aber nicht alle Bits kommentiert. Für unsere Liste hätten wir aber gern eine komplette Beschreibung aller Bits. Da kommt uns eine Variable mit einer Arraystruktur gerade recht. Zusätzlich brauchen wir einen Hilfsstring zur Aufnahme des Kommentartextes bis zum nächsten Leerzeichen und eine Indexvariable für den Zugriff auf das Array, um komplizierte und unleserliche Konstrukte zu vermeiden.

```
Dim Bit_Liste(7) as String
Dim Bit_Info as String
Dim Bit_Nr as Integer
```

Nun, warum die Arrayvariable nur mit Dimension sieben?

Wenn man bedenkt, das ein Array mit 0 zu zählen beginnt, sind es auch 8 Felder. Zuerst besetzen wir bei jeder Zeile mit Variablennamen und der Formateinstellung Byte das Array mit **nicht benutzt.**

Hier der betreffende Programmabschnitt:

```
        If DP_Pos >0 then
          If Format_Str="Byte" then ' ist Vorgängerformat ="Byte" dann übertragen
            For I = 0 to 7
              LB_Variablen.Items.Add(Bit_Liste(i))
              LB_Formate.Items.Add("Bit")
            next i
          End If
          Var_Name = Mid(Zeile, 1, DP_Pos-1)
          ' hier muss die Auswertung für Format erfolgen
          Format="Integer8"
          If InStr(Zeile, „; Byte")>0 then Format_Str ="Byte"
            If InStr(Zeile, „; ASCII")>0 then Format_Str ="ASCII"
              If InStr(Zeile, „; Hex")>0 then Format_Str ="Hex"
                Byte_Cnt=""
                If Format_Str="Byte" then                    ' Liste für 8 Bit vorbesetzen
                  For I = 0 to 7
                    Bit_Liste(i)=Var_Name+"."+Str(i)+" = nicht verwendet" ' Defaultwert
                  Next I
                End If
' hier steht nun der Rest der Bearbeitung der Zeile mit dem Variablennamen
              End If
'diese Zeilen werden zugefügt zur Auswertung von Bitinformationen
' Das Format bleibt bis zur nächsten Variablen erhalten
              If Format_Str="Byte" then
                Bit_Pos=InStr(Zeile,"; Bit ")
                If Bit_Pos>0 then
                  Bit_Pos=Bit_Pos +5              ' den Zeiger auf die Bitnummer setzen
                  Bit_Nr=Val(Zeile(Bit_Pos))
```

```
            Bit_Info=Var_Name+”.”+Zeile(Bit_Pos)+” = “
            Bit_Pos=InStr(Zeile,”=“)            ' Kommentaranfang suchen
            Zeile =Mid(Zeile,Bit_Pos+1,Len(Zeile)-Bit_Pos-1) ' hier die Zeile kürzen
            Bit_Pos=InStr(Zeile,” “)            ' und das nächste Leerzeichen suchen
            Bit_Info=Bit_Info+Mid(Zeile, 1,Bit_Pos) ' damit ist der Kommentartext begrenzt
            Bit_Liste(Bit_Nr)=Bit_Info ' zusammengesetzte Info in das Array eintragen
          end If
        end If
```

Da die Bitnummer immer nur einstellig sein kann, steht sie immer an der mit Bit_Pos +5 ermittelten Stelle. Mit dieser Kenntnis ist es nicht mehr schwer, die Bitinformation zu erhalten und in den Listen abzulegen. Der Filter ist nun komplett.

Hier die gesamte Routine, um aus dem Text in der ***RichTextBox*** alle relevanten Informationen herauszulösen.

```
Private Sub BT_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles BT_Filter_Click
  Dim Work_Buf As String
  Dim Zeile As String
  Dim Var_Name As String
  Dim Byte_Cnt As String
  Dim Format_Str As String
  Dim Bit_Info As String
  Dim Bit_Liste(7) As String
  Dim Bit_Pos As Integer
  Dim Bit_Nr As Integer
  Dim I As Integer
  Dim LF_Pos As Integer
  Dim Dp_Pos As Integer
  Dim Byte_Pos As Integer
  Dim StartFlag as Boolean
  Work_Buf = RT_Buffer.Text
  Zeile = ""                                  ' mit Leerstring vorbesetzen
  Var_Name = ""                               ' mit Leerstring vorbesetzen
  Format_Str = "Int8"                         ' Defaultwert Format setzen
  Startflag=false                             ' Startflag vorbereiten
  LB_Variablen.Items.Clear()
  While Len(Work_Buf) > 0
    LF_Pos = InStr(Work_Buf, Chr(10))
    If LF_Pos > 0 Then
      Zeile = Mid(Work_Buf, 1, LF_Pos - 1)
      Work_Buf = Mid(Work_Buf, LF_Pos + 1, Len(Work_Buf) - LF_Pos)
    Else
      Zeile = Work_Buf
      Work_Buf = ""
   End If
   If Zeile <> " " Then
'*************** Zeile auf Variablennamen prüfen *************
      Dp_Pos = InStr(Zeile, ":")
      If Dp_Pos > 0 Then
        If (Format_Str = "Byte") and Startflag Then 'Vorgänger Variable hatte Format Byte
'nun noch die Einzelbitdeklaration in die Variablenliste zufügen und das Format auf “Bit”
setzen
```

```
        For I = 0 To 7
' Bitbeschreibung Vorgänger in Variablen- und Formatlisten ergänzen
          LB_Variablen.Items.Add(Bit_Liste(I))
          LB_Formate.Items.Add("Bit")
        Next I
      End If
      StartFlag=true
      Var_Name = Mid(Zeile, 1, Dp_Pos - 1)
'**************** Format erfassen *****************
      Format_Str = "Int8"                                  'Defaultformat
      If InStr(Zeile, "; Byte") > 0 Then
        Format_Str = "Byte"
        For I = 0 To 7 'Bitliste setzen mit Defaultwert
          Bit_Liste(I) = Var_Name + ".Bit" + Str(I) + " : nicht verwendet"
           Next i
       End If
       If InStr(Zeile, "; ASCII") > 0 Then
         Format_Str = "ASCII"
       End If
       If InStr(Zeile, "; Hex") > 0 Then
         Format_Str = "Hex"
       End If
'Berechnen der Felddimension.
       Byte_Cnt = ""
       Byte_Pos = InStr(Zeile, ".")
       If Byte_Pos > 0 Then
         Byte_Pos = Byte_Pos + 6
         While (Mid(Zeile, Byte_Pos, 1) >= "0") And (Mid(Zeile, Byte_Pos, 1) <= "9")
           Byte_Cnt = Byte_Cnt + Mid(Zeile, Byte_Pos, 1) ' String addition
           Byte_Pos = Byte_Pos + 1                          ' Zeiger nachführen
         End While
         If Byte_Cnt = "1" Then
           LB_Variablen.Items.Add(Var_Name)
           LB_Formate.Items.Add(Format_Str)
         Else
           For I = 0 To Val(Byte_Cnt) - 1
             LB_Variablen.Items.Add(Var_Name + "[" + Str(I) + "]")
             LB_Formate.Items.Add(Format_Str)
           Next I
        End If
      End If
    End If
' Zeile mit Variablennamen ist abgearbeitet. Nun wird geprüft, ob explizite Bitdeklarationen
vorliegen.
     If Format_Str = "Byte" Then
       Bit_Pos = InStr(Zeile, "; Bit ")
       If Bit_Pos > 0 Then
         Bit_Pos = Bit_Pos + 5
         Bit_Nr = Val(Zeile(Bit_Pos))
         Bit_Info = Var_Name + ".Bit " + Zeile(Bit_Pos) + " : "
         Bit_Pos = InStr(Zeile, "=")
         Zeile = Mid(Zeile, Bit_Pos + 1, Len(Zeile) - Bit_Pos)
         Zeile = Trim(Zeile)
         Bit_Info = Bit_Info + Zeile
         Bit_Liste(Bit_Nr) = Bit_Info
       End If
```

```
      End If
    End If
  End While
  If (Format_Str = "Byte") and Startflag Then        'letzte Variable hatte Format Byte
    For I = 0 To 7
' Bitbeschreibung Vorgänger in Variablen- und Formatlisten ergänzen
      LB_Variablen.Items.Add(Bit_Liste(I))
      LB_Formate.Items.Add("Bit")
    Next I
  End If
End Sub
```

Diese Befehlszeilen passen nicht mehr auf eine Seite. Na ja, für den Job, der in dieser Routine erledigt wird, ist das wirklich nicht zu viel Code. Testen wir erst einmal das Ergebnis indem wir den Text aus dem Codebeispiel in das laufende Programm in die dafür vorgesehene ***Richtextbox*** kopieren und den Filter starten. In den Listen sollten nun die Variablennamen, die Formate und evtl. auch Einzelbitbeschreibungen aufgeführt sein.

Wenn die Testergebnisse korrekt sind, wäre eine Überlegung angebracht, den Code etwas übersichtlicher zu gestalten und diese komplexe Ereignisroutine durch kleinere Programmmodule aufzuteilen. Allerdings sind ein paar Probleme zu lösen, um wirklich einen Nutzen daraus zu ziehen. Was hier in einer einzigen Routine mit lokalen Variablen gelöst ist, wird kompliziert, wenn man nur Zeile für Zeile herauslöst und die Arbeit Subroutinen übergibt. Es sind die Kommentarzeilen der Bit-Beschreibung, die auf Inhalte der lokalen Variablen zugreifen. Diese Variablen sind in aufgerufenen Subroutinen nicht gültig. Hier ist ein völlig anderes Konzept erforderlich. Also beginnen wir noch einmal mit der Zerlegung in Zeilen. Die lokale Variable *Work_Buf* wird dabei weiterhin benutzt. Diesmal werden die Zeilen zur Auswertung an Unterprogramme weitergeleitet. Kopieren wir erst einmal die Subroutine aus der Ereignisbearbeitung *BT_Filter_Click* und entfernen alles bis auf die Variablendeklaration und die erforderlichen Befehle, um Zeile für Zeile aus dem Arbeitspuffer herauszulösen. Übrig bleiben die beiden Fälle

```
Zeile mit Variablenname und Zeile mit Bit-Beschreibung
Alles andere braucht nicht ausgewertet zu werden.
```

Unsere Ereignisroutine *BT_Filter_Click* ist nun schon wesentlich übersichtlicher:

```
Private Sub BT_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles BT_Filter_Click
  Dim Work_Buf As String                    ' Arbeitspuffer zum Abarbeiten
  Dim Zeile As String                       ' wird für Zeilenweise Abarbeitung gebraucht
```

```
  Dim Var_Name As String                    'wird für Bit-Beschreibung benötigt
  Dim Byte_Cnt As String                    ' wird hier nicht benötigt
  Dim Format_Str As String          ' Wird zur Auswertung der Bit-Beschreibung gebraucht
  Dim Bit_Info As String                    ' behalten wir
  Dim Bit_Liste(7) As String                ' behalten wir
  Dim Bit_Pos As Integer                    ' erforderlich zur Bit-Beschreibung
  Dim Bit_Nr As Integer                     ' behalten wir
  Dim I As Integer                          ' benötigen wir zur Vorbesetzung der Bit_Liste
  Dim LF_Pos As Integer                     ' Zur Zeilenbearbeitung erforderlich
  Dim Dp_Pos As Integer                 ' benötigt zur Erkennung eines Variablennamens
  Dim Byte_Pos As Integer                   ' wird hier nicht benötigt
  Work_Buf = RT_Buffer.Text                 ' Kopie der Assembler Variablendeklaration
  Zeile = ""                                ' mit Leerstring vorbesetzen
  Var_Name = ""                             ' mit Leerstring vorbesetzen
  Format_Str = "Int8"                       ' Defaultwert Format setzen
  LB_Variablen.Items.Clear()                ' Variablenliste leeren
  While Len(Work_Buf) > 0                   ' Beginn der Auswerteschleife
    LF_Pos = InStr(Work_Buf, Chr(10))       ' Zeilenende abfragen
    If LF_Pos > 0 Then
      Zeile = Mid(Work_Buf, 1, LF_Pos – 1) ' Zeile herauskopieren und Work_Buf kürzen
      Work_Buf = Mid(Work_Buf, LF_Pos + 1, Len(Work_Buf) - LF_Pos)
    Else
      Zeile = Work_Buf ' letzte Zeile und Work_Buf leeren
      Work_Buf = ""
    End If
    If Zeile <> " " Then                    ' Zeile vorhanden, dann Auswertung beginnen
      Dp_Pos = InStr(Zeile, ":")            ' zuerst Zeile auf Variablennamen prüfen
      If Dp_Pos > 0 Then                    ' neuer Variablenname
        For I = 0 To 7                      ' hier erst mal Bitliste setzen mit Defaultwert
          Bit_Liste(I) = Var_Name + ".Bit" + Str(I) + " : nicht verwendet"
        Next i
              ' hier kommt der Aufruf der Auswertung hin
      end if
      If Format_Str = "Byte" Then           ' prüfen, ob Bit-Beschreibung erforderlich
              ' hier kommt der Aufruf der Auswertung hin
        Bit_Liste(Bit_Nr) =Bit_Info ' Nr und Info liefert die Auswertung
      end if
    end if
  end While
end Sub
```

Diesen Programmblock entrümpeln wir erst nun. Dazu prüfen wir erst einmal, welche Variablen noch benötigt werden. Damit wird nun der Filter etwas kleiner und überschaubarer. Danach überlegen wir, wie die erforderliche Information gewonnen werden kann.

Die Auswertung bekommt immer nur eine Zeile. Nun ist aber für die Zusammenstellung der Bit-Beschreibung der Variablenname erforderlich. Dieser muss also von der Auswertung der Zeile mit dem Variablennamen zurückgeliefert werden. Um eine Bit-Beschreibung in die vorbereitete Liste

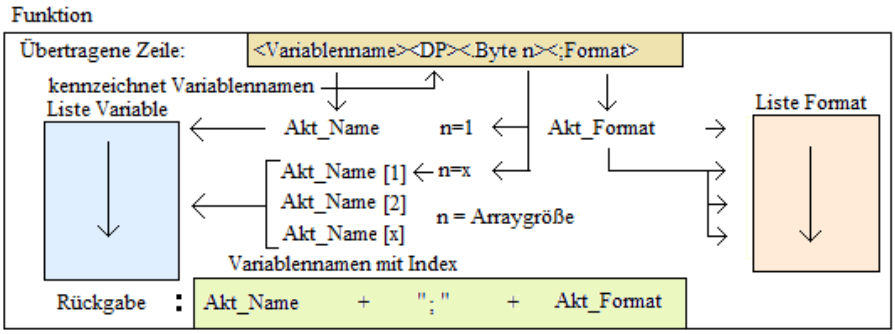
einzutragen, ist auch diese Information von der Auswertung der Bit-Beschreibung notwendig.

### *1.2.4.6 Mit Funktionen arbeiten*

Die Subroutinen, die eine solche Information liefern können, sind Functions. Sie bekommen die aktuell herausgelöste Zeile und evtl. weitere Parameter übergeben und sie liefern in einem String zusammengefasst die benötigte Information. Starten wir mit einer Function, die uns den Variablennamen und das Format liefert. Auch eine Auswertung, ob ein Variablenarray vorliegt, kann diese Routine durchführen.

```
Public Function Get_Var_Name(ByVal Akt_Zeile as String, ByVal Akt_DP as Integer) as String
  Dim Akt_Name as String
  Dim Akt_Format as String
  Dim Byte_Cnt as String
  Dim Zeile as String
  Dim Ergebnis as String
  Dim Byte_Pos as Integer
  Dim I as Integer
  Zeile=Akt_Zeile
  Akt_Name = Mid(Zeile, 1, Akt_Dp - 1)
  Akt_Format = "Int8"                                    'Defaultformat
  If InStr(Zeile, "; Byte") > 0 Then Akt_Format=“Byte“
  If InStr(Zeile, "; ASCII") > 0 Then Akt_Format = "ASCII"
  If InStr(Zeile, "; Hex") > 0 Then Akt_Format = "Hex8"
  If InStr(Zeile, "; Hex15") > 0 Then Akt_Format = "Hex16"
  If InStr(Zeile, "; Hex32") > 0 Then Akt_Format = "Hex32"
  If InStr(Zeile, "; Int16") > 0 Then Akt_Format = "Int16"
  If InStr(Zeile, "; Int32") > 0 Then Akt_Format = "Int32"
  Byte_Cnt = ""
  Ergebnis =““                                           ' Wenn Ergebnis ungültig
  Byte_Pos = InStr(Zeile, ".")
  if Byte_Pos > 0 Then
    Byte_Pos = Byte_Pos + 6
    While (Mid(Zeile, Byte_Pos, 1) >= "0") And (Mid(Zeile, Byte_Pos, 1) <= "9")
      Byte_Cnt = Byte_Cnt + Mid(Zeile, Byte_Pos, 1)          ' String addition
      Byte_Pos = Byte_Pos + 1                                 ' Zeiger nachführen
    End While
    If Byte_Cnt = "1" Then
      LB_Variablen.Items.Add(Akt_Name)                       ' Variable in Liste einfügen
      LB_Formate.Items.Add(Akt_Format)                       ' Format in Liste einfügen
    else
      For I = 0 To Val(Byte_Cnt) - 1
        LB_Variablen.Items.Add(Akt_Name + "[" + Str(I) + "]")    ' Variablenarray einfügen
        LB_Formate.Items.Add(Akt_Format)                     ' und die zugehörigen Formate
      Next I
    end if
    Ergebnis= Akt_Name+“;“+ Akt_Format ' Name und Format an Filter im String zurückliefern
  End If
  Return (Ergebnis)
End Function
```

Die Arbeitsweise ist nun auch leicht erkennbar und mit einer kleinen Grafik darzustellen.



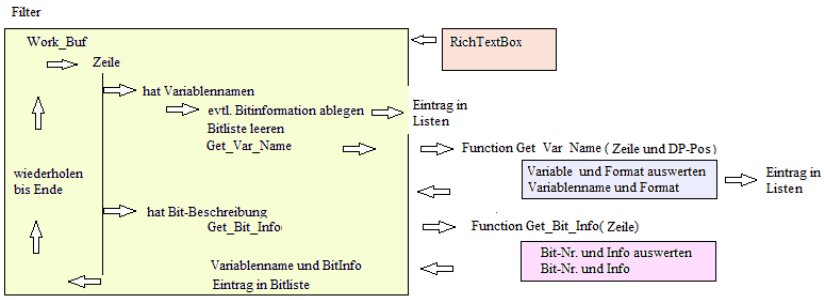

*Auswertung Variable*

Nun wenden wir uns der Auswertung von Bit-Beschreibungen zu. Hier benötigen wir die Zeile für die Übergabe in die Funktion. Zurückgeliefert werden auch hier zwei Informationen in einem String, wie es bereits schon in der Auswertung einer Variablendeklaration ausgeführt wurde. Allerdings wird hier kein Trennzeichen für eine nachfolgende Zerlegung erforderlich, da die Bitnummer an einer festen Stelle im Rückgabetext steht. Mit dieser Bitnummer lässt sich die Position in der Liste bestimmen, an die diese Information geschrieben werden soll.

```
Public Function Get_Bit_Info(ByVal Akt_Zeile as String) as String
  Dim Bit_pos as Integer
  Dim Bit_Info as String
  Dim Info as String
  Dim Ergebnis as String
  Bit_Info =""                                          ' wenn Ergebnis ungültig
  Bit_Pos = InStr(Akt_Zeile, "; Bit ")                  ' Position Bitnummer auffinden
  If Bit_Pos > 0 Then
    Bit_Pos = Bit_Pos + 6                             ' Bitnummer steht 5 Stellen weiter hinten
    Bit_Info = ".Bit " + Akt_Zeile(Bit_Pos) + " : "   ' Rückgabe vorbereiten mit Bitnummer
    Bit_Pos = InStr(Akt_Zeile, "=")                   ' Information suchen
    Info = Mid(Akt_Zeile, Bit_Pos + 1, Len(Akt_Zeile) - Bit_Pos) ' und herauskopieren
    Info = Trim(Info)                                 ' Leerzeichen entfernen
    Bit_Info = Bit_Info + Info                        ' Information Rückgabe hinzufügen
  End If
  Return (Bit_Info)
End Function
```

Wenn in einer Zeile weder eine Variable noch eine Bit-Beschreibung enthalten ist, dann wird der Rückgabewert keine Information enthalten. Die aufrufende Programmstelle kann dieses auswerten und entsprechend reagieren. Nur wenn ein Rückgabetext mit einer Information gefüllt ist, kann die Bitnummer bestimmt und damit der Platz für den Eintrag im Array bestimmt werden. Es wird lediglich noch der vorher gefundene Variablenname hinzugefügt, so dass diese Bit-Beschreibung auch einer Variablen zugeordnet werden kann.

An dieser Stelle ist es nun ein leichtes, den Filter neu zu skizzieren und den Ablauf sichtbar zu machen.




*Strukturübersicht Filter*

Fügen wir nun diese Aufrufe in unser Programmgerüst des Filters und entfernen alle unnötigen Variablendeklarationen..

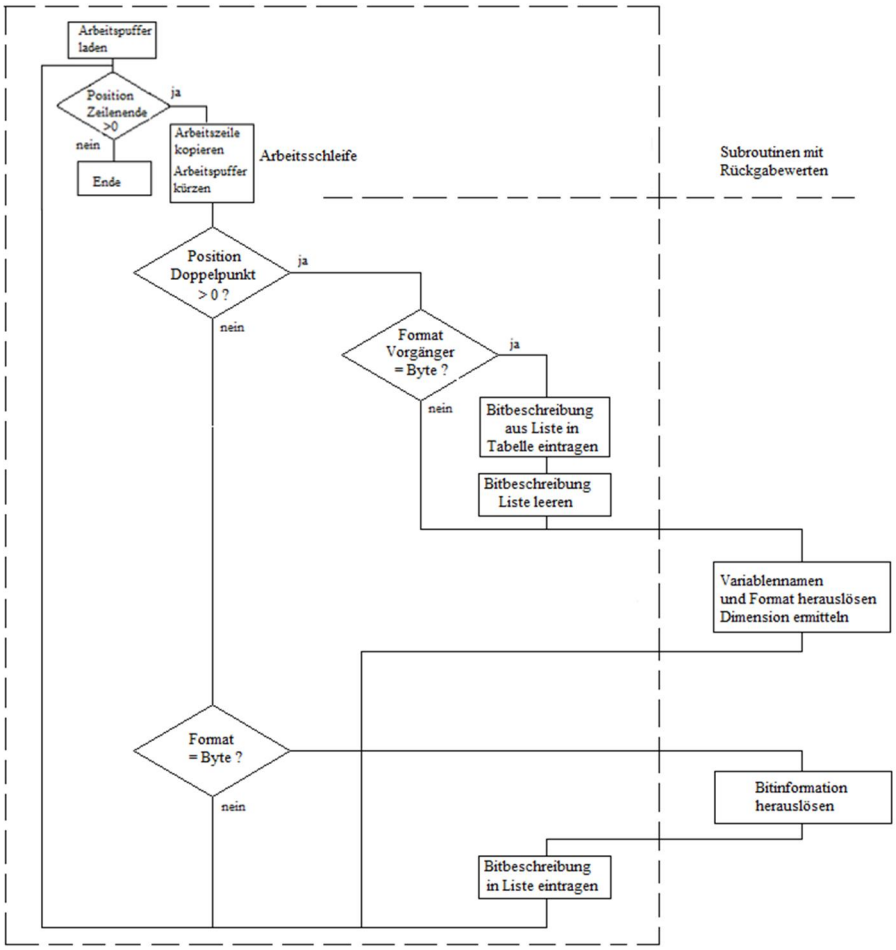
```
Private Sub BT_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles BT_Filter.Click
  Dim Work_Buf As String                    ' Arbeitspuffer zum Abarbeiten
  Dim Zeile As String                       ' wird für Zeilenweise Abarbeitung gebraucht
  Dim Var_Name As String                    'wird für Bit-Beschreibung benötigt
  Dim Format_Str As String   ' Wird zur Auswertung der Bit-Beschreibung gebraucht
  Dim Bit_Info As String                    ' Ergebnis aus Bit-Beschreibung
  Dim Bit_Liste(7) As String                ' behalten wir als Zwischenspeicher
  Dim Bit_Nr As Integer                     ' behalten wir für Index zum Zwischenspeicher
  Dim I As Integer                          ' benötigen wir zur Vorbesetzung der Bit_Liste
  Dim LF_Pos As Integer                     ' Zur Zeilenbearbeitung erforderlich
  Dim Dp_Pos As Integer                     ' benötigt zur Erkennung eines Variablennamens
  Dim Startflag As Boolean  ' Format Byte beim ersten Durchlauf nicht berücksichtigen
  Work_Buf = Rt_Buffer.Text                 ' Kopie der Assembler Variablendeklaration
  Zeile = ""                                ' mit Leerstring vorbesetzen
  Var_Name = ""                             ' mit Leerstring vorbesetzen
  Startflag = False                         ' Startflag löschen
  Format_Str = „Int8“                       ' Defaultwert Format setzen
  Lb_Variablen.Items.Clear()                ' Variablenliste leeren
  While Len(Work_Buf) > 0                   ' Beginn der Auswerteschleife
```

```
    LF_Pos = InStr(Work_Buf, Chr(10))      ' Zeilenende abfragen
    If LF_Pos > 0 Then
      Zeile = Mid(Work_Buf, 1, LF_Pos - 1) ' Zeile herauskopieren und Work_Buf kürzen
      Work_Buf = Mid(Work_Buf, LF_Pos + 1, Len(Work_Buf) - LF_Pos)
    Else
      Zeile = Work_Buf                     ' letzte Zeile und Work_Buf leeren
      Work_Buf = ""
    End If
    If Zeile <> " " Then                   ' Zeile vorhanden, dann Auswertung beginnen
      Dp_Pos = InStr(Zeile, ":")           ' zuerst Zeile auf Variablennamen prüfen
      If Dp_Pos > 0 Then                   ' neuer Variablenname
        If (Format_Str = "Byte") And Startflag Then     ' Vorgänger hat Bitliste
          For I = 0 To 7 ' Bitbeschreibung Vorgänger in Variablen- und Formatlisten ergänzen
            Lb_Variablen.Items.Add(Bit_Liste(I))
            Lb_Formate.Items.Add("Bit")
          Next I
        End If
        Startflag = True
        Zeile = Get_Var_Name(Zeile, Dp_Pos)          ' Zeile hat Name und Format
        If Zeile <> "" Then                          ' Zeile gültig?
          Var_Name = Mid(Zeile, 1, Dp_Pos - 1)       ' Name eintragen
          Dp_Pos = InStr(Zeile, ";")                 ' Trennzeichen ist „;“
          Format_Str = Mid(Zeile, Dp_Pos + 1, Len(Zeile) - Dp_Pos)
          If Format_Str = "Byte" Then
            For I = 0 To 7                           ' dann Bitliste mit Defaultwert besetzen
              Bit_Liste(I) = Var_Name + ".Bit" + Str(I) + " : nicht verwendet"
            Next I
          End If
        End If
      Else
        If Format_Str = "Byte" Then                  ' prüfen, ob Bit-Beschreibung erforderlich
          Bit_Info = Get_Bit_Info(Zeile)             ' hier kommt der Aufruf der Auswertung
          If Bit_Info <> "" Then                     ' Bitinfo liegt vor
            Bit_Nr = Val(Mid(Bit_Info, 6, 1))        ' Bitnummer holen
            Bit_Liste(Bit_Nr) = Var_Name + Bit_Info  ' in Liste eintragen
          End If
        End If
      End If
    End If
  End While
  If (Format_Str = "Byte") And Startflag Then        ' Vorgänger hat Bitliste
    For I = 0 To 7 ' Bitbeschreibung Vorgänger in Variablen- und Formatlisten ergänzen
      Lb_Variablen.Items.Add(Bit_Liste(I))
      Lb_Formate.Items.Add("Bit")
    Next I
  End If
end Sub
```

Nun passt auch das Programm vom Filter auf eine DinA4 Seite

### *1.2.4.7 Abschlussbetrachtung Filterprogrammierung*

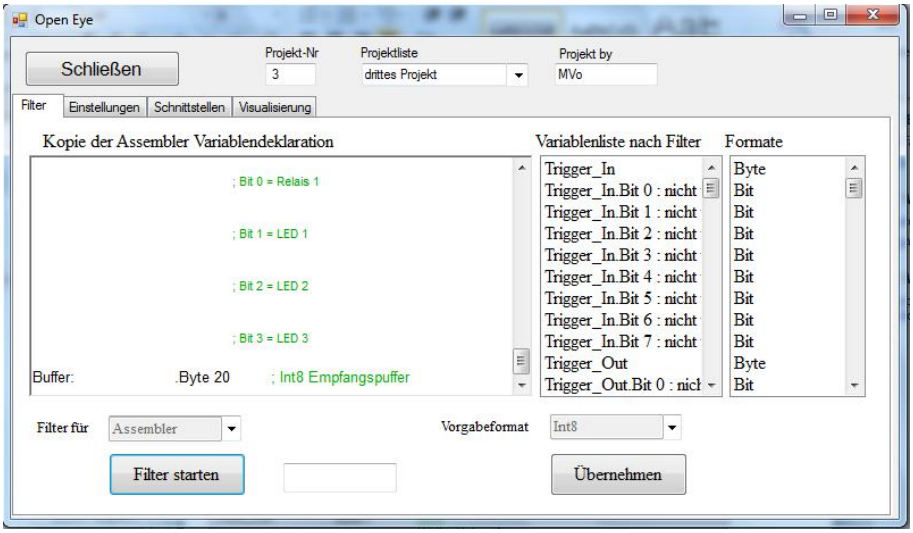
Bevor die nächste ***TabPage*** aufgebaut wird, möchte ich die Programmschritte noch einmal in einem Ablaufplan skizzieren.




*PAP Filterfunktion*

Eine solche Übersicht verlangt nicht unbedingt jeden Schritt. Würden hier alle einzelnen Entscheidungen einfließen, reichte der Platz nicht aus und das Gebilde wird unübersichtlich. Trotzdem hilft eine solche Übersicht, den Ablauf zu verstehen.

Ein Test sollte das folgende Ergebnis liefern.




*Filter abschließend testen*

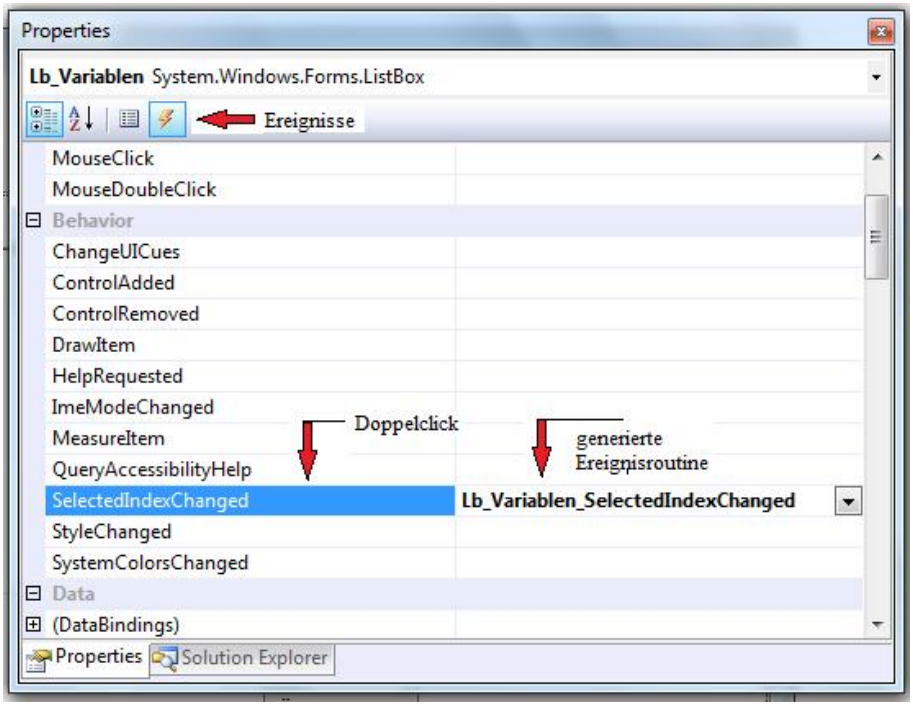
Sind für diese Seite alle vorgesehenen Aufgaben nun erfüllt?

Wir können einen Text aus einem Assemblerlisting mit Variablendeklarationen zerlegen, die vorgegebenen Formate erkennen und sogar die Beschreibung einzelner Bits herausfiltern.

Das sollte für diese Seite doch ausreichen. Etwas unschön ist die Tatsache, dass es eine Variablenliste und eine Formatliste gibt, die scheinbar keinerlei Bezug zueinander haben. Das ist insofern richtig, das die Variablenliste gescrollt und auch eine Variable selektiert werden kann, aber die Formatliste dazu in keinerlei Weise reagiert. Mit wenigen Befehlen werden wir es so ändern, dass ein Ereignis, welches in der Variablenliste auf eine Variable verweist, das zugehörige Format in der Formatliste anzeigt.

### *1.2.4.8 Abgleich zweier Listeneinträge*

Dazu werfen wir einen Blick auf verfügbare Ereignisse der Tb_Variablen.




*Format- und Variablenliste verbinden*

Das Problem ist nun herauszufinden, welcher Eintrag in der Variablenliste selektiert und welcher Index das ist. Den Index erhalten wir über die Objektfunktion

```
Lb_Variablen.Items.IndexOf(<Eintrag>)
```

und den Eintrag bekommen wir mit

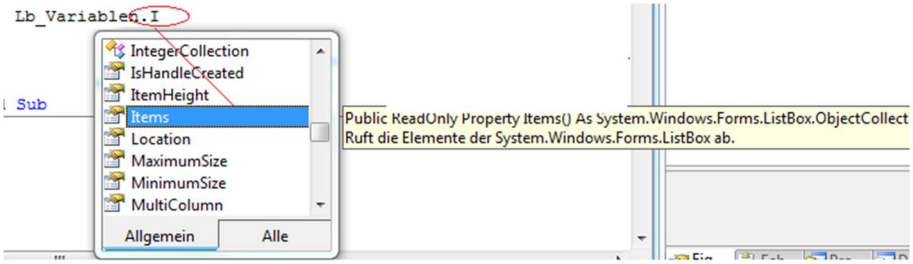
```
Variablen.SelectedItem.
```

Das in eine Zeile gepackt und <Eintrag> ersetzt ergibt

```
List_Nr= Lb_Variablen.Items.IndexOf(Lb_Variablen.SelectedItem)
```

Nun möchten wir in der Liste Formate den zugehörigen Eintrag markieren. Dazu brauchen wir den **Index** in der Liste und die Angabe, ob der Eintrag markiert werden soll. Der entsprechende Befehl dazu ist eine Routine *SetSelect* vom Objekt *Lb_Formate*. Der Aufruf dieser Routine mit Angabe von **Index** und Status Wahr setzt in der Formatliste den entsprechenden Eintrag auf *selektiert*. Alles klar?

Vielleicht hilft hier eine etwas genauere Erklärung. Woher soll man den wissen, dass es einen Befehl *IndexOf* gibt und dass dieser hier benötigt wird. Denken wir einmal kurz über Listeneinträge nach. Eine Liste ist eine Aufzählung von Einträgen und in einem Programm hat jeder Eintrag in einer Liste eine Nummer, den **Index**. So lässt er sich wieder erfassen, wenn die Information benötigt wird. Also sind Eigenschaften, die mit **Index** in Verbindung zu bringen sind, möglicherweise für die Aufgabe geeignet. Nun wissen wir auch, dass nach der Eingabe des Objektnamens und einem Punkt die Online-Hilfe alle möglichen Funktionen, Eigenschaften und Ereignisroutinen anbietet. Suchen wir daher zuerst nach den Listeneinträgen, den *Items*. Mit einer Eingabe von I erhalten wir dann unter Anderem schon einmal den Vorschlag Items.

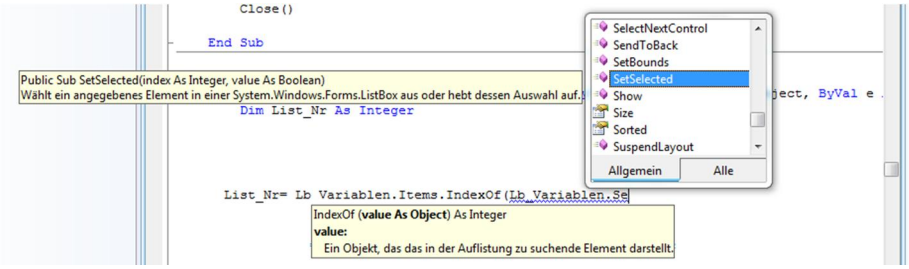



*Onlinehilfe Eigenschaft Items*

Nun ist der **Index**, also die Nummer des Eintrags in der Liste interessant, weil ja zu jedem Listeneintrag auch ein Formateintrag in der Formatliste steht. Also, die Variablenangabe mit der Listennummer 5 hat einen Formateintrag in der Formatliste auch mit der Nummer 5. Der *Index* der Einträge ist also identisch. Geben wir also einen Punkt und dann wieder ein I ein

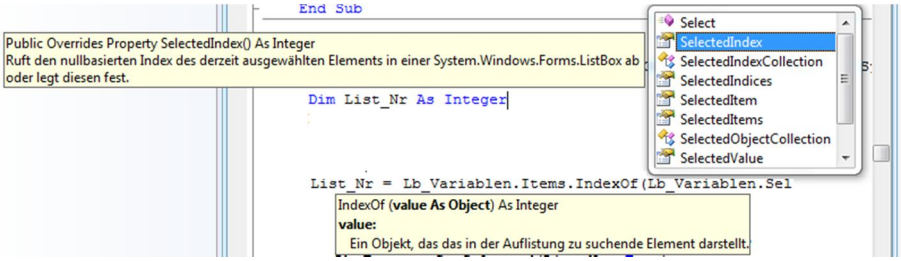
*Onlinehilfe Eigenschaft IndexOf*

Es bieten sich zwei mögliche Einträge an *IndexOf* und *Insert*. *Insert*, zu Deutsch einfügen, ist hier nicht gefragt, aber *IndexOf* hört sich nach **Nummer von** an. Ein Blick in das gelbe Erklärungsfeld bestätigt die Annahme, dass die Nummer eines bestimmten Eintrags, der anschließend in einer Klammer übergeben werden muss, zurückgeliefert wird. Aha, der Aufruf liefert einen **Index**, also sollte die Zahl einer Variablen zugewiesen werden. Deklarieren wir also eine Integer-Variable. Lokal natürlich, da nur in dieser Ereignisroutine diese Variable benötigt wird. Somit fällt die Auswahl nicht schwer und die angebotene Funktion *IndexOf* wird genommen. Anschließend noch der Inhalt der angeklickten Listenzeile in die Klammer. Auch dabei ist die Online-Hilfe sehr nützlich. Kurz nachdenken. Welche denkbare Funktion könnte mir einen Listeninhalt liefern. Einmal kann ich über Indexangabe auf einen Eintrag zurückgreifen, aber den kenn ich ja nicht. Einzig die Zeile markiert sich, wenn ich sie anklicke. Sie wird selektiert. Führen wir einmal die Eingabe weiter fort und öffnen nach *IndexOf* eine Klammer.




*Online-Hilfe bei IndexOf Werteauswahl*

Der Erklärungstext verweist mit Value as Objekt auf eine Objektangabe. Nun Strings sind auch Objekte, aber hier ist ja nur der selektierte Eintrag der Liste gegeben. Kein Problem, auch ein Listeneintrag ist ein Objekt, also geben wir, um das Auswahlangebot einzugrenzen schon mal den Anfang des Namens der ***Listbox*** an. Mit LB_ haben wir dann diese Eingrenzung. *LB_Variablen* ist die richtige Auswahl, denn wir wollen ja die Nummer des Variableneintrages, damit wir das zugehörige Format in der Formatliste markieren können. Auch hier wird nun hinter *LB_Variablen* ein Punkt gesetzt. Nun ist der Text erforderlich, der in der selektierten Zeile von *LB_Variablen* steht. Wieder gibt es eine Funktion, die den Inhalt liefert. Aber welche kommt da in Frage. Eigentlich ganz einfach: die Zeile ist **selektiert**. Also beginnen wir nach dem Punktwieder mit der Eingabe von **Sel** wie im folgenden Screenshot




*Onlinehilfe Eigenschaft SelectedIndex*

*SelectedIndex* entspricht genau der benötigten Funktion. Hier habe ich auch die Zuweisung zu einer Variablen **List_Nr** mit abgebildet.

Noch einmal das Vorgehen schrittweise:

Für die Zuweisung der **List_Nr** brauchen wir den **Index** von z.B. **Trigger_Out**. Der Aufruf der Funktion mit der Zuweisung an List_Nr erfolgt mit: List_Nr=Lb_Variablen.IndexOf(„Trigger_Out“)

Da auch andere Listeneinträge gesucht werden können, darf der Text nicht fest vergeben, sondern aus dem markierten Eintrag gewonnen werden. Diese Funktion mit einer Hilfsvariablen dargestellt lautet dann :

```
Hilfstext=Lb_Variable.SelectedItem
```

Nun ändern wir schrittweise die Zeile mit der Listennummer

```
List_Nr = Lb_Variablen.IndexOf(Hilfstext)
```

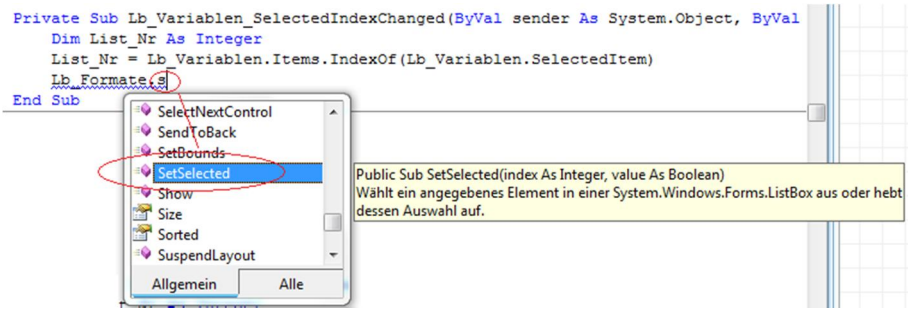
Und statt Hilfstext setzen wir die Funktion in die Klammer

```
List_Nr = Lb_Variable.IndexOf(LB_Variable.SelectedItem)
```

Was fangen wir nun mit dem erarbeiteten Wert in List_Nr an? Na ja, wir möchten, das in der Formatliste das Format passend zur ausgewählten Variablen in der Variablenliste angezeigt wird. Es soll markiert oder auch

selektiert werden. Nun auch diesmal eine ähnliche Vorgehensweise mit der Formatliste. Dort soll nun der passende Eintrag selektiert werden oder anders gesagt, die Selektierung soll entsprechend **gesetzt** werden. Es ist wichtig zu wissen, möchte ich etwas tun oder will ich etwas erfahren. Aktion oder Information, das ist immer zu bedenken. Hier ist es eine Aktion, das setzen einer Markierung.

Beginnen wir mit LB_Formate. Eine Weiterführung mit *Items* führt zu keinem Ergebnis, also lassen wir *Items* weg und setzen hinter Formate und dem Punkt ein S, um herauszubekommen, welche Eigenschaften oder Funktionen damit verbunden sind. Das **S** steht hier für **SET** oder zu Deutsch **setzen**




*Onlinehilfe Eigenschaft SetSelected*

Wie im Screenshot sichtbar, werden zwei mögliche Set-Funktionen angeboten und nur eine ist plausibel für unseren Zweck verwendbar. Der Hinweistext beschreibt, dass es zwei Parameter gibt, einmal den **Index** und einmal ob der mit dem **Index** benannte Eintrag markiert werden soll. Boolean-Eigenschaften werden immer mit **True** für Wahr und **False** für Falsch angegeben. Den **Index** haben wir vorher mit List_Nr ermittelt. Alles was für diese Funktion gebraucht wird ist also verfügbar und so vervollständigen wir diesen Befehl

```
LB_Formate.SetSelected(List_Nr, True)
```

Die Ereignisroutine *SelectedValueChange* der Variablenliste ist nun komplett:

```
Private Sub Lb_Variablen_SelectedValueChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Lb_Variablen.SelectedValueChanged
  Dim List_Nr As Integer                    ' Indexvariable
  List_Nr = Lb_Variablen.Items.IndexOf(Lb_Variablen.SelectedItem)
```

```
                                                    ' Index in der Variablenliste
  Lb_Formate.SetSelected(List_Nr, True)      ' Eintrag in der Formatliste selektieren
End Sub
```

Wenn nun in der Variablenliste gescrollt wird, ändert sich in der Formatliste immer noch nichts. Wird aber ein Eintrag in der Variablenliste angeklickt, wird automatisch das zugehörige Format selektiert und angezeigt. Damit kann schon einmal kontrolliert werden, ob die Variablen und Formatzuordnung zueinander passt.

### *1.2.4.9 Ereignisbearbeitung Combobox*

Nun sind zwei ***Comboboxen*** auf der Filter-Seite. In einer soll die Auswahl des Filters für mehrere Sprachen stattfinden und die andere dient dazu, den Variablen, die nicht über ihre Deklaration im Assembler-Kommentar einem Format zugeordnet werden können, eine Voreinstellung zu geben. Die eigentliche Einstellung wird dann noch im Einstellbereich verfeinert.

Beginnen wir zuerst mit der ***Combobox*** für die Sprache. Sie hat den Namen *Cb_Sprache* und die Eigenschaft *Text* bekommt den Eintrag *„Assembler“*. Die darüber liegende ***Textbox*** mit dem Namen *Tb_Sprache* erhält ebenfalls in der Eigenschaft *Text* den Eintrag *„Assembler“*. Den Grund für die darübergelegte Textbox hatte ich bereits erklärt. So bleibt die ***Combobox*** bedienbar aber eine Eingabe ist durch die gesperrte ***Textbox*** unmöglich. Nun muss nur noch dafür gesorgt werden, dass der Inhalt vom Textfeld der ***Combobox*** auch in der ***TextBox*** angezeigt wird. Das erledigt die Ereignisroutine *TextChanged* der ***Combobox*** für die Sprache sowohl auch für das Vorgabeformat. Auch diese ***Combobox*** hat ja eine gesperrte ***Textbox*** über dem eigenen Textfeld liegen. Der Eintrag in das Textfeld dieser beiden Objekte ist **Int8** als erste Vorgabe.

Die zwei folgenden Ereignisroutinen der ***Comboboxen*** werden nun die Textinhalte an die ***TextBoxen*** übertragen. Erst die Sprachauswahl:

```
Private Sub CB_Sprache_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles CB_Sprache.TextChanged
    TB_Sprache.Text = CB_Sprache.Text                    'Textübergabe an Textbox
End Sub
```

Dann die Formatauswahl

```
Private Sub Cb_DefaultFormat_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Cb_DefaultFormat.TextChanged
    Tb_DefaultFormat.Text = Cb_DefaultFormat.Text          'Textübergabe an Textbox
End Sub
```

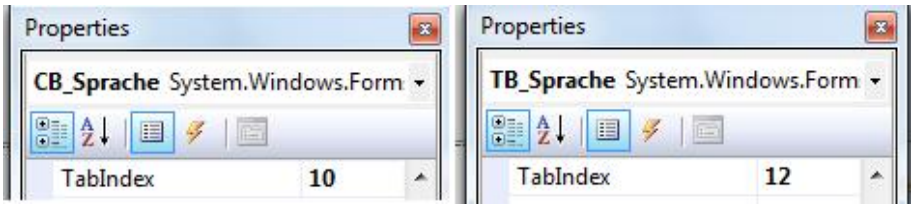
Um diese Information im Filter zu bewerten müssen wir noch in der Routine *BT_Filter_Click* die Zeile

```
Format_Str = "Int8"   ' Defaultwert Format setzen
```

ändern in

```
Format_Str = Tb_DefaultFormat.Text   ' Defaultwert Format setzen
```

Prüfen wir nun mit dem laufenden Programm unsere neuen Funktionen. Dabei müssen die ***TextBoxen*** mit der Eigenschaft *Enabled* auf **False** gesetzt werden. Es darf jetzt auch nicht mehr möglich sein, Änderungen in den ***TextBoxen*** vorzunehmen. Sollte das der Fall sein, dann liegen die ***TextBoxen*** unter der ***Combobox***. In diesem Fall hilft es, die Eigenschaft *TabIndex* anzupassen. Das darüber liegende Objekt hat einen höheren **Index**.

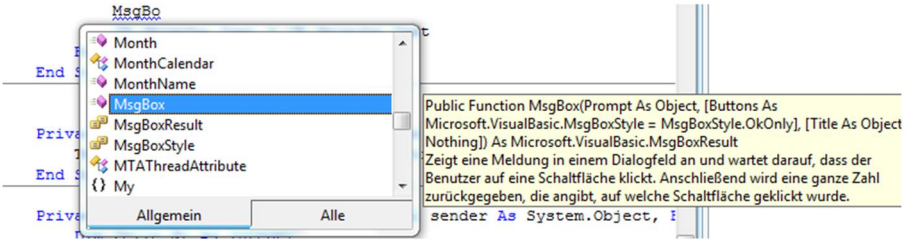



*Eigenschaft TabIndex*

Die Auswertung der Auswahl durch die Vorgabe mit den ***Comboboxen*** erfolgt im Programm. Lediglich die ***Combobox*** Sprache soll sofort auf eine gültige Auswahl überprüft und mit einem Hinweis, das zur Zeit nur Assembler verfügbar ist, jede andere Vorwahl abgelehnt.

### *1.2.4.10 Information im Programmablauf*

Um dies zu erreichen brauchen wir einen Hinweis, damit der Grund für eine Verweigerung einer Bearbeitung für den Bediener deutlich wird. Das Programmobjekt ist eine Messagebox. Leider finden wir diese nicht in der Toolbox, sondern hier ist das Wissen um die Programmelemente gefragt. Wer nicht über das Wissen verfügt hat die Möglichkeit, für die erforderliche Aufgabenlösung in der Hilfedatei nachzuschlagen. Unter Messagebox allerdings werden wir da nichts finden, aber eine durchaus denkbare Abkürzung könnte **Msg** für Message sein. Damit findet man auch dann den Begriff ***MsgBox***. Na ja, auch Visual Basic muss gelernt werden. Wie baut sich nun eine ***MsgBox*** auf. Hier möchte ich mich erst einmal nur auf die abzugebende Information beziehen. Mit einem Blick auf den Screenshot der Online-Hilfe findet man eine etwas verwirrende Beschreibung.




*Onlinehilfe MsgBox*

```
Public Function MsgBox(Prompt as Object,[…], [Title as ObjectNothing]) as Microsoft.VisualBasic.MsgBoxResult
```

Ok, mit *Prompt* ist der Informationstext gemeint und mit *MsgBoxStyle* bietet sich eine Auswahl an. Ich hab mich für Information entschieden. Die Überschrift, den Titel, hab ich mit *Hinweis* definiert. Dem entsprechend lautet die Befehlszeile

```
MsgBox("Es ist nur der Filter für Assembler Variablen verfügbar. Auswahl wird daher abgewiesen.", MsgBoxStyle.Information, "Hinweis")
```

Den Aufruf dieser Information kann man nun in der Ereignisroutine der ***Combobox*** einbauen

```
Private Sub CB_Sprache_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Sprache.TextChanged
  If CB_Sprache.Text = “Assembler” then
    TB_Sprache.Text= CB_Sprache.Text                    'Textübergabe an Textbox
  Else
    MsgBox("Es ist nur der Filter für Assembler Variablen verfügbar. Auswahl wird daher
   abgewiesen.", MsgBoxStyle.Information, "Hinweis")
  End_If
End Sub
```

und eine ungültige Sprachauswahl mit diesem Hinweis ablehnen.

Der Screenshot auf das aktive Programm stellt sich dann entsprechend so dar.

Trotzdem bereiten wir die Bearbeitung vor. Damit eine einfache Implementierung vorgenommen werden kann, rufen wir in unserem Programm in der Ereignisroutine des ***Button*** **Filter starten** je nach Textinhalt der ***Textbox*** *TB_Sprache* Subroutinen auf. Dazu erstellen wir drei Grundgerüste:

```
Public Sub Filter_Assembler

End Sub
```

Dahinein wird der gesamte Code einschließlich der Variablendeklaration aus der Ereignisroutine BT_Filter_Click hineinkopiert und anschließend in der Ereignisroutine gelöscht.

```
Public Sub Filter_Assembler()
  Dim Work_Buf As String                  ' Arbeitspuffer zum Abarbeiten
  Dim Zeile As String                     ' wird für Zeilenweise Abarbeitung gebraucht
  Dim Var_Name As String                  'wird für Bit-Beschreibung benötigt
  Dim Format_Str As String   ' Wird zur Auswertung der Bit-Beschreibung gebraucht
  Dim Bit_Info As String                  ' Ergebnis aus Bit-Beschreibung
  Dim Bit_Liste(7) As String              ' behalten wir als Zwischenspeicher
  Dim Bit_Nr As Integer                   ' behalten wir für Index zum Zwischenspeicher
  Dim I As Integer                        ' benötigen wir zur Vorbesetzung der Bit_Liste
  Dim LF_Pos As Integer                   ' Zur Zeilenbearbeitung erforderlich
  Dim Dp_Pos As Integer                   ' benötigt zur Erkennung eines Variablennamens
  Dim Startflag As Boolean  ' Format Byte beim ersten Durchlauf nicht berücksichtigen
  Work_Buf = Rt_Buffer.Text               ' Kopie der Assembler Variablendeklaration
  Zeile = ""                                  ' mit Leerstring vorbesetzen
  Var_Name = ""                               ' mit Leerstring vorbesetzen
  Startflag = False                           ' Startflag löschen
  Format_Str = Tb_DefaultFormat.Text      ' Defaultwert Format setzen
  Lb_Variablen.Items.Clear()              ' Variablenliste leeren
```

```
  While Len(Work_Buf) > 0                        ' Beginn der Auswerteschleife
    LF_Pos = InStr(Work_Buf, Chr(10))      ' Zeilenende abfragen
    If LF_Pos > 0 Then
      Zeile = Mid(Work_Buf, 1, LF_Pos - 1) ' Zeile herauskopieren und Work_Buf kürzen
      Work_Buf = Mid(Work_Buf, LF_Pos + 1, Len(Work_Buf) - LF_Pos)
    Else
      Zeile = Work_Buf                           ' letzte Zeile und Work_Buf leeren
      Work_Buf = ""
    End If
    If Zeile <> " " Then                    ' Zeile vorhanden, dann Auswertung beginnen
      Dp_Pos = InStr(Zeile, ":")         ' zuerst Zeile auf Variablennamen prüfen
      If Dp_Pos > 0 Then                  ' neuer Variablenname
        If (Format_Str = "Byte") And Startflag Then     ' Vorgänger hat Bitliste
          For I = 0 To 7 ' Bitbeschreibung Vorgänger in Variablen- und Formatlisten ergänzen
            Lb_Variablen.Items.Add(Bit_Liste(I))
            Lb_Formate.Items.Add("Bit")
          Next I
        End If
        Startflag = True
        Zeile = Get_Var_Name(Zeile, Dp_Pos)           ' Zeile hat Name und Format
        If Zeile <> "" Then                                  ' Zeile gültig?
          Var_Name = Mid(Zeile, 1, Dp_Pos - 1)          ' Name eintragen
          Dp_Pos = InStr(Zeile, ";")                          ' Trennzeichen ist „;“
          Format_Str = Mid(Zeile, Dp_Pos + 1, Len(Zeile) - Dp_Pos)
          If Format_Str = "Byte" Then
            For I = 0 To 7                                ' dann Bitliste mit Defaultwert besetzen
              Bit_Liste(I) = Var_Name + ".Bit" + Str(I) + " : nicht verwendet"
            Next I
          End If
        End If
      Else
        If Format_Str = "Byte" Then                   ' prüfen, ob Bit-Beschreibung erforderlich
          Bit_Info = Get_Bit_Info(Zeile)               ' hier kommt der Aufruf der Auswertung
          If Bit_Info <> "" Then                       ' Bitinfo liegt vor
            Bit_Nr = Val(Mid(Bit_Info, 6, 1))          ' Bitnummer holen
            Bit_Liste(Bit_Nr) = Var_Name + Bit_Info       ' in Liste eintragen
          End If
        End If
      End If
    End If
  End While
  If (Format_Str = "Byte") And Startflag Then          ' Vorgänger hat Bitliste
    For I = 0 To 7 ' Bitbeschreibung Vorgänger in Variablen- und Formatlisten ergänzen
      Lb_Variablen.Items.Add(Bit_Liste(I))
      Lb_Formate.Items.Add("Bit")
    Next I
  End If
end Sub
```

Stattdessen wird dort dann der Aufruf  Filter_Assembler() eingetragen.

```
Private Sub Bt_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Bt_Filter.Click
  If TB_Sprache.Text = "Assembler" Then
    Filter_Assembler()
  End If
  If TB_Sprache.Text = "Bascom" Then     ' Auswahl nicht möglich
    Filter_Basic()
  End If
  If TB_Sprache.Text = "C" Then          ' Auswahl nicht möglich
    Filter_C()
  End If
End Sub
```

Auch wenn die Auswahl von C und Basic gar nicht möglich ist, fertigen wir die Grundgerüste der Subroutinen an.

```
Public Sub Filter_Basic

End Sub
```

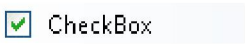
```
Public Sub Filter_C

End Sub
```

Das Programm wird gestartet und die Änderung auf korrekte Funktion geprüft. Das Programm muss sich genau so verhalten wie vor der Änderung.

Mit dieser letzten Maßnahme ist der Filter erst einmal abgearbeitet. Das Click-Ereignis des ***Button*** Übernahme werden wir erst später bearbeiten. Wenn alle Tests ein einwandfreies Ergebnis liefern, können wir uns dem Aufbau der zweiten Seite zu wenden.

### *1.2.5 Aufbau der zweiten Seite*

Neue Objekte kommen hinzu:





*Icon CheckBox* *Icon DataGridView*

Auch hier stehen wir vor einem völlig leeren Blatt und es ist wichtig, eine Strategie für den Aufbau der Nachbearbeitung zu entwickeln. Die erste Überlegung sollte nun den geeigneten Objekten gelten. Was werden wir benötigen? Wie bereits beim Filter ist erst einmal ein *Gedankensturm* erforderlich. Alles, was uns dazu einfällt, sollte auf einem Blatt Papier aufgeschrieben werden. Bisher haben unsere Variablen nur eine Formatierung über die Einträge in den Kommentarzeilen oder aber eine fest vorgegebene Defaulteinstellung erhalten. Möglicherweise gibt es aber die Notwendigkeit, einzelnen Variablen ein anderes Format zu zuordnen. Dafür brauchen wir eine Darstellung der Variablen mit ihren bereits vorgegebenen Formaten. Bisher kennen wir ***TextBoxen***, ***Listboxen***, ***RichTextBoxen*** und ***Buttons***. Ach ja, das ***TabControl*** und die ***TabPages*** nicht vergessen. Diese Aufgabe ließe sich ganz gut mit einer Tabelle umzusetzen, in denen die Daten zeilenweise abgelegt sind. Das Objekt für Tabellen ist ein ***DataGridView***, auf gut deutsch einfach Datentabelle. Bevor wir aber mit der Gestaltung beginnen, ziehen wir ein ***Panel*** auf die Seite und nennen es **Pn_Einstellung**. Es erlaubt uns, alle Objekte dieser Seite für einen Zugriff zu sperren. Der Grund ist zwar noch nicht ersichtlich, dennoch rechnen wir damit, die Bedienung sperren zu müssen. Man kann das zwar auch noch zu einem späteren Zeitpunkt nachholen, ist aber nicht notwendig, wenn man im Vorfeld diesen Schritt macht. Nun können wir mit der Gestaltung beginnen.

Schreiben wir also erst einmal auf „Tabelle für Variablen“.

Nun wissen wir noch, dass es Beschreibungen von Einzelbits gibt. Diese möchten wir ebenfalls editieren und die Grundlage bietet auch hier eine Tabelle, die bei einem Format *Byte* einer Variablen aufgeschlagen wird und die Zweckbeschreibung der Bits enthält. Ein Byte hat immer 8 Bit und daher generieren wir uns entsprechend eine Tabelle für die Bitbeschreibung, die exakt 8 Einträge aufnehmen kann.

Was darf nun editierbar sein? Der Variablenname nicht, denn der steht in direktem Bezug zum Assemblerlisting. So bleibt lediglich das Format. Dazu kommt noch etwas, das bisher nicht so explizit erklärt wurde und

das ich nun nachhole. Man muss nicht immer alle Variablen ausgeben, sondern kann auch diejenigen ausblenden, an deren Inhalt kein Interesse besteht. Allerdings überträgt der Datentransfer immer den gesamten Variablenblock und deshalb dürfen wir diese nicht einfach löschen. Aber anzeigen brauchen wir sie nicht. Dies wird mit einer Spalte **Aktiv** festgelegt. Eine weitere Spalte benötigen wir, um bei einer Zusammenfassung von mehreren Bytes zu einer entsprechend großen Variablen den Zugriff auf die folgenden Bytes zu sperren. Diese Spalte definiert die Freigabe zur Korrektur. Bisher ist dies für eine Variableneinstellung noch einleuchtende Spaltenvergabe. Neu und bisher nicht erwähnt ist eine weitere Eigenschaft, die ich einer Variablen zuordne, den **Trigger**. Was aber hat es mit **Trigger** auf sich?

#### *1.2.5.1 Datenübertragung triggern*

Das Triggern der Datenübertragung ist ein kleines Highlight dieser Software. Wenn wir vom PC einen Triggerwert an den Controller senden, sind wir in der Lage, dieses Bit in einem wichtigen Programmbereich auszuwerten und wenn es gesetzt ist, sofort eine Übertragung an den PC durchzuführen. Das bedeutet, wir bekommen exakt den Variablenstatus beim Durchlauf einer Programmsequenz. Die Triggervariable kann 8 verschiedene Trigger setzen. Dementsprechend kann der Wert im Tabellenfeld Trigger von 0 bis 8 angegeben werden. Das entspricht den acht Bits in einem Byte. Wird ein Bit gesetzt und in einem Programmzweig dieses Bit abgefragt, kann sofort eine Datenübertragung durchgeführt werden. Das bedeutet, der Inhalt der Variablen wird exakt zu diesem Zeitpunkt an den PC übertragen und in Open_Eye visualisiert.

Aber wieso ein Wert von 0-8? Sind das nicht neun verschiedene Bits? Nun, dazu gleich die Erklärung.

Die 0 lässt sich abfragen und es wird kein Triggerbit gesetzt. Ist eine Zahl zwischen 1 und 8 eingetragen, so ist das Setzen eines Bits ganz einfach mit Bitnummer - 1. In der Mathematik ist jede Zahl hoch 0 eine 1. Die binäre Mathematik arbeitet mit der Basis 2. Dementsprechend ist 2^0 =1, 2^1=2, 2^3=4, 2^3=8,2^4=16, 2^5=32, 2^6=64 und 2^7 = 128. Betrachten wir einmal die Zahlen in einem Byte ist bei jeder Zahl nur ein Bit gesetzt. 2^0 setzt Bit 0, 2^1 setzt Bit 1, 2^3 setzt Bit 2 usw. Das ergibt einen ganz einfachen Algorhytmus, um in einem Byte ein Bit mit einer Bitnummer zu setzen..

Die Funktion ist ganz einfach:

```
Public Function Set_Bit(ByVal Bit_Nr as Integer) as Byte
  Dim Ergebnis as Byte
  If Bit_Nr= 0 then
    Ergebnis = 0
  Else
    Ergebnis = 2^(Bit_Nr-1)
  End if
  Return(Ergebnis)
End Sub
```

Der Controller hat nun die Aufgabe nur eine einzige Antwort auf dieses Triggerbit zurück zu senden um die Daten nicht gleich wieder zu überschreiben. Auch wollen wir sehen, welche Variablen durch dieses

Triggerereignis einen aktuell interessanten Wert enthalten. Und diese Zuordnung findet hier in der Tabelle statt. Der Controller liefert das Triggerereignis zurück und die betreffenden Felder der Variablen, die später die Werte anzeigen werden farblich abgesetzt. Hat also eine Variable den Trigger 3 und ist das zurückgelieferte Triggerereignis auch 3 wird dies entsprechend bei der Visualisierung sichtbar. Na ja, auf dem Weg dahin müssen wir noch einige Arbeit leisten.

### *1.2.5.2 Schlüsselfelder und Tabellenbezüge*

Zwei weitere Spalten helfen uns den Editiervorgang ordentlich zuzuordnen. Das ist eine Spalte für die Identnummer des Eintrags in der Datenbank und einmal die laufende Nummer in der Tabelle. Schlüsselfelder liefern eine Referenz auf einen Datensatz, die nur vom Programm generiert werden darf. Eine Änderung dieser Werte würde die Datenstruktur zerstören. Es ist auch nicht erforderlich, diese Information für den Anwender sichtbar zu gestalten. Bei dem geringen Platz ist das Ausblenden sicher sinnvoll. Trotzdem wird bei den ersten Experimenten ein sichtbarer Schlüssel viel zum Verständnis beitragen.

Warum Schlüssel? Da noch keine Datenbank hinterlegt ist, betrachten wir einmal eine simple Tabelle. Die Daten einer Zeile (Row) stellen den Datensatz dar. Sie gehören zusammen. Bei einer Adressliste ist es ja auch so, das Name, Vorname, Straße, Hausnummer, PLZ und Wohnort den Datensatz einer Person bilden.

| Lfd.Nr | Name | Vorname | Strasse | Haus-Nr. | PLZ | Wohnort |
|---|---|---|---|---|---|---|
| 1 | König | Horst | Am Markt | 5 | 33213 | Irgendwo |
| 2 | Meier | Fritz | Bahnhofstr | 13 | 87338 | Bergdorf |
| 3 | Bäcker | Ulli | Gartenweg | 34 | 2977 4 | Heidetal |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

*Einfache Adresstabelle*

Die Beispieltabelle zeigt die Spalten (Column) mit den Überschriften und die Zeilen (Row) mit den Datensätzen. So gehören die Daten in Zeile 2 zusammen und die 2 ist ein eindeutiger Schlüssel für den Datensatz des Herrn Meier. Er darf auch kein zweites Mal in dieser Tabelle vorkommen. Aber einen herrn Meier kann es ein weiteres Mal geben.

Eine lfd. Nr wird kein zweites Mal gelistet sein, aber als Schlüssel taugt sie dennoch nicht. Bezieht sich die Nummer auf die Position in einer Tabelle ist das kein Problem, wird aber mit dieser Nummer auf eine andere Tabelle referenziert, würde sich die Zuordnung Lfd. Nr und Datensatz ändern.

Betrachten wir einmal eine Bestellliste und die Referenz zum Kunden.

Solange kein Datensatz gelöscht wird, ist die Referenz ok, aber sobald ein Kunde herausgenommen wird, ändert sich die laufende Nummer und die Referenz ist nicht mehr gültig. Eine Identnummer ist immer unveränderbar, und ist fest mit dem Datensatz verbunden, so wie der Name zur Adresse gehört. Eine Verschiebung von Schlüssel und Datensatz ist unmöglich.

Wird ein Datensatz gelöscht, geht auch diese Schlüsselnummer verloren.

**Materialliste**

| Lfd.Nr | Bez. | Menge | Preis | Gewicht | Kunde |
|---|---|---|---|---|---|
| 1 | Hammer | 1 | 3,95 € | 300 gr | 3 |
| 2 | Schrauben | 100 | 2,50 € | 1 kg | 8 |
| 3 | Regale | 5 | 10,95 € | 15 kg | 2 |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

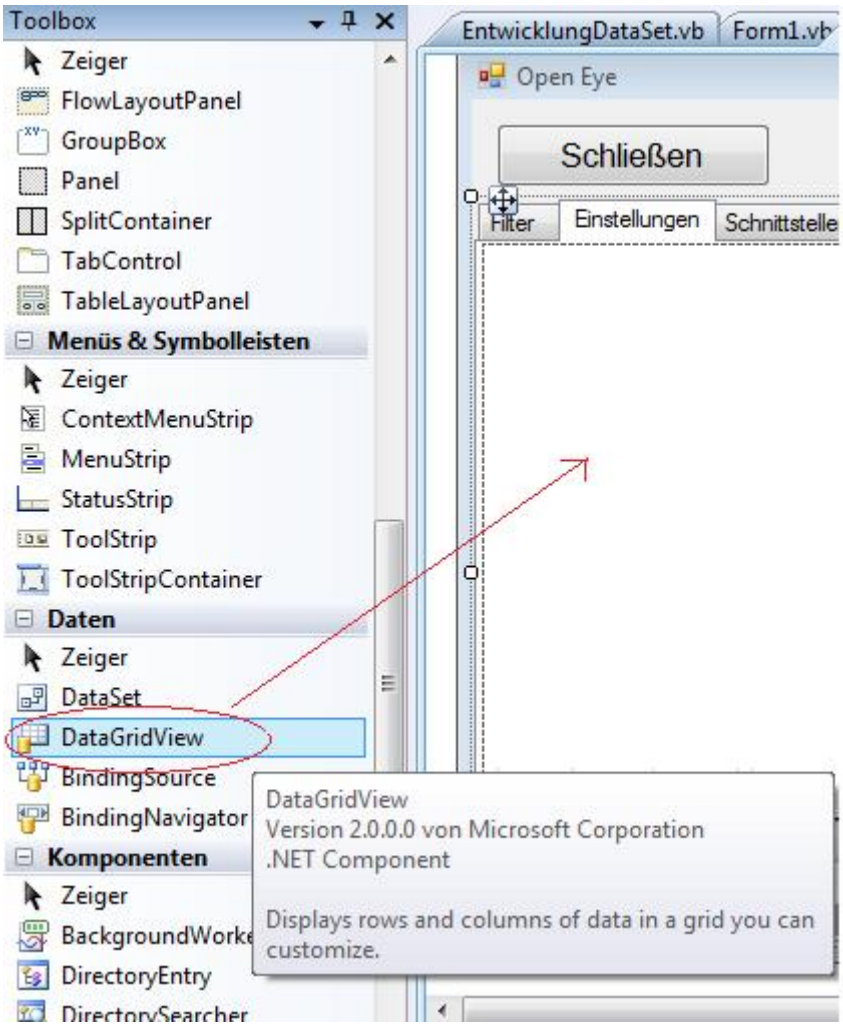
**Adressliste**

| Lfd.Nr | Name | Vorname | Strasse | Haus-Nr. | PLZ | Wohnort |
|---|---|---|---|---|---|---|
| ~~1~~ | ~~König~~ | ~~Horst~~ | ~~Am Markt~~ | ~~5~~ | ~~33213~~ | ~~Irgendwo~~ |
| ~~2~~ 1 | Meier | Fritz | Bahnhofstr | 13 | 87338 | Bergdorf |
| ~~3~~ 2 | Bäcker | Ulli | Gartenweg | 34 | 2977 4 | Heidetal |
| 3 | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

*Adresstabelle mit falschem Bezug*

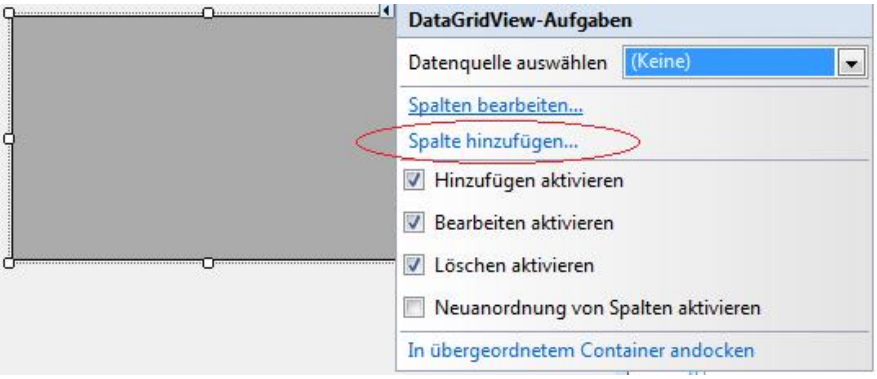
### *1.2.5.3 Tabellen einrichten*

Wechseln wir nun mit einem Click auf die Überschrift „Einstellungen“ auf die zweite Seite. Es zeigt sich ein leeres Blatt. Auf dieses Blatt ziehen wir nun aus der Toolbox zweimal eine DataGridView. Auch wenn die Objekte noch unbekannt sind so geben die Icons in der Toolbox etwas Aufschluss, ob ein Objekt für die Aufgabe geeignet ist.



*Toolbox Auswahl DataGridView*

Nachdem das DataGridView auf der TabPage installiert ist, will es eingerichtet werden. Eine Datenquelle haben wir nicht verfügbar und darum wird diese Einstellung auch nicht verändert. Aber Spalten werden gebraucht und müssen nun hinzugefügt werden. Der Screenshot zeigt den Vorgang.



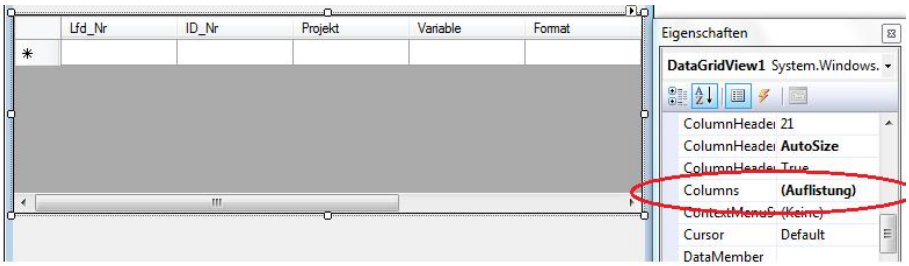

*DataGridView einrichten*

Für die Tabelle der Variablen werden die Spalten **lfd_Nr**, **Id_Nr**, **Projekt**, **Variable**, **Format**, **Freigabe**, **aktiv** und **Trigger** eingerichtet. Die Spalten **Id_Nr**, **lfd_Nr** und **Projekt** werden über die Eigenschaftseinstellung auf unsichtbar geschaltet. Die brauchen wir nur, um die Datensätze in der Datenbank zuzuordnen.

Die gleiche Vorgehensweise ist bei der Tabelle der Bitbeschreibung anzuwenden. Die Spalten **Id_Nr**, **Projekt_Id**, **Variable_ID**, **Bit** und **Funktion** müssen die Referenzen zum Projekt und Variablennamen sowie die eigene Referenz auf den Datensatz in der Datenbank enthalten.

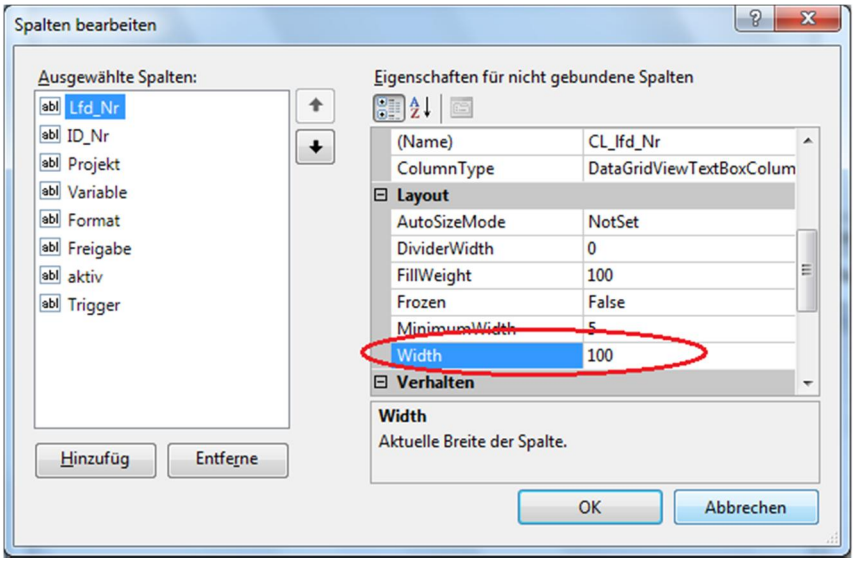
Der Assistent liefert die Schablone dafür gleich mit dazu. Sollte der Bereich DataGridView-Aufgaben abgeblendet werden, kann mit einem rechten Mausklick in das Objekt ein Auswahlmenü aufgeschlagen werden, in welchem der Punkt Spalte hinzufügen enthalten ist. Auch im Eigenschaftsbereich des Objektes findet sich diese Möglichkeit wieder.

Schließlich erhalten wir das folgende Tabellenobjekt.




*Tabellenspalten einfügen*

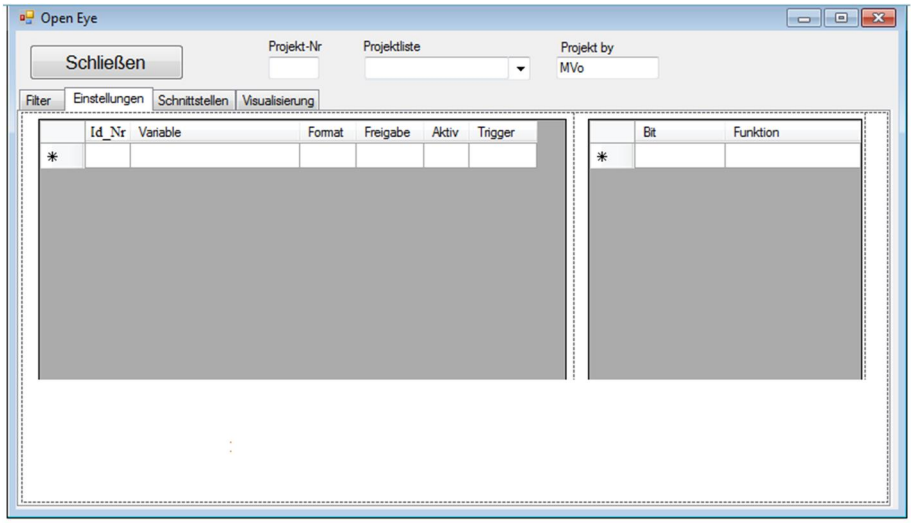
Bei dieser Ansicht ist noch nicht jede Spalte sichtbar. Allerdings bestünde die Möglichkeit mit einem Scrollbalken die nicht sichtbaren Bereiche herüber zu schieben. Es geht aber auch etwas eleganter und dafür brauchen wir die Eigenschaften der Spalten, die wir mit der Auflistung der Columns aus der Objekteigenschaft aufrufen können. Dann bekommen wir ein Eigenschaftsfenster mit dem Zugriff auf die Spalteneigenschaften. Mit dem Parameter Width lässt sich so auch eine Spaltenbreite variieren. Auch die Eigenschaft Visible ist hier mit dem gewünschten Status zu besetzen.




*Tabellenspalten parametrieren*

Auf diese Weise werden die beiden Tabellen für die Variablen und die Einzelbitdarstellung aufgebaut und in das TabPage-Objekt der zweiten Seite eingepasst. Es ist nicht gleich erforderlich, die Eigenschaft Visible gleich zu besetzen, da es manchmal noch wichtig ist, die Informationen in den Tabellenfeldern zu kontrollieren oder visuell abzurufen.

Schließlich sollten wir dieses Bild erhalten




*Erster Entwurf Seite 2*

Die Tabelle für die Variablen bekommt den Namen **Dg_Variablen** und die Tabelle für die Einzelbitdarstellung bekommt den Namen **Dg_Einzelbit**.

Die Spalte Lfd_Nr ist eigentlich nicht unbedingt erforderlich, da eine Tabelle einen eigenen Zeilenindex besitzt. Natürlich kann bei Korrekturen und Referenzen auch darauf zurückgegriffen werden. Doch so wird die Zugehörigkeit der Spalte zum Datensatz etwas deutlicher und die Datensätze der Tabelle mit den Bitinformationen wird diese Lfd_Nr als Referenz benutzen. Das ist kein Problem, da wir ja keine Datensätze löschen und diese Bindung auch nur in dieser Bearbeitung gültig ist. Bei jedem Neuaufbau wird diese Bindung auch neu erzeugt.

### *1.2.5.4 Korrekturobjekte einfügen*

Die Tabellen sind erstellt und nun ist die Frage, wie eine Korrektur vorgenommen werden soll. Dazu nehmen wir eine Textbox, in der die Zeilennummer der zu bearbeitenden Tabellenzeile steht. Die Nummer wird aus dem Tabellenparameter **Row** gewonnen und darf nicht verändert werden. Deshalb bekommt diese Textbox auch die Eigenschaft *Enabled = False* und wird dadurch für den Zugriff gesperrt. Auf gleiche Weise wird eine zweite Textbox eingebaut, die den eindeutigen Index des Datensatzes auf der Datenbank enthält. Im Moment ist er noch nicht vorhanden, da eine Datenbanktabelle noch gar nicht installiert ist. Deshalb sind **Lfd_Nr** und **Id_Nr** noch gleich.

Nun ist zu überlegen, welche Spalteneinträge für den Korrekturvorgang angezeigt werden sollen und welche Möglichkeiten zur Korrektur angebracht sind.

Nur informativ ist der Variablenname, der in dem Editierbereich mit aufgeführt wird und daher wird auch diese Textbox gesperrt. Die Namen der drei Textbox-Objekte werden mit **Tb_Lfd_Nr**, **Tb_Var_ID** und **Tb_Variable** vergeben

Das Format ist für die Auswertung in einer Combobox fest vorgegeben, damit kein Schreibfehler eine fehlerhafte Auswertung verursacht. Darum ist hier eine Combobox mit einer darüber liegenden gesperrten Textbox angebracht, wie wir es ja bereits schon auf der Filterseite und mit den Projektnamen aufgebaut haben. Gleiches gilt für den Trigger, der nur feste Werte von 0 - 8 eintragen darf. Die Namen sind entsprechend mit **Tb_Format**, **Cb_Format**, Tb_Trigger und **Cb_Trigger** gewählt

Die Spalte **Aktiv** ist der letzte veränderbare Parameter und wird mit einer Checkbox dargestellt. Sie liefert über ihren Status nur zwei Zustände und das reicht für die Aussage ja und nein. Das Objekt bekommt den Namen **Cb_Aktiv**.

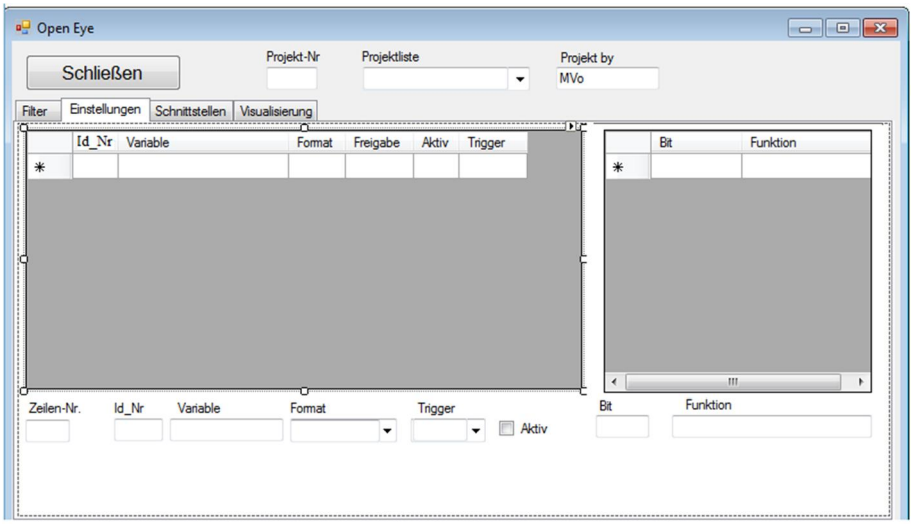
Ein Blick hinüber zur Tabelle mit der Bitinformation. Hier haben wir nur einen einzigen Eintrag, der geändert werden darf. Es ist die Bitbeschreibung. Diese ist allerdings völlig frei und ohne Einschränkung vorzunehmen. Eine Textbox für die Funktion des Bits ist hier die richtige Wahl.

Auch dieses Mal wird die Zeilennummer mit einer gesperrten Textbox angezeigt. Die Namen sind **Tb_Bit** und **Tb_Funktion**.

Ein Hinweis: Es ist nicht erforderlich, die Textboxen mit den Zeilennummern der Tabelle anzuzeigen. Diese Objekte können auch mit der Eigenschaft Visible ausgeblendet werden. So ist der Zugriff durch das Programm auf den Inhalt gegeben, aber die Ansicht verrät es nicht.

Mit der Tabelle der Bitbeschreibung und den Objekten, die zur Korrektur der Bitbeschreibung eingerichtet wurden, verhält es sich ähnlich. Nur wenn das Format der Variablen **Byte** ist, soll diese Tabelle und die Korrekturelemente erscheinen.

Die Ansicht stellt sich nun so dar:




*Editbereich für Variablen*

Die Labels über den Textfeldern sind nicht explizit erwähnt, da sie keinerlei Besonderheiten haben. Darum besitzen sie auch keinen Extranamen. Hier reicht die automatische Benennung mit Labelxx völlig aus.

### *1.2.5.5 Eine Gruppe von Objekten ein- und ausblenden*

Im Kapitel 1.1.5 habe ich einen Bezug zu Lego-Bausteinen und Objekten aufgezeigt. Es ist klar, ich kann einen einzelnen Baustein betrachten, aber auch ein Gebilde aus mehreren Bausteinen auf einer Grundplatte. Bewege ich diese, folgen alle einzelnen Bausteine der Bewegung.

Auch in unserem Programm können Bausteine als Grundplatte eingebaut werden und wenn diese *Grundplatte* unsichtbar geschaltet ist, sind auch alle darauf befindlichen Objekte unsichtbar. Betrachten wir dazu die Tabelle für die Einzelbitbeschreibung und die zugehörigen Textboxen für die Korrektur.

Nun ist es möglich, wenn eine Variable als Byte definiert ist, mit drei Befehlen jedes einzelne Objekt sichtbar und bei einem anderen Format unsichtbar zu schalten.

```
If Format = „Byte” then
  Dg_Einzelbit.Visible = True
  TB_Bit.Visible= true
  Tb_Funktion.Visible = True
Else
  Dg_Einzelbit.Visible = True
  TB_Bit.Visible= true
  Tb_Funktion.Visible = True
End if
```

Dabei sind aber noch nicht die Labels berücksichtigt, die für Bitnummer und Funktion den beschreibenden Text liefern. Legt man aber alle Objekte auf ein Panel und gibt diesem dann den Namen **PN_Bitfeld** ist nur ein einziger Befehl erforderlich:

```
Pn_Bitfeld.Visible = Tb_Format.Text=”Byte”
```

Das Ergebnis der Zuweisung an **TB_Bitfeld.Visible** ist das Ergebnis aus dem Vergleich **Tb_Format.Text = „Byte“.** Nicht einfach zu durchschauen, aber das *Visible* erwartet *Wahr* oder *Falsch*, also einen booleschen Ausdruck und der ist das Ergebnis aus dem Vergleich auf Byte. Dieser Befehl wird in dem Ereignis **TextChanged** der Textbox **Tb_Format** eingebaut. So wird er bei Änderung durch die Combobox **Cb_Format** genau so wie durch Datenübertrag aus der Tabelle ausgeführt.

```
Private Sub Tb_Format_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Tb_Format.TextChanged
    Pn_Bitfeld.Visible = Tb_Format.Text = "Byte"
End Sub
```

Nicht immer ist es erforderlich, einen Vergleich mit If umzusetzen, wie im ersten Beispiel mit der separaten Behandlung der Sichtbarkeit der einzelnen Objekte Die Ereignisbehandlung von Tb_Format_TextChanged zeigt den Weg über das Ergebnis einer Berechnung.

Die Klausel

Pn_Bitfeld.Visible = Tb_Format.Text = "Byte"

Werde ich mal etwas deutlicher machen:

Pn_Bitfeld.Visible kann den Wert Wahr oder Falsch annehmen. So ist z. B. die Aussage 3=5 falsch oder hier mit Text  Int8 = Byte. Wenn nun Im Textfeld der Textbox Int8 oder Word steht, dann ist die Aussage

Tb_Format.Text = „Byte“

falsch und dieses Falsch wird einfach der Eigenschaft Visible des Panels zugewiesen. Ist aber der Inhalt der Textbox „Byte“ dann wäre die Aussage Wahr und mit der Zuweisung an die Eigenschaft Visible des Panel wird dieses dann natürlich auch sichtbar. Man könnte auch sagen, das Programm bewertet eine Behauptung.

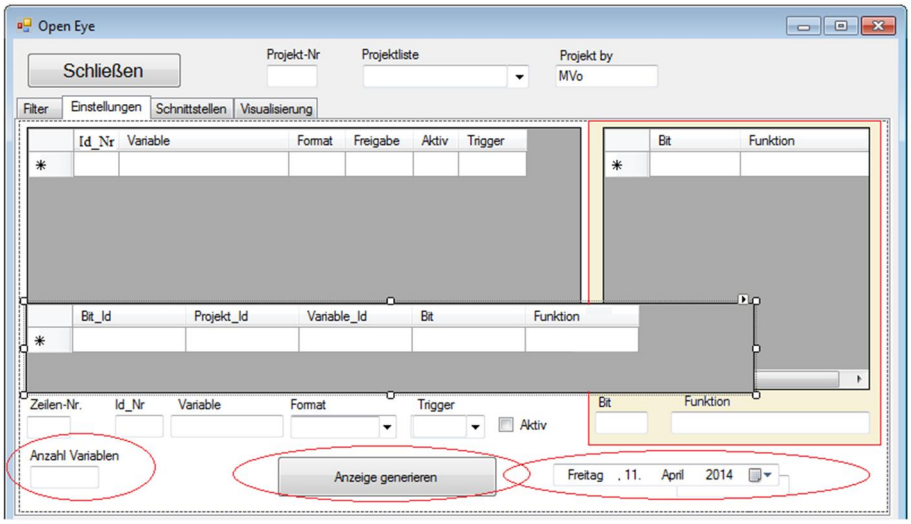
### *1.2.5.6 Ansicht der fertigen Seite Einstellungen*

Grob betrachtet ist nun soweit alles eingebaut. Vielleicht ist eine Anzeige der Zeilenanzahl der Variablentabelle brauchbar, denn die Anzahl der Variablen müssen wir später im Mikrocontroller der Übertragungsprozedur mitgeben. Ein einfaches Textfeld erledigt diese Aufgabe und mit **Tb_Anzahl_Var** haben wir auch gleich einen Namen für die Textbox, die übrigens auch für den Zugriff gesperrt wird.

Wenn nun alle Einstellungen vorgenommen sind, können wir die Objekte erzeugen, die uns später auf der Seite Visualisierung die Werte der Variablen anzeigen. Dies erledigen wir einfach mit einem Button, das den Namen **Bt_Gen_Anzeige** bekommt. Um den Inhalt der Ereignisroutine Click kümmern wir uns später.

Vielleicht kommt auch noch ein Datumsfeld hinzu, welches den Änderungsstand festhält. Dafür ist in der Toolbox ein Objekt **DateTimePicker.** Den Namen dafür können wir leicht mit **DTP_Datum** ableiten. In der Eigenschaft wird noch das Format auf **Long** gestellt, um das Datum in der gezeigten Form zu bekommen.

Schließlich wird noch eine Tabelle gebraucht, die alle Einzelbitinformationen beinhaltet. Die Tabelle zur Ansicht und zur Korrektur enthält ja nur die Information zur gerade angewählten Variablen. Also ist die Tabelle für alle bekannten Einzelbitbeschreibungen nicht zur Ansicht gedacht, sondern der Datencontainer, aus dem die Auswahl der Information herausgeholt wird. Darum liegt sie unsichtbar im Hintergrund. Nur zur Entwurfszeit ist sie sichtbar. Sie bekommt den Namen **Dg_All_Bits**. Der Screenschot zeigt die vollständige 2. Seite




*Alle Objekte der Seite 2*

Das Panel ist zur besseren Ansicht etwas eingefärbt und mit einem roten Rahmen versehen. Im Programm selbst ist die Eigenschaft BackColor auf Transparent gestellt.

Damit sind alle für eine Korrektur erforderlichen Elemente aufgezählt. Auf ein Button für **Korrektur übernehmen,** wie es normalerweise bei Programmen üblich ist, verzichte ich. Bei dieser kleinen Anwendung spricht nichts gegen eine sofortige Übernahme von geänderten Tabelleneinträgen und die Objekte liefern dazu die entsprechenden Ereignisse, die dann den Speichervorgang auslösen.

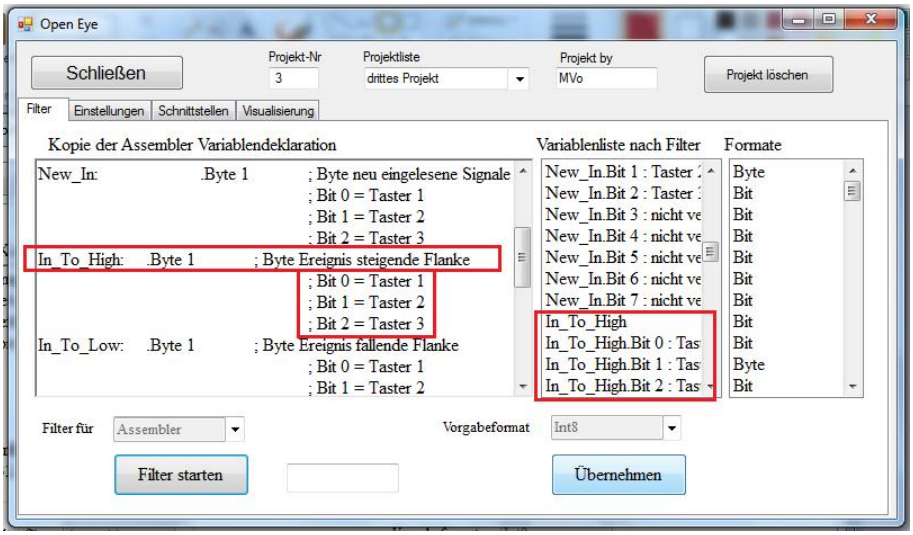
### *1.2.5.7 Übertragen der Daten aus dem Filter zur Bearbeitung*

Damit wir mit dieser Seite weiter arbeiten können, brauchen wir die Daten aus der ersten Seite, dem Filter. Lassen wir den Filter noch einmal mit unseren Testdaten durchlaufen und schalten mit dem Button Übernehmen auf die zweite Seite. Allerdings ist diese Aufgabe noch nicht erteilt. Blättern wir also noch einmal zurück zu Seite Filter und erzeugen mit einem Doppelklick auf das Button Übernehmen die Ereignisroutine Click.

```
Private Sub Bt_Uebernahme_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Uebernahme.Click

End Sub
```

Nun müssen wir die erzeugten Variablennamen in die Tabelle **Dg_Variablen** und die Einzelbitinformationen in die Tabelle **Dg_All_Bits** übertragen.

Werfen wir dazu noch einmal einen Blick in den Screenschot mit dem Ergebnis, das uns der Filter liefert.



*Ergebnis Filter Seite 1*

Wir können hier eine feste Schleife zur Übertragung einsetzen, doch nicht alle Einträge aus der Variablenliste dürfen in die Variablentabelle. Die rot markierten Einträge weisen darauf hin. Es gibt eine Variable mit dem

Namen In_To_High, die als Byte deklariert ist. Dazu gibt es aber auch Einzelbitbeschreibungen, die direkt dahinter in der Variablenliste mitgeführt sind. Die müssen natürlich in der Tabelle **Dg_All_Bits** eingetragen werden, denn in der Variablentabelle haben diese Einträge nichts zu suchen. Deshalb deklariere ich zusätzlich zum Schleifenzähler noch für jede Tabelle einen Indexzähler.

### *1.2.5.8 Gerüst entwerfen*

Fassen wir kurz zusammen

Wir benötigen einen Schleifenzähler, einen Index für Variablennamen und einen Index für Bitbeschreibungen.

```
Dim I As Integer                ' Schleifenzähler
Dim lfd_Nr_Var As Integer       ' Zähler für Variableneinträge
Dim lfd_Nr_Bit As Integer       ' Zähler für Biteinträge
```

Hinzu kommen die Variable für die Anzahl der Einträge in der Variablenliste.

```
Dim Anzahl As Integer       ' Anzahl Einträge in Variablenliste
```

Ebenfalls Integer ist die Position vom Punkt im String, um eine Bitbeschreibung von der Variable zu lösen

```
Dim Trenn_Pos As Integer    ' Position vom Punkt
```

Schließlich müssen wir uns einen Text für die Bitbeschreibung herausfiltern. Das tun wir auch wieder in kleinen Häppchen und deklarieren deshalb eine Stringvariable

```
Dim Ref_Str As String       ' Kopie Listeneintrag
```

Damit können wir nun starten:

```
Private Sub Bt_Uebernahme_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Uebernahme.Click
  Dim i As Integer                ' Schleifenzähler
  Dim Anzahl As Integer
  Dim lfd_Nr_Var As Integer       ' Zähler für Tabellenzeile Variablen
  Dim lfd_Nr_Bit As Integer       ' Zähler für Tabellenzeile Einzelbit
  Dim Trenn_Pos As Integer        ' Hilfsvariable für Textbearbeitung
  Dim Ref_Str As String           ' Hilfsvariable für Listeneintrag
  DG_Variablen.Rows.Clear()
  Dg_All_Bits.Rows.Clear()
  lfd_Nr_Var = 0
  lfd_Nr_Bit = 0
  Anzahl = Lb_Variablen.Items.Count    ' Schleifengrenze ermitteln
  For i = 0 To Lb_Variablen.Items.Count – 1
```

```
    ' hier werden die Listeneinträge bearbeitet
  Next i
End Sub
```

### *1.2.5.9 Daten aus Filter übernehmen*

So, bis zur Schleife haben wir nun schon die Basis für die Datenübertragung in die Tabellen. Werfen wir nun den Blick in die Schleife. Hier soll ja jeder Eintrag geprüft und der richtigen Tabelle zugeordnet werden. Also laden wir erst einmal den Listeneintrag in den Referenzstring. Anschließend prüfen wir, ob ein Punkt in diesem Eintrag vorhanden ist. Die Funktion **InStr** kennen wir bereits aus dem Filter. So ist das Ergebnis > 0 wenn im String ein Punkt enthalten ist. Damit haben wir die Abfrage zur Entscheidung ob Variablenname oder Bitposition. Der folgende Abschnitt kommt in die Schleife:

```
    Ref_Str = Lb_Variablen.Items(i)     ' Eintrag aus VAriablenliste holen
    Trenn_Pos = InStr(Ref_Str, ".")     ' Position vom Punkt ermitteln
     If Trenn_Pos = 0 Then              ' Kein Punkt, dann ist das ein Variablenname
      DG_Variablen.Rows.Add()
      DG_Variablen.Item("CL_Id_Nr", lfd_Nr_Var).Value = lfd_Nr_Var + 1
      DG_Variablen.Item("CL_Projekt", lfd_Nr_Var).Value = 0
      DG_Variablen.Item("CL_Variable", lfd_Nr_Var).Value = Ref_Str
      DG_Variablen.Item("CL_Format", lfd_Nr_Var).Value =
                                                Lb_Formate.Items(i)
      DG_Variablen.Item("CL_Freigabe", lfd_Nr_Var).Value = "Ja"
      DG_Variablen.Item("CL_aktiv", lfd_Nr_Var).Value = "Ja"
      DG_Variablen.Item("CL_Trigger", lfd_Nr_Var).Value = "0"
      lfd_Nr_Var = lfd_Nr_Var + 1
     Else    ' hier ist es eine  Einzelbitinfo
      Dg_All_Bits.Rows.Add()
      Dg_All_Bits.Item("CL_Bit_Id", lfd_Nr_Bit).Value = lfd_Nr_Bit + 1
      Dg_All_Bits.Item("CL_Proj_Id", lfd_Nr_Bit).Value = 0
      Dg_All_Bits.Item("CL_Var_Id", lfd_Nr_Bit).Value = lfd_Nr_Var
      Dg_All_Bits.Item("CL_BitNr", lfd_Nr_Bit).Value = "Bit " + Mid(Ref_Str, Trenn_Pos
                                                                        + 5, 1)
      Trenn_Pos = InStr(Ref_Str, ":")
      Ref_Str = Mid(Ref_Str, Trenn_Pos + 1, Len(Ref_Str) - Trenn_Pos)
      Dg_All_Bits.Item("CL_BitFunktion", lfd_Nr_Bit).Value = Ref_Str
      lfd_Nr_Bit = lfd_Nr_Bit + 1
     End If
```

Die Tabelle für die Variablennamen wird mit festen Vorgaben für die noch nicht bekannten Einträge ergänzt und der Zähler für den Schlüssel zum Variableneintrag hochgezählt.

Der zweite Teil ordnet die Bitbeschreibung zu. Die Spalte **Bit_Id** bekommt den Wert von der Zählvariablen **Id_Nr_Bit+1**. Auch die Einträge mit den Variablennamen aus dem Assemblerlisting wurden so indiziert. Das +1

muss nicht sein, aber ich lasse die 0 gern weg und beginne mit der Zählung bei 1.

Es folgt nun die Referenz zum Namen der Assemblervariablen. Da die **Id_Nr** der des Namens nicht verändert wurde, habe ich hier einen Bezug zur Tabellenzeile.

Die Bitnummer ist im Referenzstring immer an der gleichen Stelle, daher können wir direkt darauf zugreifen und den Tabelleneintrag an dieser Stelle einfach mit dem Text + Bitnummer zusammenstellen. Danach wird alles vor dem Punkt einschließlich Punkt aus dem Referenzstring entfernt und übrig bleibt die Bitbeschreibung.

Die letzte Aktion ist es den Indexzähler für die Einzelbittabelle zu erhöhen. Damit ist in der Schleife alles erledigt.

Damit ist aber auch die Seite Filter abgeschlossen und wir können hier direkt auf die Seite Einstellungen wechseln. Auch die Anzahl der Variablen entspricht den Einträgen in der Variablentabelle und diese Zahl können wir gleich in die Textbox eintragen. Minus 1 ist klar, es ist immer die letzte Zeile in der Tabelle frei.

```
TC_Auswahl.SelectTab(1) ' hier erfolgt der Wechsel zur nächsten Seite
TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
Set_Edit_Variable(0)        ' erste Zeile in den Editbereich übernehmen
```

Die letzte Zeile ruft eine Routine auf, die noch nicht existiert. Ein Test würde also eine Fehlermeldung des Compilers liefern.

Eigentlich gehört diese Routine zur Seite Einstellungen, aber natürlich darf sie auch hier aufgerufen werden. Genau genommen hat das Programm ja bereits die Seite gewechselt. Aber wofür ist sie gedacht, was ist die Funktion?

Der Name dieser Sub sollte es eigentlich ausdrücken. Setze den Editierbereich mit den Variablendaten. Und weil eine 0 in der Klammer steht, bedeutet das, die Daten aus der ersten Tabellenzeile sind betroffen. Immer daran denken, eine Tabelle fängt mit der Indizierung der Zeilen bei 0 an.

Hier noch einmal die zusammenhängende Ereignisroutine vom Übernahmebutton

```
Private Sub Bt_Uebernahme_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Uebernahme.Click
  Dim i As Integer                  ' Schleifenzähler
  Dim Anzahl As Integer
  Dim lfd_Nr_Var As Integer    ' Zähler für Tabellenzähler Variablen
  Dim lfd_Nr_Bit As Integer      ' Zähler für Tabellenzeile Einzelbit
  Dim Trenn_Pos As Integer   ' Hilfssvariable für Textbearbeitung
  Dim Ref_Str As String          ' Hilfsvariable für Listeneintrag
  DG_Variablen.Rows.Clear()
  Dg_All_Bits.Rows.Clear()
  lfd_Nr_Var = 0
  lfd_Nr_Bit = 0
  Anzahl = Lb_Variablen.Items.Count    ' Schleifengrenze ermitteln
  For i = 0 To Lb_Variablen.Items.Count – 1
    Ref_Str = Lb_Variablen.Items(i)     ' Eintrag aus Variablenliste holen
    Trenn_Pos = InStr(Ref_Str, ".")      ' Position vom Punkt ermitteln
    If Trenn_Pos = 0 Then                 ' kein Punkt, dann ist das Variablenname

      DG_Variablen.Rows.Add()
      DG_Variablen.Item("CL_Id_Nr", lfd_Nr_Var).Value = lfd_Nr_Var + 1
      DG_Variablen.Item("CL_Projekt", lfd_Nr_Var).Value = 0
      DG_Variablen.Item("CL_Variable", lfd_Nr_Var).Value = Ref_Str
      DG_Variablen.Item("CL_Format", lfd_Nr_Var).Value =
                                  Lb_Formate.Items(i)
      DG_Variablen.Item("CL_Freigabe", lfd_Nr_Var).Value = "Ja"
      DG_Variablen.Item("CL_aktiv", lfd_Nr_Var).Value = "Ja"
      DG_Variablen.Item("CL_Trigger", lfd_Nr_Var).Value = "0"
      lfd_Nr_Var = lfd_Nr_Var + 1
    Else    ' hier ist es eine  Einzelbitinfo
      Dg_All_Bits.Rows.Add()
      Dg_All_Bits.Item("CL_Bit_Id", lfd_Nr_Bit).Value = lfd_Nr_Bit + 1
      Dg_All_Bits.Item("CL_Proj_Id", lfd_Nr_Bit).Value = 0
      Dg_All_Bits.Item("CL_Var_Id", lfd_Nr_Bit).Value = lfd_Nr_Var
      Dg_All_Bits.Item("CL_BitNr", lfd_Nr_Bit).Value = "Bit " + Mid(Ref_Str, Trenn_Pos
                                                                          + 5, 1)
      Trenn_Pos = InStr(Ref_Str, ":")
      Ref_Str = Mid(Ref_Str, Trenn_Pos + 1, Len(Ref_Str) - Trenn_Pos)
      Dg_All_Bits.Item("CL_BitFunktion", lfd_Nr_Bit).Value = Ref_Str
      lfd_Nr_Bit = lfd_Nr_Bit + 1
    End If
    TC_Auswahl.SelectTab(1)  ' Hier erfolgt der Wechsel zur nächsten seite
    TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
     Set_Edit_Variable(0)
  Next i
End Sub
```

### *1.2.5.10 Daten zum Editieren bereitstellen*

Bauen wir nun endlich die Subroutine Set_Edit_Variable auf.

```
Public Sub Set_Edit_Variable(ByVal Row_Nr As Integer)

End Sub
```

Auch hier könnten wir vielleicht lokale Variablen benötigen. Aber beginnen wir erst einmal damit, die Aufgabe zu lösen. Die Variablen sind noch nicht erkennbar.

Zuerst setzen wir den Index der selektierten Zeilennummer in die Textbox für die laufende Nummer. Dabei ist es Geschmackssache, die Zeile mit dem Index, Wert ist Row_Nr, oder mit der Zeilennummer, Wert ist dann Row_Nr+1 zu füllen. Das muss dann lediglich beim Zurückschreiben der Daten berücksichtigt werden. Ich lege es jetzt fest, dass der Index der Zeile eingesetzt wird. Da die Textbox einen Text enthält, muss die Zahl in einen String gewandelt werden. Das erledigt die Funktion Str(<Zahl>)

```
Tb_Lfd_Nr.Text = Str(Row_Nr)
```

Nun kommt eine der wesentlichen Programmaufgaben, die Kontrolle, ob ein Editieren überhaupt erlaubt ist. Zuerst erfolgt die Prüfung, ob überhaupt ein Datensatz existiert Der zweite Schritt prüft, ob aufgrund der Freigabe editieren erlaubt ist. Ist die Zeile freigegeben werden die Tabelleninhalte der Zeile in die Textboxen übertragen.

```
If DG_Variablen.Rows.Count > 0 Then
  If DG_Variablen.Item("CL_Freigabe", Row_Nr).Value = "Ja" Then
     Tb_Var_ID.Text = Str(DG_Variablen.Item("CL_Id_Nr", Row_Nr).Value)
     TB_Variable.Text = DG_Variablen.Item("CL_Variable", Row_Nr).Value
     Cb_Format.Text = DG_Variablen.Item("CL_Format", Row_Nr).Value
     CB_Aktiv.Checked = DG_Variablen.Item("CL_Aktiv", Row_Nr).Value = "Ja"
***     Trigger = DG_Variablen.Item("Cl_Trigger", Row_Nr).Value
     CB_Trigger.Text = CB_Trigger.Items(Trigger)
     Tb_Trigger.Text = CB_Trigger.Text
   Else
***     Infosatz = "Formatänderung nicht möglich. Byte gehört zu einem anderen Format."
        Infosatz = Infosatz + Chr(13) +
                   "Es muss erst das vorhergehende Format geändert werden,"
        Infosatz = Infosatz + Chr(13) + "um Zugriff auf dieses Byte zu bekommen."
        MsgBox(Infosatz, MsgBoxStyle.Information, AcceptButton)
    End If
 End If
```

Die meisten Befehle erklären sich von selbst. Da wird einer Textbox zum Beispiel der Wert der Tabellenzelle übergeben, nichts Außergewöhnliches. Auch beim Format wird der Wert in den Textbereich von Cb_Format geschrieben. Die Zuweisung ab die Checkbox Cb_Aktiv allerdings ist schon interessanter. Ein ähnliches Vorgehen ist ja bereits bei der Sichtbarkeit des Panels PN_Bitfeld angewendet worden.

```
Cb_Aktiv.Checked = DG_Variablen.Item("CL_Format", Row_Nr).Value = "Ja"
```

CB_Aktiv.Checked ist ein boolescher Begriff und ist entweder wahr oder falsch genau wie die nachgeschaltete Behauptung, dass der Inhalt der Tabellenzelle **Ja** ist. Kann sein oder auch nicht. Entsprechend wird dies in der Checkbox angezeigt. Beim Aufbau der Befehle haben wir eine Zeile, in der ein Eintrag aus der Combobox CB_Trigger selektiert werden soll. Der Index steht in der Tabelle. Allerdings ist die Zeile

```
CB_Trigger.Text = CB_Trigger.Items(Dg_Variablen.Item("Cl_Trigger",Row_Nr).Value)
```

ziemlich unübersichtlich. Da eine lokale Variable kein Geld kostet und die Lesbarkeit des Codes wesentlich erhöht, darf eine Variable deklariert werden, die einzig und allein den Zweck hat, die Befehlszeilen übersichtlich zu gestalten. Das Ergebnis noch mal explicit

```
Trigger = DG_Variablen.Item("Cl_Trigger", Row_Nr).Value
CB_Trigger.Text = CB_Trigger.Items(Trigger)
```

Außerdem kann hier auch ein Haltepunkt eingesetzt werden, um den Wert der Variablen Trigger an dieser Stelle zu prüfen.

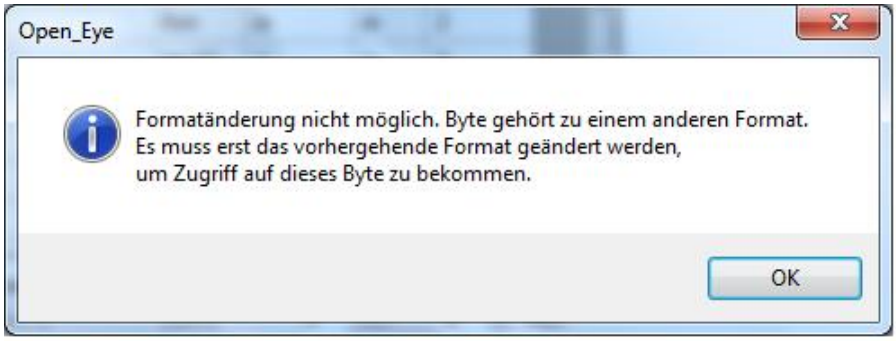
Die zweite Variable Infosatz hat einen anderen Grund. Möchte ich in einer Msgbox einen etwas umfangreicheren Text anzeigen und vielleicht auch noch Zeilen explizit absetzen, kann dies mit einer Textvariablen leicht erledigt und ein Zeilenende kann mit einem ASCII-Code hinzugefügt werden.

Der Code dafür ist Chr(<ASCII-Code>) und da ein ASCII Zeichen mit dem Code 13 einen Zeilenwechsel auslöst, kann mit Chr(13) eine mehrzeilige Textanzeige erstellt werden.

```
Infosatz = "Formatänderung nicht möglich. Byte gehört zu einem anderen Format."
Infosatz = Infosatz + Chr(13) +
                "Es muss erst das vorhergehende Format geändert werden,"
Infosatz = Infosatz + Chr(13) + "um Zugriff auf dieses Byte zu bekommen."
```

Der Inhalt der Variablen Infosatz stellt sich dann so dar. Die MsgBox bekommt nun die Variable und nicht einen Text zugewiesen.

Diese MsgBox wird nun aufgerufen, wenn eine Zeile nicht editierbar ist und liefert den Hinweis dass keine Änderung erlaubt ist.




*Info Ablehnung Formatänderung*

Diese Programmteile dienen nicht der Funktion, sondern der Information. Ich halte es für sinnvoll, dass eine abgelehnte Bearbeitung begründet wird. Stellt euch mal vor, diese Hinweise gäbe es nicht. Klar, ihr wisst schon, warum nix passiert, ihr habt ja das Programm geschrieben, aber ein Freund oder Kunde versteht erst einmal nicht, warum sich diese Tabellenzeile nicht editieren lässt. Dazu muss er nachdenken und selbstständig den Grund erkennen. Also immer daran denken, Programme schreibt man nie für sich selbst, sondern für absolute Laien. Und außerdem kann man auch nach Jahren noch seine eigenen Programme verstehen.

Hier noch einmal die komplette Subroutine

```
Public Sub Set_Edit_Variable(ByVal Row_Nr As Integer)
  Dim Trigger As Integer
  Dim Infosatz As String
  Tb_lfd_Nr.Text = Str(Row_Nr)
   If DG_Variablen.Rows.Count > 1 Then
     If DG_Variablen.Item("CL_Freigabe", Row_Nr).Value = "Ja" Then
       Tb_Var_ID.Text = Str(DG_Variablen.Item("CL_Id_Nr", Row_Nr).Value)
       TB_Variable.Text = DG_Variablen.Item("CL_Variable", Row_Nr).Value
       Cb_Format.SelectedIndex =
           Cb_Format.Items.IndexOf(DG_Variablen.Item("CL_Format", Row_Nr).Value)
        Cb_Format.Text = Cb_Format.Items.Item(Cb_Format.SelectedIndex)
        CB_Aktiv.Checked = DG_Variablen.Item("CL_Aktiv", Row_Nr).Value = "Ja"
        Trigger = DG_Variablen.Item("Cl_Trigger", Row_Nr).Value
        CB_Trigger.Text = CB_Trigger.Items(Trigger)
```

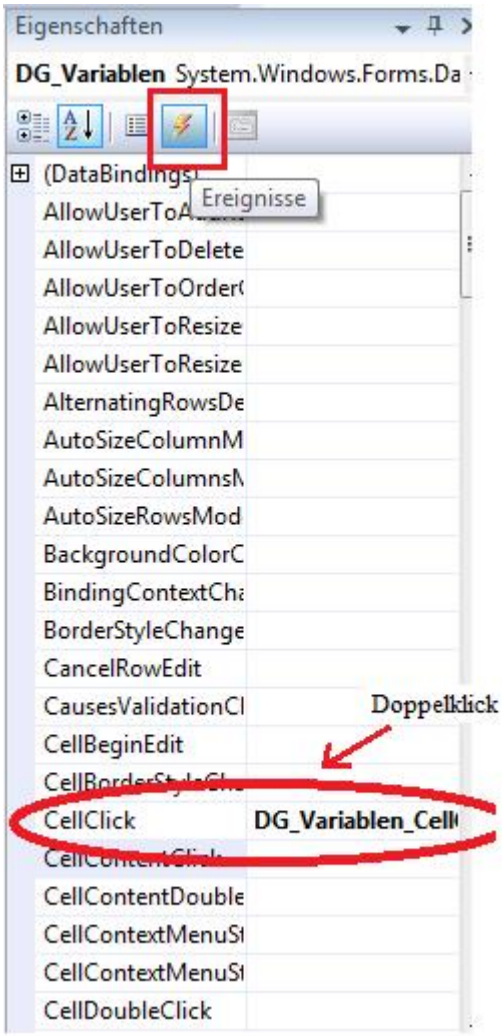
```
        Tb_Trigger.Text = CB_Trigger.Text
      Else
        Infosatz = "Formatänderung nicht möglich. Byte gehört zu einem anderen Format."
        Infosatz = Infosatz + Chr(13) +
                                "Es muss erst das vorhergehende Format geändert werden,"
        Infosatz = Infosatz + Chr(13) + "um Zugriff auf dieses Byte zu bekommen."
        MsgBox(Infosatz, MsgBoxStyle.Information, AcceptButton)
      End If
    End If
  End Sub
```

Wir rufen diese Routine auch auf, wenn eine Zeile in der Variablentabelle angeklickt wird.

Beginnen wir nun damit, im Abschnitt Einstellungen die erforderlichen Ereignisroutinen und Unterprogramme für Aufgaben mit Anweisungen zu füllen.

Fast fertig ist die Aufgabe, Daten einer Zeile in den Editierbereich zu bringen. Wenn wir nun eine Tabellenzeile anklicken, soll natürlich diese Zeile die Daten liefern. Dazu nehmen wir das Ereignis CellClick der Tabelle Dg_Variablen.

Der Weg ist einfach beschrieben. Der Screenschot zeigt die Liste der Ereignisse und den Eintrag Cellclick. Zu dieser Liste noch einmal der Hinweis: rechte Maustaste in das entsprechende Objekt, Eigenschaften anwählen und dann oben bei den Icons die Eigenschaften oder Ereignisse auswählen.




*Ereignis CellClick*

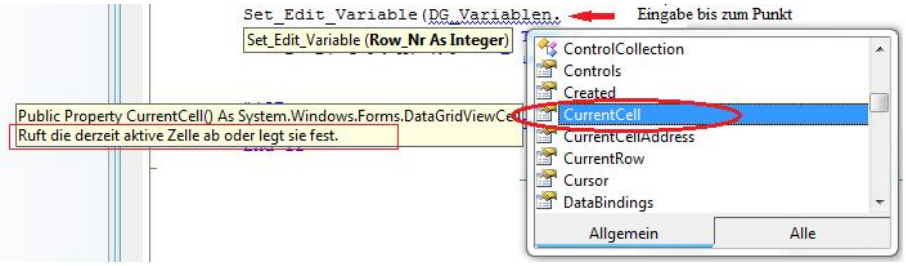
Nach dem Doppelklick wird der Rahmen für die Ereignisroutine **Dg_Variablen_CellClick** erstellt.

```
Private Sub DG_Variablen_CellClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DataGridViewCellEventArgs) Handles DG_Variablen.CellClick
```

```
End Sub
```

Was muss nun geschehen? Klar, die Daten in der gewählten Zeile müssen in den Editbereich. Dafür haben wir die Subroutine bereits fertig. Nur, wie bekommen wir den Index der Tabellenzeile, den wir der Subroutine mitteilen können?

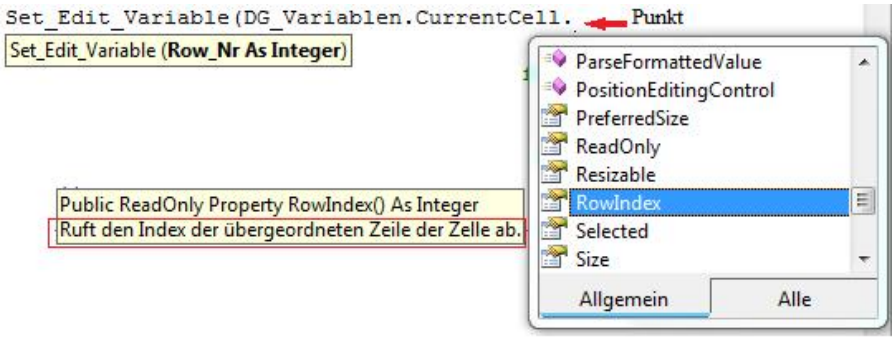
Auch hier wieder mal ganz langsam. Ein Objekt hat Eigenschaften und eine davon ist zum Beispiel der Index einer bestimmten Zeile. Verlassen wir uns einfach auf unsere Sprachkenntnisse in Englisch und die Online-Hilfe sowie auf unser Gefühl. Wir brauchen die Eigenschaften des Objektes, welches dieses Ereignis generiert hat, also das DataGrid Dg_Variablen. Schreiben wir also zuerst den Aufruf der Subroutine Set_Edit_Variable und setzen in der Klammer den Namen vom DataGrid Dg_Variablen ein, gefolgt von einem Punkt.




*Eigenschaft CurrentCell*

Es ist nicht schwer zu erraten, dass Zellen mit Cell benannt sind. Wenn wir diese Liste einmal durchgehen und alles, was mit Cell angegeben ist, uns ansehen, erkennen wir schon, welche Eigenschaft zum Ziel führt. CurrentCell liefert dann beim Anklicken die zusätzliche Information, dass wir auf dem richtigen Weg sind. Aber noch fehlt eine Information, nicht die Zelle wollen wir, sondern den Index der Zeile, in der diese Zelle steht. Also wieder ein Punkt hinter CurrentCell

Diesmal suchen wir nach einem Hinweis auf die Zeile. Row ist die englische Vokabel dafür und Index ist auch genau der Begriff, den wir brauchen, Starten wir einmal einen Versuch. Vom Compiler wird schon mal nix angemeckert. Also stimmt zumindest der Syntax.




*Eigenschaft RowIndex*

Fügen wir nun in die Ereignisroutine den Aufruf ein und testen das Ergebnis.

```
Private Sub DG_Variablen_CellClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DataGridViewCellEventArgs) Handles DG_Variablen.CellClick
   Set_Edit_Variable(DG_Variablen.CurrentCell.RowIndex)
End Sub
```

Sind die Daten richtig übertragen? Stimmen die Einträge. Wenn ja, dann bleibt noch eines zu tun. Die Assemblervariablen mit dem Format Byte sollen auch eine Tabelle mit der Einzelbitbeschreibung liefern. Ist das Format anders, soll die Tabelle mit den Einzelbits nicht sichtbar sein.

Also brauchen wir nur eine Abfrage, ob der Text in der Textbox **Tb_Format** Byte ist, dann lassen wir das Panel erscheinen und mit ihm alles, was sich darauf befindet.

```
Private Sub DG_Variablen_CellClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DataGridViewCellEventArgs) Handles DG_Variablen.CellClick
   Set_Edit_Variable(DG_Variablen.CurrentCell.RowIndex)
   If Tb_Format.Text = "Byte" Then
      Pn_Bitfeld.Visible = True   ' Bitinformation anzeigen
   Else
      Pn_Bitfeld.Visible = False  ' keine Bitinformation
   End If
End Sub
```

Aber die Tabelle zeigt nach einem anschließenden Test keine Einträge. Also schreiben wir eine Subroutine, die abhängig von der ausgewählten Variablen die Einzelbits darstellen kann. Ich nenne diese Routine

Show_Einzelbit. Den Aufruf fügen wir schon mal in die CellClick Ereignisroutine ein.

```
Private Sub DG_Variablen_CellClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DataGridViewCellEventArgs) Handles DG_Variablen.CellClick
   Set_Edit_Variable(DG_Variablen.CurrentCell.RowIndex)
   If Tb_Format.Text = "Byte" Then
      Pn_Bitfeld.Visible = True   ' Bitinformation anzeigen
      Show_Einzelbit()
   Else
      Pn_Bitfeld.Visible = False  ' keine Bitinformation
   End If
End Sub
```

Wie müssen wir dann weiter vorgehen? Nun ja, fangen wir einfach mal wie immer mit dem Gerüst an.

```
Public Sub Show_Einzelbit(ByVal Var_Id As Integer)

End Sub
```

Anschließend die Variablen deklarieren, von denen man weiß, das man sie braucht. Im Moment sehe ich nur einen Schleifenzähler und einen Grenzwert, der aus der Anzahl Einträge in einer Tabelle besetzt wird.

Definieren wir also

```
Dim I as Integer
Dim Anzahl as Integer
```

Dann leeren wir erst einmal die Tabelle Dg_Einzelbit

```
Dg_Einzelbit.Rows.Clear
```

um dann in einer Schleife erst einmal alle acht Bits mit einem Defaultwert einzurichten.

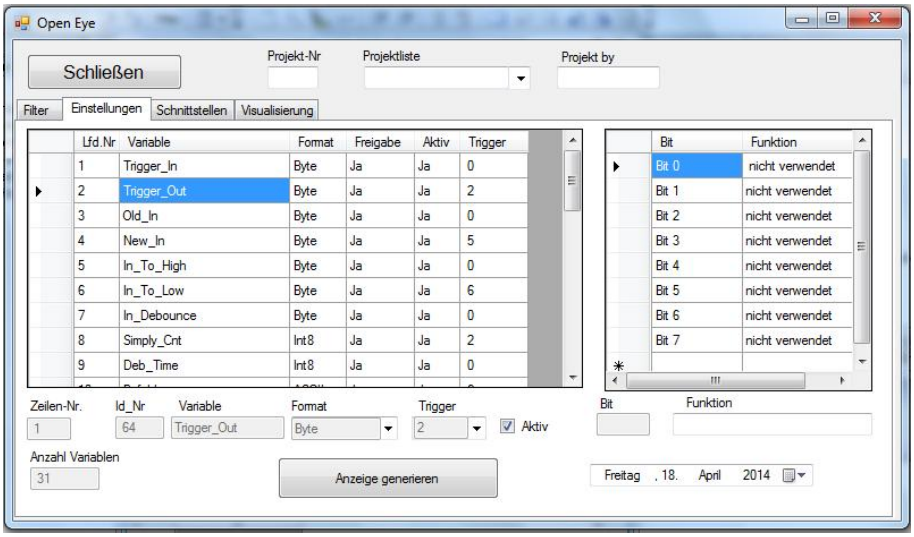
```
      For i = 0 To 7                    ' Tabelle vorbesetzen
         Dg_Einzelbit.Rows.Add()
         Dg_Einzelbit.Item("CL_BitId_Nr", i).Value = -1
         Dg_Einzelbit.Item("CL_Variable_Id", i).Value = Str(Var_Id)
         Dg_Einzelbit.Item("Cl_Bit", i).Value = "Bit" + Str(i)
         Dg_Einzelbit.Item("CL_Funktion", i).Value = "nicht verwendet"
      Next
```

Es kommt nun die Abfrage, ob in der nicht sichtbaren Tabelle Dg_All_Bits Bitbeschreibungen zur aktuellen Assemblervariable enthalten sind. Diese werden über den Vergleich mit der Schlüsselnummer zum Variablenamen und der Referenznummer in der Tabelle Dg_All_Bits gefunden.

Dazu brauchen wir nun die Anzahl der Einträge in dieser Tabelle. Auch hier setzen wir eine Schleife ein.

```
Anzahl = Dg_All_Bits.Rows.Count - 1     ' Anzahl Einträge Einzelbits
For i = 0 To Dg_All_Bits.Rows.Count - 2 ' Tabelle mit allen Einzelbits durchsuchen
  If Dg_All_Bits.Item("CL_Var_Id", i).Value = Var_Id Then
        ' Dieses Einzelbit hat eine Referenz auf die Variable
    Zeilen_Nr = Val(Mid(Dg_All_Bits.Item("CL_BitNr", i).Value, 5, 1))
        ' Zeilennummer wird aus Bitnummer gewonnen
    Dg_Einzelbit.Item("CL_BitId_Nr", Zeilen_Nr).Value = i
    Dg_Einzelbit.Item("Cl_Bit", Zeilen_Nr).Value = "Bit" + Str(Zeilen_Nr)
    Dg_Einzelbit.Item("CL_Funktion", Zeilen_Nr).Value =
                                  Dg_All_Bits.Item("CL_BitFunktion", i).Value
  End If
Next
```

Dieser Programmbereich ist nur solange erforderlich, bis eine Datenbank eine direkte Zusammenstellung der Information ermöglicht. Ansonsten wäre eine solche Vorgehensweise ungeschickt, da eine feste Schleife bis zum letzten Eintrag durchlaufen würde, auch wenn bereits alle acht Einzelbitbeschreibungen längst gefunden wurden. Testen wir nun das Ergebnis unserer Arbeit.



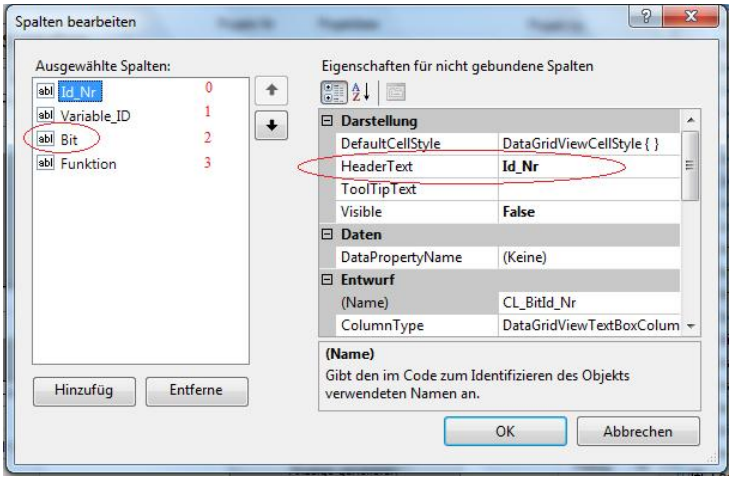
*Testergebnis Seite 2*

Sieht schon ganz gut aus, doch wenn man genau hinschaut, fällt vielleicht etwas auf. Die Einzelbitinformation, zu welchem Variablennamen gehören sie? Klar, hier sind die Tabellenfelder markiert, aber das ist nicht immer so. Also suchen wir nach einer Möglichkeit, der Tabelle den zugehörigen Variablennamen mitzuteilen und auch anzuzeigen.

Es gibt bei einer Tabelle die Eigenschaft Column (Spalten) und diese wiederum hat u.a. die Eigenschaft Headertext (Kopftext, Überschrift). Man findet diese z.B. in der Liste der Eigenschaften und wir haben sie auch bereits mit festen Einträgen besetzt. Dort, wo jetzt **Funktion** als Kopftext einer Spalte in der Einzelbitbeschreibung zu lesen ist, könnte man ja auch den Variablennamen hinschreiben und mit Funktion ergänzen. Mit dem Befehl

```
Dg_Einzelbit.Columns.Item(3).HeaderText =
"Variable: " + TB_Variable.Text +Chr(13) +"Funktion"
```

wird dieses Problem gelöst. Na ja, auch hier ist die eingeblendete Hilfe bei der Eingabe zu beachten und die Hinweistexte, die entsprechend aufgeblendet werden. Ich denke, bis Columns ist alles klar. In der Eigenschaftsliste steht hinter Columns Auflistung und ein Objekt in einer Liste ist ein Item. Die 2 gibt den Index an. Werfen wir einen Blick in die Auflistung der Columns und beginnen mit dem Zählen bei 0.




*Spaltenbeschriftung*

Der Screenshot zeigt noch einmal die Eigenschaft der Spaltenüberschrift.

Ergänzen wir nun unser Programm mit dem Befehl

```
Public Sub Show_Einzelbit(ByVal Var_Id As Integer)
    Dim i As Integer
    Dim Zeilen_Nr As Integer
    Dim Anzahl As Integer
    Dg_Einzelbit.Columns.Item(3).HeaderText = "Variable: " + TB_Variable.Text
    Dg_Einzelbit.Rows.Clear()              ' Tabelle löschen
    For i = 0 To 7                         ' Tabelle vorbesetzen
      Dg_Einzelbit.Rows.Add()
      Dg_Einzelbit.Item("CL_BitId_Nr", i).Value = -1
      Dg_Einzelbit.Item("CL_Variable_Id", i).Value = Str(Var_Id)
      Dg_Einzelbit.Item("Cl_Bit", i).Value = "Bit" + Str(i)
      Dg_Einzelbit.Item("CL_Funktion", i).Value = "nicht verwendet"
    Next
    Anzahl = Dg_All_Bits.Rows.Count - 1      ' Anzahl Einträge Einzelbits
    For i = 0 To Dg_All_Bits.Rows.Count - 2 ' Tabelle mit allen Einzelbits durchsuchen
      If Dg_All_Bits.Item("CL_Var_Id", i).Value = Var_Id Then
        ' Dieses Einzelbit hat eine Referenz auf die Variable
        Zeilen_Nr = Val(Mid(Dg_All_Bits.Item("CL_BitNr", i).Value, 5, 1))
        ' Zeilennummer wird aus Bitnummer gewonnen
        Dg_Einzelbit.Item("CL_BitId_Nr", Zeilen_Nr).Value = i
        Dg_Einzelbit.Item("Cl_Bit", Zeilen_Nr).Value = "Bit" + Str(Zeilen_Nr)
        Dg_Einzelbit.Item("CL_Funktion", Zeilen_Nr).Value =
                                    Dg_All_Bits.Item("CL_BitFunktion", i).Value
      End If
    Next
    Set_Edit_Bit(0)      ' Inhalt erster Zeile in den Editierbereich
  End Sub
```

Aber noch ist eine Kleinigkeit offen. Der Übertrag in den Editierbereich, so wie wir es bereits bei der Tabelle der Variablennamen vorgenommen haben. Den Aufruf habe ich bereits in die Routine Show_Einzelbit eingefügt. Die separate Subroutine, die dafür zuständig ist, die Daten einer bestimmten Tabellenzeile in die dafür vorgesehenen Textboxen zur Korrektur zu übertragen muss nun noch erstellt werden. Na ja, eigentlich gibt es nur einen Wert, der korrigiert werden darf, die Bitbeschreibung.

Die Subroutine ist nichts Besonderes und schnell erstellt.

```
Public Sub Set_Edit_Bit(ByVal Row_Nr As Integer)
    TB_Bit.Text = "Bit" + Str(Row_Nr)
    TB_Funktion.Text = Dg_Einzelbit.Item("CL_Funktion", Row_Nr).Value
  End Sub
```

Wie bereits bei der Tabelle der Variablennamen wird die Zeilennummer der Routine mitgegeben. So ist es auch kein Problem, ein CellClick-Ereignis der Dg_Einzelbittabelle für diese Aufgabe einzurichten.

```
Private Sub Dg_Einzelbit_CellClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DataGridViewCellEventArgs) Handles Dg_Einzelbit.CellClick
    Set_Edit_Bit(Dg_Einzelbit.CurrentCell.RowIndex)
End Sub
```

Bei den Tests wird uns aufgefallen sein, dass nach der Filterung und der Übernahme die erste Assemblervariable zwar vom Format Boolean und auch die Tabelle der Einzelbitbeschreibung sichtbar ist, jedoch keine Daten enthält. Das werden wir nun ändern, indem wir die Routine Show_Einzelbit mit in das Ereignis Bt_UebernahmeClick einbinden. Direkt hinter den Aufruf von

Set_Edit_Variable(0)

```
If Tb_Format.Text = "Byte" Then
    Show_Einzelbit(Val(Tb_Var_ID.Text))
End If
```

### *1.2.5.11 Durchführen von Korrekturen*

Damit sind schon wichtige Voraussetzungen der Funktionalität dieser Seite abgearbeitet. Wie werden wir nun editieren? Nun, eigentlich ist es egal, denn alle Korrekturen müssen mit einem Ereignis verbunden werden können. So sind wir in der Lage, bei einer Änderung diese in der Tabelle auch gleich mit einzubringen. Der schwierigste Part wird wohl das Format sein. Warum? Nun, eine Korrektur von Byte nach Int8, ASCII oder Hex8 ist kein Thema. Das wär einfach, aber da wir auch mit Werten größer 255 in einer Variablen arbeiten wollen, müssen wir den Zugriff auf folgende Zeilen unter Umständen verhindern. Und dafür haben wir die Spalte Aktiv erfunden, die diesen Sperrvermerk mitführt.

Beginnen wir zuerst mit dem zugeordneten Ereignis. Es wird durch ein Combobox-Ereignis SelectedIndexChange ausgelöst. Den Rahmen erhalten wir wieder über die Eigenschaft und Ereignisliste.

```
Private Sub Cb_Format_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Format.SelectedIndexChanged

End Sub
```

Nun zum Inhalt. Es gibt drei Gruppen zu bewerten

Formate mit 8 Bit

Formate mit 16 Bit

Formate mit 32 Bit

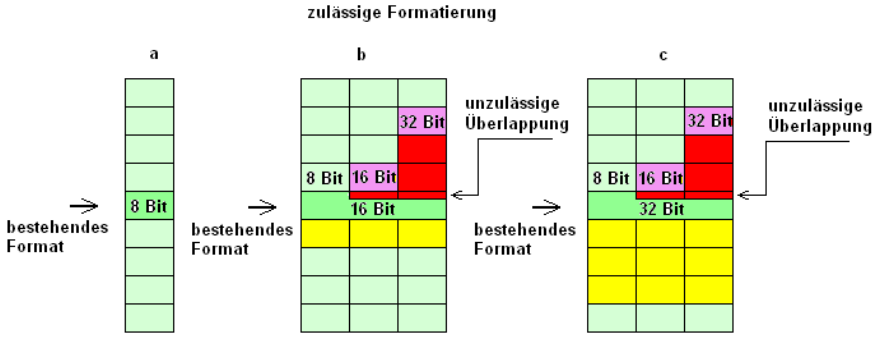
Betrachten wir einmal die Problematik dieser drei Formate bezüglich auf die Korrekturfreigabe.

Bei neu gesetzten 8 Bit ist zu prüfen,, ob das alte Format eventuell 16 Bit oder gar 32 Bit eingeschlossen hat. Dann sind die ehemals gebundenen und gesperrten Tabellenzeilen wieder freizugeben.

Bei neu gesetzten 16 Bit ist ebenfalls eine vorherige 32 Bit Formatbindung aufzulösen und die neuen 16 Bit zu binden. Bestand vorher noch keine Bindung darf die folgende Zeile keine Bindung aufgrund eines vorhandenen 16 Bit oder 32 Bit Formates besitzen. Nur wenn die folgende Zeile freigegeben ist, kann das Format 16 Bit eingerichtet werden.

Ein neues 32 Bit Format ist mit den nächsten 3 zeilen zu verbinden, das bedeutet, keine dieser Zeilen darf gesperrt oder ein Format > 8 Bit besitzen.

Ddazu eine kleine Skizze, die auf die Problematik etwas deutlicher eingeht:




*Freigabe Formatbereiche*

Um mehrere Zeilen bei 16 Bit und 32 Bit Formaten zu korrigieren brauche ich einen unabhängigen Indexzeiger. Ich nenne ihn Zeilen_Nr, obwohl er nichts mit der aktuellen Zeile zu tun hat. Eine weitere Variable mit dem Namen Bereich legt die Zahl der von der Änderung betroffenen Zeilen fest. Für Variablen mit einem Wert >255 definiere ich zwei boolesche Variablen Frei16 und Frei32. Außerdem soll ein Eintrag nur erfolgen, wenn das Format auch geändert wurde und dafür brauche ich den Inhalt vor der Korrektur Old_Format ist aussagekräftig und so werden die Variablen deklariert:

```
Dim Zeile_Nr As Integer
Dim Bereich As Integer          ' Wert für 1-, 2- oder 4- Byte Zahl
Dim Frei16 As Boolean           ' Hilfsvariable für Bereichsprüfung Zweibyte Format
Dim Frei32 As Boolean           ' Hilfsvariable für Bereichsprüfung Vierbyte Format
Dim old_Format As String        ' zur Prüfung, ob manueller Formatwechsel oder von Tabelle
```

Die folgenden Anweisungen betreffen das Vorbesetzen der Variablen und die Entscheidung, ob eine Änderung durchgeführt wurde.

```
Zeile_Nr = Val(Tb_lfd_Nr.Text)  ' aktuelle Zeilennummer
    Bereich = 1                 ' Bereich vorbesetzen für Einbyte-Format
```

```
old_Format = DG_Variablen.Item("CL_Format", Zeile_Nr).Value
If old_Format <> Cb_Format.Text Then


End If
```

Lösen wir nun den Bereich innerhalb der If-Abfrage. Hier ist die Prüfung auf die Variablengröße unterzubringen. Zuerst nehmen wir uns die 16 Bit-Variablen vor. Wenn ein solches Format vorliegt, dann ist der Bereich mit 2 zu besetzen und in der Tabelle die Folgezeile, ob dort eine 8 Bit Variable definiert ist. Bei anderen Variablen würden wir auch wieder zusammenhängende Bytes beeinflussen. Die Vorgehensweise ist wieder die Zuweisung an eine boolesche Variable mit einer Behauptung. **Frei16** ist wahr, wenn die Zelle in der nächsten Zeile das Wort **Byte** enthält.

Die nächste Zeile weist **Frei16** die Behauptung zu, das der Inhalt der Zelle in der nächste Seite das Wort **ASCII** enthält. Da zusätzlich das Ergebnis mit dem vorherigen über eine Oder-Verknüpfung erhalten bleibt, steht am Ende der vier Zuweisungen an **Frei16** nur **True** an, wenn eine Bedingung Wahr ergab.

Schließlich wird der Status von **Frei16** abgefragt und wenn ok, dann die Routine **Chk_Format16** aufgerufen. Die gesamte Bearbeitung hier einzubringen würde den Codeabschnitt unübersichtlich gestalten. Deshalb habe ich mich für eine separate Routine dafür entschieden.

Auch dieser Abschnitt wird mit einer **MsgBox** versehen, wenn die Korrektur abgewiesen wird.

```
        If (Cb_Format.Text = "Int16") Or (Cb_Format.Text = "Hex16") Then
            Bereich = 2              '  Zweibytewert
            Frei16 = Zeile_Nr + 1 < DG_Variablen.Rows.Count - 1
            If Frei16 Then
                Frei16 = DG_Variablen.Item("CL_Format", Zeile_Nr + 1).Value = "Byte"
                Frei16  =  Frei16  Or  (DG_Variablen.Item("CL_Format",  Zeile_Nr  +  1).Value  =
"ASCII")
                Frei16  =  Frei16  Or  (DG_Variablen.Item("CL_Format",  Zeile_Nr  +  1).Value  =
"Int8")
                Frei16  =  Frei16  Or  (DG_Variablen.Item("CL_Format",  Zeile_Nr  +  1).Value  =
"Hex8")
            End If
            If Frei16 Then
                Chk_Format16(Tb_Format.Text, Cb_Format.Text, Zeile_Nr)
            Else
                MsgBox(" Bereich zu klein", MsgBoxStyle.Information, AcceptButton)
            End If
        End If
```

Kommen wir zu den ganz großen Werten, den 32 Bit Variablen. Natürlich betrifft es den Bereich 4 und somit wird diese Variable auch auf 4 gesetzt. Danach ist erst einmal nachzusehen, ob nicht eventuell die Tabellengrenze überschritten wird. Durch die relative Position der aktuellen Zeile zur Tabellengrenze kann dies leicht ermittelt werden. Hierbei wird mit der boolschen Variable **Frei32** nach gleichem Muster wie mit **Frei16** gearbeitet. Logisch ist, das wenn die zweite nachfolgende Zeile frei ist, in der nachfolgenden Zeile keine größere Variable deklariert sein kann. Deshalb ist der erste Vergleich auch auf die Freigabe in der übernächsten Zeile ausgerichtet. Ein weiterer Vergleich ist auf diese Zeile nicht erforderlich, da die nächste Folgezeile ja auch freigegeben sein muss. Damit sind in der Zeile 2 keine größeren Variablen möglich, wenn Zeile 3 freigegeben ist. Allerdings ist in Zeile 3 nach der altuellen wieder der Vergleich auf gößere Formate erforderlich. So kommt hier eine Negation und eine Und_Verküpfung in die Auswertung. Natürlich wäre auch eine Oder-Verknüpfung mit einem **1-Byte** Format möglich gewesen. **Frei32** wird so Vergleich für Vergleich durchgereicht und am Ende ist **Frei32** nur noch Wahr, wenn auch keine weitere größere Variable von der Korrektur betroffen wird. Somit ist die Korrektur erlaubt und die separate Routine **Chk_Format32** kann diese Aufgabe übernehmen. Andernfalls informiert die **MsgBox** über die nicht erfolgte Korrektur.

```
If (Cb_Format.Text = "Hex32") or  (Cb_Format.Text = "Int32") Then
  Bereich = 4              '  Vierbytewert
  Frei32 = Zeile_Nr + 3 < DG_Variablen.Rows.Count - 1
  If Frei32 Then
    Frei32 = (DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 2).Value = "Ja")
    Frei32 = Frei32 And (DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 3).Value = "Ja")
    Frei32 = Frei32 And
             Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value = "Int16")
    Frei32 = Frei32 And
             Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value = "Hex16")
    Frei32 = Frei32 And
             Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value = "Hex32")
    Frei32 = Frei32 And
             Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value = "Int32")

  End If
  If Frei32 Then
   Chk_Format32(Cb_Format.Text, Zeile_Nr)
  Else
     MsgBox(" Bereich zu klein", MsgBoxStyle.Information, AcceptButton)
  End If
End If
```

Bleibt noch das Format mit der Größe von einem **Byte**. Da die Variable Bereich nicht geändert wurde, weil vorher keine Zuweisung erfolgte, darf der Wert 1 als Entscheidung herangezogen werden.

```
    If Bereich = 1 Then
        Chk_Format8(Tb_Format.Text, Cb_Format.Text, Zeile_Nr)
    End If
```

Bauen wir nun die aufgerufenen Routinen **Chk_Format16**, **Chk_Format32** und **Chk_Format8** auf.

Welche Aufgabe müssen diese erledigen. Zuerst kommt in Betracht, das eine 4-Byte Variable in eine 2-Byte oder gar 1-Byte Variable geändert wird. Das bedeutet, es müssen evtl. Zeilen wieder freigegeben werden.

Beginnen wir mit **Chk_Format8**. Dieser Routine teilen wir das alte und neue Format mit sowie die aktuelle Zeilennummer.

Zum Arbeiten braucht diese Routine nochmal eine Bereichsvariable und einen Schleifenzähler. Die Bereichsvariable hier weicht von der Bereichsvariablen der aufrufenden Routine ab, denn sie bezieht sich auf das alte Format, um entsprechend Zeilen wieder freizugeben. Dementsprechend wird sie auch mit der Bewertung vom alten Format besetzt. Nun kann in einer festen Schleife die Freigabe aller gesperrten Zeilen erfolgen. Eine separate Auswertung, ob überhaupt Zeilen freigegeben werden müssen können wir uns sparen. Ist der Bereich 1 wird die Schleife auch nur einmal durchlaufen. Die Zuweisung an die Tabelle mit dem aktuellen Format erfolgt ebenfalls in der Schleife, wobei die freigegebenen Zeilen das Defaultformat von der Filterseite erhalten. Nur die aktuelle Zeile bekommt das ausgewählte Format.

```
Public Sub Chk_Format8(ByVal Old_Format As String, ByVal New_Format As String, ByVal
Zeile_Nr As Integer)
   Dim Bereich As Integer
   Dim i As Integer
   Bereich = 1
   If (Old_Format = "Int16") Or (Old_Format = "Hex16") Then
      Bereich = 2
   End If
   If (Old_Format = "Hex32") or (Old_Format = "Int32") Then
      Bereich = 4
   End If
   For i = 0 To Bereich - 1
     DG_Variablen.Item("CL_Freigabe", Zeile_Nr + i).Value = "Ja"
     If i = 0 Then
       DG_Variablen.Item("CL_Format", Zeile_Nr + i).Value = New_Format
     Else
      DG_Variablen.Item("CL_Format", Zeile_Nr + i).Value = Cb_DefaultFormat.Text
     End If
   Next
End Sub
```

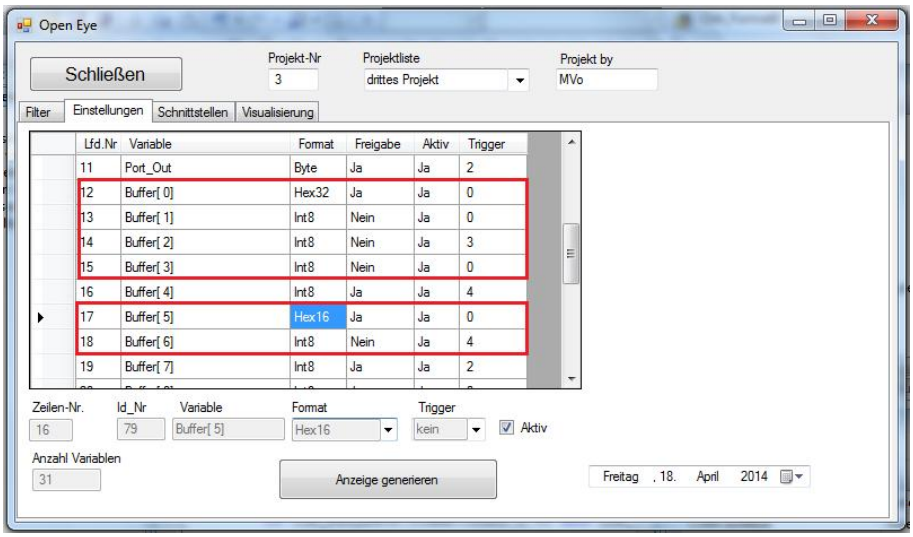
Die nächste Routine wäre dann die **Chk_Format16**. Auch hier kommt eine Freigabe bereits gesperrter Zeilen aufgrund eines alten größeren Formates in Frage. Das kann ein **16 Bit** oder **32 Bit** Format sein. Ist der Bereich größer **16 Bit** werden einfach die zweite und dritte Zeile zur Bearbeitung freigegeben und mit dem Defaultformat besetzt. Einer weiteren speziellen Abfrage bedarf es nicht und die aktuelle Zeile kann mit dem aktuellen Format besetzt und die Folgezeile gesperrt werden..

```
Public Sub Chk_Format16(ByVal Old_Format As String, ByVal New_Format As String, ByVal
Zeile_Nr As Integer)
  If (Old_Format = "Hex32") or (Old_Format = "Int32") Then
    DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 2).Value = "Ja"     ' Zeile freigeben
    Update_Variable(Zeile_Nr + 2)                                   ' für jede Zeile ein Update
    DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 3).Value = "Ja"     ' Zeile freigeben
  End If
  DG_Variablen.Item("CL_Format", Zeile_Nr).Value = New_Format
  DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 1).Value = "Nein"      ' Zeile sperren
End Sub
```

Zuletzt erfolgt die Formatänderung auf einen 4- Byte Wert. Das ist einfach nur die Formatübernahme auf der aktuellen Zeile und das Sperren der 3 folgenden Zeilen. Geschickter Weise erledigen wir das mit einer Schleife.

```
Public Sub Chk_Format32(ByVal New_Format As String, ByVal Zeile_Nr As Integer)
  Dim i As Integer                        ' Schleifenzähler
  DG_Variablen.Item("CL_Format", Zeile_Nr).Value = New_Format
  For i = 1 To 3
    DG_Variablen.Item("CL_Freigabe", Zeile_Nr + i).Value = "Nein"   ' Zeile sperren
  Next
 End Sub
```

Nun sind wieder umfangreiche Tests angesagt, die Fehler in der bisherigen Programmarbeit und Lücken in der Darstellung aufdecken.



*Einfluss Formatierung*

Mit meinem Ergebnis bin ich erst einmal zufrieden. Auch die Rücknahme der Sperrvermerke in den verbundenen Zeilen funktioniert einwandfrei. Ein Ereignis in Verbindung zum Format brauchen wir noch. Die Combobox überträgt zum Schluss ihren Inhalt des Textfeldes in die darüber liegende Textbox. Dazu ein Blick auf die zusammenhängende Ereignisroutine

```
Private Sub Cb_Format_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Format.SelectedIndexChanged
  Dim Zeile_Nr As Integer
  Dim Bereich As Integer          ' Wert für 1-, 2- oder 4- Byte Zahl
  Dim Frei16 As Boolean           ' Hilfsvariable für Bereichsprüfung Zweibyte Format
  Dim Frei32 As Boolean           ' Hilfsvariable für Bereichsprüfung Vierbyte Format
  Dim old_Format As String        ' zur Prüfung, Formatwechsel
   Zeile_Nr = Val(Tb_lfd_Nr.Text)  ' aktuelle Zeilennummer
   Bereich = 1                  ' Bereich vorbesetzen für Einbyte-Format
   old_Format = DG_Variablen.Item("CL_Format", Zeile_Nr).Value
   If old_Format <> Cb_Format.Text Then
    If (Cb_Format.Text = "Int16") Or (Cb_Format.Text = "Hex16") Then
       Bereich = 2              '  Zweibytewert
       Frei16 = Zeile_Nr + 1 < DG_Variablen.Rows.Count - 1
       If Frei16 Then
         Frei16 = DG_Variablen.Item("CL_Format", Zeile_Nr + 1).Value = "Byte"
         Frei16 = Frei16 Or
                (DG_Variablen.Item("CL_Format", Zeile_Nr + 1).Value = "ASCII")
         Frei16 = Frei16 Or (DG_Variablen.Item("CL_Format", Zeile_Nr + 1).Value = "Int8")
         Frei16 = Frei16 Or (DG_Variablen.Item("CL_Format", Zeile_Nr + 1).Value = "Hex8")
       End If
       If Frei16 Then
         Chk_Format16(Tb_Format.Text, Cb_Format.Text, Zeile_Nr)
       Else
         MsgBox(" Bereich zu klein", MsgBoxStyle.Information, AcceptButton)
       End If
     End If
     If (Cb_Format.Text = "Hex32") or (Old_Format = "Int32") Then
       Bereich = 4              '  Vierbytewert
       Frei32 = Zeile_Nr + 3 < DG_Variablen.Rows.Count - 1
       If Frei32 Then
         Frei32 = (DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 2).Value = "Ja")
         Frei32 = Frei32 And (DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 3).Value = "Ja")
         Frei32 = Frei32 And
                Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value = "Int16")
         Frei32 = Frei32 And
                Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value = "Hex16")
         Frei32 = Frei32 And
                Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value = "Hex32")
         Frei32 = Frei32 And
                Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value = "Int32")
       End If
       If Frei32 Then
         Chk_Format32(Cb_Format.Text, Zeile_Nr)
       Else
```

```
            MsgBox(" Bereich zu klein", MsgBoxStyle.Information, AcceptButton)
          End If
        End If
        If Bereich = 1 Then
          Chk_Format8(Tb_Format.Text, Cb_Format.Text, Zeile_Nr)
      End If
    End If
      Tb_Format.Text = Cb_Format.Text
  End Sub
```

Mit dem Ereignis der Textbox **Tb_TextChanged** können wir nun einfach die Einzelbittabelle einblenden.

```
Private Sub Tb_Format_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Tb_Format.TextChanged
  If Tb_Format.Text = "Byte" Then
     Pn_Bitfeld.Visible = True
     Show_Einzelbit(Val(Tb_Var_ID.Text))
   Else
      Pn_Bitfeld.Visible = False
   End If
End Sub
```

Die anderen editierbaren Tabelleneinträge sind einfach korrigierbar. Der **Trigger** wird einfach aus der Combobox gewählt und eingetragen. Dabei ist die Nummer des Eintrags, der **IndexOf** vom Eintrag im Tabellenfeld. Das Ereignis ist einfach aufzubauen du abzuarbeiten.

```
Private Sub CB_Trigger_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles CB_Trigger.TextChanged
    Dim Zeile_Nr As Integer
    Dim Trigger As Integer
    Zeile_Nr = Val(Tb_lfd_Nr.Text)
    Tb_Trigger.Text = CB_Trigger.Text
    Trigger = CB_Trigger.Items.IndexOf(CB_Trigger.Text)
    DG_Variablen.Item("CL_Trigger", Zeile_Nr).Value = Str(Trigger)

  End Sub
```

## Die Checkbox **Aktiv** ist auch ein Kinderspiel

```
Private Sub CB_Aktiv_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles CB_Aktiv.CheckedChanged
   Dim Zeile_Nr As Integer
   Zeile_Nr = Val(Tb_lfd_Nr.Text)
   If CB_Aktiv.Checked Then
       DG_Variablen.Item("CL_Aktiv", Zeile_Nr).Value = "Ja"
   Else
       DG_Variablen.Item("CL_Aktiv", Zeile_Nr).Value = "Nein"
   End If
 End Sub
```

### *1.2.5.12 Einzelbitbeschreibung ändern oder hinzufügen*

Fehlt nun noch das Editieren der Einzelbitinformation. Dabei ist zu beachten, das eine Korrektur, soll sie einfach durch die Änderung im Textfeld auch gleich übernommen werden, nicht im Ereignis **TextChanged** bearbeitet werden sollte. Da das Feld beschrieben wird, ist mit der Eingabe eines jeden Zeichens ein **TextChanged**-Ereignis gegeben. Hier brauchen wir aber eine vollständige Eingabe der Korrektur, bevor die Änderung an die Tabellen zurückgeführt wird. Ja, es ist genau genommen von zwei Tabellen die Rede. Einmal die visualisierte **Dg_Einzelbit** und zum anderen die Quelle der Daten. Die unsichtbare Tabelle **Dg_All_Bits**. In beiden Tabellen muss die Korrektur übernommen werden. Doch immer noch sind wir auf der Suche nach einem geeigneten Ereignis.

Sehen wir uns unter den Eigenschaften der Textbox **TB_Funktion** einmal alle Ereignisse an. Alle Drag- und Mouse-Ereignisse können wir gleich ausschließen, denn die Texteingabe wird mit der Tastatur erledigt. Gut, wir könnten grundsätzlich die Eingabe mit Enter bestätigen und dabei das **Keypress**-Ereignis anwenden. Dann muss aber auch ein Tabulator, also den Focus auf das nächste Objekt leiten, abgefragt werden. Doch, wenn wir mit einer Korrektur fertig sind, was tun wir grundsätzlich als nächstes? Wir wechseln auf ein anderes Objekt. Ob eine weitere Eingabe an anderer Stelle, oder der nächste Tabelleneintrag gewählt oder gar die Seite gewechselt wird, immer wird zuerst der Focus vom Eingabeelement **Tb_Funktion** genommen. Erst muss dieses Objekt verlassen werden, bevor ein anderes Objekt aktiv werden kann. Das Ereignis **Leave** scheint da die richtige Wahl.. Einen Versuch wäre es ja wert.

Immer, wenn man sich nicht sicher ist,. Ob ein Ereignis die Aufgabe erledigen kann, erzeugt man es, schreibt eine wilde Zuweisung hinein und setzt einen Haltepunkt.

```
Private Sub TB_Autor_Leave(ByVal sender As System.Object, ByVal e As
    Dim test As Integer
    test = 99
End Sub
```

*Haltepunkt Test*

Anschließend wird das Programm gestartet und das Ereignis ausgelöst. Das Programm wird dann, an dieser Stelle angehalten und so kann überprüft werden, ob die Wahl erfolgreich war.

Für unsere Aufgabe, die Korrektur der Funktionsbeschreibung eines Bits automatisch auch abzulegen, war die Wahl des Ereignis **Leave** richtig und nun können wir die Schritte programmieren.

Zuerst wieder ein kleiner Gedankensturm: welche lokalen Variablen werden benötigt?

Da ist die Referenz zur Tabelle **Dg_All_Bits**, die in der Tabelle **Dg_Einzelbit** mitgeführt sein sollte. Mit **Id_Nr** kann eine Variable deklariert werden, die diese Referenz führt. Auch die Zeilennummer der Tabelle **Dg_Einzelbit** ist vielleicht wichtig, also auch eine Variable **Zeile_Nr** wird eingetragen. Die Zeilennummer, sie entspricht der Bitnummer, wird aus dem Inhalt der Textbox über eine Stringfunktion **Mid(<Text>,<Anfangsposition>,<Anzahl>)** herauskopiert. Daraufhin ist die **Id_Nr**, also die Nummer, unter der dieses Bit in der Tabelle **Dg_All_Bits** gelistet ist, aus der selektierten Zeile abrufbar. Nun wird auch der Inhalt der Textbox mit der neuen Funktionsbeschreibung in die Tabelle **Dg_Einzelbit** übertragen. Das ist aber nur ein Teil der Aufgabe. Schließlich sollen diese Korrekturen auch erhalten bleiben und wir erinnern uns: wird ein Format auf **Byte** geändert, ist in der Tabelle **Dg_All_Bits** noch gar keine Information hinterlegt. Lediglich die Tabelle **Dg_Einzelbit** wird für alle acht Bits mit einer Standartbeschreibung vorbesetzt. Und da es noch keine Referenz auf die Tabelle **Dg_All_Bits** gibt, wird dort eine -1 eingetragen. Das können wir im nächsten Schritt auswerten. Wenn die Referenz < 0 ist, dann müssen auf der Tabelle **Dg_All_Bits** die Bitbeschreibungen hinzugefügt und die Referenzen eingetragen werden. Das erledigt am besten eine Funktion, die dann die neue Referenz zurückliefert. Nun kann dort auch der neue Funktionstext hinterlegt werden und die neu erstellten Bitbeschreibungen haben eine Referenz auf die Basistabelle.

```
Private Sub TB_Funktion_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TB_Funktion.Leave
  Dim Zeile_Nr As Integer
  Dim Id_Nr As Integer
  Zeile Nr = Val(Mid(TB Bit.Text, 5, 1))
  Id Nr = Val(Dg Einzelbit.Item("CL BitId Nr", Zeile Nr).Value)
  Dg_Einzelbit.Item("CL_Funktion", Zeile_Nr).Value
                              = TB_Funktion.Text
  Id_Nr = Dg_Einzelbit.Item("CL_BitId_Nr", Zeile_Nr).Value
  If Id_Nr < 0 Then
     Zeile Nr = Add New Rows(Val(Mid(TB Bit.Text, 5, 1)))
  End If
  Dg_All_Bits.Item("CL_BitFunktion", Zeile_Nr).Value
          = TB_Funktion.Text
End Sub
```

Allerdings fehlt noch der Aufbau der Funktion **Add_New_Row**, bevor wir auch diesen Programmabschnitt testen können. Rufen wir uns noch einmal die Aufgabe einer Funktion ins Gedächtnis:

Eine Funktion liefert einen Wert zurück.

In unserem Fall soll sie die Zeilennummer der **Dg_All_Bits** zurückliefern, da er nicht auf der Tabelle **Dg_Einzelbit** verfügbar war. Sie muss für diesen Zweck auch neue Zeilen an die Tabelle anhängen und neue Referenzen zur Tabelle **Dg_Einzelbit** herstellen. Natürlich könnte diese Aufgabe auch eine Subroutine übernehmen, aber so können wir einfach anschließend die allgemeine Anweisung zum Eintrag des neuen Funktionstextes in die Tabelle eintragen.

Eine Function wird wie eine Sub programmiert, nur das der Deklarationstext etwas abweicht. So wird auch zuerst der Rumpf verfasst und da der zurückgelieferte Wert vom Typ Integer sein muss, das Resultat auch so deklariert. Als Parameter liefern wir die altuelle Bitnummer mit, damit auch die Referenz in den neuen Tabelleneinträgen zurückgeliefert werden kann. Nun gibt eis zwei Werte, die in lokalen Variablen für die Bearbeitung dieser Funktion abgelegt werden. Eine ist die bekannte Schleifenvariable i, denn immerhin müssen acht neue Zeilen zur Tabelle **Dg_All_Bits** hinzugefügt werden, und ein Wert ist der Index der letzten Zeile vor der Bearbeitung. Den Wert können wir auch gleich zuweisen. Er wird aus der Anzahl der bestehenden Zeilen ermittelt minus 1, da die Zählung bei 0 beginnt.

```
Public Function Add_New_Rows(ByVal Bit_Nr As Integer) As Integer
  Dim i As Integer                                  ' Variable für Schleifenzähler
  Dim Last_Index As Integer                         ' Variable für letzen Tabellenindex
  Last_Index = Dg_All_Bits.Rows.Count - 1           ' letzenTabellenindex holen
End Function
```

Der Schleifenaufbau ist einfach. Die Aufgabe ist es eine neue Zeile an die Tabelle **Dg_All_Bits** zu hängen und mit Vorgabewerten zu besetzen. Zusätzlich ist in der Tabelle **Dg_Einzelbit** die Referenz einzutragen.

```
 For i = 0 To 7                                     ' nicht "-1, weil I schon mit 0 beginnt
   Dg_All_Bits.Rows.Add()
      ' Referenzindex setzen
   Dg_All_Bits.Item("CL_Bit_Id", Last_Index + i).Value = Last_Index + i + 1
```

```
        ' noch nicht relevant
    Dg_All_Bits.Item("CL_Proj_Id", Last_Index + i).Value = 0
      ' Referenzindex der Variablen
    Dg_All_Bits.Item("CL_Var_Id", Last_Index + i).Value = Val(Tb_Var_ID.Text)
        ' Bitnummer eintragen
    Dg_All_Bits.Item("CL_BitNr", Last_Index + i).Value = "Bit" + Str(i)
        ' Kommentar eintragen
    Dg_All_Bits.Item("CL_BitFunktion", Last_Index + i).Value = "nicht verwendet"
            ' Referenz zur Einzelbittabelle
    Dg_Einzelbit.Item("CL_BitId_Nr", i).Value = Last_Index + i + 1
  Next i
```

Schließlich erfolgt die Rückgabe der neuen referenzierten Zeile für diese Änderung. Dafür haben wir mit **Last_Index** und dem Übergabeparameter die exakte Position in der Tabelle **Dg_All_Bits** für den Korrektureintrag.

```
    Return (Last_Index + Bit_Nr)
```

Noch einmal die gesamte Funktion

```
Public Function Add_New_Rows(ByVal Bit_Nr As Integer) As Integer
   Dim i As Integer                                  ' Variable für Schleifenzähler
   Dim Last_Index As Integer                         ' Variable für letzen Tabellenindex
   Last_Index = Dg_All_Bits.Rows.Count - 1           ' letzenTabellenindex holen
   For i = 0 To 7                                    ' nicht "-1, weil i mit 0 beginnt
      Dg_All_Bits.Rows.Add()
       ' Referenzindex setzen
      Dg_All_Bits.Item("CL_Bit_Id", Last_Index + i).Value = Last_Index + i + 1
        ' noch nicht relevant
      Dg_All_Bits.Item("CL_Proj_Id", Last_Index + i).Value = 0
       ' Referenzindex der Variablen
      Dg_All_Bits.Item("CL_Var_Id", Last_Index + i).Value = Val(Tb_Var_ID.Text)
       ' Bitnummer eintragen
      Dg_All_Bits.Item("CL_BitNr", Last_Index + i).Value = "Bit" + Str(i)
       ' Kommentar eintragen
      Dg_All_Bits.Item("CL_BitFunktion", Last_Index + i).Value = "nicht verwendet
        ' Referenz zur Einzelbittabelle
      Dg_Einzelbit.Item("CL_BitId_Nr", i).Value = Last_Index + i + 1
    Next i
    Return (Last_Index + Bit_Nr)                     ' neue Referenz für Zeilen_Nr
 End Function
```

Nun können wir das Ergebnis testen.

## Hinweis zum Programmaufbau

Unser Programmlisting ist schon ganz schön gewachsen und so langsam muss man schon ganz schön blättern, wenn man eine Programmstelle

sucht. Ein paar Möglichkeiten, die Übersicht nicht zu verlieren gibt es. Man muss sich dabei nur an Regeln halten. Diese stellt man letztlich selbst auf. Auch ich habe meine Methode, die ich auch gern zur Anwendung weitergebe.

*Programmlisting:*

### *Globale Variablen*

Möglichst keine globalen Variablen. Wenn doch welche benötigt werden, kommen diese ganz vorn in das Listing.

Beginnend mit “Private” gefolgt von “Public”
(Ist bei dieser Anwendung nicht relevant.)

### *Funktionen*

Dann folgen selbstverfasste Funktionen.
Beginnend mit “Private” gefolgt von “Public”

### *Unterprogramme*

Im zweiten Block sind die eigenen Subroutinen
Beginnend mit “Private” gefolgt von “Public”

### *Größe von Programmblöcken*

Alle Programmbausteine sollten auf eine DinA4 Seite passen. Das erleichtert die Arbeit, wenn man Paperware von getesteten und abgeschlossenen Programmroutinen für die Dokumentation erstellt und bei der Programmierung zur Hand nimmt.
Ein wenig Erziehung ist auch dabei, denn man verhindert so Programmabschnitte, die kaum noch zu überblicken sind. So ist auch irgendwann genügend Übung vorhanden, zusammenhängende Blöcke zu erkennen und auszulagern. Beim Filter wurde diese Vorgehensweise angewendet.

### *Namensvergabe Objekte*

Objekte haben von Hause aus einen Namen und werden bei der Installation mit einer laufenden Nummer gekennzeichnet, So sind die Namen von 5 Installierten Textboxen **Textbox1** bis **Textbox5**. Das ist nicht sehr hilfreich, wenn im Programm einer Textbox ein Inhalt zugewiesen werden soll. Daher verwende ich eine Abkürzung des Objektnamens z.B. bei der TextBox **Tb**, bei einer Combobox **Cb** und bei einem Button **Bt** gefolgt von einem Unterstrich. Dann folgt ein bezeichnender Name, z.B. Projekt. So ergeben sich Name wie **Tb_Projekt**, **Cb_Formate**, **Dg_All_Bits** usw.

Bei Labels, die zur Überschrift dienen verzichte ich auf eine Namensvergabe, ebenfalls bei Panels, die nur der Optik dienen. Werden diese Objekte jedoch vom Programm beeinflusst, z. B. über die Eigenschaft Visible oder bei Änderung einer Überschrift durch eine Textanweisung an ein Label, bekommen diese Objekte selbstverständlich einen bezeichnenden Namen. So ist der Zugriff im Programm einfach.

### *Kommentarzeilen*

Kommentarzeilen sind nicht nur zur Kommentierung einzelner Zeilen verwendbar, sondern sie können auch zur optischen Abgrenzung von Programmabschnitten und zur ausführlichen Kommentierung von Unterprogrammen benutzt werden. Dadurch wird das Programm nicht größer oder gar langsamer. Sie sind nur Text, der vom Compiler komplett ignoriert wird. Die Kommentare in den folgenden Beispielen helfen, die Übersicht zu behalten

```
'****************  Abschnitt globaler Variablen **********************


'*************************************************************
'*                    Abschnitt Funktionen                   *
'*************************************************************

'****************    Abschnitt Unterprogramme    ********************

'********************    Abschnitt Ereignisse    **********************
```

```
'*****************************************************************
'* Subroutine ist für einen Variablenfiler              *
'* eines C-Controllerprogrammes vorgesehen    *
'*****************************************************************
Public Sub Filter_C()

End Sub
```

## Zwischenbilanz der ersten Programmierung

Ziehen wir einmal eine Zwischenbilanz:

Wir können Variablennamen aus einem kopierten Bereich des Assemblerlisting herausfiltern.

Wir können mit Tabellen arbeiten und Inhalte editieren

Wir können Formate verändern und mehrere Bytes zu großen Zahlenbereichen verbinden.

Wir können Meldungen zusammenstellen und Informationen zum Programmverlauf abgeben

Wir sind in der Lage, ganze Objektgruppen mit nur einem Befehl aus- oder einzublenden.

Wir können die Funktionsbeschreibung einzelner Bits ändern, ja sogar neue Einzelbits erfassen und deren Funktion beschreiben

Aber es ist schon nervig, jedes Mal von vorn anzufangen, wenn wir den Text aus Abschnitt 2.2.2.7 durch den Filter jagen.

Nun, er sollte ja auch nur zu Testzwecken dienen. Jetzt ist es an der Zeit, die Ergebnisse auch abzuspeichern. Sicherlich ist eine Datenbank nicht unbedingt erforderlich, aber sie erlaubt gezielte Zugriffe auf einzelne Datensätze und uns schadet es nicht, wenn wir etwas über die SQL-Anweisungen lernen, die uns die gewünschte Information gezielt von einer Datenbank holt.

## 1.3 Die Datenbank

### Beschreibung

Eine Datenbank ist eine Ablage der Daten in Tabellen. Wer bereits mit Tabellenprogrammen gearbeitet hat, weiß, dass die Daten in Zeilen stehen, deren Spalten eine Überschrift tragen. Ganz links ist in der Regel eine Zeilennummer. Auf diese Weise lassen sich Informationen leicht und übersichtlich ablegen. Es ist sogar möglich, Werte mit hinterlegten Formeln aus verschiedenen Feldern zu berechnen. Nicht selten kalkulieren Firmen damit ihre Einkaufslisten. Selbst wissenschaftliche Anwendungen sind denkbar, um verschiedene Messreihen mit simulierten Vorgaben zu prüfen. Nähere Details sind aber hier nicht von Bedeutung. Lediglich die Tabelle und ihr Aufbau bleibt relevant.

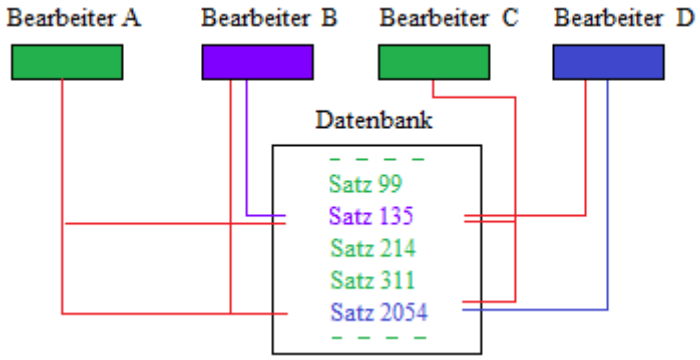
Wenn in einer Tabelle unter Angabe der Zeilennummer ein Datensatz genannt wird, fällt es uns nicht schwer, die zugehörigen Daten zu bestimmen. Allerdings müssen wir die Tabelle durchblättern und die Daten manuell erfassen. Wenn nun in einer Spalte eine Zahl steht, die eine Lieferanten - Nummer darstellt und auch der Lieferanten ermittelt werden soll, so ist eine Tabelle mit Namen, Adressen, Kontaktpersonen etc. der verschiedenen Firmen die richtige Ablage. Dort steht dann in Zeile (<Lieferanten – Nummer>) die Firma mit allen erforderlichen Daten. Auch das ist manuell überhaupt kein Problem und lässt sich mit Tabellen leicht umsetzen.

Eine Datenbank nimmt uns diese Arbeit ab. Wenn unser Auftrag heißt: „Bestelle 100 Transistoren (xyz)“ so wird der Artikel in der Artikeltabelle herausgesucht und bei entsprechender Firma bestellt. Vielleicht noch telefonisch bei der Kontaktperson über den Preis verhandelt oder Lieferfristen abgefragt. Bei der Datenbank wird ebenfalls ein Auftrag erstellt. Die Bezeichnung dafür ist „SQL-Abfrage“ oder auch „SQL-Anweisung“. Mit diesen Anweisungen werden die Daten in den Tabellen herausgesucht und einem Programm zur Verfügung gestellt. Allerdings kann eine Datenbank wesentlich mehr. So können unerwünschte Zugriffe durch Passwortabfrage geschützt werden. Auch ist der Zugriff durch verschiedene Programme und mehrere Anwendungen möglich.

Am Beispiel Reisebüro wird die daraus entstehende Problematik deutlich:

Alle Reisebüros weltweit haben Zugriff auf Angebote verschiedener Unternehmen. Ein Kunde interessiert sich irgendwo für ein

Urlaubsangebot. Damit kein anderes Reisebüro auf dieses Angebot zugreifen kann, muss es, solange der Kunde in der Beratung ist, gesperrt werden. Erst nach Abschluss wird der Zugriff beendet. Nun kann die Reise anderweitig vermittelt werden oder ist durch den Kunden gebucht und nicht mehr verfügbar. Die Grafik zeigt mit den roten Linien, dass ein Zugriff auf die Datensätze nicht möglich ist. Dem Bearbeiter bleiben sie




verborgen oder werden als „reserviert“ angezeigt.

*Datenbankzugriff mehrerer Anwender*

In unserer Applikation ist die Datenbank überschaubar. Auf Schutzmechanismen kann vollständig verzichtet werden, es sei denn, eure Projekte sind „geheim“ oder ihr arbeitet mit verschiedenen Personen an Projekten, die ihre Daten in eurer Datenbank ablegen. Dies ist hier nicht berücksichtigt und alle diesbezüglichen Maßnahmen sind nicht Bestandteil in diesem Abschnitt.

### *1.3.1 Datenbanktabellen*

Hat man den Mechanismus der Datenbank erst einmal erkannt, wird man die Arbeit mit Tabellen nicht mehr missen wollen. Es ist auch nicht sonderlich schwer, die erforderlichen Abfragen zu erstellen. Doch beginnen wir erst einmal mit einigen grundsätzlichen Begriffen.

Tabellen: Hier werden Informationen hinterlegt. Dazu ein ziemlich simples Beispiel:

10 kg Äpfel kosten 8,00 €, 4 kg Birnen kosten 5,00 €, 700 gr Tomaten kosten 1,50 €, 4 to Kies etc

So könnte ein Einkaufszettel aussehen.

Damit ich es mir merken kann, fertige ich eine Tabelle mit der Beschreibung der Artikel an. Die Tabelle nenne ich einfach Liste

| Gewicht | Einheit | Artikel | Preis | Währung |
|---|---|---|---|---|
| 10 | Kg | Äpfel | 8,00 | € |
| 4 | Kg | Birnen | 5,00 | € |
| 700 | Gr | Tomaten | 1,50 | € |
| 4 | To | Kies | 127,65 | € |

Stellt euch diese Liste mit weit über tausend Einträgen vor. Nun wollt ihr wissen, wie viel Kilo Äpfel eingekauft wurden. Wie geht man da vor? Vermutlich wird jeder die Antwort wissen. Klar, man sucht einfach alle Zeilen, in denen in der Spalte „Artikel“ Äpfel eingetragen sind und rechnet alle Zahlen aus der Spalte „Gewichte“ unter Beachtung der Einträge in der Spalte „Einheit“ zusammen.

So kann auch ermittelt werden, wie viel für einen Artikel ausgegeben wurde. Eine Datenbankabfrage funktioniert genauso. Der Abfrageauftrag wird mit SQL-Anweisungen einfach an die Datenbank geschickt und als Ergebnis bekommt man dann eine Liste der mit der gewünschten Information. Nun brauchen wir nicht gleich Panik zu bekommen, wenn es um eine neue Art von Befehlen geht. SQL-Anweisungen sind gar nicht so kompliziert, wie es vielleicht den Anschein hat.

### *1.3.2 Die SQL-Anweisung*

Wie immer ist hier Englisch die Programmiersprache. Das Wort Select leitet eine Datenbankabfrage für eine Auswahl bestimmter Inhalte ein. Setzen wir einen Platzhalter „*“ ein, wird der Inhalt jeder Spalte geliefert. Es ist aber auch möglich, nur Inhalte bestimmter Spalten auszuwählen. Dies wird aber später, wenn wir eine SQL-Anweisung schreiben, sehr schnell deutlich.

Die Anweisung

```
Select * from Liste
```

liefert alle Datensätze der Tabelle **Liste** Nun wollen wir aber nur den Artikel *Äpfel* haben. Dann müssen wir der Datenbank mitteilen, dass wir nur an den Einträgen *Äpfel* in der Spalte **Artikel** interessiert sind.

Diese Anweisung lautet.

```
Select * from Liste Where Artikel =“Äpfel“
```

Ist doch gar nicht so schwer und mit etwas Grundwissen in Englisch auch verständlich. Nun ist eine Datenbank auch mit Inhalten gefüllt, die veränderbar sind. Der gezielte Zugriff auf eine bestimmte Zeile ist mit den gegebenen Spalten nicht möglich. Keine der Information ist so eindeutig, das ein Schreibbefehl auf der Datenbank durchgeführt werden könnte. Zeilennummern würden sich anbieten, aber auch wenn es auf den ersten Blick die Lösung scheint, funktioniert das nicht. Sobald ein Eintrag gelöscht wird, verändern sich die Zeilennummern. Das ist bei der Verwendung einer einzigen Tabelle kein Problem, aber dann könnten wir unsere Information auch in Files ablegen.

Wir möchten aber auch den Lieferanten des Artikels erfahren. Nun hat ein Lieferant nicht nur einen Namen, sondern auch eine Adresse und Kontaktdaten. Diese bei jedem Artikel mitzuführen ist Speicherplatzverschwendung. Wenn 10 mal der Eintrag *Äpfel* gefunden würde, gäbe es auch 10 mal die gleiche Information zum Lieferanten. Wir können aber eine Tabelle mit den Daten der Lieferanten schreiben und die Zeilennummer dann beim Artikel mitführen. Das wäre nur eine einzige Spalte mehr. Gute Idee, aber von der Zeilennummer müssen wir uns dann verabschieden. Statt dessen bekommt die Tabelle eine Spalte für eine

einzigartige unveränderbare **ID-Nummer**. Und nun ist die Tabellenzeile eindeutig identifizierbar.

Im Laufe dieser Datenbankbehandlung werden wir auch noch zu SQL-Anweisungen kommen, die Einträge hinzufügen, ändern und löschen.

### *1.3.3 Tabellenaufbau*

Hier hilft zuerst eine Skizze, die diese Zusammenhänge verdeutlicht.

Artikeltabelle

| Id_Nr | Artikel-Nr | Artikelbeschreibung | Menge | Einheit | Preis | Lieferant |
|---|---|---|---|---|---|---|
| 1 | 4 | Schrauben | 12 | Stk | 0,23 | 4 |
| 4 | 8 | Nägel | 234 | Stk | 1,33 | 33 |
| 7 | 4 | Schrauben | 100 | Stk | 2,53 | 4 |
| 12 | 2 | Haken | 233 | Stk | 2,10 | 65 |
| 22 | 6 | Dübel | 154 | Stk | 1,23 | 38 |

Id-Nr. von Lieferantentabelle ist Referenz

Lieferantentabelle

| Id_Nr | Name | Firma | PLZ | Ort | Strasse | Tel | FAX | Kontakt |
|---|---|---|---|---|---|---|---|---|
| 4 | ABC | ABC | 321 | A | Weg | 34 | 34 | Meier |
| 33 | CDE | CDE | 543 | B | Platz | 46 | 46 | Pelz |
| 38 | EFG | EFG | 123 | C | Allee | 66 | 66 | Müller |
| 65 | HIJ | HIJ | 222 | D | Markt | 73 | 73 | Wolf |

*Tabellenbezüge*

Auf einer Datenbank sind die Tabellen noch wesentlich tiefer miteinander verbunden, doch für unsere Zwecke sollte diese Darstellung erst einmal ausreichen. Wir wollen ja keine Handelskette eröffnen. Unsere Daten sind etwas anders geartet. Aber Tabellen brauchen wir auch. Denken wir einmal kurz nach. Sicherlich werden wir viele Controller programmieren wollen und verschiedene Projekte umsetzen. Für jedes Projekt gibt es:

Einen Namen
Einen Controller
Eine Variablenliste
eine Parametereinstellung
Skizzen, Bilder und Schaltpläne
eine Bauteileliste

Auch wenn uns noch etwas einfallen sollte, können wir es hier mit auflisten.

Beginnen wir mit dem Projekt.

Da gibt es den Namen, den Controller und die Parameter, die für ein Projekt nur einmal angegeben werden müssen

Eine Tabelle mit den Variablennamen mit der Referenz auf das Projekt. Hier stehen alle Variablen, auch von anderen Projekten

Eine Tabelle für die Darstellung von Einzelbits. Diese Tabelle enthält eine Auflistung der Bits unter Verwendung einer Referenz auf die Variable. Die Identifizierung erfolgt durch die Variablen-ID. Das Projekt wird ebenfalls als Referenz mit eingebunden. Das erlaubt, die komplette Liste der Kommentare für die Bits zum Projekt zu laden.

Eine Tabelle für alle Bilder, Skizzen und Schaltpläne mit Referenz zum Projekt

Option:

Tabelle für Bauteile

Tabellen für Stücklisten mit Referenz zum Projekt und zu Bauteilen

Diese optionalen Tabellen lasse ich erst einmal weg und komme später darauf zurück. Bauteile erfassen und Listen zusammenstellen ist zwar ein interessanter Bereich, erfordert allerdings viel akribische Handarbeit. Trotzdem kann es sinnvoll sein, wenn man bedenkt, dass sogar eine Kostenverfolgung der Projekte mit solchen Tabellen problemlos möglich ist.

Auf den nächsten Seiten werden wir die für unser Projekt erforderlichen Tabellen entwerfen und anschließend die Datenbank erstellen.

### *1.3.3.1 Vier Grundtabellen*

Der Aufbau beginnt bei allen Tabellen mit einer Spalte **Id_Nr** (Ganzzahl)

Die Projekttabelle wird mit den folgenden Spalten ergänzt:

zweite Spalte ist Projektname (Text)
Spalte 3 für verwendete Controller (Text)
Spalte 4 für Angabe der Taktfrequenz (Text)
Nun die Parameter für die Schnittstelle
Spalte 5 für die Baudrate (Ganzzahl)
Spalte 6 Datenbit, (Ganzzahl)
Spalte 7 Stoppbit (Ganzzahl)
Spalte 8 Parity (Text)
Nun spezielle Übertragungsinformationen
Spalte 9 Übertragungskopf (Telegrammkopf) (Text)
Spalte 10 Auftrag (Text)
Noch ein paar Projektparameter
Elfte Spalte für die Sprachauswahl (optional) (Text)
Spalte 12. und 13. für Schreib- und Lesepuffergröße (Ganzzahl)
Schließlich noch Spalte 14 für Autor (Text)
und Spalte 16 für Datum (Text)

Nun die Tabelle für die Variablen:

Spalte 2 ist für die Referenz zur Projekttabelle (Ganzzahl)
Spalte 3 ist für den Variablennamen (Text)
Spalte 4 für das Format (Text)
Spalte 5 für die Freigabe (Text)
Spalte 6 für den Vermerk Aktiv (Text)
Spalte 7 für den Trigger (Ganzzahl)

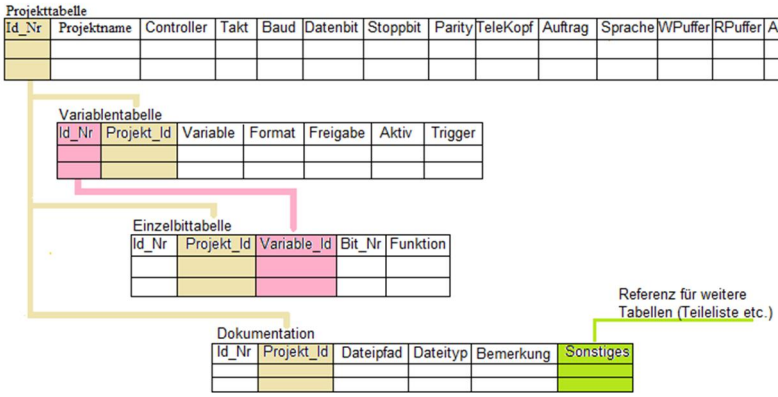
Die dritte Tabelle ist die Tabelle für die Einzelbitbeschreibung

Auch hier zuerst die Referenzfelder
Spalte 2 für Referenz zum Projekt (Ganzzahl)
Spalte 3 für Referenz zur Variablen (Ganzzahl)
Spalte 4 für die Bitnummer (Ganzzahl)
und Spalte 5 für die Funktionsbeschreibung (Text)

die vierte Tabelle ist für Zeichnungen, Bilder und Skizzen

Spalte 2 für die Referenz zum Projekt (Ganzzahl)
Spalte 3 für den Dateipfad des Bildes oder der Skizze (Text)
Spalte 4 für den Dateityp (Ganzzahl)
Spalte 5 für eine Bemerkung und (Text)
Spalte 6 für sonstiges (Ganzzahl)

Ein Blick auf die Tabellendarstellung mit farblich markierter Tabellenbeziehung verdeutlicht das Zusammenspiel.



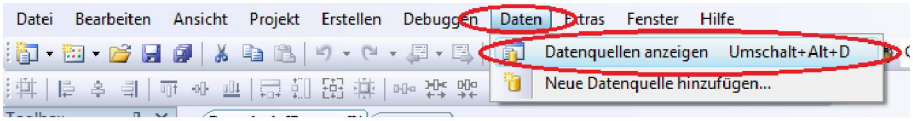

*Zusammenhänge bei Tabellenbindung*

Damit sollte man in der Lage sein, weitere Tabellen über Referenzfelder einzugliedern.

### *1.3.4 Datenbank erstellen*

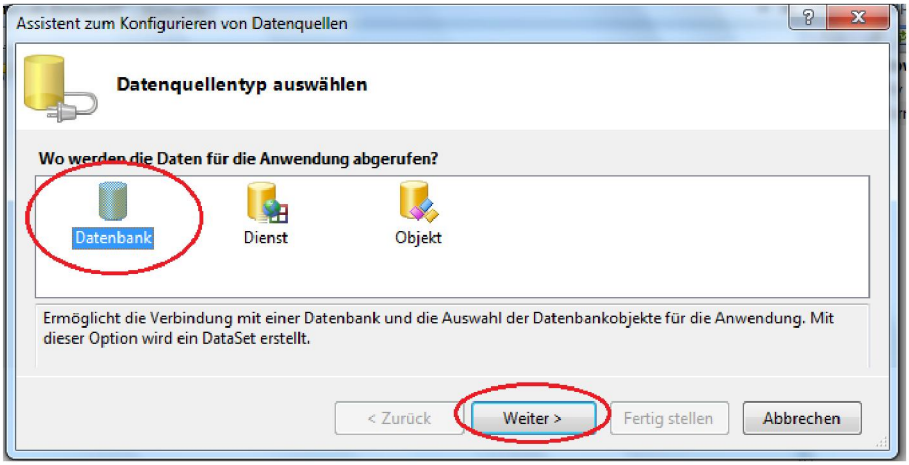
Soweit die erforderlichen Grundlagen zum Verständnis der Datenbankstruktur und so können wir mit der Gestaltung der Datenbank beginnen.

Die Vorgehensweise ist in einer Folge von Screenshots abgebildet. Der Vorgang beginnt mit der Erstellung der Datenbank durch die Anwahl **Daten** in der Menüleiste und dem Click auf den Eintrag **neue Datenquelle hinzufügen**.




*Datenbank erstellen*

Es öffnet sich der Assistent zur Auswahl des Datenquellentyps. Die

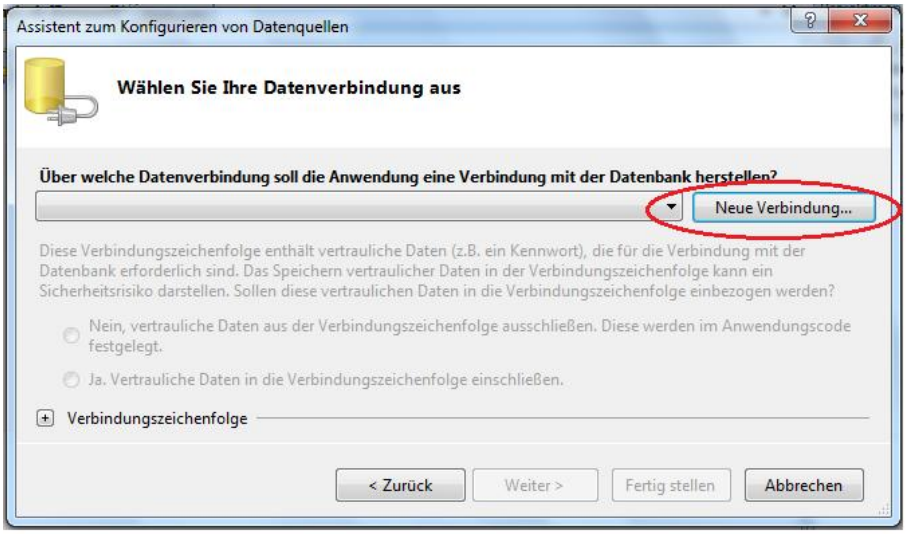



*Datenquelle generieren*

Auswahl fällt auf **Datenbank**.

Mit dem Button **Weiter** wird fortgesetzt.

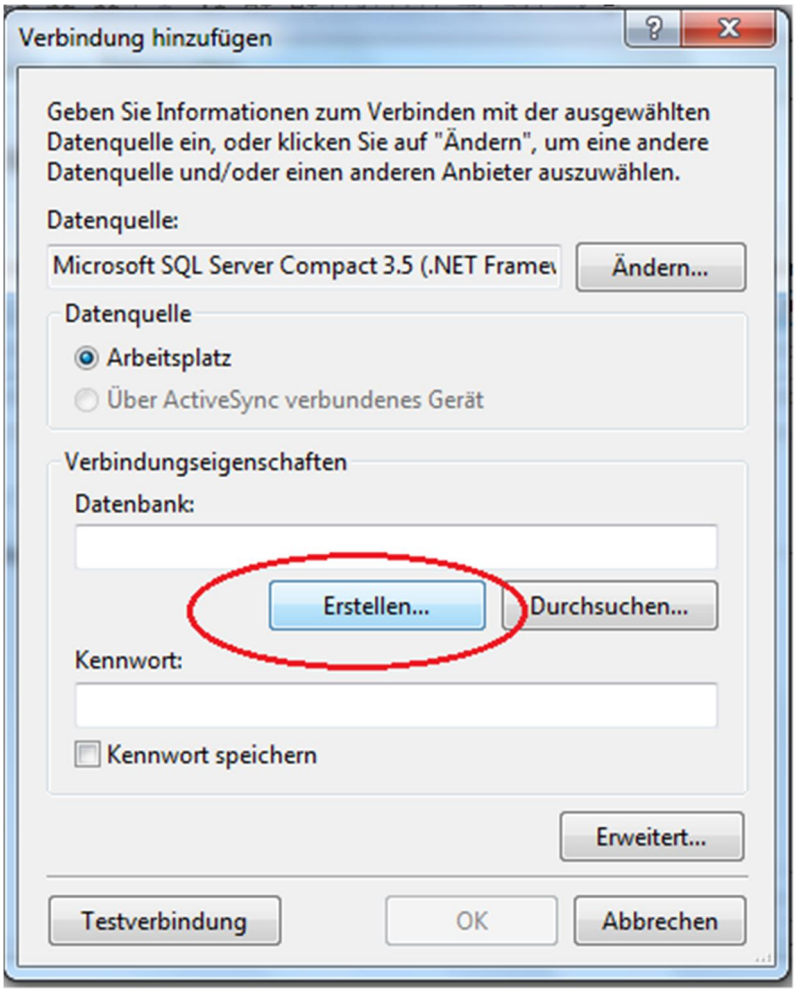
Da wir noch keine Datenbank im System haben wählen wir im nächsten Schritt **neue Verbindung**



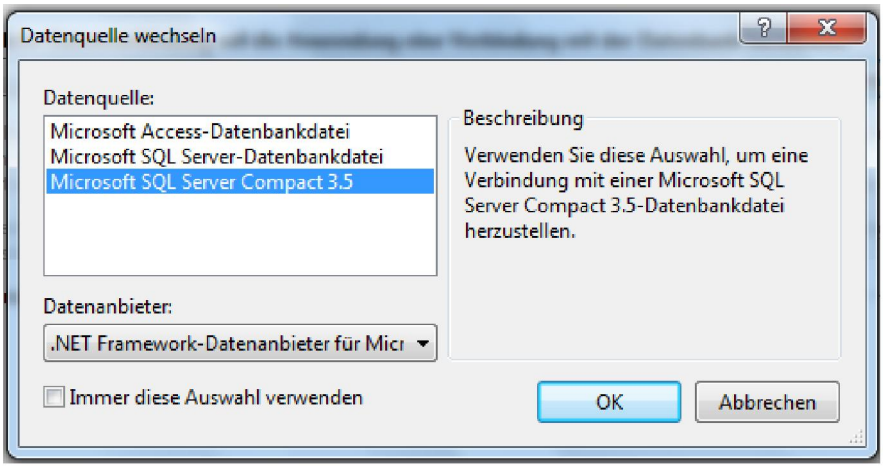

*Verbindung einrichten*

I

m folgenden Fenster ist man aufgefordert, eine vorhandene Datenbank anzugeben oder eine neue Datenbank zu erstellen. Ein Blick in die Auswahl der Datenbanken kann nicht schaden. Klicken wir einmal auf das Button **Ändern** um alle verfügbaren Datenbanktypen zu sehen.

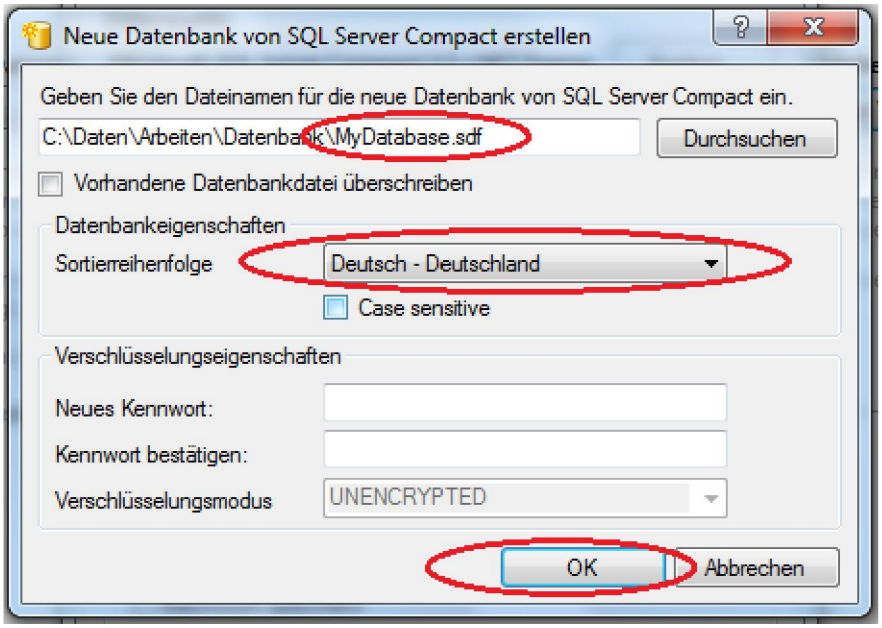



*Datenbanktyp wählen oder vorgeschlagene übernehmen*



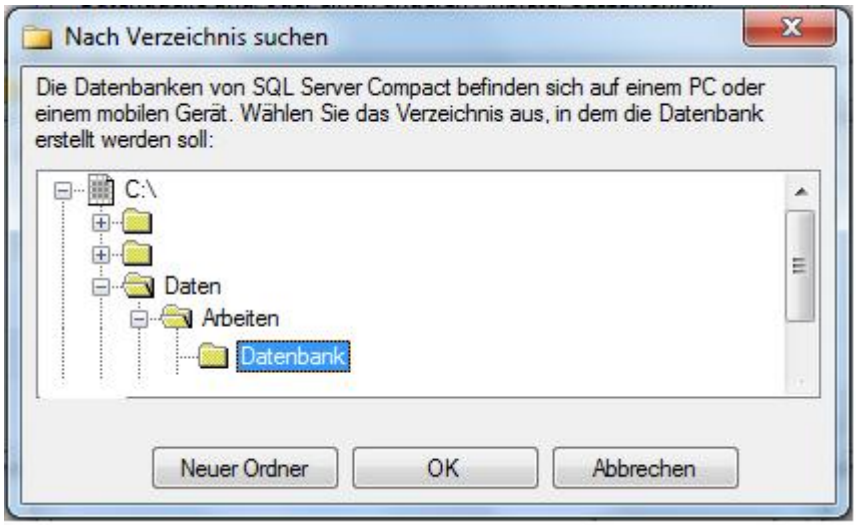

Wir belassen aber die Einstellung und brechen hier wieder ab.

Zurück im Fenster **Verbindungen hinzufügen** wird das Button **Erstellen** betätig, da ja noch keine Datenbank vorhanden ist.




*Neue Datenbank erstellen und Namen vergeben*

Nur noch wenige Einstellungen in den folgenden Fenstern sind erforderlich. Den Ablagepfad legen wir durch anklicken des ***Button*** **Durchsuchen** fest.

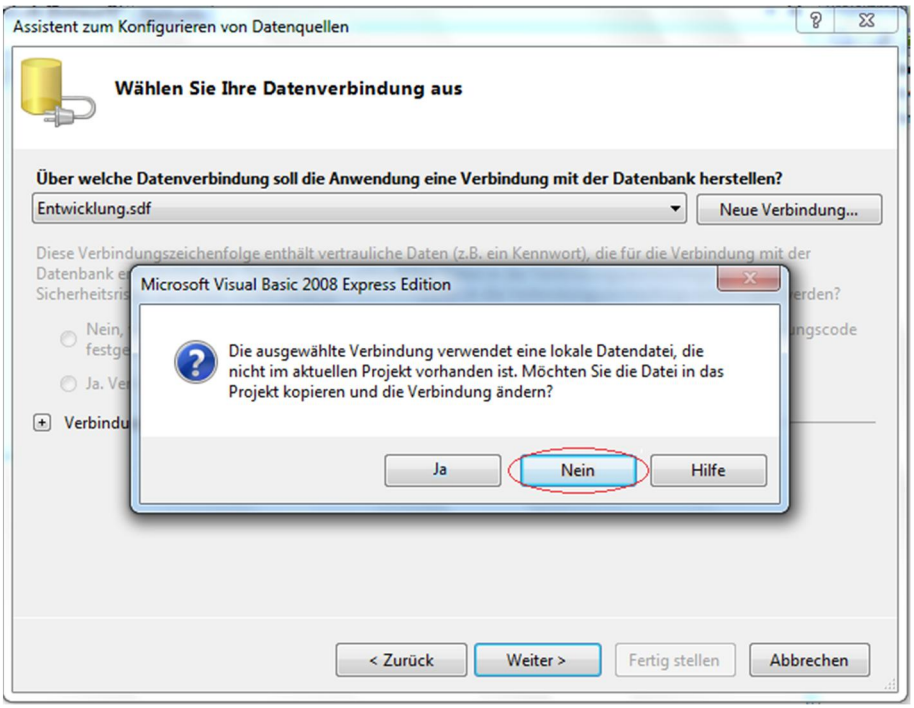



*Ablageort Datenbank*

Eventuell muss ein neuer Ordner angelegt werden. Mit **OK** geht es dann zurück. Dann kann der Pfad mit dem Datenbanknamen, den wir verwenden wollen ergänzt werden. Dafür wird **MyDatabase** überschrieben. Ich habe die Datenbank **Elektronik** benannt.

Wenn diese Einstellung mit dem Button **OK** dann abgeschlossen wird, kommt der Hinweis, dass kein Passwort vergeben wurde. Wir können dies ignorieren, da ein Passwort hier nur lästig ist und mit **Ja** fortfahren.

Die Datenbank wird im Fenster **Verbindung hinzufügen** mit **OK** übernommen und ist nun im Assistenten sichtbar. Mit dem Button **Weiter** erhalten wir nun eine Meldung

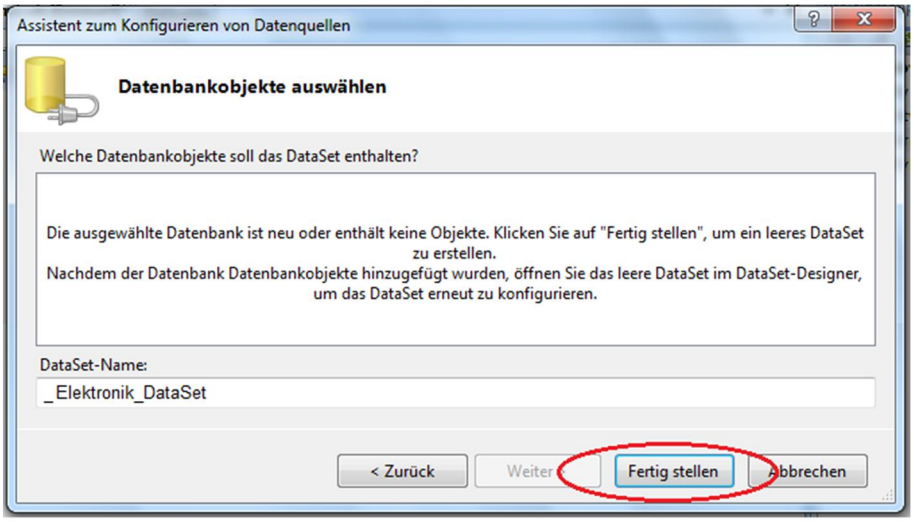



*Datenbankverbindung festlegen*

Den Einstellung im nächsten Schritt übernehmen wir  wir ohne Änderung mit dem Button **Weiter**.


*Verbindungszeichenfolge festlegen*

Schließlich landen wir im letzten Schritt. Da die Datenbank neu ist und noch keine Tabellen angelegt sind, wird eine entsprechende Mitteilung abgegeben.

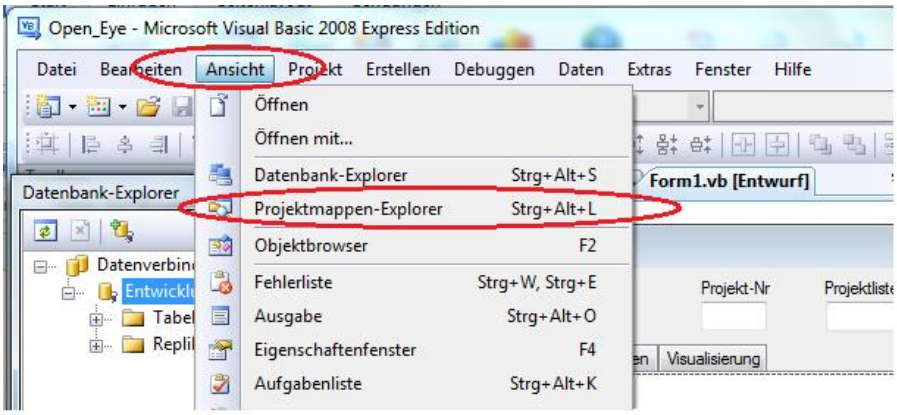



*Datenbank fertigstellen*

Mit **Fertig stellen** ist nun die Voraussetzung für das Anlegen von Datenbanktabellen geschaffen.

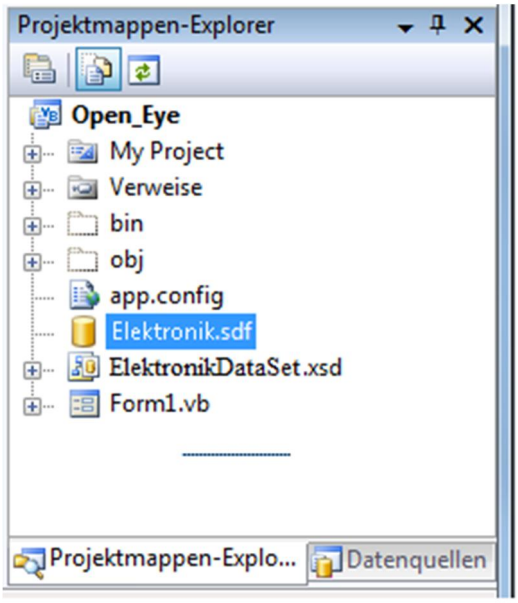
### *1.3.5 Datenbanktabellen anlegen*

Wieder zurück zu unserer Anwendung. In der Menüleiste unter Projekt finden wir den Menüpunkt Projektmappenexplorer.




*Projektmappenexplorer öffnen*

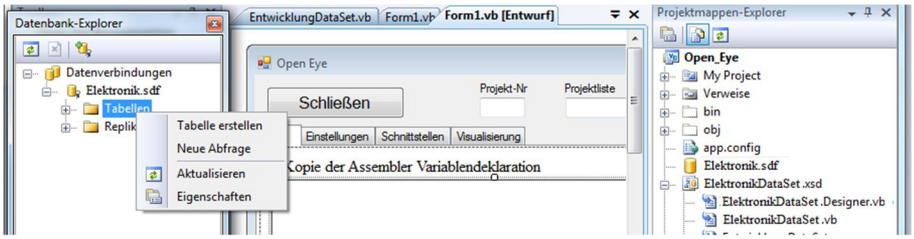
Wenn er nicht schon auf der rechten Bildschirmseite sichtbar ist, können wir ihn nun aufschlagen.




*Verzeichnisbaum Objektmappen Explorer*

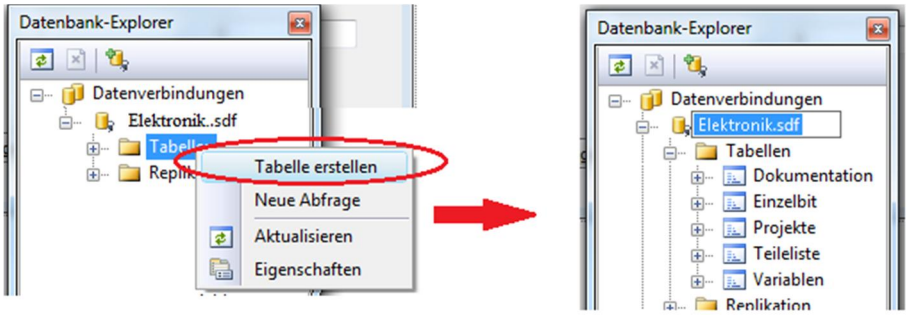
Mit einem Doppelklick in den Eintrag der von uns erstellten Datenbank **Elektronik.sdf** öffnen wir den Datenbankexplorer.

Ein rechter Mausklick in die Rubrik **Tabellen** öffnet ein Auswahlmenü und wir finden den Eintrag **Tabellen erstellen**.



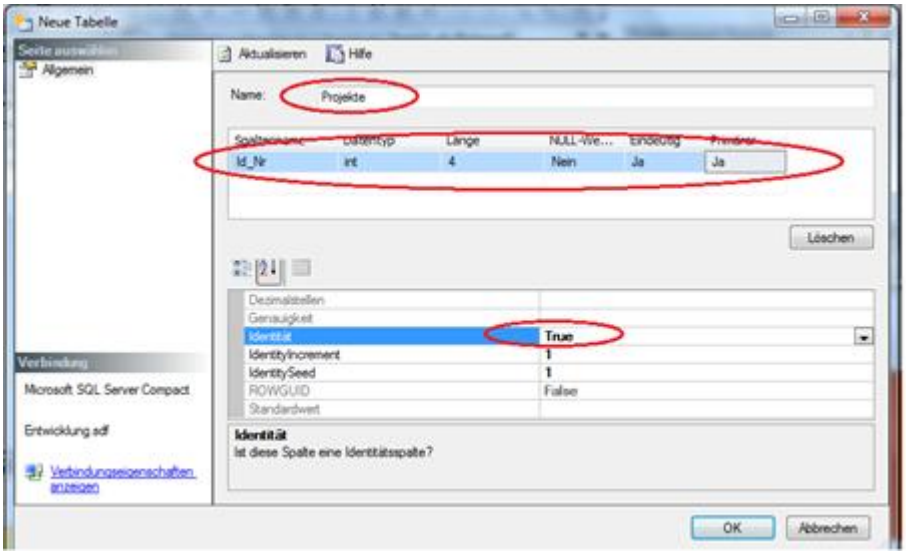

*Datenbanktabellen erstellen*

Wir bekommen nun die Möglichkeit, unsere Tabellen auf der Datenbank anzulegen.




*Tabellen anlegen*

Zuerst tragen wir den Namen ein. Die einzelnen Spalten sind bereits erarbeitet. Bei jeder Tabelle beginnen wir mit der *Identitätsspalte*. Sie ist in der im Screenshot dargestellten Parametrierung anzulegen.




*Die Identitätsspalte*

Die Auswahl des Datentyps **int** reicht für unsere Zwecke völlig aus. Bei einer Datenbank mit vielen Einträgen ist aber **bigint zu** wählen.

Anschließend folgen die Spalteneinträge wie sie bereits erarbeitet wurden. Nach dem Namen ist der **Datentyp** anzugeben. Wenn hier von Text die Rede ist kommt immer **nvarchar** zum Einsatz. Für einen Projektnamen reichen 30 Zeichen in einem **Text** aus. Auch der Controllername dürfte kaum länger als 30 Zeichen sein. Beim Takt sollten selbst bei krummen Zahlen 20 Zeichen reichen. Das ist auch mit ein Grund, warum hier der Datentyp **Text** eingestellt ist. Es ermöglicht die Eingabe von 16 MHz genauso wie 31.243 KHz. Im Programm wird darauf ja keine Auswertung erstellt und so ist die Eingabe von **Text** ausreichend bequem.

Bei den Feldern **Baud**, **Daten**, **Stopp** und **Parity** reicht der Datentyp **tinyint**, eine 8 Bit Zahl zwischen 0 und 255. Hier greifen wir nur auf den Index der Comboboxen zurück und der ist auf jeden Fall kleiner.

Der Zeile **Kopf** geben wir großzügig eine Textgröße von 10 Zeichen.

Bei der Spalte **Befehl** sind 10 Zeichen auch etwas überdimensioniert, aber so haben wir später alle Möglichkeiten offen.

Bei Sprache reichen 10 Zeichen auch völlig aus.

Die beiden Puffer Lese- und Schreibpuffer sind allerdings unterschiedlich zu betrachten. Schreibend werden wir in etwa mit der Größe von der Textlänge **Befehl** arbeiten, lesend ist es wesentlich mehr. Zuerst der **Kopf** gefolgt von dem Datenstrom. Da ist der Datentyp **Tinyint** ausreichend. Der je nach Anzahl der Variablen erheblich sein kann. Auch ist es denkbar, dass sogar ein zweiter oder dritter Datensatz zwischengespeichert werden muss. Also, hier reicht ein **Tinyint** nicht mehr aus.

Die Aufnahme von **Autor** ist eigentlich nicht erforderlich, aber wer vielleicht mit Freunden gemeinsame Projekte durchzieht, hat hier die Chance, seine Entwicklung festzuhalten.

Für das **Datum** reicht in der Regel eine Textgröße von 10 Zeichen, aber es ist denkbar, dass auch ein Zeitstempel mitgeführt werden soll. Dann sollten die 20 Zeichen ausreichen.

Bleibt noch ein zusätzliches Feld für einen **Kommentar**. Damit wäre die Tabelle vollständig.

Name: Projekte

| Spaltenname | Datentyp | Länge | NULL-We... | Eindeutig | Primärer ... |
|---|---|---|---|---|---|
| Id_Nr | int | 4 | Nein | Ja | Ja |
| Projekt | nvarchar | 30 | Nein | Ja | Nein |
| Controller | nvarchar | 30 | Nein | Nein | Nein |
| Takt | nvarchar | 20 | Nein | Nein | Nein |
| Baud | tinyint | 1 | Nein | Nein | Nein |
| Daten | tinyint | 1 | Nein | Nein | Nein |
| Stop | tinyint | 1 | Nein | Nein | Nein |
| Parity | tinyint | 1 | Nein | Nein | Nein |
| Kopf | nvarchar | 10 | Nein | Nein | Nein |
| Auftrag | nvarchar | 10 | Nein | Nein | Nein |
| Sprache | nvarchar | 10 | Nein | Nein | Nein |
| Schreibpuffer | tinyint | 1 | Nein | Nein | Nein |
| Lesepuffer | smallint | 2 | Nein | Nein | Nein |
| Autor | nvarchar | 30 | Nein | Nein | Nein |
| Datum | nvarchar | 20 | Nein | Nein | Nein |
| Kommentar | nvarchar | 100 | Nein | Nein | Nein |

Löschen

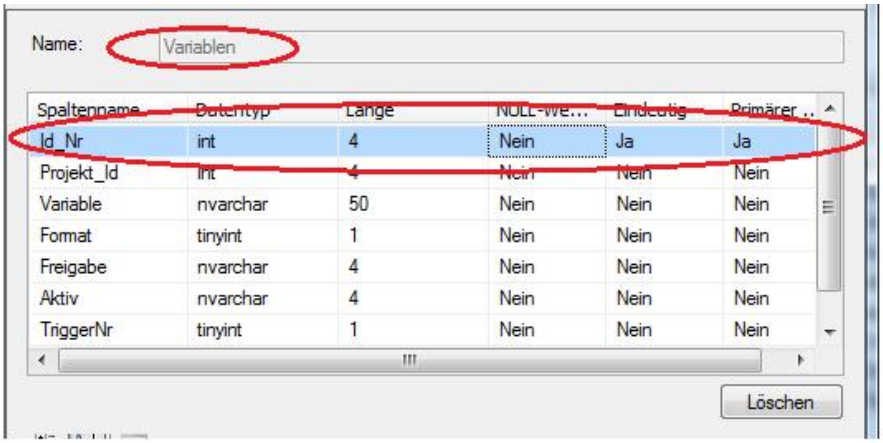
*Tabelleneinstellung Projekte*

In der Spalte **Null Werte zulassen** setze ich grundsätzlich **nein** ein. Natürlich kann man Null Werte außer in der ersten Spalte zulassen, aber schließlich wird eine Entscheidung verlangt und da sollte man nicht unbedingt mischen Wenn für Felder keine Daten vorliegen, schadet es nicht, einen Bindestrich in Textfelder und Felder mit numerischen Daten mit einer 0 zu füllen. Dies kann dann auch im Programm abgefragt und entsprechend behandelt werden.

Auch die beiden Folgezeilen sind in erster Linie für die **Id_Nr**, also die Schlüsselspalte von Bedeutung. Da ist klar, dass ein doppelter Eintrag nicht stattfinden darf. Aber auch beim Projektnamen macht es Sinn, doppelte Namen zu verhindern. Bei allen anderen Spalten können aber mehrfach gleiche Einträge stehen.

Mit OK erzeugen wir nun die Datenbanktabelle **Projekte**.

Die Tabelle für **Variablen** bauen wir nach gleichem Muster auf. Nachdem wir wieder mit einem rechten Mausklick auf Tabellen den Eintrag Tabellen hinzufügen gewählt haben, vergeben wir den Namen und legen die Spalten für die Tabelle **Variablen** fest.




*Ansicht Tabelle Variablen*

Die **Identitätsspalte** ist die wichtigste Zeile einer Tabelle und auch immer mit der gleichen Struktur aufzubauen.

Die Spalte **Projekt_Id** ist ebenfalls eine wichtige Spalte. Sie dient der Referenzierung dieser **Variablen** zum **Projekt**. So ist der Datentyp mit

dem Datentyp der Identitätsspalte gleichzusetzen, also int. Aber ansonsten ist diese Spalte eine ganz normale Spalte.

Für die Namen der Variablen sollten 50 Zeichen reichen.

Die Spalte Format ist mit der Auswahl einer Combobox besetzt, also kommt auch ein Datentyp **tinyint** in Frage, der auf den Eintrag in der Combobox zeigt. Wer es lieber mit Klartext hätte, kann natürlich auch einen Datentyp **nvarchar** wählen. Muss dieses allerdings im Programm entsprechend umschreiben.

Die beiden Spalten Freigabe und Aktiv sind als boolesche Werte vom Datentyp **Bit**, aber hier bevorzuge ich den Typ **nvarchar** mit 4 Zeichen. Auch **Bit** belegt immer ein ganzes Byte auf der Datenbank und einen Grund für besondere Sparsamkeit haben wir bei dieser Anwendung nicht.

Bleibt der Datentyp von der Spalte Trigger. Da hier nur ein Byte verwendet wird, reicht ein Datentyp **tinyint** völlig aus. Wir können damit sowohl das Bit (Werte 0, 1, 2, 4, 8, 16, 32, 64 und 128) als auch die Zeile in der Combobox (Werte von 0 bis 8) eintragen. Mit OK wird auch diese Tabelle auf der Datenbank erstellt.

Nun zur Tabelle Einzelbit. Die Struktur ist nun hinlänglich erklärt.

| Spaltenname | Datentyp | Länge | NULL-We... | Eindeutig | Primärer ... |
|---|---|---|---|---|---|
| Id_Nr | int | 4 | Nein | Ja | Ja |
| Projekt_Id | int | 4 | Nein | Nein | Nein |
| Variable_Id | int | 4 | Nein | Nein | Nein |
| Bit | tinyint | 1 | Nein | Nein | Nein |
| Funktion | nvarchar | 50 | Nein | Nein | Nein |

*Aufbau der Tabelle Einzelbit*

Die Spalten **Projekt_Id** und **Variable_Id** bekommen den gleichen Datentyp wie die eigene **Identitätsspalte**. Auch hier ist eine Beziehung zu den Beiden Tabellen **Projekte** und **Variable** beabsichtigt.

Die Spalte Bit ist ganz klar **tinyint**, da nur Werte von 0 bis 7 in Betracht kommen und 50 Zeichen sollten zur Funktionsbeschreibung reichen.

Bleibt noch die Tabelle für Dokumentation. Auch hier wieder zuerst die Identitätsspalte.

Name: Dokumentation

| Spaltenname | Datentyp | Länge | NULL-We... | Eindeutig | Primärer ... |
|---|---|---|---|---|---|
| Id_Nr | int | 4 | Nein | Ja | Ja |
| Projekt_Nr | int | 4 | Nein | Nein | Nein |
| Dateipfad | nvarchar | 100 | Nein | Nein | Nein |
| Dateityp | tinyint | 1 | Nein | Nein | Nein |
| Bemerkung | nvarchar | 100 | Nein | Nein | Nein |
| sonstiges | int | 4 | Nein | Nein | Nein |

Löschen

*Die Tabelle Dokumentation*

Natürlich hat eine Dokumentation auch einen Bezug zum Projekt und so ist die zweite Spalte für die Referenz mit dem Datentyp der Identitätsspalte int vorgesehen.

Der Dateipfad kann sehr lang sein, doch 100 Zeichen sollten ausreichen. Damit Zeichnungen und Texte differenziert behandelt werden können, kommt noch eine Spalte für den Dateityp hinzu. Hier reicht eine Tinyint, da der Typ in einer Combobox hinterlegt wird.

Schließlich noch ein Feld für eine Bemerkung. Auch hier brauchen wir nicht unbedingt sparen und setzen 100 Zeichen dafür an.

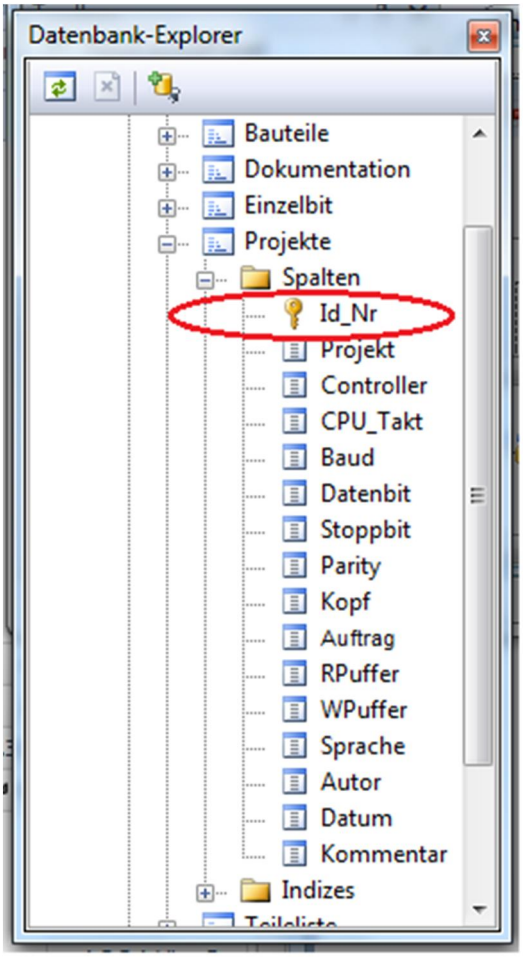
Die Spalte sonstiges ist eine Möglichkeit für eine weitere Referenz.

Damit sind die Datenbanktabellen eingerichtet.

Eine abschließende Bemerkung noch zu den Formaten: die verwendete Bytezahl angegeben im Feld „Länge“ ist für jeden Datensatz gültig, Speziell bei den Spalten mit Text, also nvarchar, werden bei einer Länge 100 immer 100 Byte reserviert, auch wenn ein Wort nur 8 Zeichen hat. Sollte aber ein Text mit einer größeren Länge eingetragen werden, gehen die überzähligen Zeichen verloren. Bei unserer Anwendung ist dies kein Problem. Werden Datenbanken mit viel Datensätzen angelegt, muss dies allerdings berücksichtigt werden.

### *1.3.6 Tabellenschema im Datenbankexplorer*

Im Datenbankexplorer sind nun die Tabellen eingetragen. Wie ein Verzeichnisbaum ist auch hier die Struktur bis zur Spaltendefinition gegeben. Der Schlüssel vor der Spalte „Id_Nr“ weist auf die besondere Funktion dieser Spalte hin.




*Baumstruktur Tabelle Projekt*

Mit einer solchen Baumstruktur ist es ähnlich, wie mit einem Dateiverzeichnis. Soll ein Zugriff auf den Inhalt einer Spaltenzelle erfolgen, ist der Weg dorthin wie bei einem Dateizugriff. Es ist die Verbindung zur Datenbank erforderlich, die Rubrik Tabellen muss

angewählt werden, eine Tabelle benannt und schließlich das Feld einer Spalte mit einem Index angesprochen werden. Hinter dieser Struktur steckt aber ein Datenbankprogramm, welches den Zugriff auf seine eigene Art koordiniert. Eine Datenbank ist ja schließlich für einen Zugriff von verschiedenen Applikationen ausgerichtet, z. B. wie bei einem Reisebüro. Ich sprach diese Problematik bereits an. Das bedeutet, so direkt mit

```
Mein_Projekt=Entwicklung.Tabellen.Projekte.Columns(„Projekte“).Items(3)
```

kann der Zugriff nun nicht erfolgen. Eine Datenbank erwartet eine Abfrage, etwa in der Art:

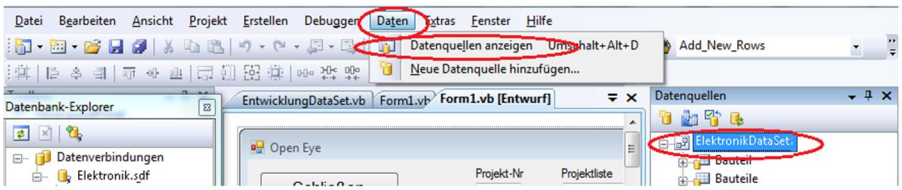
```
Zeig mir mal den Projektnamen in der Tabelle Projekte mit der Schlüsselnummer 17
```

Oder

```
Liefere mir alle Variablen vom Projekt mit der Schlüsselnummer 12
```

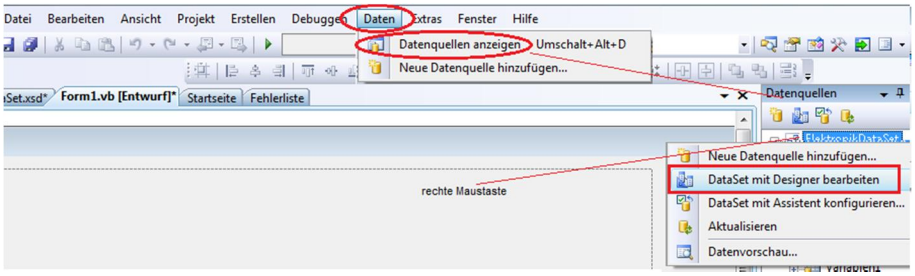
Na ja, hört sich gar nicht so kompliziert an, und das ist es auch nicht. Wenn das System erst einmal verstanden ist, macht eine Datenbank so richtig Spaß.

Solche Abfragen nennt der Fachmann SQL-Abfrage. Wenn man die Struktur selbst erstellen sollte, würde so mancher das Handtuch werfen, aber Visual Basic ist ein komfortables Programm und wie immer gibt es Assistenten, die hier gewaltig Arbeit abnehmen. Dafür brauchen wir erst einmal den Datenbankdesigner, damit wir unsere erstellten Tabellen anwenden können. Wir finden ihn über den Weg Menüleiste Daten und den Eintrag Datenquellen anzeigen. ( Siehe auch Besondere Hinweise




*Aufruf Dataset Elektronik*

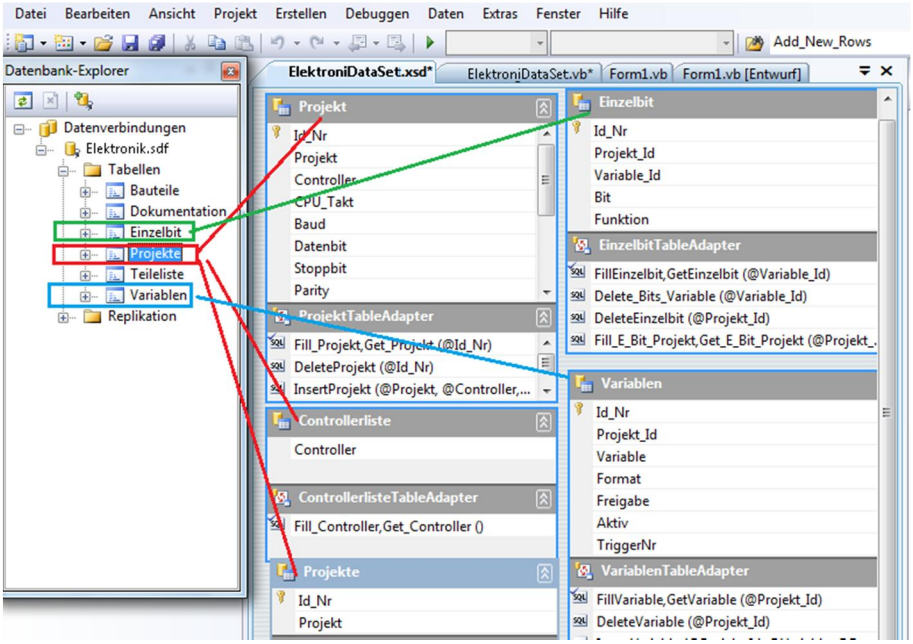
Im nächsten Schritt wird mit der rechten Maustaste in den Datenquellen der Eintrag EntwicklungDataSet angeklickt. Im Aufgeschlagenen Menü finden wir nun den Punkt DataSet mit Designer bearbeiten. Diese Auswahl öffnet nun eine neue Seite, den Datenbankdesigner.




*Aufruf Dataset-Designer*

### *1.3.7 Tabellen im DataSetDesigner bearbeiten*

Es gilt nun unsere Tabellen dem Programm zugänglich zu machen. Fügen wir der noch leeren Seite unsere Tabellen hinzu. Dies geht einfach, indem die Tabellen aus dem Datenbankexplorer angeklickt und mit gedrückter Maustaste auf die Designer-Seite gezogen werden.




*Ansicht des Datasetdesigner*

Die Tabelle Projekte ziehen wir dreimal in die neue Seite (rote Linien) und die anderen jeweils einmal.

Nun vergeben wir den Tabellen Projekte neue Namen: Projekt, Projekte und Controller. Die Erklärung dazu folgt in den weiteren Abschnitten.

Das war erst einmal der erste Schritt, die Datenbanken dem Programm verfügbar zu machen. Nun müssen die Tableadapter eingerichtet werden.

### *1.3.8 Der Tableadapter*

Um die Funktion eines Tableadapters zu verstehen, müssen wir grundsätzlich wissen, dass ein Programm von einer Datenbank nur die Daten abruft, die es auch benötigt. Die Tabelle wird in den seltensten Fällen komplett gefüllt. Bei großen Datenbanken wäre das auch kaum möglich. Dazu ein Beispiel:

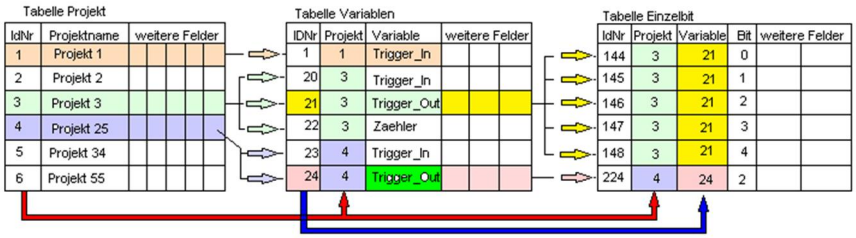
Stellt euch einen großen Topf mit einer Suppe vor. Auf dem Tisch ist ein Teller. Nun ist der Teller nicht so groß, dass die ganze Suppe hinein passt. Deshalb gibt es eine Suppenkelle, um die Suppe auf den Teller zu portionieren. Nun mag es sein, das in der Suppe besonders leckere Zutaten enthalten sind oder aber auch welche, die man gar nicht gern mag. Daher wird auch manchmal etwas gezielt aus dem Topf herausgefischt. So ungefähr müssen wir uns die Tabellen und Tableadapter vorstellen. Vergleichbar dabei ist der Topf die Tabelle, der Tabelleninhalt die Suppe, die Suppenkelle die Datenbankabfrage und der Teller der Tableadapter mit dem Ergebnis.

Mit dieser Vorstellung machen wir uns nun ein paar Gedanken, wie unsere erwarteten Ergebnisse auszusehen haben. Da ist eine einfach gefüllte Suppenkelle bunt gemischt mit allen Inhalten der Suppe, aber eben nur eine Kelle. Dies wäre vergleichbar mit einer Tabellenzeile und den Inhalten aller Spalten. Ein einzelner Datensatz sozusagen. Den brauchen wir, um in der Projekttabelle die Einstellungen eines Projektes zu verwalten. Welcher Controller und Taktfrequenz verwendet wurde. Wie die Schnittstelle eingestellt ist und was auch immer zu einem Projekt genau nur einmal dazugehört.

Wir möchten aber auch eine Übersicht über bestehende Projekte haben. Da reichen Projektnamen und zugehörige ID-Nummern, um gezielt Datensätze aus der Tabelle herauszulösen. Dies wäre vielleicht vergleichbar, als würde man alle Fleischbrocken aus der Suppe fischen. Die passen vermutlich auch alle auf einen Teller. Doch nun soll es genug sein, diese Vergleiche aufzuführen. Eine Datenbanktabelle ist ja auch keine Suppe. Aber die Abfrage nach allen Projekten macht schon Sinn. So können die Projektnamen in einer Liste abgelegt und zur bequemen Auswahl verfügbar sein. Schließlich bleiben noch die Anweisungen, Daten abzulegen, zu korrigieren und zu löschen. Hierbei ist immer der Zugriff auf einen kompletten Datensatz in der Tabelle erforderlich, der den betroffenen Datensatz über die ID-Nummer referenziert. Die Beschreibung der Tabelle für die Variablen ist ähnlich. Auch hier wird der Zugriff auf einen einzelnen Datensatz erforderlich, wenn Einstellungen

verändert werden. Im Gegensatz zur Projekttabelle ist hier aber auch der Zugriff auf alle Einträge zu einem Projekt erforderlich. Dafür wird die Spalte **Projekt_ID** mitgeführt. Hier steht die Nummer des zugehörigen Projektes und über diese Nummer wird die Datenselektion auch durchgeführt. Bleibt von den drei Haupttabellen noch die Verwaltung der Einzelbits. Hier wird nicht die Projektnummer, sondern die ID-Nummer der Variablen gefiltert. Ist ja klar, das Projekt hat eine Nummer, zum Beispiel 4. Nun werden alle Variablen, die in der Spalte **Projekt_ID** eine 4 stehen haben in die Anwendung geladen. Die Einzelbitbeschreibung greift nun auf die **Variablen_ID**, denn die Variablen sind ja nur für dieses Projekt gültig. Somit sind natürlich auch die zugehörigen Bitbeschreibungen nur für dieses Projekt, referenziert über die **Variablen_Id**.

Deutlich wird dies hier noch einmal durch eine kleine Skizze

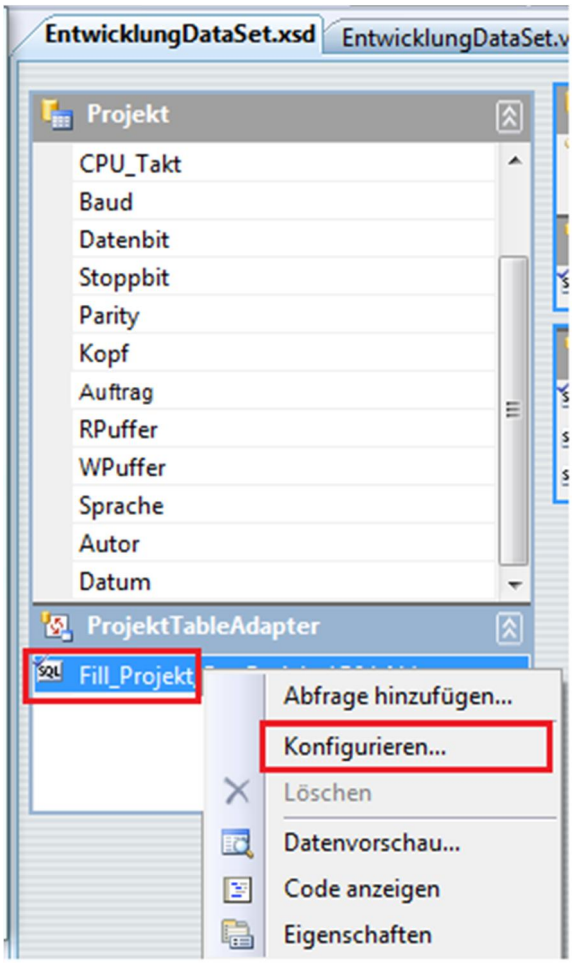



*Tabellenverbindungen*

Das die Einzelbittabelle auch die Projekt_ID mitführt ist scheinbar nicht sinnig. Die Einfärbung zeigt den eindeutigen Zusammenhang. So können wir mit einem Tableadapter über die Projekt_Id 4 alle zugehörigen Assemblervariablen in die Tabelle Variablen auf unserer zweiten Seite laden. Auch alle Einzelbitbeschreibungen, die zum Projekt gehören, werden mit der 4 aus der großen Datenbanktabelle herausgesucht. Aber warum ist das erforderlich. Reicht es nicht, nur die Einzelbits, die zum gerade aktivierten Variablennamen gehören hervor zu holen? Nein, aber soweit sind wir noch nicht.

### *1.3.9 Abfrage für die Daten von einem Projekt*

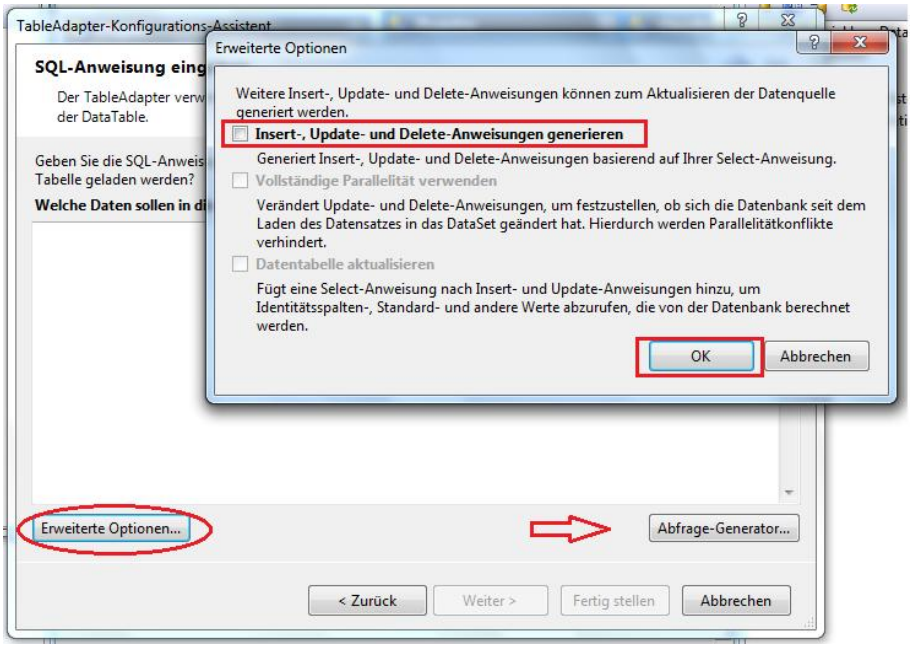
Den ersten Tableadapter richten wir für die Projekttabelle ein. Dabei wollen wir eine volle Zeile der Tabelle in den Tableadapter laden. Nur so funktionieren auch die Korrekturanweisungen. Schließlich haben wir bei allen Feldern darauf bestanden, dass dort Inhalte stehen werden. Sicher, ein Datum oder einen Autor muss man nicht jedes Mal eintragen, aber will man unzulässige Werte bei einer Verarbeitung meiden, ist eine Vorbesetzung mit Defaultwerte sowieso erforderlich.



*Tableadapter Fill Projekt*

Ein Klick mit der rechten Maustaste auf die Zeile unter *ProjektTableadapter* in die Zeile *Fill, GetData()* öffnet den Assistenten nach der Auswahl Konfigurieren.

Der Assistent ist eine bequeme und komfortable Einrichtung. Allerdings sind auch einige Schritte zu beachten. Dazu sehen wir uns erst einmal den Screenshot an

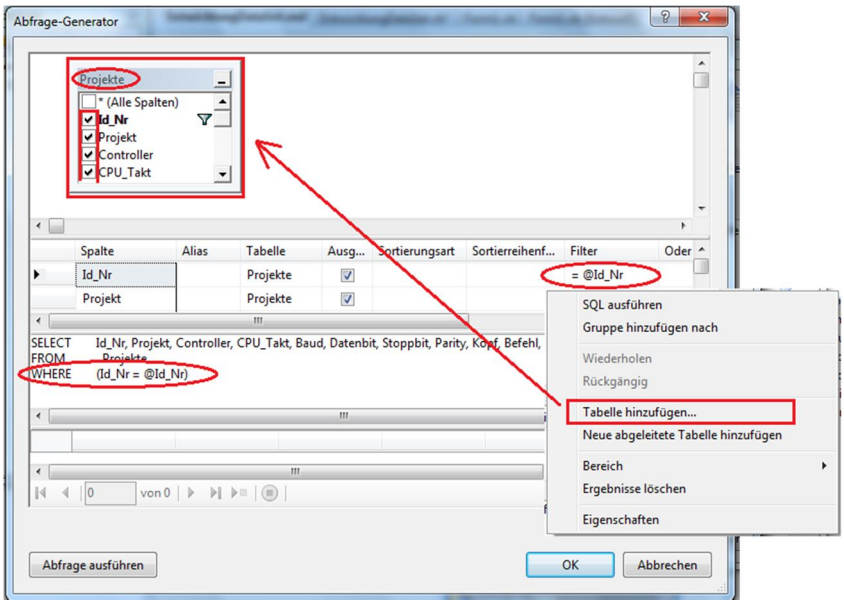



*Schreibfunktionen abwählen*

Der erste Schritt ist ein Blick auf erweiterte Optionen. Hier entfernen wir den Haken aus der Checkbox **Insert-, Update- und Delete-Anweisungen generieren.**

Mit **OK** kehren wir zurück und klicken auf das Button **Abfragegenerator.**

Nun ist es möglich, dass eine Fehlermeldung erscheint und die obere Fläche keine Tabelle enthält.




*Datenbankabfrage Tabelle auswählen*

Ein Klick mit der rechten Maustaste in den Tabellenbereich und es erscheint ein kleines Menü. Klickt man den Punkt Tabellen hinzufügen an, bekommt man eine Liste mit den von uns auf der Datenbank erstellten Tabellen geliefert. Die Auswahl fällt auf **Projekte**


*Projekttabelle wählen*

Ist die Tabelle eingetragen werden alle Spalten ausgewählt. Die Auswahl einzelner Spalten kann auch durch eine Anwahl ***alle Spalten** erledigt werden. Allerdings fehlt dann auch die Auflistung der einzelnen Spalten und die brauchen wir, um einen Filter auf die **Id_Nr** zu setzen.

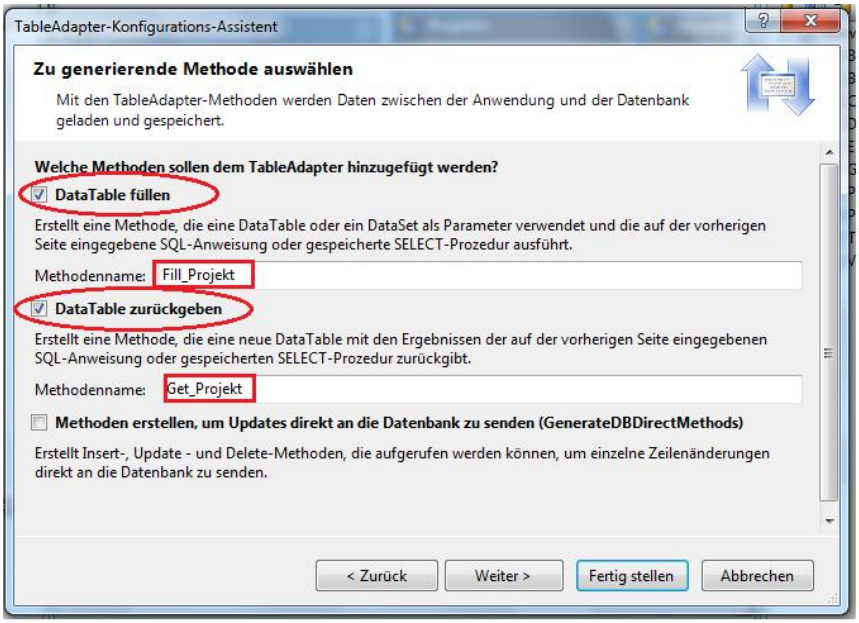
Im Mittelteil wird die automatisch generierte SQL-Anweisung angezeigt. Durch den Eintrag unter Filter **=@Id_Nr** wird die Select-Klausel mit Where Id_Nr = @Id_Nr.

**@Id_Nr** ist ein Übergabeparameter. Hier dient er zur Selektion einer bestimmte **Id_Nr**.

Die SQL-Anweisung an die Datenbank für einen einzigen Datensatz lautet also

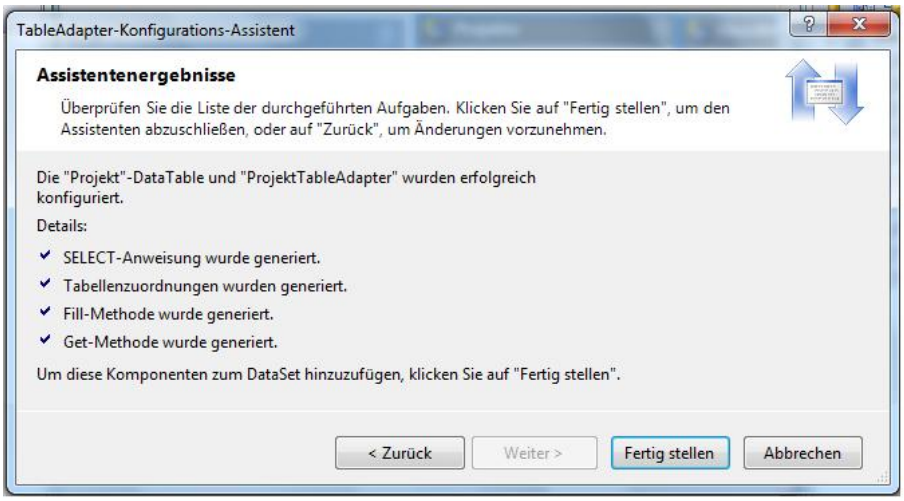
```
SELECT      Id_Nr, Projekt, Controller, CPU_Takt, Baud, Datenbit, Stoppbit, Parity, Kopf,
Befehl, RPuffer, WPuffer, Sprache, Autor, Datum
FROM        Projekte
WHERE       (Id_Nr = @Id_Nr)
```

Mit Ok geht es zurück zum Assistenten und mit weiter zum nächsten Schritt.




*Tableadapter Get-und Fill Projekte*

In diesem Fenster werden nur die beiden eingekreisten Methoden angewählt und eigene Namen vergeben. Im Prinzip ist damit alles erledigt und wir könnten das Button Fertig stellen aktivieren, doch das letzte Fenster des Assistenten will ich euch nicht vorenthalten.




*Tableadapter Get Projekte fertigstellen*

Mit dieser Ergebnisinformation haben wir unseren ersten Tableadapter erstellt.

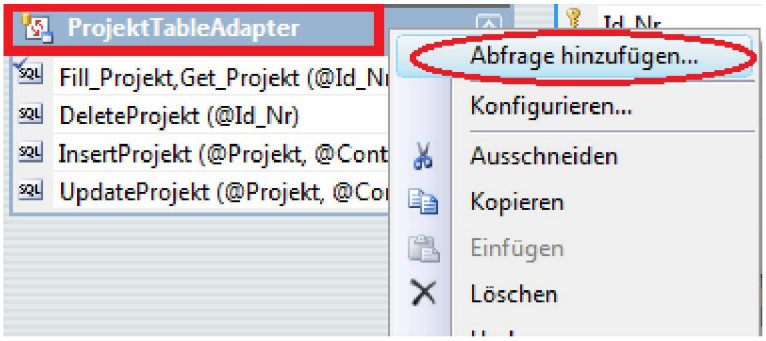
Doch leider können wir noch gar nichts testen. Klar, im Abfragegenerator gab es ein Button Abfrage ausführen und vielleicht war ja auch einer neugierig und hat dieses Button betätigt. NA, was habt ihr gesehen?

Ihr habt es nicht ausprobiert? Na dann schnell mal nachgeholt. Also mit der rechten Maustaste wieder in die Fill-Methode und mit Konfigurieren in den Assistenten wechseln. Nun noch den Abfragegenerator öffnen und die Abfrage ausführen.

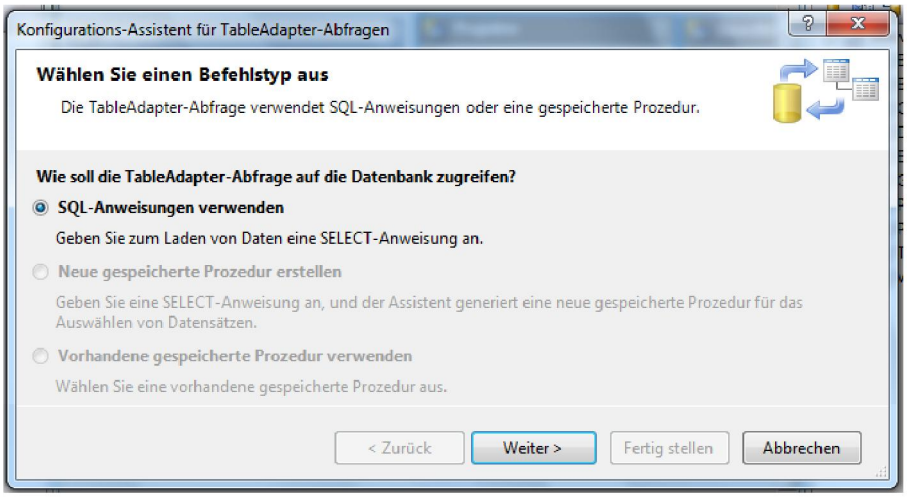
Es öffnet sich ein Fenster und nur die Spalte Id_Nr wird angezeigt mit dem Inhalt **NULL**. Ist ja auch klar, die Tabelle ist leer. Und ohne Inhalt kommt da nicht viel Information. Daher werden wir nun den Tableadapter um die Methode **Insert** erweitern.

### *1.3.9.1 Die Insert-Methode*

Um eine neue Abfrage zu generieren klicken wir mit der rechten Maustaste nicht auf die **Fill-Methode**, sondern auf den Eintrag **ProjektTableAdapter** und im anschließend geöffneten Menü auf **Abfrage hinzufügen**.



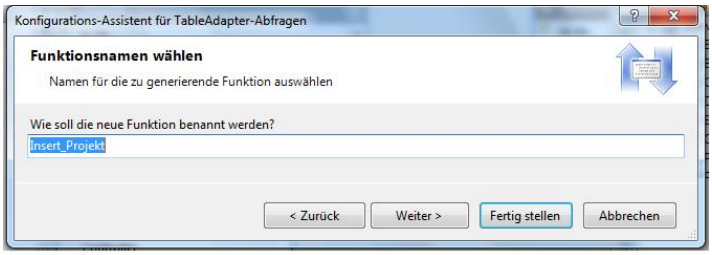

*Abfragen hinzufügen*




*Tableadapter SQL-Anweisungen erstellen*

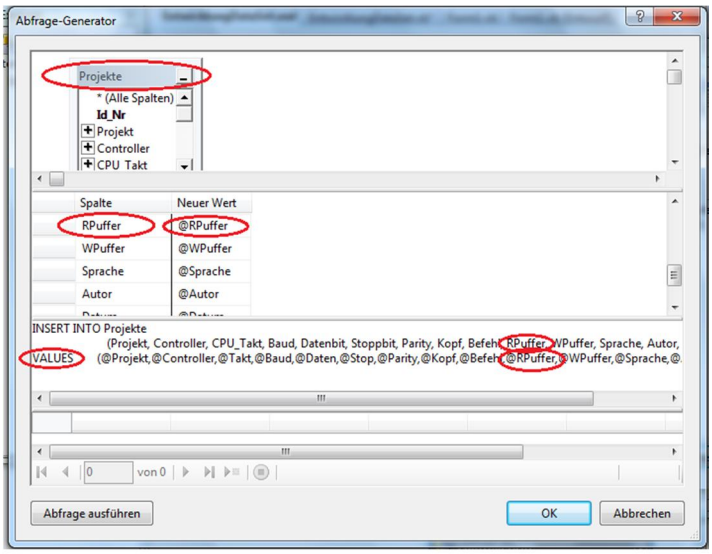
Diesmal können wir das erste Fenster des Assistenten direkt übernehmen, und mit **Weiter** zum nächsten Schritt gehen.

Im folgenden Screenshot werden wir vor die Wahl gestellt, welche Anweisung gewünscht wird.




*SQL Anweisung Insert Projekt*

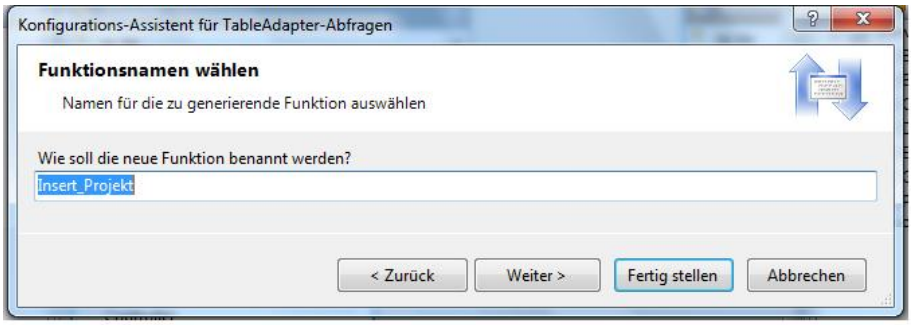
Bis hierher werden nun die Schritte für eine Löschfunktion und eine Korrektur des Datensatzes gleich sein. Wir beginnen mit **Insert**, also einen neuen Datensatz einfügen. Schließlich wird mit **Weiter** und dem Klick auf **Abfrage-Generator** wieder das bekannte Fenster zur Abfrageerstellung geöffnet. Auch hier ist es möglich, dass sich keine Tabelle im oberen Bereich befindet und eine Fehlermeldung erscheint. Diese wird in gewohnter Weise mit einem rechten Mausklick auf den Tabellenbereich aus der Liste hinzugefügt. Schließlich erhalten wir, nachdem die neuen Werte mit dem **@** und dem Spaltennamen parametriert wurden, folgendes Bild.




*Insert Übergabe mit @-Parameter*

Es ist richtig, das in der Spalte **Neuer Wert** jedes Mal ein Übergabeparameter eingetragen werden muss. Der Blick auf die **SQL-Anweisung** zeigt, dass nun zu jeder Spalte auch ein Wert mit **Values** zugewiesen wird bis auf die Spalte **ID_Nr.** Warum bekommt diese Spalte keinen Wert? Nun, das erledigt die Datenbank für uns. Da bekannt ist, dass dies der **Schlüssel** dieser Spalte ist, wird automatisch ein Inkrement auf die höchste Nummer in der Spalte **Id_Nr** erzeugt und dem neuen Datensatz hinzugefügt. Somit ist er eindeutig referenziert. Selbst wenn ein Datensatz gelöscht wird, geht der niederwertige Schlüssel zwar verloren, aber alle anderen Datensätze behalten ihren Wert in der Spalte **Id_Nr**.

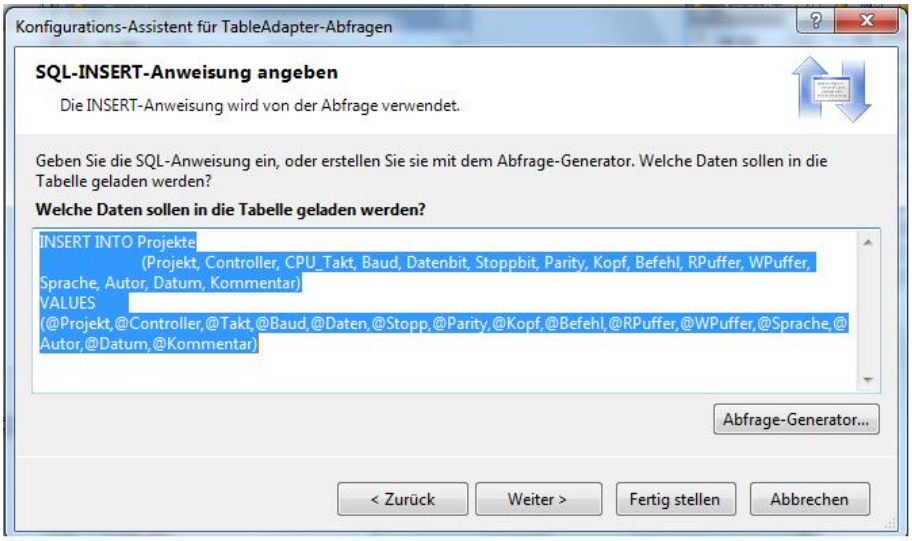
Nachdem wir den Abfragegenerator verlassen haben klicken wir noch einmal auf Weiter, um dieser Anweisung einen Namen zu geben.




*SQL-Anweisung Insert_Projekt*

Mit Insert_Projekt haben wir einen passenden Namen und können die **SQL-Anweisung** fertigstellen.

Es ist interessant, wie sein solche SQL-Anweisung aufgebaut ist. Nehmen wir uns doch einmal die Zeit und werfen einen Blick auf die generierte SQL-Struktur.



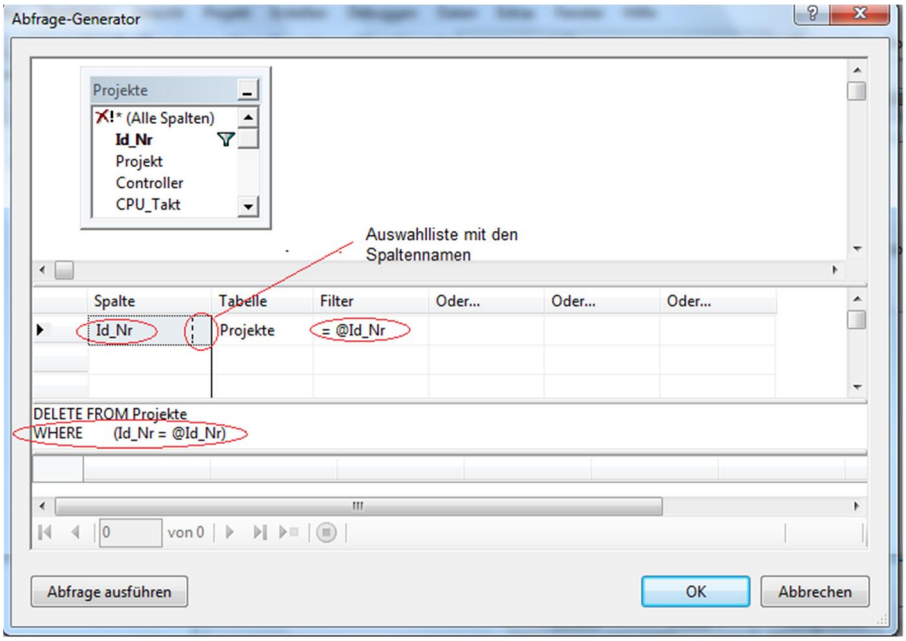

*SQL Anweisung InsertProjekt*

Mit **Insert** beginnt diese Anweisung gefolgt vom Tabellennamen und der Aufzählung der Spalten. Anschließend werden die Werte durch die Übergabeparameter eingetragen

### *1.3.9.2 Die Delete Anweisung*

So wie ein Datensatz abgelegt wird, so muss er auch entfernbar sein. Dazu gehen wir wieder mit einem rechten Mausklick in den Tableadapter und unter Abfrage hinzufügen wieder bis zur Auswahl der Anweisungen. Diesmal wählen wir **Delete**.

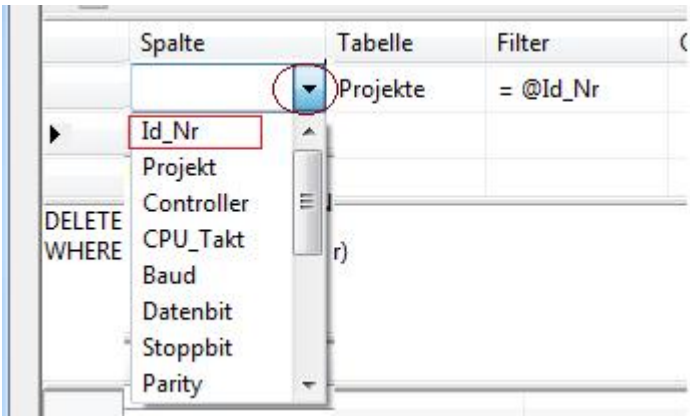
Genau wie vorher müssen wir vermutlich im Abfrage-Generator erst die Tabelle Projekte hinzufügen. Hier braucht gar nicht viel eingestellt werden, da eine **Delete** Anweisung immer den ganzen Datensatz entfernt, lediglich die Referenz auf den Datensatz müssen wir angeben. Dazu einmal den Screenshot mit der entsprechenden Maßnahme.#




*Delete Projekt*

Nach dem Zufügen der Tabelle sind erst einmal alle Spalten ausgewählt und das ist so auch korrekt. Aber welcher Datensatz soll gelöscht werden. Uns ist klar, dass es der ist, der grad aktuell mit der **Id_Nr** 17 aufgeschlagen ist. Weiß das die Datenbank auch? Um sicher zu gehen, teilen wir der **Delete-Anweisung** mit, dass wir den Datensatz mit der Nummer 17 löschen wollen und die dazu gehörende Anweisung ist im Filter untergebracht. Wenn Gleich dem Übergabeparameter lautet grob übersetzt das **=@Id_Nr**. In der SQL-Anweisung wird dies mit dem Wort

**Where** dargestellt. Da aber die Spalte zuerst gar keine Spaltennamen anzeigt, müssen wir diese selbst auswählen. Dazu ist ein etwas genauerer Blick auf das Fenster erforderlich, denn in jedem Feld in der Rubrik Spalte ist ein **Dropdown-Element** untergebracht, mit dem ich den Namen der Spalte **Id_Nr** auswählen und eintragen kann.

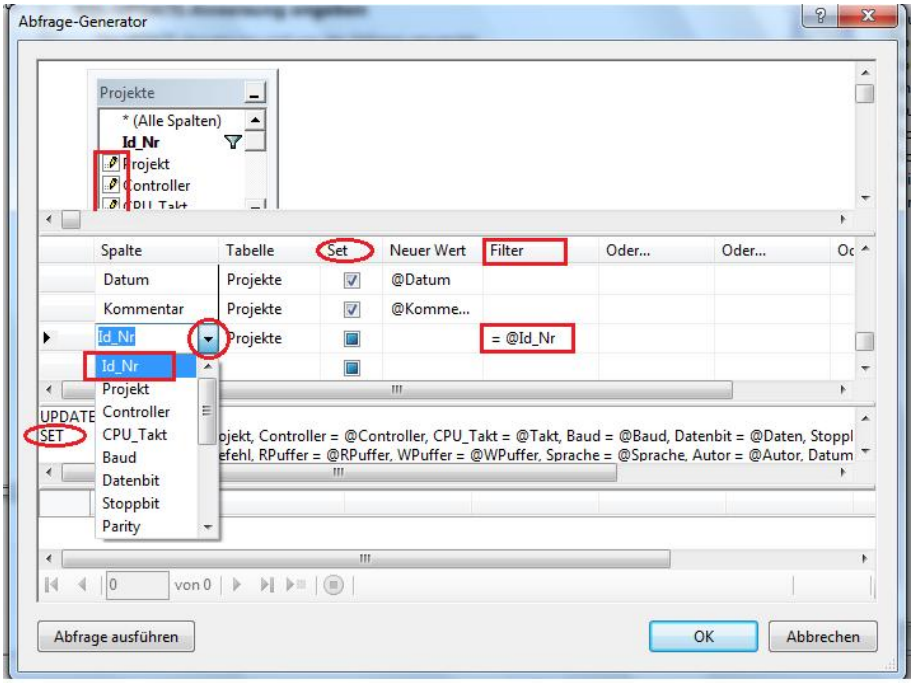



*Schlüsselspalte Id_Nr vom Projekt*

Damit ist die **Delete**- Anweisung fertig und wir gehen mit **Weiter** bis zur Namensvergabe. Klar, die Wahl ist nicht schwer **Delete_Projekt** und **Fertig stellen** schließt diese Anweisung ab.

### *1.3.9.3 Die Update-Anweisung*

Bleibt uns noch die Anweisung für eine Korrektur. Der Vorgang wiederholt sich bis zur Auswahl der Anweisung. Diesmal wird **Update** gewählt und bis zum Abfragegenerator durchgeschaltet.




*Update Projekt*

Diesmal müssen wir alle Spalten angeben, die von dieser **Update-Anweisung** betroffen ist. Ich übernehme alle bis auf **Id_Nr**. Diese kann auch gar nicht angewählt werden. Der Grund sollte mittlerweile auch klar sein. Die **Identitätsspalte** darf nicht korrigiert werden. Natürlich könnten wir den Projektnamen auch herausnehmen, aber auch wenn er nur einmal in der Liste aufgenommen werden darf, könnte ich mich ja für einen anderen Projektnamen entscheiden.

Schließlich kommen wir an das Ende der Liste. Hier gilt das Selbe, wie bei der **Delete-Anweisung**. Welcher Datensatz ist zu korrigieren? Klarheit verschafft der Schlüssel und dazu müssen wir wieder diese Liste in einem leeren Feld öffnen und den Spaltennamen **Id_Nr** auswählen. Der Filter ist dann auch **=@Id_Nr.**

Interessant ist auch der Aufbau der Anweisung. In der **Insert-Anweisung** gab es erst eine Liste der Spaltennamen und dann die Parameternamen hinter der Anweisung **Values**. Hier ist einfach nur **Update from Projekt Set**

**Projekt =@Projekt** etc. Es wird also jeder Spalte einzeln ein Wert zugewiesen. Am Schluss der Anweisung taucht dann wieder der **Where-Abschnitt** auf.


*SQL Anweisung UpdateProjekt*

Nun noch den Namen **Update_Projekt** eintragen und **Fertig stellen**.

Jetzt haben wir alle erforderlichen Zugriffe auf die Datenbanktabelle Projekte eingerichtet.

Um nun das Ergebnis der Arbeit zu testen, müssen wir wieder in das VB Programm.

### *1.3.4 Testen der Datenbankmethoden für die Projekttabelle*

Der einfachste Weg ist es erst einmal die vier Aufgaben **Fill**, **Insert**, **Delete** und **Update** in kleine Unterprogramme zu packen. Darum werden auch erst nur die vier Grundgerüste für die Subroutinen eingetragen. Für **Fill** setze ich den passenderen Namen **Load** ein.

```
Public Sub Load_Projekt(ByVal Id_Nr As Integer)

End Sub

Public Sub Insert_Projekt()

End Sub

Public Sub Delete_Projekt(ByVal Id_Nr As Integer)

End Sub

Public Sub Update_projekt(ByVal Id_Nr As Integer)

End Sub
```

### *1.3.4 Projektdaten laden*

Beginnen wir wie bei der Erstellung der Datenbankmethoden auch in dieser Reihenfolge und betrachten die Subroutine **Load_Projekt**. Da noch nicht alle Objekte zur Ansicht der Spalteninhalte existieren, werden wir, um die Inhalte zuweisen zu können, erst einmal in der Subroutine lokale Variablen deklarieren und ihnen die Spaltennamen geben. Da später auch Zahlenwerte in Textboxen angezeigt werden, definieren wir einfach alles Strings, bis auf die Integerwerte, die in den Comboboxen den Inhalt über einen Index adressieren.

```
Public Sub Load_Projekt(ByVal Id_Nr As Integer)
    Dim Projekt As String          ' Ausgabe in Textbox
    Dim Controller As String       ' Ausgabe in Textbox
    Dim Takt As String             ' Ausgabe in Textbox
    Dim Baud As Integer            ' Ausgabe über Combobox
    Dim Daten As Integer           ' Ausgabe über Combobox
    Dim Stopp As Integer           ' Ausgabe über Combobox
    Dim Parity As Integer          ' Ausgabe über Combobox
    Dim Kopf As String             ' Ausgabe in Textbox
    Dim Befehl As String           ' Ausgabe in Textbox
    Dim WPuffer As String          ' Ausgabe in Textbox
    Dim RPuffer As String          ' Ausgabe in Textbox
    Dim Sprache As String          ' Ausgabe in Textbox
    Dim Autor As String            ' Ausgabe in Textbox
    Dim Datum As String            ' Ausgabe in Textbox
    Dim Kommentar As String        ' Ausgabe in Textbox

  End Sub.
```

Nun werden wir den lokalen Variablen die Werte aus der Datenbanktabelle zuweisen. Damit der Zugriff auch sicher ist und nicht ins Leere greift, prüfen wir zuerst, ob mit der angegebenen **Id_Nr** überhaupt ein Datensatz existiert. Dabei hilft eine Variable mit dem Namen Anzahl, die noch zusätzlich deklariert wird. Hier die ganze Subroutine

```
Public Sub Load_Projekt(ByVal Id_Nr As Integer)
    Dim Projekt As String          ' Ausgabe in Textbox
```

```
    Dim Controller As String          ' Ausgabe in Textbox
    Dim Takt As String                ' Ausgabe in Textbox
    Dim Baud As Integer               ' Ausgabe über Combobox
    Dim Daten As Integer              ' Ausgabe über Combobox
    Dim Stopp As Integer              ' Ausgabe über Combobox
    Dim Parity As Integer             ' Ausgabe über Combobox
    Dim Kopf As String                ' Ausgabe in Textbox
    Dim Auftrag As String             ' Ausgabe in Textbox
    Dim WPuffer As String             ' Ausgabe in Textbox
    Dim RPuffer As String             ' Ausgabe in Textbox
    Dim Sprache As String             ' Ausgabe in Textbox
    Dim Autor As String               ' Ausgabe in Textbox
    Dim Datum As String               ' Ausgabe in Textbox
    Dim Kommentar As String           ' Ausgabe in Textbox
    Dim Anzahl As Integer             ' prüfen, ob Daten vorhanden
    Anzahl = ProjektTableAdapter.Get_Projekt(Id_Nr).Rows.Count
    If Anzahl > 0 Then
        Projekt = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Projekt
        Controller = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Controller
        Takt = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).CPU_Takt
        Baud = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Baud
        Daten = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Datenbit
        Stopp = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Stoppbit
        Parity = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Parity
        Kopf = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Kopf
        Auftrag = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Auftrag
        RPuffer = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).RPuffer)
        WPuffer = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).WPuffer)
        Sprache = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Sprache
        Autor = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Autor
        Datum = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Datum
        Kommentar = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Kommentar
    End If
End Sub
```

**Hinweis:** Sollte die Zeile ProjektTableAdapter schon bei der Eingabe nicht unterstützt werden und eine Fehlermeldung verursachen, ist der Tableaddapter dem Programm nicht bekannt. Abhilfe schafft wie in Besondere Hinweise erklärt, eine Datenquelle aus dem Dataset auf die Programmoberfläche zu ziehen. Dabei wird ein DataGridview-Objekt erzeugt und die Referenzen zum Tableadapter. Das DataGridView – Objekt wird gelöscht und im unteren Bereich lassen wir nur den/ die Tableadapter und das Dataset stehen.

Das war eigentlich schon alles.

Nun suchen wir für die Subroutine **Load_Projekt** ein geeignetes Ereignis. Also, wann soll ein Projekt geladen werden

A Beim Programmstart

B mit einem Wechsel in der Combobox, wo alle Projekte abgelegt werden.

Ok, dann nehmen wir erst einmal das Ereignis SelectedIndexChanged der Combobox Projekte und lassen uns den ersten Datensatz anzeigen.

```
Private Sub Cb_Projekte_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Projekte.SelectedIndexChanged
        Load_Projekt(1)
 End Sub
```

Nun noch einen Haltepunkt auf die Zeile End If in der Subroutine Load_Projekt und dann das Ereignis auslösen. Allerdings ist in dieser Liste noch nichts enthalten und so kann das Ereignis nicht eintreten. Aber da wissen wir uns zu helfen. Mit ein paar Items, die durch die Eigenschaft Auflistung eingetragen werden ist dieses Problem erst einmal gelöst.

Nun ist ein Wechsel mit der Combobox Cb_Projekte möglich und das Programm wird auch an der Zeile End If angehalten. Die Inhalte der Variablen sind allerdings noch leer. Ist ja auch klar, es gibt noch gar keinen Eintrag.

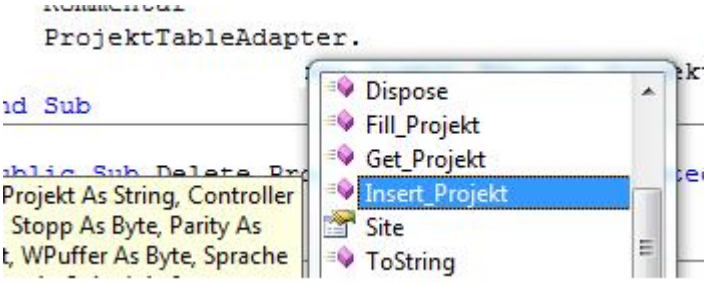
Sollte trotz aller Sorgfalt etwas nicht funktionieren, muss der Fehler ausgemerzt werden. Erst, wenn das Programm wie erwartet reagiert, ist mit dem nächsten Schritt fortzusetzen und der Aufruf der Anweisung Insert_Projekt auf dem TableAdapter umzusetzen.

### 1.3.4 Projektdaten speichern

Damit sind wir bei der Subroutine Insert. Auch hier deklarieren wir erst einmal für alle Spalten eine Variable. Der Einfachheit halber kopieren wir den Variablenbereich der Subroutine Load ohne die Variable Anzahl. Die wird hier nicht benötigt.

Nun weisen wir jeder Variablen einen Wert zu. Beim Datum greifen wir mit Date.Today auf das aktuelle Datum zurück.

Bis hier war alles einfach, oder? Nun folgt der Aufruf der Anweisung Insert_Projekt auf dem TableAdapter. Auch diesmal unterstützt die Onlinehilfe. Beginnen wir mit dem Namen des Tableadapters und einem Punkt.




*Onlinehilfe InsertProjekt*

Nun, einfacher geht es wirklich nicht. Der Hilfstext im gelben Feld weist darauf hin, das anschließend eine Klammer geöffnet und die Werte in die Parameterliste einzutragen sind. Wir setzen dafür die bereits vorbesetzten Variablen ein. Dabei wird der aktuell zu übergebene Parameter hervorgehoben. Sind alle Parameter eingetragen wird die Klammer geschlossen.

Die komplette Subroutine ist nun erstellt.

```
Public Sub Insert_Projekt()
    Dim Projekt As String       ' Wert aus Textbox
    Dim Controller As String    ' Wert aus Textbox
    Dim Takt As String          ' Wert aus Textbox
    Dim Baud As Integer         ' Wert aus Index Combobox
    Dim Daten As Integer        ' Wert aus Index Combobox
```

```
    Dim Stopp As Integer        ' Wert aus Index Combobox
    Dim Parity As Integer       ' Wert aus Index Combobox
    Dim Kopf As String          ' Wert aus Textbox
    Dim Auftrag As String       ' Wert aus Textbox
    Dim WPuffer As Integer      ' Wert aus Textbox
    Dim RPuffer As Integer      ' Wert aus Textbox
    Dim Sprache As String       ' Wert aus Textbox
    Dim Autor As String         ' Wert aus Textbox
    Dim Datum As String         ' Wert vom System
    Dim Kommentar As String     ' Wert aus Textbox
    Projekt = "Erstes"
    Controller = "Atmega"
    Takt = "16 MHz"
    Baud = 1
    Daten = 2
    Stopp = 1
    Parity = 1
    Kopf = "VALUE"
    Auftrag = "AB"
    RPuffer = Val("100")
    WPuffer = Val("100")
    Sprache = "Assembler"
    Autor = "ich"
    Datum = Date.Today
    Kommentar = "-"
    ProjektTableAdapter.
    ProjektTableAdapter.Insert_Projekt
        (Projekt, Controller, Takt, Baud, Daten, Stopp, Parity, Kopf, Auftrag, RPuffer,
                                        WPuffer, Sprache, Autor, Datum, Kommentar)
End Sub
```

Schließlich suchen wir uns noch ein geeignetes Ereignis, um die Routine Insert aufzurufen. Wie wäre es mit dem Button Übernehmen auf der Seite Filter. An dieser Stelle werden doch die aus dem Assemblerlisting gewonnenen Variablen in die Tabelle auf der zweiten Seite übertragen. Eigentlich der richtige Zeitpunkt, um das Projekt anzulegen und auch die Variablen auf der Datenbank einzutragen. Dafür braucht es aber erst das Projekt.

Da wir nur diese Routine testen wollen und im Button Übernehmen schon einiges untergebracht ist, setzen wir einfach ein neues Button und rufen in

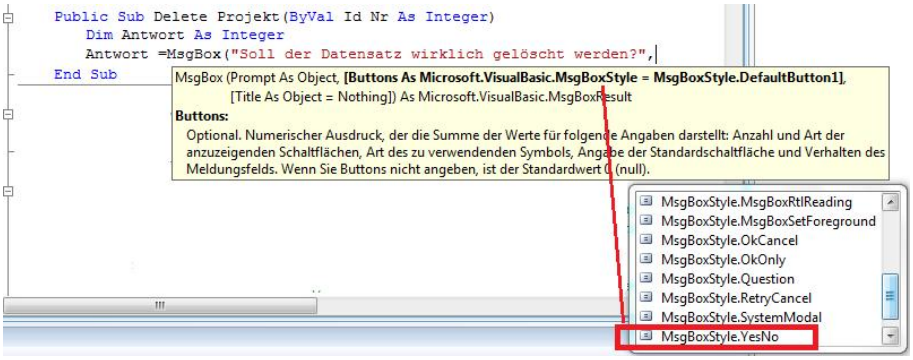
der Click Ereignisroutine diese Subroutine auf. Einen Namen braucht dieses Button nicht, da es nur dem Test dient.

Wenn wir nun anschließend mit der Combobox CB_Projekte einen Wechsel der Auflistung durchführen, wird wieder das Programm am Haltepunkt angehalten und wir können uns die Werte in den Variablen ansehen, indem wir einfach mit dem Mauszeiger eine überdecken, aber nicht anklicken. Ein klein wenig warten, dann wird nach kurzer Zeit der Inhalt angezeigt.

### *1.3.4 Einen Datenbankeintrag löschen*

Das klappt ja schon ganz gut und so wenden wir uns gleich der nächsten Aufgabe zu, einen Eintrag wieder löschen. Dazu hatten wir ja bereits den Rahmen für die Subroutine geschrieben. Lokale Variablen brauchen wir nicht einzutragen, denn wir wollen ja weder Werte eingeben noch erwarten wir welche. Trotzdem ist hier nicht einfach die Tableadapterdelete-Anweisung zu schreiben, sondern es könnte ja sein, dass diese Aktion aus Versehen ausgelöst wurde. Wenn da keine Vorkehrung getroffen würde, wäre der Datensatz futsch. In allen mir bekannten Programmen haben die Entwickler daran gedacht, wenigstens einmal nachzufragen. Aber wie wird diese Nachfrage erstellt?

Nun, wir hatten bereits die Bekanntschaft mit der MsgBox gehabt. Auch für einen solchen Zweck ist sie geeignet. Nur müssen wir einen anderen Stil einsetzen, denn es muss mindestens zwei Möglichkeiten geben. Die Frage mit Ja oder Nein zu beantworten. Anschließend ist auch das Ergebnis interessant, denn davon ist ja abhängig, welche Aktion durchgeführt werden soll. Darum lesen wir die Hinweistexte und sehen, dass eine MsgBox einen Zahlenwert zurückliefern kann. Also doch eine lokale Variable, die diesen Wert zugewiesen bekommt und mit der dann die Entscheidung ausgewertet wird. Aber welchen Stil muss man ansetzen. Visual Basic macht es einfach, wie ein Blick auf den Screenshot zeigt:




*Entscheidung mit MsgBox*

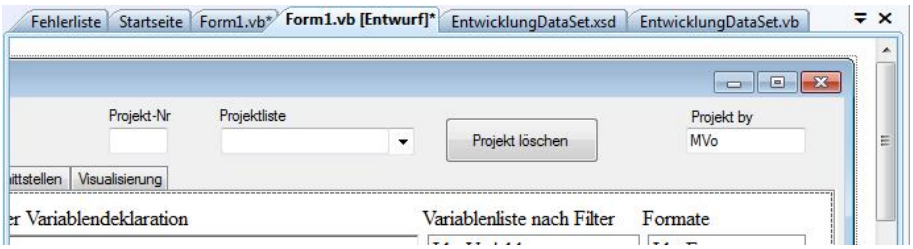
In der aufgeschlagenen Liste mit den Möglichkeiten finden wir den Eintrag MsgBoxStyle.YesNo. Das ist doch das, was wir möchten, also auswählen und für den Titel noch „Warnung“ eintragen.

Was liefert aber die Funktion MsgBox zurück. Die Hilfstexte weisen auf eine Zahl hin. Um das zu prüfen, schreiben wir erst einmal die Subroutine bis auf die Tableadapter Anweisung Delete.

```
Public Sub Delete_Projekt(ByVal Id_Nr As Integer)
      Dim Antwort As Integer
      Antwort = MsgBox("Soll der Datensatz wirklich gelöscht werden?", MsgBoxStyle.YesNo,
"Warnung")
      If Antwort = 1 Then

      End If
   End Sub
```

Nun können wir auf die Zeile **End If** einen Haltepunkt setzen.

Um diese Subroutine zu prüfen ist wieder ein Ereignis erforderlich. Diesmal nehmen wir ein Button und setzen es neben die Textbox **TB_Projek**t im oberen Teil unserer Anwendung.



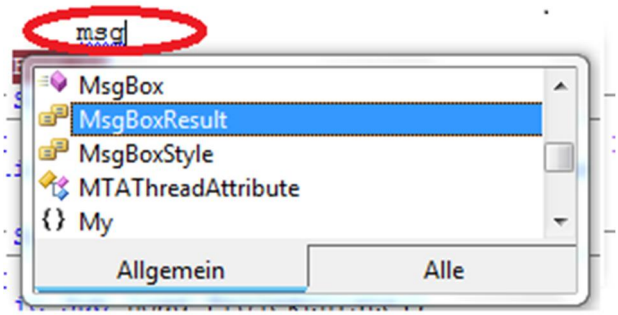

*Button Löschen Projekt*

Das Button wird benötigt und deshalb wird es auch beschriftet und mit dem Namen **Bt_Delete** versehen. Mit einem Doppelklick in das Objekt erhalten wir auch das Ereignis **Bt_Delete_Click**. Dort tragen wir nun den Aufruf der Subroutine **Delete_Projekt** ein.

```
Private Sub Bt_Delete_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Del_Projekt.Click
      Delete_Projekt(1)
End Sub
```

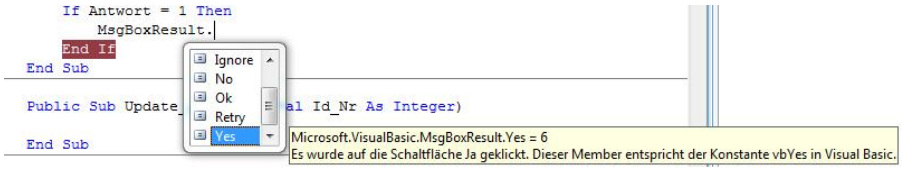
Starten wir nun das Projekt und lassen uns über den Haltepunkt den Wert anzeigen, den die **MsgBox** bei Ja und Nein zurückliefert. Bei mir war es eine 6 für **Ja** und eine 7 für **Nein**. Ist das Zufall? Nein, natürlich nicht. Im Programm sind Konstanten hinterlegt, die solche Werte abbilden. Um

nicht mit Werten, die niemand wirklich zuordnen kann zu arbeiten, werden wir nun die Konstante suchen, die mit dem Inhalt von der Antwort verglichen werden kann. Es ist auf jeden Fall ein Name, der mit **MsgBox** etwas gemeinsam hat. Beginnen wir mit der Eingabe **Msg** und erhalten sofort die Liste der möglichen Objekte.




*Onlinehilfe Auswahl MsgBoxResult*

**MsgBox** ist klar, die haben wir. Mit **MsgBoxStyle** haben wir auch schon gearbeitet, bleibt **MsgBoxResult**. Ok, **Result** -> **Ergebnis**, sehen wir es uns einmal an und versuchen mehr zu erfahren, in dem wir erst einmal einen Punkt anfügen. Wieder erhalten wir Hinweise und finden auch den




*Die Button einer Msg_Box*

Wenn wir ihn anklicken, bekommen wir wieder einen Hinweis, dass dieser Eintrag der Konstanten VBYes von Visual Basic entspricht. Nun ist klar, dass die Antwort einfach nur mit VbYes verglichen werden muss, um die Entscheidung positiv zu bewerten und die Löschaktion durchzuführen.

Die fertige Routine lautet nun

```
Public Sub Delete_Projekt(ByVal Id_Nr As Integer)
  Dim Antwort As Integer
  Antwort = MsgBox("Soll der Datensatz wirklich gelöscht werden?",
                                                    MsgBoxStyle.YesNo, "Warnung")
  If Antwort = vbYes Then
    ProjektTableAdapter.Delete_Projekt(Id_Nr)
  End If
End Sub
```

Nachdem wir diesen Programmabschnitt getestet haben, müssten wir eigentlich erst einmal die Updateroutine schreiben. Aber ein Test mit unserem provisorischen Button bringt keine Ergebnisse. Auch wenn neue Projekte eingetragen werden, es gelingt uns nicht, diese anzuzeigen.

Der Grund, wir haben eine feste Id_Nr eingetragen und diese natürlich mit dem Test der Deletefunktion auch gelöscht. Ändern wir einmal den Übergabewert der Load_Routine in 2 und rufen dann diese auf. Möglicherweise ist wieder kein Datensatz vorhanden. Das liegt ebenfalls an unserer Projektierung. Schließlich haben wir es nicht erlaubt, den Namen des Projektes doppelt einzutragen. Und mittlerweile ist auch der erzeugte Schlüssel ins Unbekannte gelaufen.

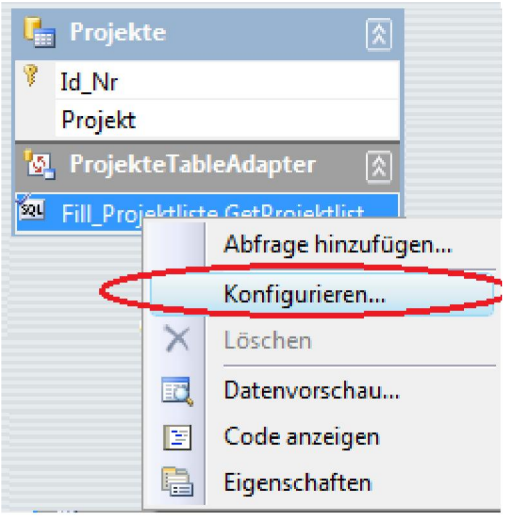
### *1.3.9.4 Eine Liste aller Projektnamen laden*

Es hilft nichts, wir müssen wissen, wie viele Datensätze mit welcher Id_Nr in der Datenbanktabelle Projekte abgelegt sind. Dafür haben wir die Combobox Cb_Projekte vorgesehen. Um die Id_Nr passend mit abzulegen, setzen wir noch eine Listbox ein und geben ihr den Namen Lb_Id_Nr. Dann bekommt sie noch die Eigenschaft unsichtbar über Visible. Dann kann die Listbox so verkleinert werden, dass sie über die Combobox passt.


*Liste Projekt_Id*

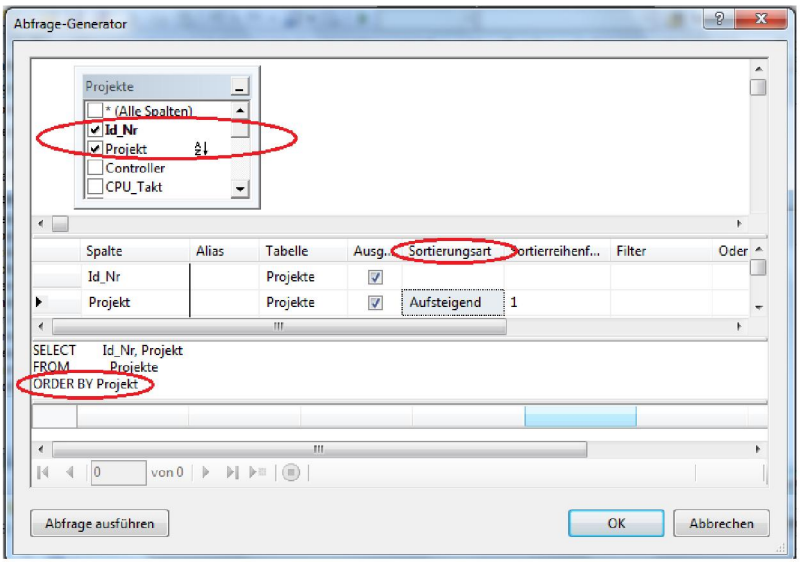
Um nun alle Projekte zu laden, brauchen wir nur die Id_Nr und den Projektnamen, um sie dann in die Listbox. Lb_Id_Nr und die Combobox CB_Projekte einzutragen

Auf der Seite 168 haben wir bereits von der Projekttabelle einen Abzug auf den Datenbankdesigner gezogen und mit dem Namen Projekte benannt. Diesen werden wir nun herrichten, dass er uns alle Projektnamen und die zugehörigen Schlüssel liefert. Die restliche Information der Zeilen wird nicht benötigt. Wir fischen sozusaggen die Fleischbrocken aus der Suppe. Dazu wechseln wir in den Datenbankdesigner und klicken mit der rechten Maustaste in die bereits vorhandene Fill Methode des Tableadapters.




*Tableadapter Projekte*

Mit konfigurieren gelangen wir in den Assistenten, der uns ja schon vom Tableadapter Projekt bekannt ist. Auch hier schließen wir die **Insert**, **Update** und **Delete** Anweisungen in den **erweiterten Optionen** aus. Anschießend werden im Abfragegenerator die zwei Spalten **Id_Nr** und **Projekt** in der Tabelle selektiert.




*SQL Anweisung nur Id_Nr und Projekt*

Weitere Spalten werden nicht benötigt. Allerdings wär es schön, die Projekte in der Liste sortiert zu bekommen. Dafür gibt es die Spalte **Sortierungsart**, in der Aufsteigend ausgewählt wird. Die SQL-Anweisung wird mit einer neuen Klausel ergänzt. **Order by Projekt** sortiert aufsteigend. Nach und nach lernen wir so einiges zu **SQL**-Anweisungen.

Mit **OK** und Weiter gelangen wir zur Namensvergabe. Erforderlich ist nur die **Get**-Methode. Deshalb wählen wir **Fill** ab und vergeben für **Get** den Namen **Get_Projektliste**.

Bleibt noch die Routine in unserer Applikation, die diesen Tableadapter aufruft und das Ergebnis in die Combobox einträgt und ebenso die **Id_Nr** in die Listbox.

Schreiben wir erst einmal den Rumpf für die Subroutine und bestimmen die erforderlichen lokalen Variablen.

Da wir sicherlich mit vielen Ergebnissen rechnen, ist eine Schleife unumgänglich, die Daten im Tableadapter in die Listen zu kopieren. Dazu deklarieren wir die bekannte Schleifenvariable **I**. Hinzu kommt der Grenzwert der Schleife, der in eine Variable **Anzahl** gepackt wird.

```
Public Sub Load_Projekte()
      Dim I As Integer
      Dim Anzahl As Integer
    Dim Projekt as String
    Dim Id_Nr as Integer
 End Sub
```

Bevor diese Routine irgendetwas macht, wird geprüft, ob Datensätze vorliegen. Dafür wird die Variable **Anzahl** benutzt. Sie bekommt den Wert von **Rows.Count**, den die Tableadapterfunktion liefert. Dieses finden wir wieder mit Hilfe der gelieferten Information, wenn hinter einem Objektnamen ein Punkt eingegeben wird. Erscheinen keine Informationen ist auch nichts mehr untergeordnet und entweder werden Parameter in einer Klammer eingetragen oder das Ende der Objekthierarchie ist erreicht. Daher ist es Anfangs wichtig, die Informationen wahrzunehmen und auch zu lesen.

```
Anzahl = ProjekteTableAdapter.GetProjektliste.Rows.Count
```

Nun haben wir einen Wert, der geprüft werden kann. Ist er >0 sind Datensätze vorhanden. Dementsprechend folgt das weitere Programm

```
If Anzahl > 0 Then
   Cb_Projekte.Items.Clear()          ' falsche Aufrufposition
   LB_Id_Nr.Items.Clear()
   For I = 0 To Anzahl - 1

   Next
End If
```

Ist das so korrekt? Ist es richtig, dass die beiden Listobjekte erst von vorherigen Einträgen befreit werden, wenn im Tableadapter Datensätze enthalten sind?

Nein, die Löschung der alten Einträge in den beiden Listen muss vorher geschehen. Warum?

Angenommen, es ist ein Eintrag in der Liste enthalten. Nun wird dieser Eintrag auf der Datenbank gelöscht und die neue Zuweisung an die Variable Anzahl ergibt 0. Dann bleiben die alten Einträge in der Liste enthalten und verursachen möglicherweise ein Fehlverhalten. Also, nochmal

```
Cb_Projekte.Items.Clear()          ' richtige Aufrufposition
LB_Id_Nr.Items.Clear()
If Anzahl > 0 Then
   For I = 0 To Anzahl - 1

   Next
End If
```

So ist es nun richtig. Sind keine Daten im Tableadapter dürfen in der Liste auch keine sein.

Nun stellt sich die nächste Frage. Warum Anzahl -1 in der For Anweisung. Auch dies wird uns immer wieder begegnen. Ein Controller beginnt seine Zählung immer bei 0. Zählt mal eure Finger, beginnend mit 0, ihr kommt nur bis 9. (10-1) Auch unser Zahlensystem arbeitet letztendlich mit Zahlen von 0 bis 9. Bei einem Wert 10 haben wir 1 Zehner und 0 Einer. Also, einen Übertrag in die höhere Stelle. Darum, immer daran denken, wenn es um Tabellen und Listeneinträge geht. Immer Anzahl – 1 ist der letzte Index der Auflistung.

Fehlen noch die Zuweisungen der Daten vom Tableadapter an die Combobox und die Listbox. Beide Listen haben Textablagen. Der Projektname ist Text, die **Id_Nr** nicht. Diese muss von Integer nach Text konvertiert werden. Diese Aufgabe ist ja nichts neues, das erledigt die Funktion **Str(<Zahl>)**. Statt Zahl kommt der Wert vom Tableadapter hinein. Schließlich werden noch die Daten des ersten gefundenen Datensatzes in den Textboxen **Tb_Projekt_Id** und **Tb_Projekte** zur Ansicht gebracht. Außerdem werden wir abhängig vom Ergebnis noch das Button **Projekt löschen** freigeben. Nun die komplette fertige Routine

```
Public Sub Load_Projekte()
  Dim I As Integer
  Dim Anzahl As Integer
  Dim Projekt As String
  Dim Id_Nr As Integer
  Anzahl = ProjekteTableAdapter.Get_Projekte.Rows.Count
  Cb_Projekte.Items.Clear()                    ‘ Listen leeren
  LB_Id_Nr.Items.Clear()
  If Anzahl > 0 Then
    For I = 0 To Anzahl – 1
      Projekt = ProjekteTableAdapter.Get_Projekte.Item(I).Projekt ‘ Projektnamen
      Id_Nr = ProjekteTableAdapter.Get_Projekte.Item(I).Id_Nr     ‘ Id_Nummer
      Cb_Projekte.Items.Add(Projekt)                              ‘ in Liste eintragen
      LB_Id_Nr.Items.Add(Str(Id_Nr))
    Next
    Tb_Projekte.Text = Cb_Projekte.Items(0)                       ‘ erstes Projekt
    Tb_Projekt_Id.Text = LB_Id_Nr.Items(0)                        ‘ anzeigen
  End If
   Bt_Del_Projekt.Enabled = Anzahl > 0                            ‘ löschen freigeben ?
   Pn_Einstellung.Enabled = Anzahl > 0                            ‘ editieren freigeben ?
End Sub
```

.Diese Zeilen verdienen eine besondere Erklärung.

```
Bt_Del_Projekt.Enabled = Anzahl >0
Pn_Einstellung.Enabled = Anzahl > 0
```

**Bt_Del_Projekt** ist ein Button, welches aktiv sein darf, wenn ein Projekt existiert und gesperrt, wenn Anzahl = 0. Nun die Erklärung zur Befehlsstruktur anhand des Button. Somit wäre ja eine Zuweisung **Enabled = True** oder **Enabled = false** logisch.

Der Programmabschnitt wäre demnach

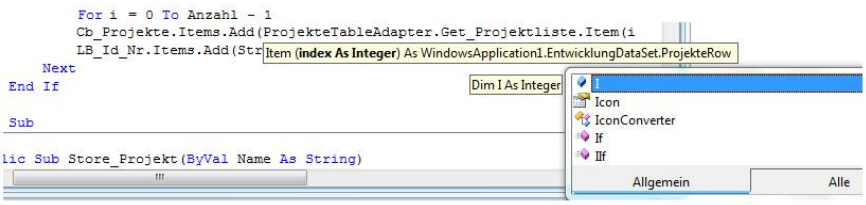
```
If Anzahl > 0 then
  Bt_Del_Projekt.Enabled =True
Else
  Bt_Del_Projekt.Enabled =false
End if
```

Nun hat die Aussage Anzahl > 0 auch die Bedeutung einer wahren oder unwahren Aussage und so betrachtet, das Ergebnis **True** oder **False**. Erinnert euch, ähnliche Anweisungen haben wir bereits zur Objekt-Eigenschaft Visible mit dem Gleich-Operator benutzt.

Genauso funktioniert auch dieses Konstrukt, da nach einer **If** Anweisung eine Aussage bewertet wird, ob sie Wahr oder unwahr ist. Ist also Anzahl <= 0, dann ist die Aussage Anzahl >0 falsch ansonsten wahr.

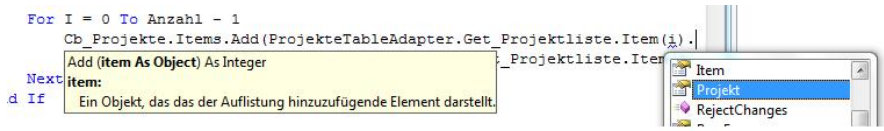
Gleiches gilt für die gesamte Seite und den Editierfunktionen. Wenn noch kein Projekt erzeugt ist dürfen keine Update – Aufrufe erfolgen. Der einfachste Weg, dies zu verhindern ist alles für den Zugriff zu sperren.

Das Erstellen von  Anweisungen, die **Items** in die Listen eintragen, mag kompliziert aussehen, doch das System liefert Informationen, die bei dem Aufbau sehr hilfreich sind, wie der Screenshot zeigt.




*Projektliste laden Onlinehilfe Item*

Das i in der Klammer kommt von der Schleifenvariable, die hier den Eintrag indiziert.  Im nächsten Screenshot sehen wir dann auch die Zuordnung zum Wert, wenn hinter der Klammer wieder ein Punkt gesetzt wird. Diesmal ist es der Eintrag Projekt.




*Projektliste Laden Onlinehilfe Spalte*

Allerdings ist nun die Liste leer, und wir müssen wieder Daten auf die Datenbank schreiben. Wenn wir es nun gut machen wollen und gleich mehrmals auf unser provisorisches Button klicken, bekommen wir eine Fehlermeldung. Warum? Nun, wir haben die Spalte Projekt nur für einen eindeutigen Eintrag zugelassen, also doppelte Einträge abgewiesen. Da wir aber immer denselben Datensatz eintragen ist der zweite schon doppelt und es kommt zum Fehler auf der Datenbank. Trotzdem müsste doch der erste Eintrag sichtbar sein. Nun ja, wäre er, wenn wir die Möglichkeit hätten, ihn erst einmal von der Datenbank herunter zu laden. Deshalb werden wir nun ein paarmal unser Programm starten und den Eintrag Projekt in Projekt 1, 2 usw. jedes Mal abändern um ihn dann in die Datenbanktabelle zu laden. Zusätzlich setzen wir hinter die Tableadapter-Insert Anweisung den Aufruf der Subroutine **Load_Projekte**.

```
Projekt = "Erstes3"            '' mehrfach ändern und dann auf datenbank abspeichern
' restliche Zuweisungen bleiben
ProjektTableAdapter.Insert_Projekt(Projekt, Controller, Takt, Baud, Daten, Stopp,
                Parity, Kopf, Befehl, RPuffer, WPuffer, Sprache, Autor, Datum, Kommentar)
Load_Projekte()            ' Nun die Liste wieder neu laden
```

Nun sollten die erfassten Einträge in der Combobox sichtbar sein.

### *1.3.4 Ein Projekt zur Bearbeitung auswählen*

Erweitern wir nun die Ereignisroutine **SelectedIndexChanged** der Combobox **Cb_Projekte**, um das angewählte Projekt in der Textbox sichtbar zu machen und die **Id_Nr** vom Datenbankschlüssel in die Textbox **Tb_Projekt_Id** zu legen. Zur Auffindung der Position des Schlüssels zum Projekt behelfen wir uns mit einer lokalen Variable Projekt_Pos.

Im ersten Schritt übertragen wir den ausgewählten Eintrag aus der Combobox in die Textbox. Mit der Funktion **IndexOf** der Combobox können wir uns den Index des Eintrags holen und in die Variable **Projekt_Pos** übertragen. Damit indizieren wir nun den zugehörigen Schlüssel in der Listbox und übertragen diesen in die Textbox **Tb_Projekt_Id**.

Nun können wir die Routine **Load_Projekt** mit der Schlüsselnummer aufrufen, wobei der Inhalt der Textbox in eine Zahl zurückgewandelt wird.

```
Private Sub Cb_Projekte_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Projekte.SelectedIndexChanged
  Dim Projekt_Pos As Integer                          ' für Position in der Liste
  Tb_Projekte.Text = Cb_Projekte.Text
  Projekt_Pos = Cb_Projekte.Items.IndexOf(Tb_Projekte.Text)   ' Position ermitteln
  Tb_Projekt_Id.Text = LB_Id_Nr.Items(Projekt_Pos)          ' und Schlüssel eintragen
  Load_Projekt(Val(Tb_Projekt_Id.Text))
End Sub
```

Damit uns die bereits gespeicherten Projekte in der Combobox verfügbar sind, tragen wir die Routine **Load_Projekte** in das Load-Ereignis unserer Anwendung ein. Das erhalten wir mit einem Doppelklick auf die Entwurfsoberfläche von frm_Open_Eye.

```
Private Sub Frm_Open_Eye_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
      Load_Projekte()

   End Sub
```

Für den nächsten Test ändern wir den Aufruf der Subroutine **Delete_Projekt** und setzen statt der festen Zahl auch **Val(Tb_Projekt_Id.Text)** ein. Nach dem Aufruf der Delete-Anweisung laden wir die Projektliste neu.

```
Private Sub Bt_Delete_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Del_Projekt.Click
    Delete_Projekt(Val(Tb_Projekt_Id.Text))
  End Sub
```

Und die Änderung in **Delete_Projekt**

```
Public Sub Delete_Projekt(ByVal Id_Nr As Integer)
    Dim Antwort As Integer
    Antwort = MsgBox("Soll der Datensatz wirklich gelöscht werden?", MsgBoxStyle.YesNo, "Warnung")
    If Antwort = vbYes Then
      ProjektTableAdapter.Delete_Projekt(Id_Nr)
      Load_Projekte()
    End If
  End Sub
```

Nun solle nach dem Löschen eines Datensatzes auch immer weniger in der Liste stehen. Diese Art der Tests ist aber nicht so komfortabel und so werden wir nun den Schritt wagen, über die Textbox **Tb_Projekt** einen neuen Projekteintrag durchzuführen. Dazu nutzen wir auch das Button Übernehmen, wie ich es ja bereits angedeutet habe.

### *1.3.9.5 Ein neues Projekt ablegen*

Angefangen hat ja alles mit dem Filter. Nachdem der Test mit dem Assemblerlisting erfolgreich war, hatten wir die Variablen, die Formate und eventuelle Einzelbitbeschreibungen herausgelöst. Mit dem Button **Übernehmen** sind wir dann zur zweiten Seite übergegangen. An diesem Punkt möchte ich das Projekt und auch später, wenn die anderen Tableadapter eingerichtet sind, Variablendaten und Einzelbits auf der Datenbank ablegen. Jedoch ist vorrangig erst ein Projekt abzulegen, damit die **Id_Nr** auch in die anderen Tabellen zur Referenzierung eingetragen werden kann. Und wir wissen, die **Id_Nr** kommt von der Datenbank.

Sehen wir uns einmal die bereits erstellte Programmierung des Button Übernehmen an

```
Private Sub Bt_Uebernahme_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Uebernahme.Click
  Dim i As Integer                  ' Schleifenzähler
  Dim Anzahl As Integer
  Dim lfd_Nr_Var As Integer    ' Zähler für Tabellenzähler Variablen
  Dim lfd_Nr_Bit As Integer     ' Zähler für Tabellenzeile Einzelbit
  Dim Trenn_Pos As Integer   ' Hilfssvariable für Textbearbeitung
  Dim Ref_Str As String          ' Hilfsvariable für Listeneintrag
  DG_Variablen.Rows.Clear()
  Dg_All_Bits.Rows.Clear()
  lfd_Nr_Var = 0
  lfd_Nr_Bit = 0
  Anzahl = Lb_Variablen.Items.Count    ' Schleifengrenze ermitteln
  For i = 0 To Lb_Variablen.Items.Count – 1
    Ref_Str = Lb_Variablen.Items(i)     ' Eintrag aus Variablenliste holen
    Trenn_Pos = InStr(Ref_Str, ".")     ' Position vom Punkt ermitteln
    If Trenn_Pos = 0 Then                ' kein Punkt, dann ist das Variablenname
      DG_Variablen.Rows.Add()
      DG_Variablen.Item("CL_Id_Nr", lfd_Nr_Var).Value = lfd_Nr_Var + 1
      DG_Variablen.Item("CL_Projekt", lfd_Nr_Var).Value = 0
      DG_Variablen.Item("CL_Variable", lfd_Nr_Var).Value = Ref_Str
      DG_Variablen.Item("CL_Format", lfd_Nr_Var).Value =
                                  Lb_Formate.Items(i)
      DG_Variablen.Item("CL_Freigabe", lfd_Nr_Var).Value = "Ja"
      DG_Variablen.Item("CL_aktiv", lfd_Nr_Var).Value = "Ja"
      DG_Variablen.Item("CL_Trigger", lfd_Nr_Var).Value = "0"
      lfd_Nr_Var = lfd_Nr_Var + 1
    Else    ' hier ist es eine  Einzelbitinfo
      Dg_All_Bits.Rows.Add()
```

```
        Dg_All_Bits.Item("CL_Bit_Id", lfd_Nr_Bit).Value = lfd_Nr_Bit + 1
        Dg_All_Bits.Item("CL_Proj_Id", lfd_Nr_Bit).Value = 0
        Dg_All_Bits.Item("CL_Var_Id", lfd_Nr_Bit).Value = lfd_Nr_Var
        Dg_All_Bits.Item("CL_BitNr", lfd_Nr_Bit).Value = "Bit " + Mid(Ref_Str, Trenn_Pos
                                                                                    + 5, 1)
        Trenn_Pos = InStr(Ref_Str, ":")
        Ref_Str = Mid(Ref_Str, Trenn_Pos + 1, Len(Ref_Str) - Trenn_Pos)
        Dg_All_Bits.Item("CL_BitFunktion", lfd_Nr_Bit).Value = Ref_Str
        lfd_Nr_Bit = lfd_Nr_Bit + 1
      End If
      TC_Auswahl.SelectTab(1)  ' Hier erfolgt der Wechsel zur nächsten seite
      TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
       Set_Edit_Variable(0)
   Next i
End Sub
```

Wenn wir mit diesem Button ein neues Projekt anlegen möchten würde ich vorschlagen, diese Programmzeilen in ein separates Unterprogramm zu packen. Die Funktionalität ist geprüft und war in Ordnung. Deshalb kopieren wir alles und schreiben eine Subroutine **Public Sub Filter_eintragen**.

```
Public Sub Filter_Eintragen()
' Hier steht nun der Inhalt der Ereignismethode vom Button Übernehmen

 End Sub
```

Was im Button-Ereignis noch auftaucht ist der Aufruf der Subroutine

```
Private Sub Bt_Uebernahme_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Uebernahme.Click
      Filter_Eintragen()
 End Sub
```

### *1.3.4 Projektnamen prüfen*

Nun gilt es diesen Aufruf nur zuzulassen, wenn ein gültiger Projektname vergeben wurde. Das soll in der Textbox **Tb_Projekt** erfolgen. Natürlich ist es möglich, ein **Leave**-Ereignis der Textbox zu nutzen, um einen gültigen Namen zu prüfen und dann erst das Button **Übernehmen** freigeben. Schließlich kann man ja auch einzelne Objekte für den Zugriff sperren. Also, mit dem Button **Filter** sowie der Ereignismethode **Textchanged** der Richtextbox wird der Text in der Textbox **Tb_Projekt** mit **Neu** überschrieben.

```
Private Sub Rt_Buffer_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Rt_Buffer.TextChanged
    Tb_Projekte.Text = "Neu"
End Sub
```

Das Button **Übernehmen** wird nur freigegeben, wenn ein gültiger neuer Projektname vorliegt. Nachdem die Daten übertragen sind wird der Zugriff wieder gesperrt. Das bedeutet aber auch, dass dieser Button schon beim Programmstart nicht freigegeben sein darf. Die Stelle, den Button zu sperren kennen wir. Es ist das **Load** Ereignis der Applikation. Tragen wir dort auch den Befehl **Bt_Uebernehmen.Enabled = False** nach.

```
Private Sub Frm_Open_Eye_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Bt_Uebernahme.Enabled = False
    Load_Projekte()
End Sub
```

Kommen wir nun zur Textbox **Tb_Projekte** und dem Ereignis **Leave**. Dieses suchen wir im Eigenschaftsfenster unter den Ereignissen und generieren mit einem Doppelclick den Rahmen für die Routine.

```
Private Sub Tb_Projekte_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Tb_Projekte.Leave

End Sub
```

Nun, was ist zu tun. Wir wissen, das bereits vorhandene Projekte in der Combobox **Cb_Projekte** gelistet sind und das es eine Funktion **Items. IndexOf** gibt, die den Index eines Eintrags zurück liefert, wenn der Text schon in der Liste steht. Ist er nicht vorhanden, wird -1 zurückgegeben. Gut, das lässt sich verwenden. Allerdings, und das ist vielleicht nicht

gerade erwünscht, wird ein Leerzeichen hinter oder vor dem Eintrag zwar nicht gelesen, aber beim Vergleich nicht erkannt. So sind **Projekt 1** und **Projekt 1** zwei unterschiedliche Einträge. Ok, schwer zu erkennen, daher nochmal

**Projekt_1** und **_ Projekt_1_** (die Unterstriche stehen für Leerzeichen)

Um das zu verhindern muss aus dem Text in der Textbox das führende und eventuell abschließende Leerzeichen entfernt werden. Dafür gibt es die Funktion **Trim**

Beginnen wir wieder damit, die lokalen Variablen zu deklarieren. Es sind verschiedene Bedingungen die geprüft werden müssen. Ein neuer Name muss vergeben werden, Das Ergebnis der Prüfung legen wir in einer booleschen Variable ab. Auch das Ergebnis der Prüfung, ob überhaupt ein Name vergeben wurde oder ob immer noch **Neu** im Textfeld steht ist vom Typ Bool und letztlich ist noch zu erfahren, ob überhaupt gefilterte Daten vorliegen. Schließlich brauchen wir noch einen Referenztext, um die Leerzeichen zu entfernen und eine weitere Textvariable für die Information an die **MsgBox**.

Somit ergibt sich das folgende Programm in der Ereignisroutine **Leave**

```
Private Sub Tb_Projekte_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Tb_Projekte.Leave
   Dim Ref_Text As String
   Dim Name_Neu As Boolean
   Dim Name_Ok As Boolean
   Dim Filter_Ok As Boolean
   Dim Information As String
   Information = ""                          ‘ Information ist leer
   Ref_Text = Tb_Projekte.Text               ‘ Kopie aus der Textbox
   Ref_Text = Trim(Ref_Text)                 ‘ Leerzeichen entfernen
   Name_Neu = Cb_Projekte.Items.IndexOf(Ref_Text) < 0   ‘ 1. Bedingung
   Name_Ok = Tb_Projekte.Text <> "Neu"                  ‘ 2. Bedingung
   Filter_Ok = Lb_Variablen.Items.Count > 0             ‘ 3. Bedingung
   If Name_Neu And Name_Ok And Filter_Ok Then
      Bt_Uebernahme.Enabled = True                      ‘ Freigabe
   Else              ‘ Information, welche Bedingung nicht erfüllt ist
      If Not Name_Neu Then Information =
 Information + "Bitte neuen Namen wählen. Projekt schon vorhanden" + Chr(13)
      If Not Name_Ok Then Information =
```

```
 Information + "Neu ist kein gültiger Name" + Chr(13)
     If Not Filter_Ok Then Information =
  Information + "Es liegen keine Daten zur Übernahme vor. Bitte erst den Filter benutzen" +
Chr(13)
     MsgBox(Information, MsgBoxStyle.OkOnly, "Information")
   End If
 End Sub
```

Diesen Abschnitt testen wir erst einmal und betrachten dabei den Button **Übernehmen**. Nun etwas ungewöhnlich ist diese Bedienung, Hat man einen Namen in der Textbox eingetragen, kann man den Button nicht anklicken, weil er ja passiviert ist. Dadurch wird aber die Ereignisroutine **Leave** gar nicht ausgelöst, denn der Focus bleibt bei der Textbox. Man muss also zusätzlich ein anderes Objekt anklicken, damit die Prüfung in der Ereignisroutine **Leave** den Button freigibt oder den Hinweis liefert, warum keine Freigabe erfolgt. Da scheint es doch eleganter, diese Prüfung in das Ereignis **Click** vom Button zu legen und auf das Ereignis **Leave** zu verzichten. Also, die Zuweisung an die Eigenschaft **Enabled** vom Button alle wieder entfernen und das Programm aus dem **Leave** Ereignis in das **Click** Ereignis übertragen. Den Befehl **Bt_Uebernehmen.Enabled =True** ersetzen wir mit dem Aufruf der Subroutine **Filter_Eintragen**.

```
Private Sub Bt_Uebernahme_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Uebernahme.Click
    Dim Ref_Text As String
    Dim Name_Neu As Boolean
    Dim Name_Ok As Boolean
    Dim Filter_Ok As Boolean
    Dim Information As String
    Information = ""
    Ref_Text = Tb_Projekte.Text
    Ref_Text = Trim(Ref_Text)
    Name_Neu = Cb_Projekte.Items.IndexOf(Ref_Text) < 0
    Name_Ok = Tb_Projekte.Text <> "Neu"
    Filter_Ok = Lb_Variablen.Items.Count > 0
    If Name_Neu And Name_Ok And Filter_Ok Then
       Filter_Eintragen()
    Else
       If Not Name_Neu Then Information = Information + "Bitte neuen Namen wählen.
Projekt schon vorhanden" + Chr(13)
```

```
        If Not Name_Ok Then Information = Information + "Neu ist kein gültiger Name" +
Chr(13)
        If Not Filter_Ok Then Information = Information + "Es liegen keine Daten zur
Übernahme vor. Bitte erst den Filter benutzen" + Chr(13)
        MsgBox(Information, MsgBoxStyle.OkOnly, "Information")
    End If
  End Sub
```

Im Ergebnis ist diese Überprüfung doch komfortabler.

Nun wollen wir erst einmal, dass der neue Projektname auf der Datenbanktabelle landet. Zu diesem Zweck müssen wir den Aufruf für die Subroutine **Insert_Projekt** noch vor dem Aufruf **Filter_Eintragen** hier einbauen, wobei der Projektname auch noch mit übergeben werden soll. Schließlich ist der Text in der Textbox noch im Rohzustand und hat eventuell noch Leerzeichen. Die Variable **Ref_Text** aber ist frei davon. Erweitern wir also die Parameterübergabe in der Subroutine wie folgt

Erst einmal die geänderte Subroutine **Insert_Projekt**

```
Public Sub Insert_Projekt(ByVal Projektname As String)
    Dim Controller As String    ' Wert aus Textbox
    Dim Takt As String          ' Wert aus Textbox
    Dim Baud As Integer         ' Wert aus Index Combobox
    Dim Daten As Integer        ' Wert aus Index Combobox
    Dim Stopp As Integer        ' Wert aus Index Combobox
    Dim Parity As Integer       ' Wert aus Index Combobox
    Dim Kopf As String          ' Wert aus Textbox
    Dim Befehl As String        ' Wert aus Textbox
    Dim WPuffer As Integer      ' Wert aus Textbox
    Dim RPuffer As Integer      ' Wert aus Textbox
    Dim Sprache As String       ' Wert aus Textbox
    Dim Autor As String         ' Wert aus Textbox
    Dim Datum As String         ' Wert vom System
    Dim Kommentar As String     ' Wert aus Textbox
    Controller = "Atmega"
    Takt = "16 MHz"
    Baud = 1
    Daten = 2
    Stopp = 1
    Parity = 1
    Kopf = "VALUE"
```

```
    Befehl = "AB"
    RPuffer = Val("100")
    WPuffer = Val("100")
    Sprache = "Assembler"
    Autor = "ich"
    Datum = Date.Today
    Kommentar = "-"
    ProjektTableAdapter.Insert_Projekt(Projektname, Controller, Takt, Baud,
                                Daten, Stopp, Parity, Kopf, Befehl, RPuffer, WPuffer,
                                            Sprache, Autor, Datum, Kommentar)
    Load_Projekte()              ' Nach Einfügen Projektliste neu laden
  End Sub
```

Vielleicht ist es aufgefallen, aber im Aufruf der Tableadapteranweisung **Insert** ist jetzt direkt der Projektname eingebaut und wird nicht mehr über eine lokale Variable geführt.

An eines müssen wir unbedingt denken. Nachdem ein neues Projekt angelegt ist, wird die Subroutine Load_Projekte aufgerufen, um alle Projektnamen in die Combobox zur Auswahl einzutragen und ebenso die zugehörigen Id_Nummern in die Listbox Lb_Id_Nr. Hier sollten wir der Subroutine Load_Projekte dem Projektnamen mitgeben, damit dieser aus der Listbox die Id_Nr fischen kann, denn sie ist erst nach dem Rückladen von der Datenbank vorhanden. Darum ergänzen wir diese Routine mit dem Parameter Projektname. Und werten am Ende aus, ob das Projekt aus der Textbox Tb_Projekte wieder hervorgeholt werden soll.

```
Public Sub Load_Projekte(ByVal Projektname as String)
  Dim I As Integer
  Dim Index as Integer
  Dim Anzahl As Integer
  Dim Projekt As String
  Dim Id_Nr As Integer
  Anzahl = ProjekteTableAdapter.Get_Projekte.Rows.Count
  Cb_Projekte.Items.Clear()                    ‘ Listen leeren
  LB_Id_Nr.Items.Clear()
  If Anzahl > 0 Then
    For I = 0 To Anzahl – 1
      Projekt = ProjekteTableAdapter.Get_Projekte.Item(I).Projekt ‘ Projektnamen
      Id_Nr = ProjekteTableAdapter.Get_Projekte.Item(I).Id_Nr     ‘ Id_Nummer
      Cb_Projekte.Items.Add(Projekt)                              ‘ in Liste eintragen
      LB_Id_Nr.Items.Add(Str(Id_Nr))
    Next
     ‘ auswerten, ob ein Projektname eingetragen ist.
```

```
    If Tb_Projekte.Text = „“ then
      Tb_Projekte.Text = Cb_Projekte.Items(0)                    ‘ erstes Projekt
      Tb_Projekt_Id.Text = LB_Id_Nr.Items(0)                     ‘ anzeigen
    Else
      Index = Cb_Projekte.Items.Indexof(Tb_Projekte.Text)        ‘ Position Projekt in Liste
     Tb_Projekt_Id.Text = LB_Id_Nr.Items(Index)                 ‘ aktuelle Projekt Id
    End If
  Bt_Del_Projekt.Enabled = Anzahl > 0                            ‘ löschen freigeben ?
End Sub
```

Ab jetzt rufen wir diese Routine entweder mit einem gültigen Projektnamen auf oder einem Leerstring.

```
Load_Projekte(Tb_Projekte.Text)
```

oder

```
Load_Projekte(“”)
```

Nun noch der Abschnitt in der **Click** Methode vom Button

```
    If Name_Neu And Name_Ok And Filter_Ok Then
       Insert_Projekt(Ref_Text)
       Filter_Eintragen()
    Else
```

Nun können wir nach belieben Testen und Projektnamen vergeben.

Auch die **Delete**-Anweisung funktioniert. Aber dabei fällt ein Fehler auf. Der Text in den Textboxen **TB_Projekt_Id** und **Tb_Projekte** ändert sich dabei nicht. Das bedeutet, wir müssen nach dem neu Laden der Projektnamen auch die Textboxen wieder mit neuen Werten füllen. Das erledigen wir sofort in der Subroutine **Delete_Projekt**. Einfach den ersten gefundenen Datensatz an die Textboxen übertragen.

```
  If Antwort = vbYes Then
```

```
    ProjektTableAdapter.Delete_Projekt(Id_Nr)
    Load_Projekte()
   ' Die Textboxen Tb_Projekt und Tb_Projekt_Id neu besetzen
   If ProjekteTableAdapter.Get_Projekte.Rows.Count > 0 Then
      Tb_Projekte.Text = ProjekteTableAdapter.Get_Projekte.Item(0).Projekt
      Tb_Projekt_Id.Text = Str(ProjekteTableAdapter.Get_Projekte.Item(0).Id_Nr)
    Else
            ' keine Projekte mehr gelistet
       Tb_Projekte.Text = ""
       Tb_Projekt_Id.Text = ""
    End If
  End If
```

Das gleiche Problem haben wir auch bei einem neuen Projekt. Der Eintrag in der Textbox **TB_Projekt_Id** wird nicht eingetragen. Die wird aber für das weitere Vorgehen benötigt. Also muss diesmal wieder die Funktion **IndexOf** der Combobox herhalten, denn den Projektnamen haben wir ja noch in der Textbox **TB_Projekte** stehen. Dafür deklarieren wir in der Subroutine **Insert_Projekt** noch eine Variable **Id_Pos**.

Mit den nachfolgenden Programmzeilen hinter den Aufruf **Load_Projekte** ist das Thema dann auch erledigt.

```
Id_Pos=Cb_Prorjekte.Items.IndexOf(Tb_Projekte.Text)
Tb_Projekt_Id.Text=Lb_Id_Nr.Items(Id_Pos)
```

### *1.3.4 Daten korrigieren*

Bleibt noch, die Projektdaten mit der Korrekturfunktion zu versehen. Den Rahmen für Update haben wir ja schon fertig.

```
Public Sub Update_projekt(ByVal Id_Nr As Integer)

  End Sub
```

Dazu erst einmal wieder überlegen, wie eine Korrektur durchgeführt werden soll. Die Subroutine selbst ist nicht besonders schwer, nachdem wir nun wissen, wie eine **Insert**- und **Delete**-Anweisung an die Datenbank geschickt wurde. **Insert** kommt der Korrektur im Aufbau sehr nahe, und deshalb kopieren wir den Inhalt der **Insert**- in die **Update**-Subroutine.

```
Public Sub Update_projekt(ByVal Id_Nr As Integer, ByVal Projekt As String)
    Dim Controller As String    ' Wert aus Textbox
    Dim Takt As String          ' Wert aus Textbox
    Dim Baud As Integer         ' Wert aus Index Combobox
    Dim Daten As Integer        ' Wert aus Index Combobox
    Dim Stopp As Integer        ' Wert aus Index Combobox
    Dim Parity As Integer       ' Wert aus Index Combobox
    Dim Kopf As String          ' Wert aus Textbox
    Dim Auftrag As String       ' Wert aus Textbox
    Dim WPuffer As Integer      ' Wert aus Textbox
    Dim RPuffer As Integer      ' Wert aus Textbox
    Dim Sprache As String       ' Wert aus Textbox
    Dim Autor As String         ' Wert aus Textbox
    Dim Datum As String         ' Wert vom System
    Dim Kommentar As String     ' Wert aus Textbox
    Controller = "Atmega"
    Takt = "16 MHz"
    Baud = 1
    Daten = 2
    Stopp = 1
    Parity = 1
    Kopf = "VALUE"
    Auftrag = "AB"
    RPuffer = Val("100")
    WPuffer = Val("100")
    Sprache = "Assembler"
```

```
    Autor = "ich"
    Datum = Date.Today
    Kommentar = "-"
    ProjektTableAdapter.Update_Projekt(Projekt, Controller, Takt, Baud,
            Daten, Stopp, Parity, Kopf, Auftrag, Sprache, WPuffer, RPuffer,
                        Autor, Datum, Kommentar, Id_Nr)
End Sub
```

Den Projektnamen übergeben wir wie auch die **Id_Nr** des Datensatzes über die Parameter an die Subroutine, Da wir ja auch den Projektnamen korrigieren wollen, müssen wir diese Korrektur genau so abfangen, wie wir es bei dem **Insert**-Aufruf getan haben. Aber wann soll nun diese **Update**-Anweisung aufgerufen werden? Ich mach es mal einfach.

Jedes Mal, wenn etwas geändert wird, was geändert werden darf und was zur Datenbanktabelle **Projekte** gehört.

Da der Projektname über allen Seiten steht, ist er nicht fest an die Seite **Filter** gebunden und kann überall verändert werden. Diesmal nehmen wir als Ereignis **Leave**, denn nach der Korrektur wird auf jeden Fall ein anderes Objekt angesprochen. Da kann sofort, wenn zulässig, auch die **Update**-Anweisung vom **ProjektTableAdapter** aufgerufen werden. Aber da gibt es einen Haken. Auch beim Erstellen neuer Daten für ein Projekt wird die Subroutine **Leave** ausgelöst, da ja auch die Textbox **Tb_Projekte** verlassen wird. Das geht aber nicht, weil dort bei einem Fehlverhalten in die Textbox **Neu** eingetragen wird. Wenn hier eine unzulässige Eingabe zum Projektnamen stattfindet, möchten wir natürlich nicht **Neu**, sondern bestenfalls den alten Projektnamen vorfinden.

Hier hilft ein kleiner Trick, wir müssen uns den alten Namen merken. Bei einem neuen Projekt war es ja so, das erst ein Ereignis **TextChanged** der **RichTextBox** ausgelöst wurde. Damit kam der Eintrag **Neu** in die Textbox für den Projektnamen. Wenn nun **Neu** als alter Eintrag irgendwo hinterlegt wird, kann das natürlich im Ereignis **Leave** ausgewertet werden. Dazu muss aber diese Ablage des alten Namens global verfügbar sein. Eine globale Variable bietet sich an, oder eine Textbox, die auf unsichtbar geschaltet wird. Da dieses Programm kein Problem mit Speicherbelegung hat, setzen wir eine Textbox und vergeben den Namen **Tb_Projekt_Alt**. Immer, wenn nun die Textbox **Tb_Projekte** mit einem gültigen Inhalt gefüllt wird, bekommt auch die Textbox **Tb_Projekt_Alt** den Namen eingetragen. Das passiert auch mit **Neu**, oder den Ereignissen aus der Combobox oder von der **Projekte_Load**-Routine. Nun kann im Ereignis **Leave** dies ausgewertet und auch der alte Inhalt wieder hergestellt werden. Die Textbox für die Ablage des alten Projektnamens ist für die

Ansicht unwichtig. Deshalb wird sie einfach durch Setzen des TabIndex auf 1 unter die Combobox gelegt.

Um alle Zuweisungen an die Textbox TB_Projekte zu finden, können wir die Schnellsuche nutzen. Diese finden wir im Menü unter Bearbeiten im Untermenü Suchen/Ersetzen. Dahinter fügen wir dann folgende Programmzeile ein.

```
Tb_Projekt_Alt.Text = Tb_Projekte.Text
```

Im Ereignis **Bt_Uebernahme_Click** wird dann hinter dem Aufruf der **MsgBox** entgegengesetzt zugewiesen.

```
Tb_Projekte.Text= Tb_Projekte_Alt.Text
```

Damit wird der alte Name wieder eingetragen, wenn die Korrektur abgelehnt wurde. .

Kommen wir nun zum Ereignis **Leave** der Textbox **TB_Projekte**. Die haben wir ja schon einmal versucht zu benutzen, hier wird es gehen, davon bin ich überzeugt.

```
Private Sub Tb_Projekte_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Tb_Projekte.Leave
  Dim Ref_Text  As String
  Dim Name_Neu As Boolean
  Dim Name_Ok As Boolean
  Dim Information As String
  If  Tb_Projekt_Alt.Text <> "Neu" then
    Information = ""                                        ‘ Information ist leer
    Ref_Text = Tb_Projekte.Text                             ‘ Kopie aus der Textbox
    Ref_Text = Trim(Ref_Text)                               ‘ Leerzeichen entfernen
    Name_Neu = Cb_Projekte.Items.IndexOf(Ref_Text) < 0      ‘ 1. Bedingung
    Name_Ok = Ref_Text <> "Neu"                             ‘ 2. Bedingung
    If Name_Neu  And Name_Ok  And  Filter_Ok Then
      If  Ref_Text <>TB_Projekt_Alt.Text then               ‘ nur bei Änderung
        Update_Projekt( Val(Tb_Projekt_Id.Text), Ref_Text)
      End if
    Else                ‘ Information, welche Bedingung nicht erfüllt ist
       If Not Name_Neu Then Information =
  Information + "Bitte neuen Namen wählen. Projekt schon vorhanden" + Chr(13)
```

```
        If Not Name_Ok Then Information =
  Information + "Neu ist kein gültiger Name" + Chr(13)
        MsgBox(Information, MsgBoxStyle.OkOnly, "Information")
        Tb_Projekte.Text = Tb_Projekt_Alt.Text        ‘ alten Projektnamen eintragen
    End If
  End If
End Sub
```

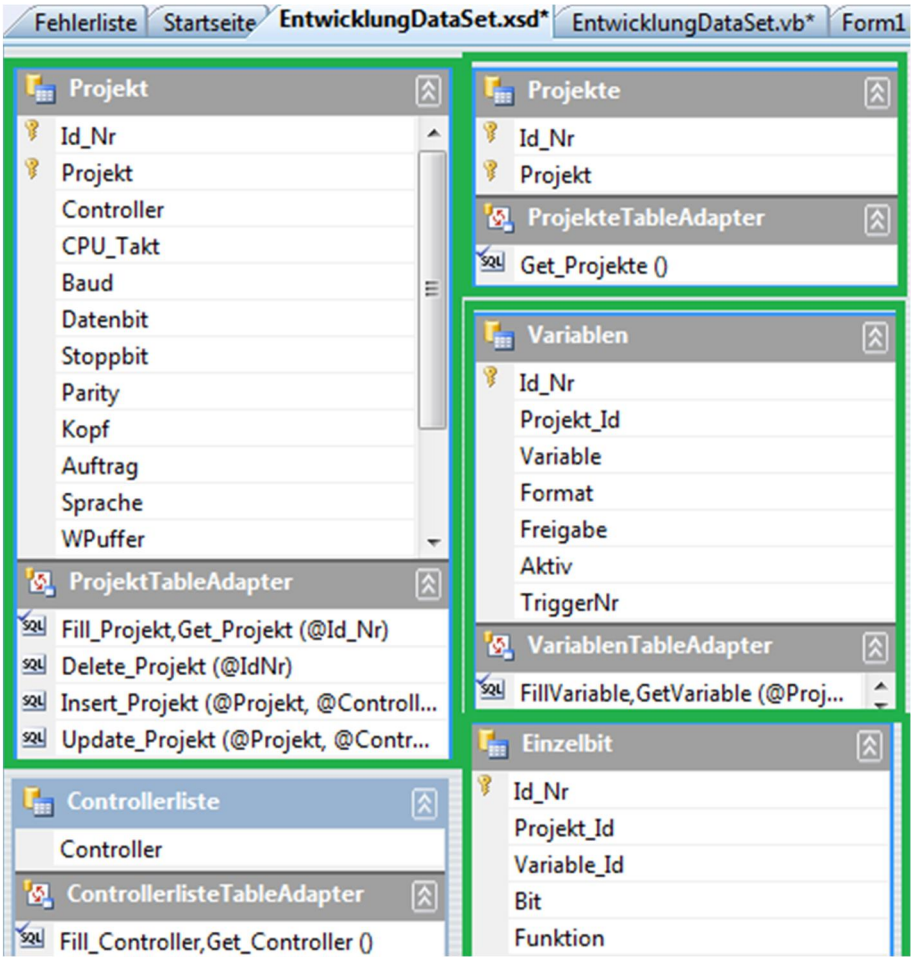
Das alte **Leave** Ereignis wird nun ein wenig abgewandelt. Zuerst wird die Ablage befragt, ob es ein neuer Datensatz ist, denn der Eintrag **Neu** ist nur vorhanden, wenn ein neues Projekt mit neuen Daten in den Filter eingetragen wurden. Dann erfolgt die Prüfung, ob der neue Name schon vorhanden oder ob **Neu** eingetragen wurde. Ist alles Ok und auch der Eintrag in der Textbox **TB_Projekte** unterschiedlich zu **Tb_Projekt_Alt**, dann wird ein Update durchgeführt. Im Moment ist der Projektname auch das einzige, was wir korrigieren können, alle anderen Werte werden erst auf der dritten Seite zugewiesen. Deshalb werden wir nun ein paar Tests durchführen und prüfen, ob unsere Überlegungen funktionieren und auch praktikabel sind. ??? Nun wird durch die Arbeit und Änderungen auf der Datenbank der bereits bestehende Bestand gelöscht und wir müssen immer wieder neue Projekte anlegen..

Will man mehrere Projekte anlegen, so ist es sinnvoll, durch neues Filtern wieder einen Eintrag **Neu** in die Textbox **Tb_Projekte** zu erzwingen. Es wird sonst das **Tb_Projekte_Leave** Ereignis zusätzlich ausgelöst, wenn nach einer Vergabe eines neuen Namens das Button **Übernehmen** angeklickt wird.

Nun sind wir wieder beim Button Übernehmen, da die erzeugten Variablen und Einzelbitbeschreibungen in die Tabellen auf der nächsten Seite Einstellungen geladen werden. Das werden wir nun erweitern, um auch die Datenbanktabellen Variablen und Einzelbit  mit diesen Informationen zu füllen. Die erforderliche Referenz zum Programm haben wir in der Textbox **Tb_Projekt_Id** vorliegen. Die Tabellen sind auf der Datenbank erstellt und auf der Oberfläche **EntwicklungDataSet** sind **TableAdapter** installiert. Beginnen wir erst einmal damit, die **SQL** Anweisungen **Get**, **Fill**, **Insert**, **Delete** und **Update** für die Datenbanktabellen zu generieren. Der Vorgang ist der gleiche wie bei der Projekttabelle und ich werde nur auf Besonderheiten speziell eingehen.

### *1.3.10 Weitere Tableadapter Variablen und Einzelbit*

Die Basic-Seite wird verlassen und in **EntwicklungDataSet.xsd** gewechselt. Sollte der Reiter nicht sichtbar sein, kann diese Seite über den Menüpunkt **Fenster** geöffnet werden. Ein Blick auf einen Ausschnitt vom Screenshot zeigt uns den Fortschritt durch eine farbliche Umrandung. Die grün markierten **Tableadapter** sind fertig und geprüft. Die gelb markierten **Tableadapter** nehmen wir uns nun vor.



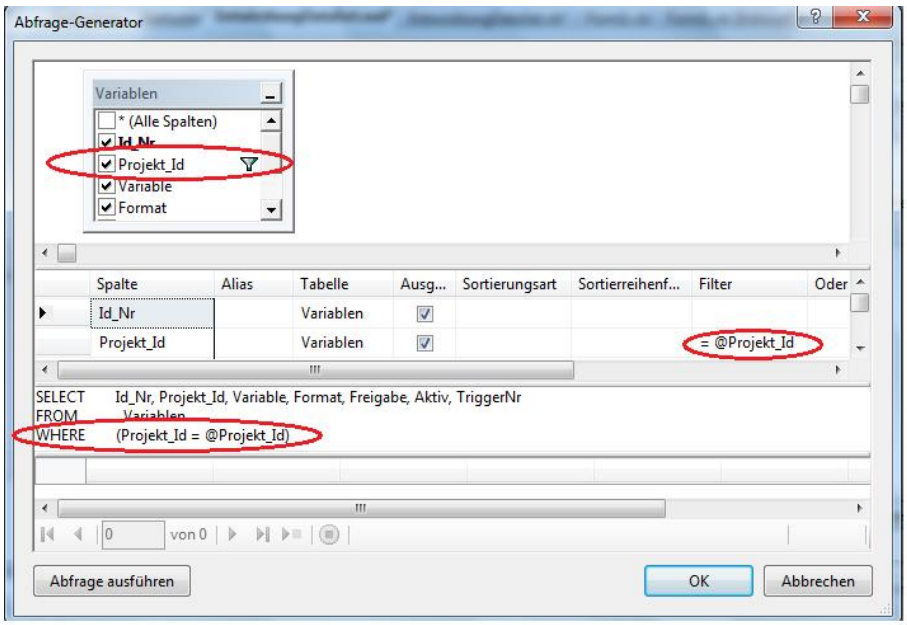

*fertige Tableadapter*

### *1.3.10.1 Tabellendaten Variablen laden*

Bearbeiten wir zuerst den **VariablenTableAdapter**, denn auch für die Ablage der Einzelbitinformationen benötigen wir die **Id_Nr** der zugehörigen Variablen.

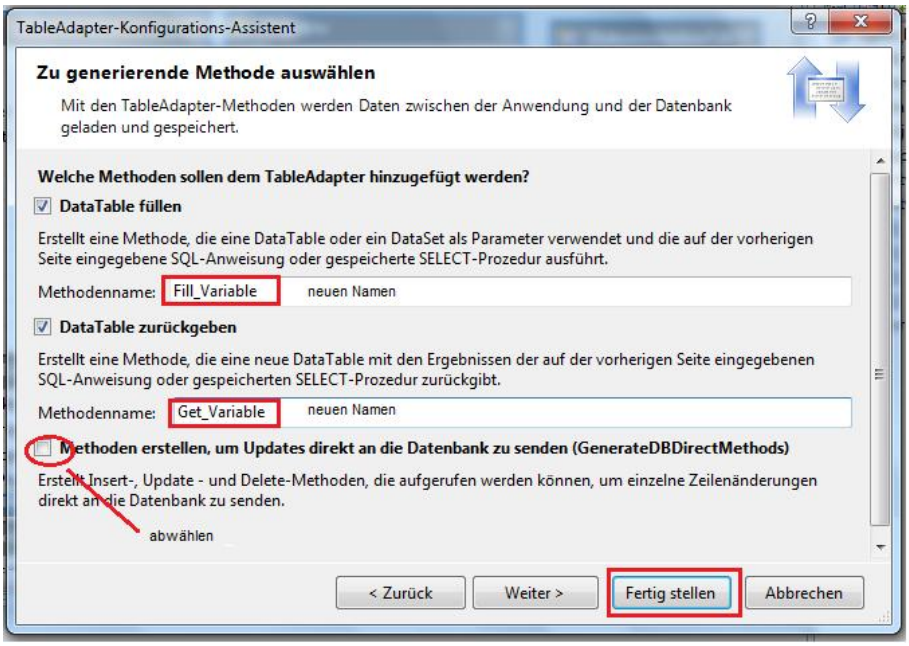
Öffnen wir also den Assistenten durch rechten Mausklick in den Eintrag **Fill-GetData** und die Auswahl **Konfigurieren**. Zuerst wieder in **erweiterten Optionen** die **Insert-Update- und Delete-Anweisung** abwählen und zurück den **Abfrage-Generator** aufrufen.

Kurz nachdenken, was macht die **Get**-Methode. Richtig, sie holt Daten von der Datenbank, und in unserem Fall immer eine ganze Zeile der Tabelle. So wie es beim Projekt auch eingestellt war, allerdings nicht nur einen Datensatz, sondern alle Datensätze, die zum Projekt gehören. Damit das funktioniert, müssen wir den **Filter** auf **Projekt_Id** setzen.




*Variablen vom Projekt laden*

Die **Select**-Anweisung wird um die Klausel **Where (Projekt_Id = @Projekt_Id)** erweitert, wenn wir in den Filter **=@Projekt_Id** setzen. Das war es schon. Nun noch einen Namen für die Methoden vergeben und **Fertig stellen**.




*Vergabe Name*

Nun folgen die **Insert**, **Delete** und **Update-Anweisungen** nach bekanntem Schema.

### *1.3.10.2 Die Insert_Variable Anweisung*

Die **Insert**-Anweisung ist wie beim Projekt aufgebaut. Hinter den Spaltennamen werden die Übergabeparameter eingetragen, das ist der Spaltenname mit vorangestelltem **@**

Wichtig ist hier die Zuweisung der **Projekt_Id**. Sie wird für die Zuordnung zum Projekt benötigt.

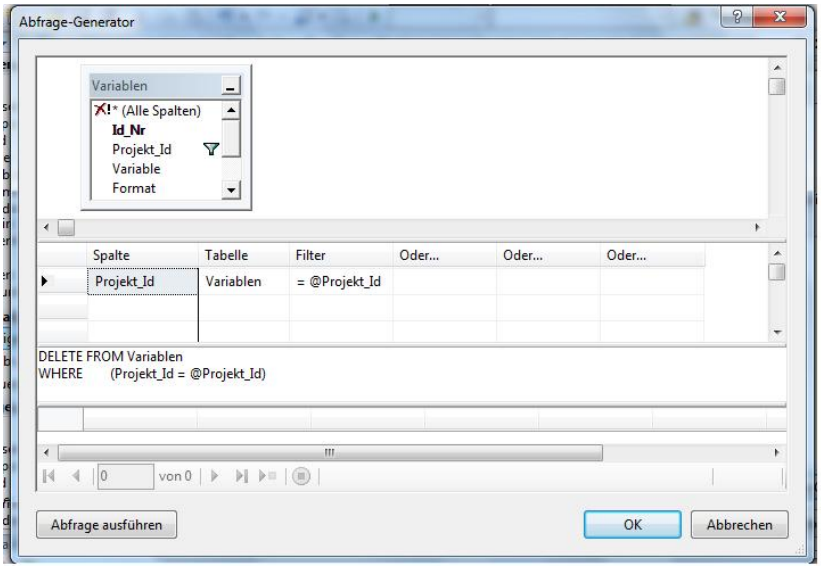
Deshalb werden wir auch gleich passend zur VariablenTableAdapter-Anweisung **Insert_Variablen** die zugehörige **Store_Variable** Subroutine beschreiben. Diesmal werden alle einzutragenden Parameter außer der **Projekt_Id** der Subroutine übergeben

```
Private Sub Store_Variable(ByVal Variable As String, ByVal Format As String,
                ByVal Freigabe As String, ByVal aktiv As String, ByVal Trigger As Integer)
  Dim Projekt_Id As Integer
  Projekt_Id = Val(Tb_Projekt_Id.Text)
     VariablenTableAdapter.Insert_Variable(Projekt_Id, Variable, Format, Freigabe,
                              aktiv, Trigger)
 End Sub
```

Damit ist schon mal die Speicherung für einen Datensatz mit dem Variablennamen abgeschlossen. Der Aufruf erfolgt in der vom Ereignis **Bt_Uebernahme_Click** aufgerufenen Subroutine **Filter_Eintragen** Die erforderlichen Änderungen in dieser Subroutine werde ich nach dem Aufbau der **Insert-Methode** für die Einzelbitinformation ansprechen und die geänderte Subroutine vorstellen.

### *1.3.10.3 Die Delete_Variable Anweisung*

Bei **Delete** macht es keinen Sinn, einen einzelnen Datensatz aus der Variablentabelle herauszulösen. Schließlich erwarten wir die Variablen in einem Block von unserem Controller. Das Löschen eines Datensatzes kann die ganze Struktur durcheinander bringen. Nein, wenn Datensätze in dieser Tabelle gelöscht werden sollen, dann alle zum Projekt gehörenden mit dem Projekt zusammen und deshalb ist der **Filter** wie im Screenshot zu sehen auch auf **Projekt_Id** zu setzen.



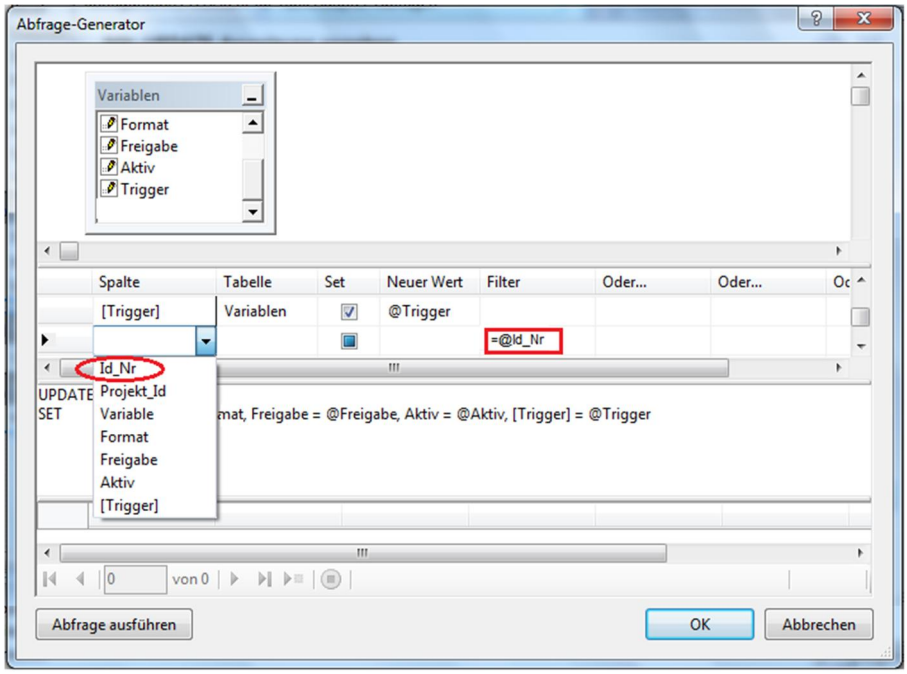

*Projektvariablen löschen*

Die **SQL**-Anweisung hat demnach auch die Klausel **Where ( Projekt_Id = @Projekt_Id)**. Das bedeutet, wird diese **Delete-Anweisung** aufgerufen, werden alle Variablen des Projekts mit einem Rutsch gelöscht. Schließlich wird noch der Name **Delete_Variable** eingetragen und mit **Fertig stellen** die Anweisung abgeschlossen.

Das zugehörige Programm in Visual Basic stellt keine besonderen Ansprüche und ist einfach aufgebaut. Deshalb wird auch keine separate Subroutine geschrieben, sondern die **TableAdapter-Anweisung** gleich in die Ereignisroutine **Bt_Del_Projekt_Click** eingebunden. Auch diese Änderung der Routine bespreche ich nach der Abhandlung der Einzelbitinformation.

### *1.3.10.4 Die Update_Variable Anweisung*

Die Update-Anweisung erwartet wieder den Zugriff auf einen bestimmten Datensatz und das erledigt der **Filter** auf die eigene **Id_Nr**. Da bei dieser Auflistung der Spaltenname **Id_Nr** nicht aufgeführt ist, muss er wieder am Ende der Spaltenliste unter Verwendung des **Dropdown-Feldes** hinzugefügt und der **Filter** gesetzt werden. Ganz wichtig ist außerdem, dass die **Projekt_Id** und der Variablenname nicht verändert werden dürfen. Ist ja klar, die Referenz zum Projekt muss erhalten bleiben. Somit sind nur die Spalten **Format**, **Freigabe**, **Aktiv** und **Trigger** veränderbar.




*Update Variable*

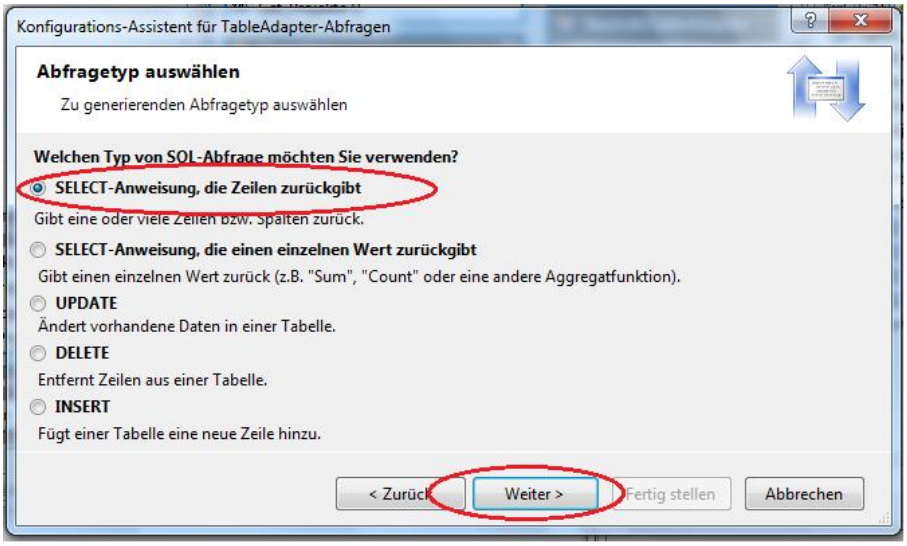
Den Namen dieser Anweisung legen wir mit Update_Variable fest und verlassen den Assistenten mit **Fertig stellen**. Damit ist die Bearbeitung des **VariablenTableadapter** abgeschlossen.

### *1.3.10.5 Der EinzelbitTableadapter*

Wie auch beim **VariablenTableAdapter** beginnen wir mit der **Fill**- und **GetData**-Methode. Diesmal brauchen wir zwei von dieser Sorte. Warum? Nun, einmal wollen wir die Einzelbitbeschreibungen zu einer Variablen holen und einmal die Einzelbitbeschreibungen des Projektes. Das ist alles kein Problem, denn wie das funktioniert wissen wir. Der Filter in der Abfrage ist entscheidend.

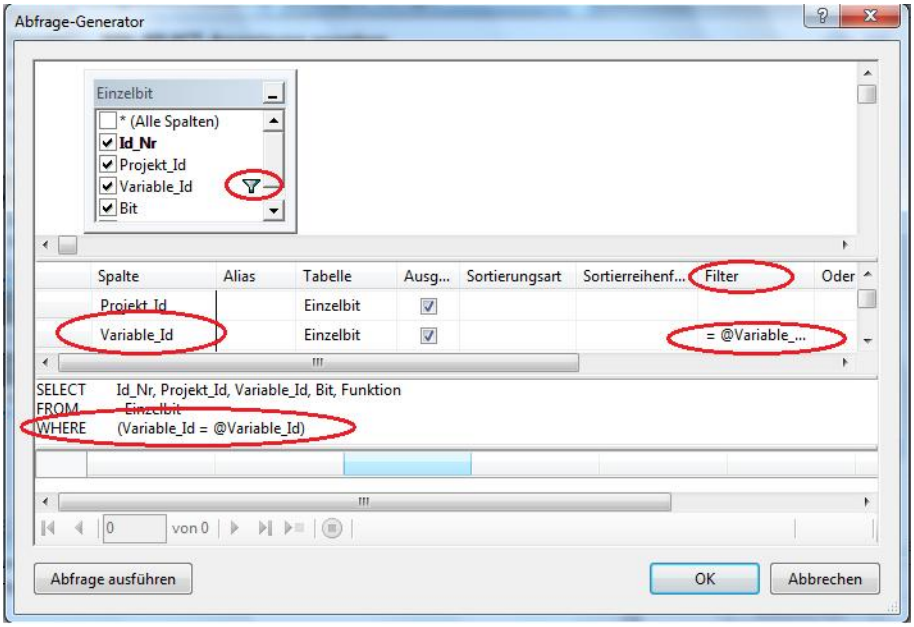
Bei der ersten Abfrage setzen wir diesen Filter auf die **Projekt_Id**, wie bei den Variablen. Diese Methoden bekommen die Namen **Fill_ProjektBits** und **Get_ProjektBits**.

Dann öffnen wir mit Abfrage hinzufügen, also rechte Maustaste in die **Fill**-zeile des **TableAdapters,** den Assistenten und wählen in der übernächsten Seite **Select-Anweisung, die Zeilen zurückgibt**




*Abfrage Einzelbit zu Variablne*

Im nächsten Fenster öffnen wir den Abfrage Generator und setzen den Abfragefilter auf **Variable_Id**.




*Filter Einzelbit Variable*

Diese Abfragen nennen wir **Get_VariablenBits**. Die Fill-Methode wählen wir ab. Diese Vorgehensweise könnte sich als praktisch erweisen. Es ermöglicht nur die zur Variablen gehörenden Bits in die Einzelbittabelle zu laden, ohne die gesamte Tabelle **Dg_All_Bits** zu durchforsten. Auch kann die gesamte Einzelbitbeschreibung in eine Liste zur späteren Visualisierung geladen werden. Und das mit nur einer Anweisung.

### *1.3.10.6 Die Insert_Einzelbit Anweisung*

Nun müssen aber auch die Einzelbitbeschreibungen erst mal auf die Datenbank geschrieben werden. Dafür gibt es wieder die **Insert**-Anweisung, die wir mit Abfrage hinzufügen mit Hilfe des Assistenten erstellen.

Das ist auch diesmal scheinbar keine besondere Herausforderung. Lediglich der Zeitpunkt der Speicherung ist einen Gedanken Wert. Wann kann eine Einzelbitbeschreibung abgelegt werden. Wir haben da zwei ganz wichtige Spalten eingerichtet und bereits bei der Abfrage auch jeweils einen Filter drauf gesetzt. Die **Projekt_Id** ist bereits nach abspeichern des Projektes in der Textbox **Tb_Projekt_Id** enthalten. Nun brauchen wir auch noch die **Variablen_Id**, um den Datensatz der Einzelbitbeschreibung mit den restlichen Daten aus dem Filter zu komplettieren. Daher muss erst die Variable selbst auf die Datenbank geschrieben werden, damit wir durch Rücklesen die **Id_Nr** der Variablen bekommen. Und genau das gestaltet sich ein wenig schwieriger als erwartet. Die Variablen werden immer komplett zum Projekt zurückgelesen. Eine **Id_Nr** kann ich zwischendurch gar nicht erfahren. Wie auch. Nach der ersten Speicherung kennt nur die Datenbank diese Nummer und den Variablennamen kann ich auch für die Abfrage nicht gebrauchen, weil in einem anderen Projekt der gleiche Variablenname vergeben sein kann. Ist ein klein wenig verzwickt, diese Situation. Da hilft aber nix, da müssen wir durch und eine Lösung suchen.

Sehen wir uns doch einmal an, wie wir die Bits beschrieben haben. War nicht erst der Variablenname gefolgt von einem Punkt und dann die Bitnummer in der Variablenliste des Filters angegeben? Kann uns das nicht einen Schritt später helfen, eine **Variablen_Id** passend zur Bitbeschreibung zu finden? Wenn wir jetzt im ersten Schritt nur diesen kompletten Bitnamen ohne eine gültige **Id_Nr** ablegen, bekommen wir es später aber wieder heraus und können diese Referenz nachtragen. Das bedeutet aber, ich muss die **Variablen_Id** für eine Korrektur zulassen und genau das ist auch kontraproduktiv, weil damit eine Verfälschung der Zuordnung gegeben sein kann. Manchmal ist es wirklich nicht einfach, aber noch haben wir Möglichkeiten.

Wenn wir nun erst mal nur die Tabelle **Dg_All_Bits** füllen und später alle Variablen von der Datenbank lesen, dann können wir doch zu diesem Zeitpunkt den Einzelbitbeschreibungen die **Id_Nr** der zugehörigen Variablen verpassen. Anschließend werden dann alle in der Tabelle **Dg_All_Bits** enthaltenen Einträge auf die Datenbank geschrieben. Die

Subroutine **Store_Einzelbits** ist aber von dieser Überlegung erst einmal noch nicht betroffen. Wie auch bei der Variablen werden die Parameter der Subroutine übergeben.

```
Public Sub Store_Einzelbit(ByVal Variable_Id As Integer, ByVal Bit As ByteByVal Funktion as
String)
    Dim Projekt_Id As Integer
    Projekt_Id = Val(Tb_Projekt_Id.Text)
     EinzelbitTableAdapter.Insert_Einzelbit(Projekt_Id, Variable_Id, Bit, Funktion)
  End Sub
```

Wo wir diese Subroutine aufrufen, legen wir später fest.

### *1.3.10.7 Die Delete_Einzelbit Anweisung*

Nach der **Insert-Anweisung** kommt wieder die **Delete-Anweisung**. Auch dafür bereiten wir zwei Anweisungen vor. Eine löscht alle Einzelbitinformationen des Projektes, wenn dieses aus unserer Datenbank entfernt wird. Dafür nutzen wir wie bei den Variablen auch den Abfragefilter auf **Projekt_Id**. Bei der zweiten **Delete_Anweisung** setzen wir den Filter der Abfrage auf **Variable_Id**. Es kann ja sein, das eine als Byte formatierte Variable in ein anderes Format gesetzt wird und wir die Einzelbitbeschreibung nicht mehr benötigen.

Die Vorgehensweise kennen wir bereits. Mit Anweisung hinzufügen den Assistenten öffnen, die Methode **Delete** auswählen, den Filter setzen und einen Namen vergeben. Der Name der ersten Anweisung sollte dann **Delete_Projektbits** und der Name der zweiten und **Delete_VariablenBits** lauten.

**Delete_ProjektBits** bauen wir wieder direkt in die Ereignisroutine **Bt_Del_Projekt_Click** ein

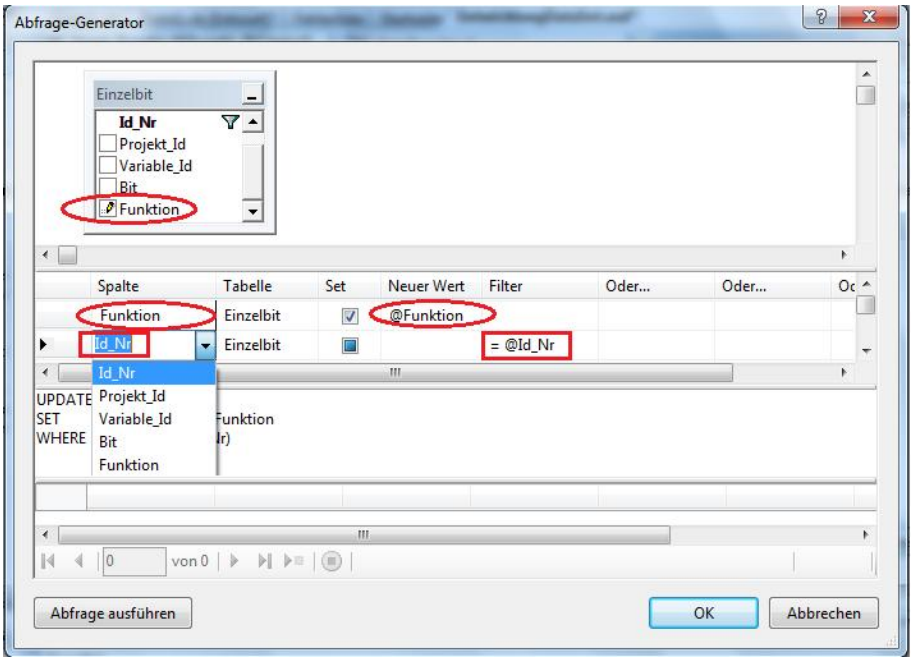
Für die Anweisung **Delete_VariablenBits** schreiben wir eine eigene separate Subroutine.

```
Public Sub Delete_Variablenbit(ByVal Variable_Id As Integer)
    If MsgBox("Sollen die Einzelbitinformationen  entfernt werden?", MsgBoxStyle.YesNo)
                = MsgBoxResult.Yes Then
      EinzelbitTableAdapter.Delete_VariablenBits(Variable_Id)
    End If
 End Sub
```

Die Abfrage ist nicht unbedingt erforderlich, doch wenn man sich nicht sicher ist, ob nicht vielleicht doch die Einzelbitdarstellung in Frage kommt, bleibt die Einstellung erhalten, wenn man die Frage verneint.

### *1.3.10.8 Die Update_Einzelbit-Anweisung*

Das ist nun wieder etwas einfacher, denn ein Update kann immer nur auf einen vorhandenen Datensatz stattfinden und da ist die eigene **Id_Nr** ja bekannt. Also, noch einmal den Assistenten belästigen, die **Update-Anweisung** wählen, die Tabelle im **Abfrage Generator** eintragen, die Spalte **Funktion** anklicken und schließlich mit der **Dropdown-Funktion** die Spalte **Id_Nr** für den Filter auswählen, der natürlich dann auch **=@Id_Nr** heißt. Mehr Spalteninhalte dürfen nicht korrigiert werden. Die Bitnummer ist fest vergeben und die Referenzen **Projekt_Id** und **Variablen_Id** müssen ebenfalls erhalten bleiben. Die Routine nennen wir einfach **Update_Einzelbit**




*Update Einzelbit*

Die SQL-Anweisung wird wieder mit der Klausel **Where (Id_Nr=@Id_Nr)** ergänzt.

Der Aufruf kann direkt in der Ereignisroutine Tb_Kommentar_Leave im Fenster Einstellungen vorgenommen werden. Dort hatten wir ja bereits die Korrektur über die Einzelbittabelle **Dg_Einzelbit** durchgeführt. Nun wird diese Routine aber komplett verändert.

### *1.3.10.9 Ein neues Projekt anlegen*

Mit den Datenbankabfragen und Anweisungen sind wir erst einmal durch und nun ist es an der Zeit, die aus dem Assemblerlisting gewonnenen Daten auch in die Datenbanktabelle zu schreiben. Der Ausgangspunkt ist das Button Bt_Übernahme auf der Filterseite. Die Projektdaten haben wir bereits auf der Datenbank angelegt und so kommen wir zu den Variablenlisten. Der Weg führt über die Subroutine **Filter_Eintragen().** Um nicht lange suchen zu müssen, mit der Maus den Aufruf anklicken und die Subroutine über die Funktion Schnellsuche finden.

Hier noch einmal den Code, den wir bisher erarbeitet haben.

```
Public Sub Filter_Eintragen()
  Dim i As Integer                ' Schleifenzähler
  Dim Anzahl As Integer
  Dim lfd_Nr_Var As Integer    ' Zähler für Tabellenzähler Variablen
  Dim lfd_Nr_Bit As Integer     ' Zähler für Tabellenzeile Einzelbit
  Dim Trenn_Pos As Integer   ' Hilfssvariable für Textbearbeitung
  Dim Ref_Str As String          ' Hilfsvariable für Listeneintrag
  DG_Variablen.Rows.Clear()
  Dg_All_Bits.Rows.Clear()
  lfd_Nr_Var = -1
  lfd_Nr_Bit = 0
  Anzahl = Lb_Variablen.Items.Count    ' Schleifengrenze ermitteln
  For i = 0 To Lb_Variablen.Items.Count – 1
    Ref_Str = Lb_Variablen.Items(i)     ' Eintrag aus Variablenliste holen
    Trenn_Pos = InStr(Ref_Str, ".")      ' Position vom Punkt ermitteln
    If Trenn_Pos = 0 Then                 ' kein Punkt, dann ist das Variablenname
      DG_Variablen.Rows.Add()
      DG_Variablen.Item("CL_Id_Nr", lfd_Nr_Var).Value = lfd_Nr_Var + 1
      DG_Variablen.Item("CL_Projekt", lfd_Nr_Var).Value = 0
      DG_Variablen.Item("CL_Variable", lfd_Nr_Var).Value = Ref_Str
      DG_Variablen.Item("CL_Format", lfd_Nr_Var).Value =
                                  Lb_Formate.Items(i)
      DG_Variablen.Item("CL_Freigabe", lfd_Nr_Var).Value = "Ja"
      DG_Variablen.Item("CL_aktiv", lfd_Nr_Var).Value = "Ja"
      DG_Variablen.Item("CL_Trigger", lfd_Nr_Var).Value = "0"
      lfd_Nr_Var = lfd_Nr_Var + 1
      Store_Variable(Ref_Str, Lb_Formate.Items(i), "Ja", "Ja", -1)
   Else    ' hier ist es eine  Einzelbitinfo
      Dg_All_Bits.Rows.Add()
      Dg_All_Bits.Item("CL_Bit_Id", lfd_Nr_Bit).Value = lfd_Nr_Bit + 1
```

```
        Dg_All_Bits.Item("CL_Proj_Id", lfd_Nr_Bit).Value = 0
        Dg_All_Bits.Item("CL_Var_Id", lfd_Nr_Bit).Value = lfd_Nr_Var
        Dg_All_Bits.Item("CL_BitNr", lfd_Nr_Bit).Value = "Bit " + Mid(Ref_Str, Trenn_Pos
                                                                                    + 5, 1)
        Trenn_Pos = InStr(Ref_Str, ":")
        Ref_Str = Mid(Ref_Str, Trenn_Pos + 1, Len(Ref_Str) - Trenn_Pos)
        Dg_All_Bits.Item("CL_BitFunktion", lfd_Nr_Bit).Value = Ref_Str
        lfd_Nr_Bit = lfd_Nr_Bit + 1
      End If
'************************** neue Anweisungen ******************************
      Load_Variable()
          ' hier Zwischenschritt zur Speicherung aller Einzelbitinfo
      Store_All_Einzelbit()
      Load_Projektbits(Val(Tb_Projekt_Id.Text))
'*****************************************************************************
      TC_Auswahl.SelectTab(1)  ' Hier erfolgt der Wechsel zur nächsten seite
      TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
       Set_Edit_Variable(0)
   Next i
End Sub
```

Zuerst besetzen wir den Zähler Lfd_Nr_Var mit -1. Er dient in dieser Subroutine erst einmal als Referenz auf die Tabelle mit den Variablennamen und diese fängt mit 0 an. Die Variablen werden aber im oberen Teil gezählt. Wird ein Format Byte erkannt, ist der Variablenzähler schon eins weiter. Mit -1 bügeln wir diesen Fehler aus. Eine andere Möglichkeit ist die Zuweisung

```
        Dg_All_Bits.Item("CL_Var_Id", lfd_Nr_Bit).Value = lfd_Nr_Var
```

um -1 zu ergänzen. Dann passt es auch.

Die Zuweisungen an die Tabelle **Dg_Variablen** kommentieren wir alle aus. Später, wenn die neue Funktion getestet ist, können wir diese Zeilen ganz entfernen. Im Moment aber genügt es, diese Zeilen von der Bearbeitung auszuschließen. Dafür rufen wir nun die **Subroutine Store_Variable** mit den erforderlichen Parametern auf. Die **Projekt_Id** brauchen wir nicht mitliefern, die befindet sich in der Textbox **Tb_Projekt_Id**. Allerdings ist das Format auf der Datenbank kein Text, sondern der Index des Eintrags in der Combobox **Cb_Format**. Dennoch übergeben wir hier den Text und ändern in der kleinen Subroutine **Store_Variable** die Zuweisung unter Zuhilfenahme von einer lokalen Variablen entsprechend ab.

### *1.3.4 Variablen in Datenbanktabelle eintragen und zurücklesen*

```
Private Sub Store_Variable(ByVal Variable As String, ByVal Format As String, ByVal
Freigabe As String, ByVal aktiv As String, ByVal Trigger As Byte)
    Dim ProjektId As Integer
    Dim FormatIndex As Byte
    FormatIndex = Cb_Format.Items.IndexOf(Format)
    ProjektId = Val(Tb_Projekt_Id.Text)
    VariablenTableAdapter.Insert_Variable
                    (ProjektId, Variable, FormatIndex, Freigabe, aktiv, Trigger)
  End Sub
```

Die Zuweisung an die Tabelle **Dg_All_Bits** bleibt erst einmal unverändert. Nachdem die Daten vom Filter übertragen sind, ist hinter der Schleife der Aufruf für die Routine **Load_Variable**. In dieser Routine muss nun der Datenbankinhalt an die Tabelle Dg_Variablen übertragen werden. So erhalten wir dann auch die **Id_Nr** der Variablen. Bauen wir uns erst mal wieder das Gerüst für diese Subroutine.

```
Public Sub Load_Variable()
    Dim Projekt_ID As Integer
    Dim Anzahl As Integer
    Dim i As Integer
    DG_Variablen.Rows.Clear()
    Projekt_ID = Val(Tb_Projekt_Id.Text)
    Anzahl = VariablenTableAdapter.Get_Variable(Projekt_ID).Count

 End Sub
```

Klar, es werden schon ein paar Datensätze erwartet, also brauchen wir eine Schleife mit einem Schleifenzähler, um das Ergebnis der Datenbankabfrage in die Tabelle zu übertragen. Und auch die ersten Schritte sind klar: die Tabelle **Dg_Variablen** vom alten Inhalt befreien, die **Projekt_Id** bereitstellen und die Anzahl der zurückgelieferten Einträge ermitteln. So haben wir mit dem Schleifenzähler drei lokale Variablen. Der Rest ist einfach. Ist der Wert in Anzahl > 0 kann die Schleife durchlaufen werden und das Ergebnis der Datenbankabfrage in die Tabelle eingetragen werden.

```
  If Anzahl > 0 Then
```

```
For i = 0 To Anzahl - 1
    DG_Variablen.Rows.Add()
    DG_Variablen.Item("CL_Id_Nr", i).Value =
            Str(VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Id_Nr)
    DG_Variablen.Item("CL_Lfd_Nr", i).Value = i + 1
    DG_Variablen.Item("CL_Projekt", i).Value =
          Str(VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Projekt_Id)
    DG_Variablen.Item("CL_Variable", i).Value =
          VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Variable

    DG_Variablen.Item("CL_Format", i).Value =
          VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Format
    DG_Variablen.Item("CL_Freigabe", i).Value =
          VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Freigabe
    DG_Variablen.Item("CL_aktiv", i).Value =
          VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Aktiv
    DG_Variablen.Item("CL_Trigger", i).Value =
           Str(VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Trigger)
  Next
End If
```

Bevor wir nun den nächsten Schritt gehen, testen wir diesen Programmabschnitt. Die Aufrufe der Subroutinen **Store_All_Einzelbit()** und **Load_Projektbits(Val(Tb_Projekt_Id.Text))** kommentieren wir erst einmal aus.

Beim Testen stoßen wir auf einen ziemlich dummen Fehler. In der Datenbank steht nicht das Format in Textform, sondern als Index auf die Formatliste. Das ist nicht schlimm, aber so erhalten wir in der Tabelle keine ordentliche Anzeige. Um der Tabellenspalte das Format in Textform einzutragen, müssen wir entweder eine ziemlich komplizierte Befehlszeile schreiben oder aber eine Hilfsvariable benutzen. Die Wahl fällt auf die bereits in der Subroutine **Store_Variable** eingesetzte lokale Variable **FormatIndex**.

Mit

```
FormatIndex= VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Format
```

und

```
DG_Variablen.Item("CL_Format", i).Value =Cb_Format.Items(FormatIndex)
```

bleiben die Befehle gut lesbar.

Nun sollten in der Tabelle **Dg_Variable** auf der nächsten Seite die Variablennamen eingetragen sein. Da wir die Zuweisung in der Subroutine **Filter_Eintragen** auskommentiert haben, kommen die Daten diesmal von der Datenbank. Das bedeutet, zu jeder Assemblervariablen gibt es auch eine korrekte **Id_Nr.**

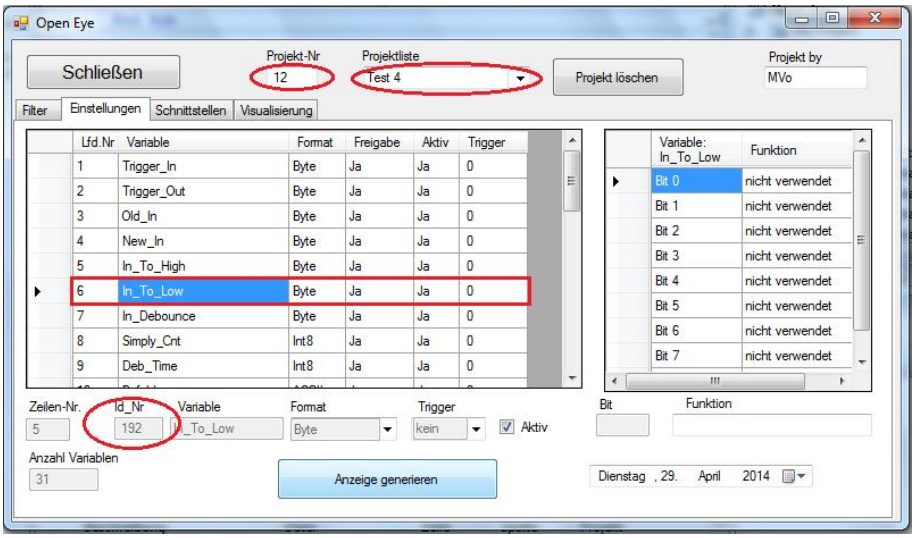
### *1.3.4 Projekt löschen*

Testen wir noch eine weitere kleine Änderung. Wir löschen das Projekt. Damit die Variablen auch entfernt werden, muss die Ereignisroutine im Button **Bt_Del_Projekt** noch etwas erweitert werden.

```
Public Sub Delete_Projekt(ByVal Id_Nr As Integer)
    Dim Antwort As Integer
    Antwort = MsgBox("Soll der Datensatz wirklich gelöscht werden?", MsgBoxStyle.YesNo,
"Warnung")
    If Antwort = vbYes Then
      ' Nachtragen, Variablen zum aktuellen Projekt löschen
      VariablenTableAdapter.Delete_Variable(Val(Tb_Projekt_Id.Text))
      ' ****************************************************
      ProjektTableAdapter.Delete_Projekt(Id_Nr)
      Load_Projekte("")
      If ProjekteTableAdapter.Get_Projekte.Rows.Count > 0 Then
        Tb_Projekte.Text = ProjekteTableAdapter.Get_Projekte.Item(0).Projekt
        TB_Projekt_Alt.Text = Tb_Projekte.Text
        Tb_Projekt_Id.Text = Str(ProjekteTableAdapter.Get_Projekte.Item(0).Id_Nr)
        Load_Variable()    ' Nachtragen, Variablen zum Projekt laden
      Else
        Tb_Projekte.Text = ""
        TB_Projekt_Alt.Text = Tb_Projekte.Text
        Tb_Projekt_Id.Text = ""
      End If
    End If
  End Sub
```

Das wird nun geprüft, ob es so wie gewünscht, funktioniert.

Nun, so wirklich ist noch kein Unterschied erkennbar. Wirklich nicht? Schaut einfach mal genau hin




*Ein Projekt von der Datenbank holen*

Die **Id_Nr** vom Projekt ist bereits bei 12 und bei den Variablen sogar schon bei 192. Diese Werte sind eindeutig von der Datenbank. Um nun auf andere Projekte zu wechseln, müssen wir das Ereignis der Combobox **Cb_Projekte** noch einmal bemühen. Wir wissen, dass mit **Load_Variablen** die Tabelle auf Seite zwei gefüllt wird. Also tragen wir diesen Aufruf einfach noch in die Ereignisroutine **SelectedIndexChanged**

```
Private Sub Cb_Projekte_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Projekte.SelectedIndexChanged
    Dim Projekt_Pos As Integer                    ' für Position in der Liste
    Tb_Projekte.Text = Cb_Projekte.Text
    TB_Projekt_Alt.Text = Tb_Projekte.Text
    Projekt_Pos = Cb_Projekte.Items.IndexOf(Tb_Projekte.Text)   ' Position ermitteln
    Tb_Projekt_Id.Text  =  LB_Id_Nr.Items(Projekt_Pos)                ' und  zugehörige
Projektnummer eintragen
    Load_Projekt(Val(Tb_Projekt_Id.Text))              ' Einstellparameter Projekt
    Load_Variable()
  End Sub
```

Wenn wir nun einen Blick auf die **Id_Nr** der Variablen werfen, erkennen wir, das es völlig andere Datensätze sind, die bei den verschiedenen Projektnamen geladen werden.

Dieser erfolgreiche Test gibt uns nun einen neuen kraftvollen Schub, mit der Programmentwicklung fortzufahren und auch die Einzelbitbeschreibung in die Datenbanktabelle einzutragen.

### *1.3.11 Einzelbitinformation in Datenbanktabelle übertragen*

Hierbei gehen wir etwas anders vor. Dabei spielen die Einträge in der Tabelle **Dg_Variablen** eine Rolle. Drei Spalten interessieren besonders: **Projekt_Id**, die ist aber auch neben dem Projektnamen im Textfeld untergebracht. **Variablen_Id**, dieser Wert ist der Tabellenzeile zugeordnet und von dort auch zu erfahren sowie das Format, denn nur Variablen mit dem Format **Byte** haben zu diesem Zeitpunkt auch eine Einzelbitinformation. Anders wie bei den Variablen werden hier aber immer acht Datensätze geschrieben.

Die Routine, die wir nun schreiben wollen und deren Aufgabe es ist, alle Einzelbitinformationen abzuspeichern nenne ich **Store_All_EinzelBit**. Hier wird die Tabelle **Dg_Variable** durchlaufen und geprüft, ob ein Format **Byte** enthalten ist. Wenn ja, wird zu dieser Variablen die Einzelbitbeschreibung aus der Tabelle **Dg_All_Bit** herausgesucht und an die Datenbanktabelle geschickt. Aufgrund dieser Aufgabenbeschreibung erkennen wir, dass es zwei Schleifen benötigt. Eine äußere, die die Variablentabelle durchforstet und eine weitere innenliegende, die die Einträge aus **Dg_All_Bit** holt und für die Datenbank zusammenstellt.

```
Public Sub Store_All_Einzelbit()
    Dim I As Integer                'Schleifenvariable für Variablentabelle
    Dim j As Integer                'Schleifenvariable für Einzelbiteinträge
    Dim Anz_Variable As Integer     ' Anzahl Einträge in der Variablentabelle
    Dim Variable_Id As Integer      ' Variablen_Id in der Variablentabelle
    Dim Projekt_Id As Integer       ' Variable zur Konvertierung der Projekt_Id
    Dim Bit_Nr As Byte              ' Bitnummer
    Dim Bitname As String           ' Text der Spalte Bit
    Dim Info As String
    Dim Varname As String
    Dim Ref_Wert As Integer         ' Zeile Variablentabelle in der Tabelle Dg_All_Bits
                                      entspricht dem Schleifenzähler i
    Anz_Variable = DG_Variablen.Rows.Count
    If Anz_Variable > 0 Then
      Projekt_Id = Val(Tb_Projekt_Id.Text)
      For I = 0 To Anz_Variable - 1
        Varname = DG_Variablen.Item("CL_Variable", I).Value
        If DG_Variablen.Item("CL_Format", I).Value = "Byte" Then
          Variable_Id = Val(DG_Variablen.Item("CL_Id_Nr", I).Value)
          For j = 0 To Dg_All_Bits.Rows.Count - 1
            Ref_Wert = Val(Dg_All_Bits.Item("Cl_Var_Id", j).Value)
            If Ref_Wert = I Then
```

```
                    Bitname = Dg_All_Bits.Item("CL_BitNr", j).Value
                    Bit_Nr = Val(Mid(Bitname, 5, 1))
                    Info = Dg_All_Bits.Item("Cl_BitFunktion", j).Value
                    If Info = "" Then Info = "nicht verwendet"
                    Trim(Info)
                    EinzelbitTableAdapter.Insert_Einzelbit(Projekt_Id, Variable_Id, Bit_Nr, Info)
                End If
            Next j
        End If
    Next I
  End If
End Sub
```

### *1.3.4 Einzelbitinformationen aus Datenbank lesen*

Nach dieser Aktion brauchen wir die Tabelle **Dg_All_Bit** für die Datenbankdaten. Deshalb werden die Einzelbits dieses Projektes auch wieder zurückgelesen. Zu diesem Zweck haben wir die Datenbankabfrage **Get_Projektbits** erstellt. In der Subroutine, die diese Abfrage aufruft, werden wir auch mit mehreren Datensätzen rechnen und so ist auch hier mit einer Schleife der Eintrag in die Tabelle **Dg_All_Bits** erforderlich. Für die Variablendeklaration ist nun nur noch eine Variable für die Anzahl der gelesenen Einträge sinnvoll.

Zuerst leeren wir die Tabelle **Dg_All_Bits**, weisen dann der Variablen **Anzahl** den Wert **Count** der Abfrage zu und wenn der Wert >0 ist, beginnen wir mit dem Kopieren der Daten. Auch hier wieder beachten: **Count** liefert die **Anzahl**, die Schleife läuft von 0 an bis **Anzahl** -1.

```
Public Sub Load_Projektbits(ByVal Projekt_Id As Integer)
   Dim Anzahl As Integer
   Dim i As Integer
   Dg_All_Bits.Rows.Clear()
   Anzahl = EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Count
   If Anzahl > 0 Then
     For i = 0 To Anzahl - 1
        Dg_All_Bits.Rows.Add()
        Dg_All_Bits.Item("Cl_Bit_Id",                    i).Value                    =
Str(EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Id_Nr)
        Dg_All_Bits.Item("Cl_Proj_Id",                    i).Value                    =
Str(EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Projekt_Id)
        Dg_All_Bits.Item("Cl_Var_Id",                    i).Value                    =
Str(EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Variable_Id)
        Dg_All_Bits.Item("Cl_BitNr",          i).Value          =          "Bit          "          +
Str(EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Bit)
        Dg_All_Bits.Item("Cl_BitFunktion",                    i).Value                    =
EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Funktion
      Next
    End If
 End Sub
```

Eine weitere Subroutine kann nach diesen Schritten auch die Einzelbits zu einer Variablen in die Tabelle **Dg_Einzelbit** übertragen. Dazu müssen wir die Routine **Show_Einzelbit** umschreiben. Die Information wird nun nicht mehr aus der Tabelle **DG_All_Bits** herausgesucht, sondern wir

überlassen diese Aufgabe der Datenbank und erwarten einfach nur die Einträge zur Variablen. Die Vorbesetzung der Tabelle belassen wir. Im zwiten Schritt besetzen wir die Variable **Anzahl** mit der Ergebnismenge der Datenbankabfrage. Wenn Ergebnisse vorliegen, werden die Daten aus der Datenbank übernommen und die Tabelle **Dg_Einzelbit** mit den gefundenen Einträgen überschrieben. Die Frage, warum vorbesetzen, kann ich gleich beantworten.

Die Subroutine wird aufgerufen, wenn im Formatfeld einer Variablen das Wort **Byte** erscheint. So auch, wenn eine Variable korrigiert und das Format abgeändert wird. Das ist ja nachträglich möglich, denn wir können nicht davon ausgehen, das ein Assemblerlisting uns die Formate liefert. Unsere Subroutine **Store_All_Einzelbit** speichert aber nur Daten, die durch den Filter in der **Dg_All_Bits** vorbereitet waren. Somit ist für diese Variable nach dem Wechsel des Formats auf **Byte** gar kein Datensatz verfügbar. Wenn wir nun diese Einzelbitinformation editieren wollen, gibt es aufgrund fehlender Datenbankeinträge auch keine **Id_Nr**, die erforderlich ist, um die Korrektur auch dem richtigen Datensatz zuzuweisen. Werfen wir mal einen Blick auf die Zuweisung in der Vorbesetzung auf die Spalte **Cl_BitID_Nr**. Dort steht die **Id_Nr** aus der Datenbank. Ist dort eine -1, kann der Datensatz nicht von der Datenbank stammen und muss dort erst angelegt werden. So können wir durch die Vorbesetzung erkennen, ob diese Information schon vorhanden und ein **Update** oder ob sie neu angelegt und ein **Insert** ausgeführt werden muss.

```
Public Sub Show_Einzelbit(ByVal Var_Id As Integer)
  Dim i As Integer
  Dim Zeilen_Nr As Integer
  Dim Anzahl As Integer
  Dg_Einzelbit.Columns.Item(2).HeaderText = "Variable: " + TB_Variable.Text
  Dg_Einzelbit.Rows.Clear()                          ' Tabelle löschen
  For i = 0 To 7                                      ' Tabelle vorbesetzen
     Dg_Einzelbit.Rows.Add()
     Dg_Einzelbit.Item("CL_BitId_Nr", i).Value = -1
     Dg_Einzelbit.Item("CL_Variable_Id", i).Value = Str(Var_Id)
     Dg_Einzelbit.Item("Cl_Bit", i).Value = "Bit" + Str(i)
     Dg_Einzelbit.Item("CL_Funktion", i).Value = "nicht verwendet"
   Next
   Anzahl = EinzelbitTableAdapter.Get_VariablenBit(Val(Tb_Var_ID.Text)).Count
   If Anzahl > 0 Then
     For i = 0 To Anzahl - 1 ' Tabelle mit allen Einzelbits durchsuchen
       ZeilenNr = EinzelbitTableAdapter.Get_VariablenBit(Val(Tb_Var_ID.Text)).Item(i).Bit
         Dg_Einzelbit.Item("CL_BitId_Nr", ZeilenNr).Value
        = Str(EinzelbitTableAdapter.Get_VariablenBit(Val(Tb_Var_ID.Text)).Item(i).Id_Nr)
         Dg_Einzelbit.Item("CL_Funktion", ZeilenNr).Value
```

```
        = EinzelbitTableAdapter.Get_VariablenBit(Val(Tb_Var_ID.Text)).Item(i).Funktion
    Next
  End If
  Set_Edit_Bit(0)      ' Inhalt erster Zeile in den Editierbereich
End Sub
```

Die Zuweisungen an die Tabelle werden nur auf die Spaltenfelder **Cl_BitId_Nr** und **Cl_Funktion** unter Verwendung der Bitnummer, die wir vorher der Variablen **ZeilenNr** zugewiesen haben, durchgeführt.

An dieser Stelle könnte man auch darüber nachdenken, ob es erforderlich ist, immer das gesamte Byte auf die Datenbank zu legen, wenn nur 3 Bits kommentiert sind. Es würde auch ausreichen, nur die Bits abzulegen, deren Informationsfeld nicht den Text **nicht verwendet** enthalten. Für den Aufbau unserer Subroutinen ist das eine nur kleine Änderung in der Abspeicherung nach dem Filter. In der Subroutine **Store_All_EinzelBits** muss lediglich der Abschnitt

```
            If Info = "" Then Info = "nicht verwendet"
            EinzelbitTableAdapter.Insert_Einzelbit(Projekt_Id, Variable_Id, Bit_Nr, Info)
```

in

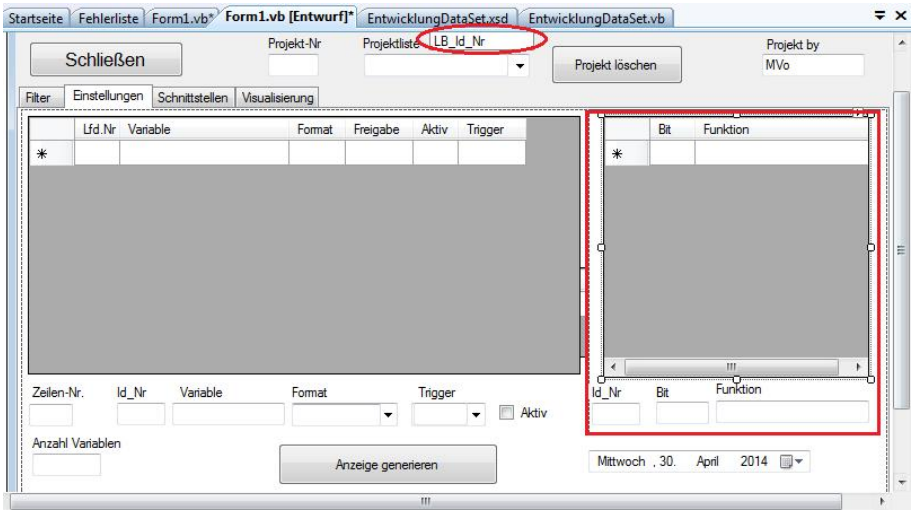
```
.           If Info = "" Then Info = "nicht verwendet"
           Trim(Info)
            If Info <> "nicht verwendet" then
              EinzelbitTableAdapter.Insert_Einzelbit(Projekt_Id, Variable_Id, Bit_Nr, Info)
            End if
```

geändert werden. Auch wenn wir später einzelne Bits nachtragen wollen, findet die Auswertung über -1 in der **Id_Nr** des Dateneintrags zur Auswahl des Aufrufs **Insert** oder **Update** statt.

Vielleicht ist es ja auch sinnvoll, nur benutzte Bits aufzunehmen, also tragen wir diese Korrektur ein und vergessen, das alle acht Bits immer abgespeichert werden sollen.

Nun sind wir wieder für die nächsten Tests bereit. Damit wir hier auch einen Erfolg sehen können, spendieren wir der zweiten Seite noch eine Textbox mit der **Id_Nr** der Einzelbitbeschreibung. Die Textbox **TB_Bit_Id** setzen wir neben die Textbox **Tb_Bit**.

Der Screenshot zeigt nun den Aufbau der zweiten Seite mit den eingebrachten Korrekturen



*Blick auf Seite 2*

Das rot umrandete Feld wird nur mit einem Format **Byte** visualisiert. Die rot eingekreiste Listbox **Lb_Id_Nr** enthält die Schlüssel zu den Projektnamen in der Combobox **Cb_Projekte.** Im Programm ist sie nicht sichtbar durch die Eigenschaft **Visible =False**.

Damit unser Programm nun auch die Informationen wie gewohnt bei einem Wechsel der Variablen oder auch bei einem Wechsel des Projekts liefert, müssen wir die entsprechenden Subroutinen anpassen. Beginnen wir mit der Projektauswahl.

Das haben wir bereits erstellt

```
Private Sub Cb_Projekte_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Projekte.SelectedIndexChanged
  Dim Projekt_Pos As Integer                          ' für Position in der Liste
  Tb_Projekte.Text = Cb_Projekte.Text
  Projekt_Pos = Cb_Projekte.Items.IndexOf(Tb_Projekte.Text)   ' Position ermitteln
  Tb_Projekt_Id.Text = LB_Id_Nr.Items(Projekt_Pos)          ' und Schlüssel eintragen
```

```
  Load_Projekt(Val(Tb_Projekt_Id.Text))
End Sub
```

Nun wollen wir auch die Variablen und die Einzelbits zum Projekt laden. Also müssen wir die Routinen **Load_Variable** und **Load_ProjektBits** einfügen. Mit der Anzahl der gefundenen Datensätze für die Variablenliste haben wir auch gleich den Wert für die Textbox **TB_Anzahl_Var**. Vielleicht brauchen wir sie im Assemblerprojekt des Controllers. Anschließend soll auch gleich der erste Datensatz in den Editbereich. Dafür hatten wir die Routine **Set_Edit_Variable** geschrieben. Hier brauchen wir sie nur aufrufen. Finden wir eine Variable mit dem Format **Byte** vor, darf auch gleich die Anzeige der Einzelbits erscheinen und wir rufen die entsprechende Subroutine **Show_Einzelbits** unter Angabe vom Schlüssel der Variable auf.

```
Private Sub Cb_Projekte_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Projekte.SelectedIndexChanged
    Dim Projekt_Pos As Integer                                  ' für Position in der Liste
   Tb_Projekte.Text = Cb_Projekte.Text
   TB_Projekt_Alt.Text = Tb_Projekte.Text
   Projekt_Pos = Cb_Projekte.Items.IndexOf(Tb_Projekte.Text)   ' Position ermitteln
   Tb_Projekt_Id.Text = LB_Id_Nr.Items(Projekt_Pos)            ' und Projekt-Nr eintragen
    Load_Projekt(Val(Tb_Projekt_Id.Text))                      ' Einstelldaten Projekt
    Load_Variable()                                            ' Variablen Projekt
    Load_Projektbits(Val(Tb_Projekt_Id.Text))                  ' Einzelbits Projekt
   TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
   Set_Edit_Variable(0)
   If Tb_Format.Text = "Byte" Then
      Show_Einzelbit(Val(Tb_Var_ID.Text))                      ' Einzelbits anzeigen
   End If
 End Sub
```

### *1.3.12 Datenbankinhalte editieren*

An dieser Stelle möchte ich auch gleich mit den Korrekturaufgaben für Projektdaten, Variablen und Einzelbitinformationen fortsetzen.

Korrekturen können sich auch auf einen ganzen Bereich von Datenbankeinträgen auswirken. Man stelle sich einmal einen Warenbestand vor und in einer Tabellenspalte befinden sich beispielsweise Steuersätze verschiedener Art, die auf den Preis einer Ware berechnet werden. Nun wird eine Warensteuer einer bestimmten Art geändert. Mit einer Datenbank kein Problem. Es wird eine Datenbankabfrage entsprechend mit einer **Update**-Anweisung auf die betreffende Warenart gesetzt und der geänderte Steuersatz übergeben. In einem Rutsch sind alle Daten aktualisiert. Nun, solche Korrekturen haben wir nicht. Bei uns wird immer nur ein einzelner Datensatz verändert. Aber es kann auch sein, das eine Formatänderung einer Variablen bis zu vier Korrekturen ausführen muss. Diese werden dann einzeln nacheinander durchgeführt. Beginnen wir aber erst einmal mit den Korrekturmöglichkeiten für das Projekt.

### *1.3.12.1 Eine Datenstruktur für Projektdaten definieren*

Für die Projektdaten sind noch nicht alle erforderlichen Objekte in unserem Programm eingebaut, da diese erst auf der dritten Seite zur Sprache kommen. Der Vorgang weicht aber nicht besonders ab, so dass diese Aufgabe auch erst später vervollständigt werden kann. Der Aufbau der Routine **Update_Projekt** wird dadurch nicht verändert.

Was also darf verändert werden? Nun, alles, bis auf die **Id_Nr**. Die **Update**-Anweisung auf der Datenbank ist schon entsprechend eingerichtet. Aus diesem Grund übergeben wir der Subroutine einfach einen kompletten Datensatz mit der **Id_Nr**. Allerdings, in der uns bekannten Art der einzelnen Übergabeparameter die Subroutine zu verfassen würde einen ziemlich unübersichtlichen Aufbau ergeben. Aber es gibt eine Möglichkeit, einen Datensatz in einem Rutsch zu übertragen. Das Zauberwort heißt **Structure** und lässt einen globalen **Datentypen** entwerfen, der alle erforderlichen Variablentypen für unser Projekt enthält. In unserer Anwendung wird dann einfach eine Variable vom **Typ** dieser **Struktur** deklariert und die Werte dann den einzelnen Elementen zugewiesen. Definieren wir erst einmal die Struktur

```
Public Structure ProjektRecord

End Structure
```

Diese Zeilen kommen direkt hinter **Public Class Frm_Open_Eye**. In diese Zeilen kopieren wir nun die deklarierten Variablen aus der bereits erstellten Subroutine **Insert_Projekt**.

```
Public Structure ProjektRecord
    Dim Id_Nr As Integer        ' Schlüssel Projekt
    Dim Projekt As String       ' Projektname
    Dim Controller As String    ' Controller
    Dim Takt As String          ' CPU Takt
    Dim Baud As Integer         ' Baudrate USART
    Dim Daten As Integer        ' Anzahl Datenbits
    Dim Stopp As Integer        ' Anzahl Stoppbits
    Dim Parity As Integer       ' Parity
    Dim Kopf As String          ' Telegrammkennung Kopfdaten
    Dim Auftrag As String       ' Befehl an Controller
    Dim WPuffer As Integer      ' Schreibpuffer Achtung, nur grade Werte
    Dim RPuffer As Integer      ' Lesepuffer Achtung nur grade Werte
```

```
    Dim Sprache As String      ' Sprachauswahl option
    Dim Autor As String        ' Projektleiter
    Dim Datum As String        ' Datum / Zeit vom System
    Dim Kommentar As String ' ergänzender Kommentar
End Structure
```

Die Id_Nr und der Projektname wird entsprechend ergänzt. Nun haben wir die Möglichkeit eine Variable vom Typ ProjektRecord zu deklarieren und haben in einer einzigen Variablen alle verschiedenen Werte für die Zuweisung an die Datenbank.

```
Dim Datensatz as ProjektRecord
```

Natürlich lässt sich diese Struktur auch als Parameter einer Subroutine übergeben

```
Public Sub Update_projekt(ByVal Projektdaten As ProjektRecord)
```

Das ist doch eine übersichtliche Sache. Die Subroutine **Insert_Projekt** und **Update_Projekt** passen wir nun den neuen Gegebenheiten an. Für die Datenübergabe an **Insert** und **Update** setzen wir einen Zwischenschritt ein, der die Werte in unserem Datensatz besetzt. Wenn wir dann unsere dritte Seite fertig haben, brauchen wir nur an einer einzigen Stelle die Werte zuweisen. Die Routine nennen wir **Set_Projektdaten** und sie ist eine Kopie der bestehenden Subroutine **Update_Projekt**.

```
Public Sub Set_ProjektDaten(ByVal Projekt_Id As Integer, ByVal Projekt As String)
    Dim Projektdaten As ProjektRecord
    Projektdaten.Id_Nr = Projekt_Id
    Projektdaten.Projekt = Projekt
    Projektdaten.Controller = "Atmega"
    Projektdaten.Takt = "16 MHz"
    Projektdaten.Baud = 1
    Projektdaten.Daten = 2
    Projektdaten.Stopp = 1
    Projektdaten.Parity = 1
    Projektdaten.Kopf = "VALUE"
    Projektdaten.Auftrag = "AB"
    Projektdaten.RPuffer = Val("100")
    Projektdaten.WPuffer = Val("100")
    Projektdaten.Sprache = "Assembler"
```

```
    Projektdaten.Autor = "ich"
    Projektdaten.Datum = Date.Today
    Projektdaten.Kommentar = "-"
    If Projekt_Id < 0 Then
      Insert_Projekt(Projektdaten)
    Else
      Update_projekt(Projektdaten)
    End If
  End Sub
```

Überall, wo die Subroutinen **Update_Projekt** oder **Insert_Projekt** aufgerufen wurden tragen wir nun den Aufruf **Set_Projektdaten** ein mit der **Id_Nr** und dem Projektnamen als Übergabeparameter. Die **Insert**-Aufrufe bekommen als **Id_Nr** eine **-1** mitgeteilt. Daraus kann in der Setzroutine dann entschieden werden, ob **Insert** oder **Update**.

Diese beiden Routinen schrumpfen nun. Es ist nicht erforderlich, die Variablen noch zu deklarieren. Die Werte der Variablen **Projektdaten** können direkt zugewiesen werden.

```
Public Sub Insert_Projekt(ByVal Projektdaten As ProjektRecord)
    Dim Id_Pos As Integer
    ProjektTableAdapter.Insert_Projekt(Projektdaten.Projekt,
                                       Projektdaten.Controller, Projektdaten.Takt,
                                       Projektdaten.Baud, Projektdaten.Daten,
                                       Projektdaten.Stopp, Projektdaten.Parity,
                                       Projektdaten.Kopf, Projektdaten.Auftrag,
                                       Projektdaten.Sprache, Projektdaten.WPuffer,
                                       Projektdaten.RPuffer, Projektdaten.Autor,
                                        Projektdaten.Datum, Projektdaten.Kommentar)
    Load_Projekte(Tb_Projekte.Text)          ' Nach Einfügen Projektliste neu laden

  End Sub
```

Bei der Insert-Anweisung wird die **Id_Nr** erst nach dem Laden aller Projekte gültig. Deshalb muss sie nach dem erneuten Laden des Projektes von der Datenbank in der Liste gefunden und in die Textbox eingetragen werden. Die Funktion **IndexOf** von **Combobox** und **Listbox** haben wir ja bereits ausführlich gesprochen.

Die **Update**-Routine besteht nun nur noch aus einer einzigen Anweisung.

```
Public Sub Update_Projekt(ByVal Projektdaten As ProjektRecord)
        ProjektTableAdapter.Update_Projekt(Projektdaten.Projekt, Projektdaten.Controller,
                                Projektdaten.Takt,Projektdaten.Baud, Projektdaten.Daten,
                                Projektdaten.Stopp, Projektdaten.Parity, Projektdaten.Kopf,
                                Projektdaten.Auftrag, Projektdaten.Sprache,
                                Projektdaten.WPuffer, Projektdaten.RPuffer,
                                Projektdaten.Autor,  Projektdaten.Datum,
                                Projektdaten.Kommentar, Projektdaten.Id_Nr)
  End Sub
```

Testen wir nun das Ergebnis und legen ein Projekt an. Eine Korrektur des Projektnamens ist der erste Schritt.

Beim Testen stellen wir fest, dass die Projektdaten immer geladen werden, auch wenn der Projektname nicht gewechselt wird. Das liegt daran, das das Ereignis der Combobox **SelectedIndexChange** aufgerufen wird. Wenn wir die beiden Inhalte der Textbox **Tb_Projekte.Text** und der Combobox **Cb_Projekte.**Text vergleichen, können wir das unnötige Laden des Projektes verhindern.

```
Private Sub Cb_Projekte_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Projekte.SelectedIndexChanged
  Dim Projekt_Pos As Integer                          ' für Position in der Liste
  If Tb_Projekte.Text <> Cb_Projekte.Text Then    ' nur neu laden, wenn unterschiedlich
    Tb_Projekte.Text = Cb_Projekte.Text
    TB_Projekt_Alt.Text = Tb_Projekte.Text
    Projekt_Pos = Cb_Projekte.Items.IndexOf(Tb_Projekte.Text)   ' Position ermitteln
    Tb_Projekt_Id.Text = LB_Id_Nr.Items(Projekt_Pos)        ' und Projek -Nr eintragen
     Load_Projekt(Val(Tb_Projekt_Id.Text))                  ' Einstellparameter Projekt
     Load_Variable()
     Load_Projektbits(Val(Tb_Projekt_Id.Text))
     TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
     Set_Edit_Variable(0)
     If Tb_Format.Text = "Byte" Then
        Show_Einzelbit(Val(Tb_Var_ID.Text))
     End If
  End If
End Sub
```

Im Moment gibt es für den Aufruf der Subroutine Update_Projekt  nur das Ereignis Tb_Projekte_Leave. Also muss, wenn id der Textbox Tb_Projekte ein anderer Name eingetragen wird, die Textbox auch verlassen werden.

Das ist aber ok, denn jede andere Aktion überschreibt sofort den aktuellen Datensatz. Auf eine Abfrage habe ich verzichtet. Warum auch, wenn ich den Projektnamen ändere ist das doch Absicht, kein Versehen. Und wenn es halt falsch war, dann schreiben wir halt den alten Namen wieder hin.

### *1.3.12.2 Editieren von Variablenparametern*

Die Editierfunktion von Variablen ist ähnlich einfach. Eine Änderung von Format wird über eine Combobox, genau so wie die Zuweisung einer Triggernummer durchgeführt. Also diese beiden Ereignisse sind gut geeignet, die Korrektur auch gleich durchzuführen. Das haben wir ja schon gemacht, mit allen Dingen, die dabei beachtet werden müssen. So brauchen wir nur die Stellen suchen, wo in die Tabelle **Dg_Variablen** die Änderungen eingebracht werden und wir haben die Stelle, um die **Update**_Anweisung an die Tabelle unterzubringen. Einen Parameter brauchen wir der Routine nicht zu übergeben. Die **Id_Nr** der Variablen steht im Textfeld, die anderen Daten sind aus den Editierobjekten abzuleiten.
Doch halt, das stimmt so nicht. Wenn ein Format geändert wird, dann werden auch andere Tabellenzeilen beeinflusst. Die Korrektur auf der Tabelle **Dg_Variablen** berücksichtigt das und ändert entsprechend Folgezeilen ab. Das können wir uns zu nutze machen und die Information zur Korrektur direkt aus dieser Tabelle abfragen. Nur dort finden wir die **Id_Nr**, um den richtigen Datensatz zu verändern.
Eine Abfrage, ob eine Korrektur erfolgen soll, braucht es nicht, denn es ist komfortabler, wenn die Datenbank gleich aktualisiert wird.
Somit ist die Subroutine **Update_Variable** schon fast komplett beschrieben.

```
Public Sub Update_Variable(ByVal Zeile As Integer)
    Dim Var_Id As Integer
    Dim ProjektId As Integer
    Dim Format As Byte
    Dim Freigabe As String
    Dim Aktiv As String
    Dim Trigger As Byte
    Dim Ref_Text As String
    Var_Id = Val(DG_Variablen.Item("CL_Id_Nr", Zeile).Value)
    ProjektId = Val(Tb_Projekt_Id.Text)
    Ref_Text = DG_Variablen.Item("CL_Format", Zeile).Value
    Format = Cb_Format.Items.IndexOf(Ref_Text)
    Freigabe = DG_Variablen.Item("CL_Freigabe", Zeile).Value
    Aktiv = DG_Variablen.Item("CL_Aktiv", Zeile).Value
    Trigger = Val(DG_Variablen.Item("CL_Trigger", Zeile).Value)
    VariablenTableAdapter.Update_Variable(Format, Freigabe, Aktiv, Trigger, Var_Id)
  End Sub
```

Die Routine wird einfach mit der Zeilennummer der Tabelle **Dg_Variablen** aufgerufen und nimmt dort die Information. Um die Indexwerte **Format**

und **Trigger** zu bekommen, habe ich eine zusätzliche Variable **Ref_Text** deklariert, die diese Aufgabe übersichtlich macht.

Fügen wir nun diese Subroutine überall dort ein, wo wir Korrekturen auf der Tabelle **Dg_Variable** eingetragen haben.

Der erste änderbare Wert ist **Format**. Das Ereignis der Combobox **cb_Format_selectedIndexChange** führt uns zu den Subroutinen **Chk_Format8**, **Chk_Format16** und **Chk_Format32**.

Dort wird nun unter Angabe der Zeilennummer die **Update** Routine eingebunden. Beginnen wir mit **Chk_Format8**

```
Public Sub Chk_Format8(ByVal Old_Format As String, ByVal New_Format As String, ByVal
Zeile_Nr As Integer)
   Dim Bereich As Integer
   Dim i As Integer
   Bereich = 1
   If (Old_Format = "Int16") Or (Old_Format = "Hex16") Then
      Bereich = 2
  End If
  If Old_Format = "Hex32" Then
     Bereich = 4
  End If
  For i = 0 To Bereich - 1
     DG_Variablen.Item("CL_Freigabe", Zeile_Nr + i).Value = "Ja"
     If i = 0 Then
       DG_Variablen.Item("CL_Format", Zeile_Nr + i).Value = New_Format
     Else
       DG_Variablen.Item("CL_Format", Zeile_Nr + i).Value = Cb_DefaultFormat.Text
     End If
     Update_Variable(Zeile_Nr + i)
   Next
 End Sub
```

Mithilfe einer Schleife wird jede betroffene Zeile geändert. Wir brauchen nur nach der Korrektur auf der Tabelle **Dg_Variablen** die **Update_Variable** -Routine aufrufen.

Die Subroutine **Chk_Format16** wird nun genau so mit dem Aufruf der Subroutine **Update_Variable** erweitert.

```
Public Sub Chk_Format16(ByVal Old_Format As String, ByVal New_Format As String, ByVal
Zeile_Nr As Integer)
    If Old_Format = "Hex32" Then
        DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 2).Value = "Ja"     ' Zeile freigeben
        Update_Variable(Zeile_Nr + 2)                                   ' für jede Zeile ein Update
        DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 3).Value = "Ja"     ' Zeile freigeben
        Update_Variable(Zeile_Nr + 3)                                   ' für jede Zeile ein Update
    End If
    DG_Variablen.Item("CL_Format", Zeile_Nr).Value = New_Format
    Update_Variable(Zeile_Nr)                                           ' für jede Zeile ein Update
    DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 1).Value = "Nein"       ' Zeile sperren
    Update_Variable(Zeile_Nr + 1)                                       ' für jede Zeile ein Update
End Sub
```

Hier brauchen wir keine Schleife. Direkt hinter die Zeile mit der Zuweisung an die Tabelle erfolgt der **UpDate_Variable**-Aufruf.

Bleibt noch die Subroutine **Chk_Format32**. Diese beschreibt auf jeden Fall vier Tabellenzeilen. Deshalb setze ich hier wieder eine Schleife ein.

```
Public Sub Chk_Format32(ByVal New_Format As String, ByVal Zeile_Nr As Integer)
    Dim i As Integer                    ' Schleifenzähler
    DG_Variablen.Item("CL_Format", Zeile_Nr).Value = New_Format
    Update_Variable(Zeile_Nr)                                           ' für jede Zeile ein Update
    For i = 1 To 3
        DG_Variablen.Item("CL_Freigabe", Zeile_Nr + i).Value = "Nein"   ' Zeile sperren
        Update_Variable(Zeile_Nr + i)                                   ' für jede Zeile ein Update
    Next
End Sub
```

Bevor nun noch die Einzelbits zur Korrektur gebracht werden, testen wir die Programmfunktion. Korrekturen müssen nun auch nach Neustart des Programms oder Projektwechsel erhalten sein. Dies lässt sich leicht testen, wenn mehrere Projekte angelegt und die Daten aus der gleichen Quelle unterschiedlich formatiert werden.

### *1.3.12.3 Korrekturen der Einzelbitbeschreibung*

Bleibt noch die Korrektur der Einzelbitbeschreibungen. Hier ist wichtig auf die Id_Nr vom Eintrag Einzelbit zu sehen. Wir haben ja nicht zu jedem Bit einer Variablen eine Funktionsbeschreibung abgelegt und neue Variablen mit dem Format Byte können jederzeit angelegt werden. Da ist ja auch klar, dass eine Korrektur entweder ein Update oder ein Insert aufrufen muss. Die Auswertung von -1 haben wir bereits angesprochen.

Welches Ereignis ist nun für diese Korrekturaufgabe geeignet. Nun, es ist ja nur ein einziger Parameter dieses Datensatzes, der für eine Änderung in Frage kommt und da darf es dann auch Leave sein, da weitere Bearbeitung auf jeden Fall die Textbox Tb_Funktion verlässt. Diese Ereignisroutine haben wir bereits erstellt.

```
Private Sub TB_Funktion_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TB_Funktion.Leave
  Dim Zeile Nr As Integer
  Dim Id Nr As Integer
  Zeile_Nr = Val(Mid(TB_Bit.Text, 5, 1))
  Id_Nr = Val(Dg_Einzelbit.Item("CL_BitId_Nr", Zeile_Nr).Value)
  Dg Einzelbit.Item("CL Funktion", Zeile Nr).Value
                              = TB Funktion.Text
  Id Nr = Dg Einzelbit.Item("CL BitId Nr", Zeile Nr).Value
  If Id_Nr < 0 Then
     Zeile_Nr = Add_New_Rows(Val(Mid(TB_Bit.Text, 5, 1)))
  End If
  Dg All Bits.Item("CL BitFunktion", Zeile Nr).Value
          = TB_Funktion.Text
End Sub
```

Nun werden wir sie etwas anders aufbauen, denn mittlerweile wird die Tabelle **Dg_Einzelbit** mit dem Ergebnis einer Datenbank gefüllt. Daher brauchen wir nur eine Entscheidung, ob ein neuer oder vorhandener Datensatz editiert wurde. Anschließend werden die **Projektbits** und **Variablenbits** in der Subroutine **Show_Einzelbit** neu geladen.

```
Private Sub TB_Funktion_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TB_Funktion.Leave
    Dim Id_Nr As Integer
    Id_Nr = Val(TB_Bit_Id.Text)
    If Id_Nr < 0 Then
```

```
        Store_Einzelbit(TB_Funktion.Text)
    Else
        EinzelbitTableAdapter.Update_Einzelbit(TB_Funktion.Text, Id_Nr)
    End If
    Load_Projektbits(Val(Tb_Projekt_Id.Text))
    Show_Einzelbit(Val(Tb_Var_ID.Text))
End Sub
```

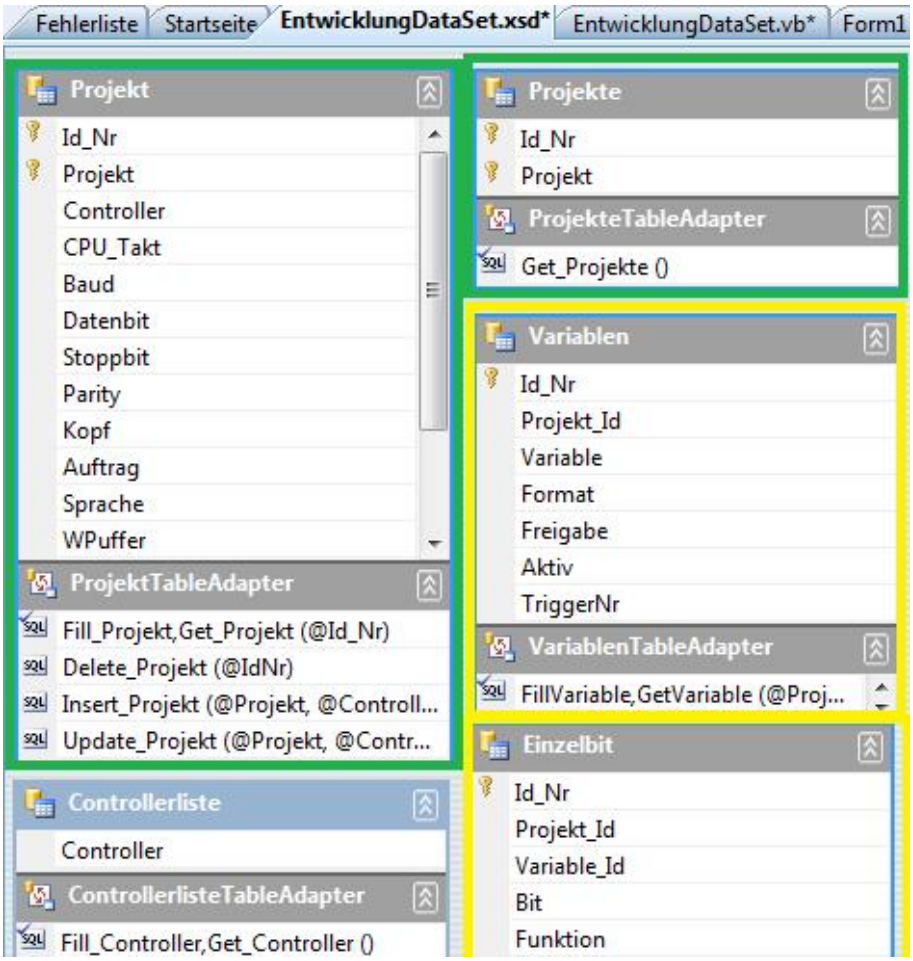
So brauchen wir nun nur noch die Subroutine **Store_Einzelbit**. Keine große Sache. Einziger Übergabeparameter ist die **Funktion**, alle anderen Daten liegen vor. Weil es einfach übersichtlicher ist, werden die Textinhalte der Textboxen über lokal deklarierte Variablen angepasst.

```
Public Sub Store_Einzelbit(ByVal Funktion As String)
    Dim Projekt_Id As Integer
    Dim Variable_Id As Integer
    Dim Bit As Byte
    Projekt_Id = Val(Tb_Projekt_Id.Text)
    Variable_Id = Val(Tb_Var_ID.Text)
    Bit = Val(Mid(TB_Bit.Text, 5))
    EinzelbitTableAdapter.Insert_Einzelbit(Projekt_Id, Variable_Id, Bit, Funktion)
End Sub
```

Nun noch abschließende Tests und die Datenspeicherung auf der Datenbank kann erst einmal als erledigt betrachtet werden.

### *1.3.13 Zwischenbilanz Dataset*

Die grün umrandeten Tableadapter sind nun fertig erstellt und bereits geprüft. Sieht doch schon ganz gut aus.




*alle fertigen Tableadapter*

Der nächste Schritt ist der Aufbau der dritten Seite für die Projektparameter.

## *1.4* Kommunikation und Visualisierung

Betreten wir nun den Bereich, der für die Kommunikation mit einem Controller eingerichtet wird. Unser Ziel ist es ja, die Variablen des Controllers auf dem PC abzubilden und deren Werte zur Laufzeit anzuzeigen. Die bisherige Arbeit diente der bequemen Aufbereitung der Information, die ein Assemblerlisting liefern kann. Nach einer Überarbeitung haben wir eine Datenbank eingerichtet und diese korrigierte Variablenliste in einem Projekt zusammengefasst und abgespeichert. Alles Arbeiten, die zwar auch interessant waren, doch mit dem Controller keine Kommunikation aufbauen konnten. Dieses Kapitel wird nun die Kopplung mit einer seriellen Datenübertragung auf der PC Seite bearbeiten und die Darstellung von übertragenen Daten organisieren.

Wie läuft eine solche Kommunikation ab und kann eine USB-Schnittstelle verwendet werden?

Im Prinzip kann ein Atmega8 oder Atmega16 zwar eine serielle Verbindung aufbauen, aber direkt mit einer USB-Schnittstelle kommunizieren geht nicht. Dafür gibt es aber ab ca. 5 € RS232-USB Konverter. Eine RS 232 ist früher die serielle Com-Schnittstelle eines PC gewesen, ausgeführt mit einem 9 pol. Sub-D Stecker. Die modernen PC haben aber kaum noch solche Ports. Uns stört es nicht, denn diese Konverter erlauben die Kommunikation ohne Probleme. Die Schnittstelle wird dann genauso eingerichtet, wie die früheren Com-Schnittstellen. Wir werden unser Programm so einrichten, das die verfügbaren Com-Schnittstellen in einer Combobox angezeigt werden.

Damit ein PC mit einem Controller kommunizieren kann, müssen ein paar grundsätzliche Dinge beachtet werden. Ein Kommunikationsbaustein, **UART** oder auch **USART** braucht erst einmal die Angabe, wie hoch die Geschwindigkeit der Datenübertragung erfolgen soll. Das ist die Baudrate. Noch zu Zeiten der TTY-Schreiber gab es Übertragungen mit 7 Bit. Die Schnittstelle lässt auch noch die Einstellung 7 oder 8 Datenbits zu, obwohl kaum eine Übertragung mit 7 Datenbit eingerichtet wird. Dennoch habe ich diese Auswahl eingebaut. Vielleicht will man ja mal eine exotische Datenübertragung ausprobieren oder einen alten TTY – Schreiber anschließen.

Des Weiteren gibt es das Stoppbit, welches eigentlich mehr oder weniger den Datenverkehr synchronisiert, Zur Wahl stehen 1 oder 2 Bit.

Schließlich kommt noch das Paritybit, das Fehler bei der Übertragung erkennen soll. Dabei wird zum Beispiel gewertet, ob die Anzahl der gesetzten Bits gerade oder ungerade ist. Sender und Empfänger müssen beide die gleiche Einstellung dieser Parameter haben.

Kommen wir zu den Kommunikationspuffern. Der Sendepuffer kann klein gehalten werden, da eine Sendung gezielt angestoßen wird. Beim Empfangspuffer ist das etwas anderes. Der PC weiß nicht, wann ein Datenstrom eintrifft. Natürlich werden die internen Schaltkreise aktiv, aber die Daten müssen irgendwo zwischengespeichert werden. Es ist ja möglich, das ein Anwenderprogramm grad nicht mit der Schnittstelle spricht oder sprechen kann. Solch eine Anwendung schaut gelegentlich mal in den Puffer, ja, aber lauert nicht auf einen Datenempfang. Dann wäre es nicht mehr bedienbar. Wie müssen wir uns das vorstellen?

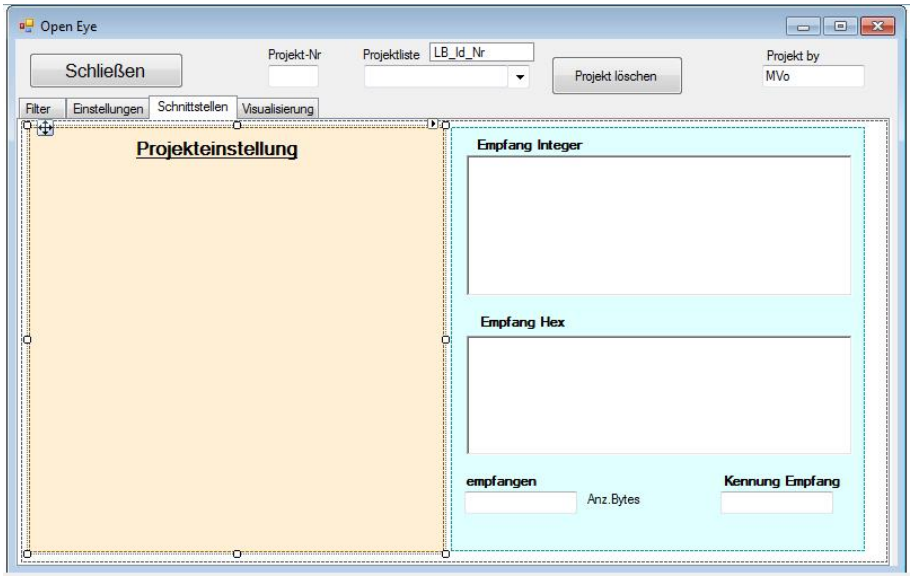
Also, der PC bearbeitet nacheinander verschiedene Programmteile. Die Schnittstelle erkennt einen Datenstrom und löst einen Interrupt aus. Die eintreffenden Daten werden über die Interrupt Routine in den Puffer geschrieben. Die Anwendung bekommt die Information, dass Daten eingetroffen sind und kann sie dann aus dem Puffer abholen. Bei einem so komplexen System, wie es ein PC ist, ist der Vorgang schwer korrekt zu erklären. Auf dem Controller werden wir selber den Schreib-Lesespeicher programmieren. Speziell der Lesespeicher oder Ringspeicher macht das Schnittstellenhandling deutlich. An dieser Stelle nehmen wir erst mal zur Kenntnis, dass es einen Schreib- und Lesepuffer gibt.

**Achtung: Die Größe muss ganzzahlig sein. Es wird sonst ein sehr unschöner Laufzeitfehler ausgelöst.**

### *1.4.1 Die Seite Projektparameter*

Auf der Seite Projektparameter werden die Schnittstellen zur Hardware eingerichtet So ist die Seite auch benannt. Zuerst trennen wir die Seite in zwei Bereiche und fügen zwei **Panels** ein, die der Optik wegen etwas eingefärbt werden. Das wird über die Eigenschaft **Backcolor** erledigt. Ein gemeinsames **Panel** benötigt diese Seite nicht, da wir uns die Möglichkeit erhalten wollen, auch ohne ein Projekt mit einem Controller zu kommunizieren. Die **Update**-Aufrufe werden einfach davon abhängig gemacht, ob in der Textbox **TB_Projekt_Id** eine **-1** steht und somit anzeigt, das noch kein Projekt erzeugt wurde.

Links kommt ein Label mit dem Text Projekteinstellung auf das Panel und auf das Panel der rechten Seite bauen wir zwei **RichTextBox** ein, in denen wir den Datenempfang visualisieren wollen. Die beiden Boxen erhalten jeweils ein **Label** als Überschrift. Interessant ist noch die Angabe, wie viele Daten empfangen wurden und wie der Kopftext aussieht. Dazu setzen wir zwei **Textboxen**.




*Erster Entwurf Seite 3*

Die rechte Seite ist schon fertig und nun werden noch die Namen für die Objekte vergeben. Die **Richtextbox** für den Empfang Integer nenne ich **Rt_Integer**, die darunter **Rt_Hex**.

Die linke Textbox **Tb_Empfang** und die rechte Textbox **Tb_Kennung**.

Nun wenden wir uns der linken Seite mit dem noch leeren **Panel** zu. Welche Objekte sind erforderlich. Betrachten wir einmal unsere Tabelle **Projekte**. Die Spalte **Id_Nr** ist intern, **Projekt** wird oben bereits bearbeitet. **Datum** und **Autor** ist ebenfalls schon zugewiesen. Was haben wir noch. Na klar, den verwendeten **Controller**, die **Taktfrequenz**, die **Schnittstellenparameter** und die Puffergrößen für **Schreib**- und **Lesepuffer** sowie die Empfängerkennung, also den **Telegrammkopf** und den **Befehl**, der gesendet werden soll.

Nun werden wir überlegen, welche Objekte erforderlich sind. Das Projekt enthält einen **Controller**, dafür setze ich eine **Combobox** und darüber eine **TextBox**. Der Grund, es sind ja bereits in anderen Projekten Controller verwendet. Warum nicht bereits erfasste Controller in einer Liste anbieten. Natürlich, wie bei den Projektnamen, muss das Programm auch neue Controller aufnehmen können. Allerdings muss auch ein neuer Controller aufgenommen werden können und deshalb wird die Textbox über der Combobox nicht gesperrt.

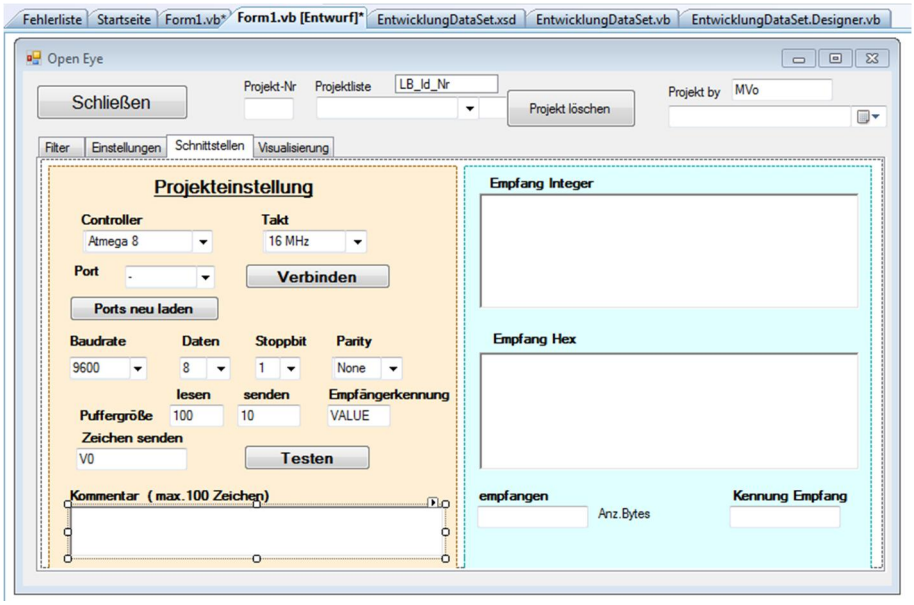
Die **Taktfrequenz** ist ebenfalls Bestandteil der Projektdatei. Hier gilt das gleiche wie für den Controller, also **Combobox** und darüber liegende **TextBox**. In dieser Liste werden die gängigsten Frequenzen vorbelegt , aber auch wieder die Möglichkeit eingeräunt, abweichende Frequenzen einzugeben. Auf ein Rücklesen bereits erfasster Frequenzen aus der Datenbank wird hier verzichtet.

**Ports** sind nicht Bestandteil der Projektdaten. Warum auch, je nach Verfügbarkeit ändert sich sowieso die Portnummer ständig. So muss man ihn nicht abspeichern. Aber trotzdem brauchen wir eine Liste verfügbarer Schnittstellen. Dafür nehmen wir wieder eine **Combobox** und darüber eine **Textbox**. Diese ist nun aber wieder gesperrt, denn Eingaben sind nicht sinnvoll und können zu Fehlern führen. Die Liste wird vom Programm zur Laufzeit mit verfügbaren Schnittstellen gefüllt.

Alle Schnittstellenparameter werden ebenfalls mit Textboxen für eine Eingabe gesperrt. Ob **Baud**, **Daten**- oder **Stoppbit** und auch **Parity**, alle Einstellungen werden vom Programm fest in die Comboboxen eingetragen. Die gesperrten **Textboxen** verhindern fehlerhafte oder unzulässige Eingaben. Für die Puffergröße **Schreiben** und **Lesen** sowie für die **Kennung** und den **Befehl** nehme ich einfache **Textboxen**. Diese Parameter gehören wieder zum Projekt und werden auch dem Projektdatensatz zugefügt.

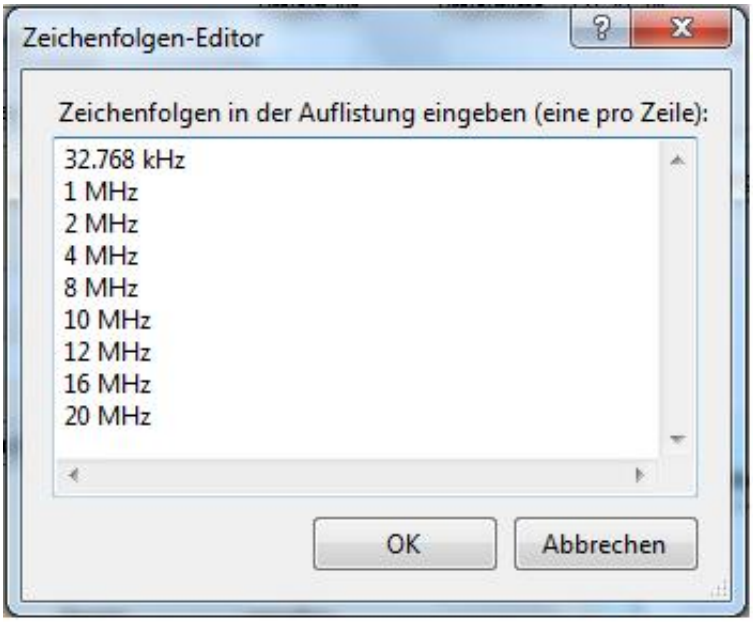
Nun braucht es noch ein paar Schalter, um Aktionen aus zu lösen.

Ein **Button**, um die gewählte Schnittstelle zu verbinden, ein **Button**, um ein Neu laden verfügbarer Schnittstellen anzustoßen und ein **Button**, um den **Befehl** an den Controller zu senden. Auch noch eine Richtextbox um unserem Projekt eine Bemerkung oder einen Kommentar zu verpassen.




*Aufbau Seite 3*

Nun vergeben wir noch die Objektnamen, angefangen mit der Combobox **Cb_Controller** und Textbox **Tb_Controller** zur Anzeige bereits verwendeter Controller. Es folgt die Combobox **Cb_Takt** mit den fest vorgegebenen Frequenzen in der Itemliste. Diese erhält man über die Eigenschaft **Items Auflistung**.



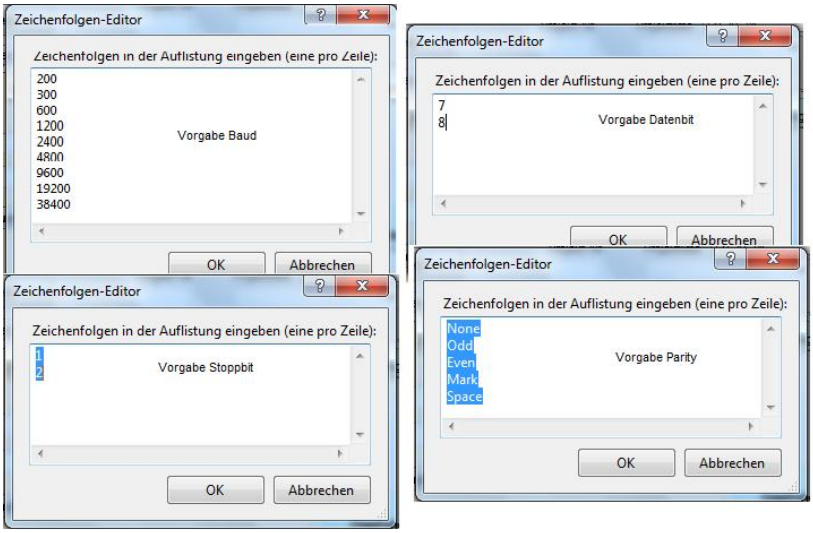

*Frequenzauswahl*

Sollten weitere Frequenzen erforderlich werden, können diese in der darüber liegenden Textbox **Tb_Takt** der Projekteinstellung mitgegeben werden. Eine Aufnahme in diese Liste ist nicht vorgesehen,

Der Inhalt der Combobox **Cb_Port**, bzw. **Tb_Port** ist nicht auf der Datenbank abgelegt. Hier werden auch keine Eingaben getätigt. Eine Systemabfrage wird die Combobox mit verfügbaren Schnittstellen füllen.

Die Schnittstellenparameter sind in **Cb_Baud**, **Cb_Daten**, **Cb_Stop** und **Cb_Parity** fest vorgegeben. Die darüber liegenden Textboxen **Tb_Baud**, **Tb_Daten**, **Tb_Stop** und **Tb_Parity** sind für den Zugriff gesperrt. Wenn diese Parametervorgabe nicht ausreicht, kann sie erweitert werden, aber sollte eine fehlerhafte Eingabe erfolgen, kann es zu einem Systemabsturz führen.

Das sind die vorbesetzten Itemlisten der Comboboxen.




*feste Vorgaben*

Für die beiden **Textboxen** für die Puffergröße werden die Namen **Tb_Read** und **Tb_Write** vergeben.

Auch die Namen **Tb_Kopf** und **Tb_Befehl** lassen sich sofort den beiden letzten **Textboxen** zuordnen. **Tb_Kopf** wird erst einmal mit **VALUE** besetzt und dient zur Synchronisierung des Datenstromes. Die Textbox **Tb_Befehl** bekommt erst einmal den Inhalt **V0**. Die Bedeutung dazu **V** bedeutet **Values** und **0** ohne Trigger. Entsprechend könnte eine Anforderung der Variableninhalte **V6** lauten. **Values**, wenn **Trigger 6** anspricht. Dies ist nur ein Beispiel. Genauso kann ein Controller **R4** - schalte **Relais 4 ein** und mit **r4** - schalte **Relais 4 aus** interpretieren.

Dann vergeben wir die Namen für die **Button**

**Bt_Connect** für das Button Verbinden
**Bt_New_Port** für das Button Ports neu laden und
**Bt_Test** für das Button Test zum Auslösen einer Datenübertragung. Dabei werden die Zeichen in der Textbox **TB_Befehl** an den Controller geschickt.
Bleibt noch die **RichTextBox** für den Kommentar **RT_Kommentar**

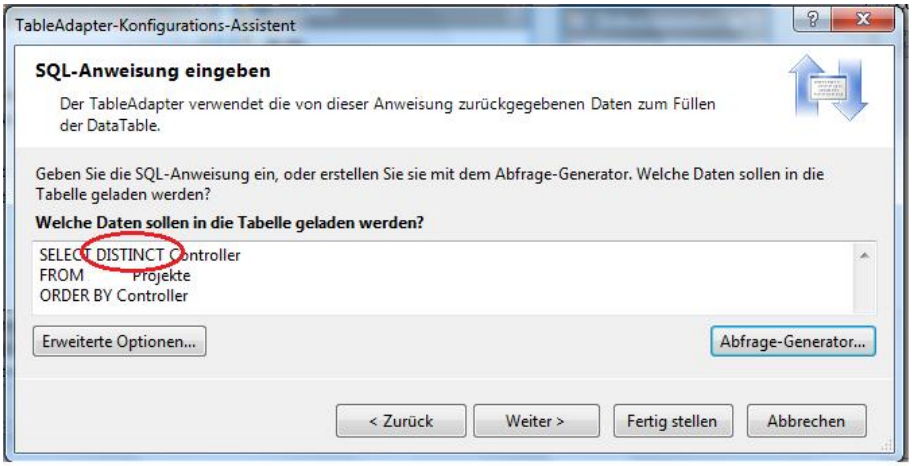
#### *1.4.1.1 Projektparameter laden*

Starten wir mit der ersten Combobox **Cb_Controller**. Hier soll eine Liste bereits verbauter oder eingesetzter Controller stehen. Diese bekommen wir aus der Datenbankabfrage des Tableadapters Controllerliste. In Kapitel 1.3.7 haben wir einen Tableadapter Controller in den Datenbankdesigner eingefügt. Wenn er noch nicht eingerichtet ist, werden wir dies jetzt nachholen. Wechseln wir in den Datenbankdesigner und klicken mit der rechten Maustaste den TableAdapter **Fill.., Get..** an

Zuerst entfernen wir wieder unter **erweiterte Optionen** die **Insert**-, Update- und **Delete**-Anweisungen

Im Abfragegenerator wählen wir nur die Spalte Controller aus und setzen die Sortierung. Nun kommt eine Anweisung, die nirgends vorgeschlagen wird - **Distinct**. Lassen wir diese Anweisung weg, bekommen wir von jedem Datensatz den Controller, ohne Rücksicht, ob vielleicht der Controller schon aufgenommen wurde. So erscheint er ohne Maßnahmen entsprechend oft in der Liste. Wir wollen aber nur unterschiedliche Einträge haben. Das Wort **Distinct** in der SQL-Anweisung erledigt dies. Auch wenn ein Atmega8 sechsmal verbaut ist, bekommen wir nur einen Atmega8 zurückgeliefert.

Die Anweisung **Distinct** fügen wir nun selbst an der gekennzeichneten Stelle in die **SQL** Anweisung.




*SQL Anweisung Distinct*

Die Namen für die Methoden vergeben wir mit **Fill_Controller** und **Get_Controller**. Nun haben wir eine Abfrage, die uns die Daten für die

Combobox **Cb_Controller** liefert. Der Eintrag erfolgt dann in einer kleinen Subroutine **Load_Controller**, die im **Frm_Open_Eye_Load** aufgerufen wird, noch vor dem Aufruf **Load_Projekte**.

Aber zuerst die Subroutine

```
Public Sub Load_Controller()
   Dim i As Integer
   Dim Anzahl As Integer
   CB_Controller.Items.Clear()                          ' Liste leeren
   Anzahl = ControllerlisteTableAdapter.Get_Controller.Rows.Count  ' Anzahl Datensätze
   If Anzahl > 0 Then                                  ' wenn ja
     For i = 0 To Anzahl - 1                           ' dann eintragen
       CB_Controller.Items.Add(ControllerlisteTableAdapter.Get_Controller.Item(i).Controller)
     Next
   End If
   TB_Controller.Text = "Atmega 8"
 End Sub
```

Das Textfeld besetzen wir pauschal mit dem Controller, mit dem wir häufig arbeiten.

Eine weitere Subroutine **Set_Default** wird für die restlichen Einstellungen angefertigt und ebenfalls noch vor **Load_Projekte** in der **Frm_Open_Eye_Load** aufgerufen

```
Public Sub Set_Default()
      TB_Takt.Text = CB_Takt.Items(1)                  ' 1 MHz
      TB_Baud.Text = "2400"
      TB_Daten.Text = "8"
      TB_Stop.Text = "1"
      TB_Parity.Text = CB_Parity.Items(0)              ' None
      TB_Read.Text = "200"
      TB_Write.Text = "10"
      TB_Kopf.Text = "VALUE"
      TB_Befehl.Text = "V0"
      TB_Datum.Text = DTP_Datum.Text
      Rt_Kommentar.Text="-"
   End Sub
```

Bei der Erstellung dieser Routine ist die ungeschickte Unterbringung des Datumfeldes aufgefallen. Klar, das Projekt braucht nur einen Datumstempel, aber der ist auf der Filterseite komplett falsch. Deshalb

kommt es auch in den oberen Bereich unter die Textbox **Tb_Autor** mit einer darüber liegenden Textbox. Sie ist hier nun als **Tb_Datum** eingetragen und wird beim Programmstart mit dem aktuellen Datum besetzt. Wird das Kalenderfeld geöffnet, überträgt das Schließen-Ereignis **DTP_Datum_CloseUp** ebenfalls das Datum in die Textbox.

```
Private Sub DTP_Datum_CloseUp(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles DTP_Datum.CloseUp
    TB_Datum.Text = DTP_Datum.Text                ' akt. Datum in Textbox übertragen
End Sub
```

So ein Ereignis ist ja schnell im Programm eingefügt. Auch das Ereignis **Frm_Open_Eye_Load** dürfte nun alle erforderlichen Aktivitäten erledigen.

```
Private Sub Frm_Open_Eye_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
  Load_Controller()                  ' bereits verwendete Controller aus der Projekttabelle
  Set_Default()                      ' weitere Defaultwerte setzen
  Init_Ports()                       ' verfügbare Ports vom System abfragen
  Load_Projekte("")                  ' Projektliste laden
  Load_Projekt(Val(Tb_Projekt_Id.Text))          ' erstes Projekt laden
  Load_Variable()                                ' zugehörige Variablen laden
  Load_Projektbits(Val(Tb_Projekt_Id.Text))      ' und die Einzelbitbeschreibungen
  TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
  Set_Edit_Variable(0)                           ' erste Variablenzeile editieren
  If Tb_Format.Text = "Byte" Then
      Show_Einzelbit(Val(Tb_Var_ID.Text))        ' Einzelbitbeschreibung öffnen
  End If
End Sub
```

### *1.4.1.2 Seriellen Port einrichten*

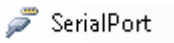
Kommen wir nun zur Liste verfügbarer Ports. Man muss schon ein wenig suchen, bis die Information, welche Wege beschritten werden müssen, vorliegt. Sollte es mal sein, das man eine Funktion, eine Aufgabe absolut nicht lösen kann, ist ein Besuch und eine nette Anfrage in einem der Foren sicherlich hilfreich. Ich kürz es hier mal ab und zeige den Aufbau der Subroutine **Init_Ports**, die bereits in der Aufrufliste der Ereignisroutine des Programmstarts **Frm_Open_Eye_Load** aufgeführt ist.

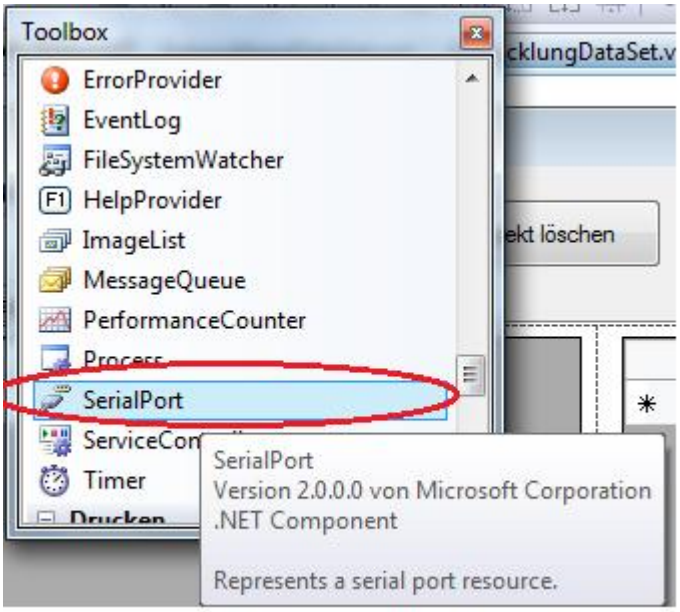
```
Public Sub Init_Ports()
      CB_Port.Items.Clear()                    ' Liste leeren
     ' In einer Schleife die Variable Sp (Serial Port) mit der Eigenschaft
     '  SerialPortNAmes besetzen
      For Each sp As String In My.Computer.Ports.SerialPortNames
         CB_Port.Items.Add(sp)
      Next
      TB_Port.Text = "-"
      If CB_Port.Items.Count > 0 Then
         CB_Port.Text = CB_Port.Items.Item(0) ' ersten Port anzeigen
         TB_Port.Text = CB_Port.Text
      End If
   End Sub
```

Das Konstrukt **for Each** ist etwas ungewöhnlich. Wer darüber mehr wissen will, darf den Cursor in das Wort **Each** setzen und **F1** betätigen. Hier führt die Erklärung zu weit. Nehmen wir einfach an, das jede gefundene Schnittstelle durch diese Schleife in unsere Combobox eingetragen wird.

Wird nun das Programm gestartet, ohne das ein USB-RS 232 Wandler angeschlossen ist, enthält die Liste keinen Port. Erst, wenn ein USB-RS 232 Wandler angeschlossen ist, bekommen wir auch einen verfügbaren Com-Anschluss.

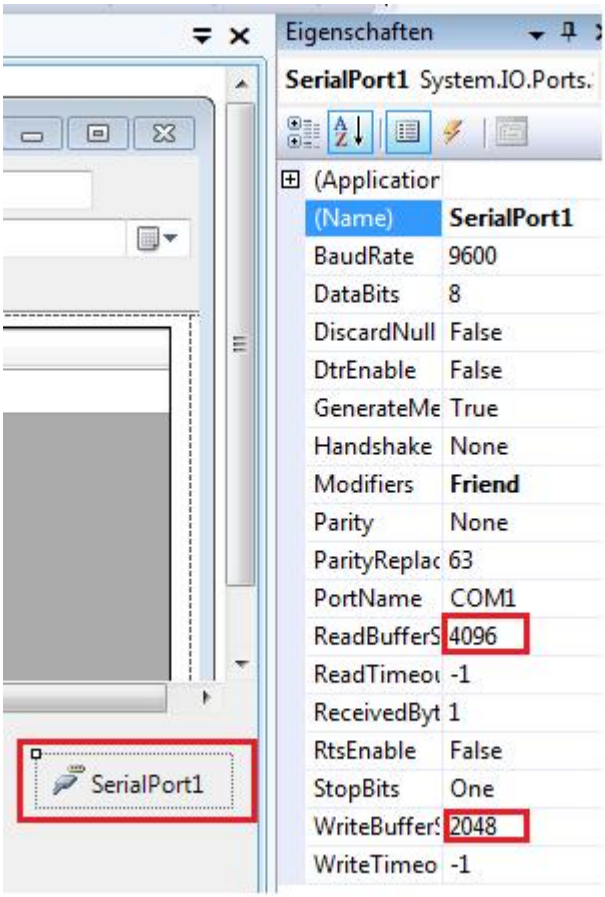
Nun brauchen wir ein Objekt, welches die Schnittstelle darstellt. Wir reden von einer seriellen Schnittstelle. Wenn wir die Toolbox durchsuchen, finden wir nur einen einzigen Eintrag, der dem entsprechen könnte. Das Objekt **SerialPort**.

*ICON SerialPort* 




*Tool SerialPort*

Ziehen wir es einmal auf unsere Anwendung. Es erscheint nicht in unserem Programm, sondern in einem Bereich darunter. Dort befinden sich auch schon die von uns erstellten **Tableadapterobjekte**. Nun gut, werfen wir einen Blick auf die Eigenschaften.




*Die Eigenschaften von SerialPort*

Hier finden wir nun die ganzen Parameter wieder, für die wir **Comboboxen** und **Textboxen** geschaffen haben. Auch die Schreib- und Lesepuffergröße. Also liegen wir richtig.

### *1.4.1.3 Schnittstellenparameter setzen*

Nun brauchen wir nur mit dem Ereignis **SelectedIndexChanged** der Comboboxen die Auswahl auf diese Parameter übertragen und die Schnittstelle ist eingerichtet. Dabei überschreiben wir auch gleich die Projektdaten, wenn in der Textbox **Tb_Projekt_Id nicht -1** steht.

Auch in der Subroutine **Set_Default** werden die Defaultparameter der Schnittstelle zugewiesen. Ergänzen wir nun zuerst die Subroutine **Set_Default**.

```
Public Sub Set_Default()
  TB_Takt.Text = CB_Takt.Items(1)
  TB_Baud.Text = "2400"
  TB_Daten.Text = "8"
  TB_Stop.Text = "1"
  TB_Parity.Text = CB_Parity.Items(0)
   TB_Read.Text = "200"
   TB_Write.Text = "10"
   TB_Kopf.Text = "VALUE"
   TB_Befehl.Text = "V0"
   TB_Datum.Text = DTP_Datum.Text
   RT_Kommentar.Text="-"
   SerialPort1.ReadBufferSize = Val(TB_Read.Text)
   SerialPort1.WriteBufferSize = Val(TB_Write.Text)
   SerialPort1.BaudRate = Val(TB_Baud.Text)
   SerialPort1.DataBits = Val(TB_Daten.Text)
   SerialPort1.StopBits = Val(TB_Stop.Text)
   If TB_Parity.Text = "None" Then
                  SerialPort1.Parity = IO.Ports.Parity.None
   If TB_Parity.Text = "Odd" Then
                  SerialPort1.Parity = IO.Ports.Parity.Odd
   If TB_Parity.Text = "Even" Then
                  SerialPort1.Parity = IO.Ports.Parity.Even
   If TB_Parity.Text = "Mark" Then
                  SerialPort1.Parity = IO.Ports.Parity.Mark
   If TB_Parity.Text = "Space" Then
                  SerialPort1.Parity = IO.Ports.Parity.Space
End Sub
```

Im nächsten Schritt ergänzen wir die Subroutine **Load_Projekt**, den auch hier sind die Schnittstellenparameter eines Projektes hinterlegt und müssen in die Parametrierung übernommen werden

```
Public Sub Load_Projekt(ByVal Id_Nr As Integer)
  Dim Anzahl As Integer
  Anzahl = ProjektTableAdapter.Get_Projekt(Id_Nr).Rows.Count
  If Anzahl > 0 Then
    Tb_Projekte.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Projekt
    TB_Projekt_Alt.Text = Tb_Projekte.Text
    TB_Controller.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Controller
    TB_Takt.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).CPU_Takt
    TB_Baud.Text = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Baud)
    SerialPort1.BaudRate = Val(TB_Baud.Text)
    TB_Daten.Text = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Datenbit)
    SerialPort1.DataBits = Val(TB_Daten.Text)
    TB_Stop.Text = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Stoppbit)
    SerialPort1.StopBits = Val(TB_Stop.Text)
    TB_Parity.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Parity
    If TB_Parity.Text = "None" Then SerialPort1.Parity = IO.Ports.Parity.None
    If TB_Parity.Text = "Odd" Then SerialPort1.Parity = IO.Ports.Parity.Odd
    If TB_Parity.Text = "Even" Then SerialPort1.Parity = IO.Ports.Parity.Even
    If TB_Parity.Text = "Mark" Then SerialPort1.Parity = IO.Ports.Parity.Mark
    If TB_Parity.Text = "Space" Then SerialPort1.Parity = IO.Ports.Parity.Space
    TB_Kopf.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Kopf
    TB_Befehl.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Auftrag
    TB_Read.Text = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).RPuffer)
    SerialPort1.ReadBufferSize = Val(TB_Read.Text)
    TB_Write.Text = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).WPuffer)
    SerialPort1.WriteBufferSize = Val(TB_Write.Text)
    TB_Sprache.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Sprache
    TB_Autor.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Autor
    TB_Datum.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Datum
  End If
End Sub
```

Um diese Daten in das Projekt zu übertragen, haben wir bereits eine Subroutine **Set_Projektdaten** vorbereitet. Zu diesem Zeitpunkt existierten allerdings die Objekte der dritten Seite noch nicht und es wurden Konstanten zugewiesen. Das ändern wir nun und weisen die Werte der neuen Textboxen zu.

```
Public Sub Set_ProjektDaten(ByVal Projekt_Id As Integer, ByVal Projekt As String)
    Dim Projektdaten As ProjektRecord
```

```
    Projektdaten.Id_Nr = Projekt_Id
    Projektdaten.Projekt = Projekt
    Projektdaten.Controller = TB_Controller.Text
    Projektdaten.Takt = TB_Takt.Text
    Projektdaten.Baud = CB_Baud.Items.IndexOf(TB_Baud.Text)
    Projektdaten.Daten = Val(TB_Daten.Text)
    Projektdaten.Stopp = Val(TB_Stop.Text)
    Projektdaten.Parity = CB_Parity.Items.IndexOf(TB_Parity.Text)
    Projektdaten.Kopf = TB_Kopf.Text
    Projektdaten.Auftrag = TB_Befehl.Text
    Projektdaten.RPuffer = Val(TB_Read.Text)
    Projektdaten.WPuffer = Val(TB_Write.Text)
    Projektdaten.Sprache = TB_Sprache.Text
    Projektdaten.Autor = TB_Autor.Text
    Projektdaten.Datum = TB_Datum.Text
    Projektdaten.Kommentar = Rt_Kommentar.Text
    If Projekt_Id < 0 Then
        Insert_Projekt(Projektdaten)
    Else
        Update_Projekt(Projektdaten)
    End If
End Sub
```

### *1.4.1.4 Schnittstellenparameter ändern*

Nun werden wir noch den Objekten dieser Seite die Ereignisse zuweisen, die eine Aktualisierung der Daten auf der Datenbank einleiten und die Ansicht aktualisieren. Beginnen wir oben mit dem Controller und der Combobox. Nur wenn die Inhalte von **Tb_Controller.Text** und **CB_Controller.Text** unterschiedlich sind, soll eine Aktion ausgeführt und die Daten nur aktualisiert werden, wenn auch ein Projekt angelegt oder von der Datenbank geladen ist.

```
Private Sub CB_Controller_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles CB_Controller.SelectedIndexChanged
   If TB_Controller.Text <> CB_Controller.Text Then
     TB_Controller.Text = CB_Controller.Text
      If Tb_Projekt_Id.Text <> "-1" Then
         Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
      End If
   End If
 End Sub
```

Da auch ein ungelisteter Controller eingesetzt sein kann, müssen wir eine Eingabe in der Textbox **Tb_Controller** ebenfalls mit einem Ereignis zur Aktualisierung der Datenbank versehen. Das benötigte Ereignis ist **Tb_Controller_Leave**

```
Private Sub TB_Controller_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TB_Controller.Leave
   If Tb_Projekt_Id.Text <> "-1" Then
      Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
  End If
End Sub
```

Hier genügt nur die Abfrage nach dem gültigen Projekt. Auf gleiche Weise verfahren wir mit der Combobox **Cb_Takt** und der Textbox **TB_Takt**.

Die Combobox **Cb_Ports** schreibt nur den ausgewählten Eintrag in die Textbox und trägt ihn in den **SerialPort** ein. Die Textbox ist für den Zugriff über die Eigenschaft **Enabled = False** gesperrt.

```
Private Sub CB_Port_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles CB_Port.SelectedIndexChanged
      TB_Port.Text = CB_Port.Text
      SerialPort1.PortName = TB_Port.Text
```

```
End Sub
```

Das Button **Ports neu laden** ruft nur die Routine **Init_Ports** auf.

```
Private Sub Bt_Port_New_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Port_New.Click
    Init_Ports()
End Sub
```

Allerdings werden wir nun die Subroutine **Init_Ports** etwas erweitern. Es macht keinen Sinn, wenn keine Ports verfügbar sind, die Combobox **Cb_Port** bedienbar zu lassen. Auch das Button **Verbinden** muss gesperrt werden, da ja keine Schnittstelle existiert. Diese beiden Anweisungen fügen wir der Subroutine **Init_Ports** hinzu.

```
Public Sub Init_Ports()
  CB_Port.Items.Clear()
  For Each sp As String In My.Computer.Ports.SerialPortNames
     CB_Port.Items.Add(sp)
  Next
  TB_Port.Text = "-"
  If CB_Port.Items.Count > 0 Then
    CB_Port.Text = CB_Port.Items.Item(0)
    TB_Port.Text = CB_Port.Text
     SerialPort1.PortName = TB_Port.Text
   End If
   CB_Port.Enabled = CB_Port.Items.Count > 0
   Bt_Connect.Enabled = CB_Port.Items.Count > 0
 End Sub
```

Die anderen Comboboxen besetzen wieder die Textboxen, aktualisieren die Datenbank und beschreiben den Serialport. Die Textboxen darüber werden für den Zugriff mit der Eigenschaft **Enabled = False** grundsätzlich gesperrt.

Das Ereignis **Cb_Baud_SelectedIndexChanged**

```
Private Sub CB_Baud_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles CB_Baud.SelectedIndexChanged
    TB_Baud.Text = CB_Baud.Text
    If Tb_Projekt_Id.Text <> "-1" Then
      Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    End If
    SerialPort1.BaudRate = Val(TB_Baud.Text)
  End Sub
```

Das Ereignis **Cb_Daten_SelectedIndexChanged**

```
  Private Sub CB_Daten_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles CB_Daten.SelectedIndexChanged
    TB_Daten.Text = CB_Daten.Text
    SerialPort1.DataBits = Val(TB_Daten.Text)
    If Tb_Projekt_Id.Text <> "-1" Then
      Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    End If
  End Sub
```

Das Ereignis **Cb_Stop_SelectedIndexChanged**

```
  Private Sub CB_Stop_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles CB_Stop.SelectedIndexChanged
    TB_Stop.Text = CB_Stop.Text
    SerialPort1.StopBits = Val(TB_Stop.Text)
    If Tb_Projekt_Id.Text <> "-1" Then
      Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    End If
  End Sub
```

Das Ereignis **Cb_Parity_SelectedIndexChanged**

```
  Private Sub CB_Parity_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles CB_Parity.SelectedIndexChanged
    TB_Parity.Text = CB_Parity.Text
    If TB_Parity.Text = "None" Then SerialPort1.Parity = IO.Ports.Parity.None
    If TB_Parity.Text = "Odd" Then SerialPort1.Parity = IO.Ports.Parity.Odd
    If TB_Parity.Text = "Even" Then SerialPort1.Parity = IO.Ports.Parity.Even
    If TB_Parity.Text = "Mark" Then SerialPort1.Parity = IO.Ports.Parity.Mark
    If TB_Parity.Text = "Space" Then SerialPort1.Parity = IO.Ports.Parity.Space
    If Tb_Projekt_Id.Text <> "-1" Then
```

```
        Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    End If
End Sub
```

Die beiden Textboxen **Tb_Kopf** und **Tb_Befehl** übertragen im **Leave** Ereignis nur die Änderung auf die Datenbanktabelle unter Abfrage der Projekt-Id.

```
Private Sub TB_Kopf_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Kopf.Leave
    If Tb_Projekt_Id.Text <> "-1" Then
        Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    End If
End Sub
```

```
Private Sub TB_Befehl_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Befehl.Leave
    If Tb_Projekt_Id.Text <> "-1" Then
        Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    End If
End Sub
```

Bei den Textboxen **Tb_Read** und **Tb_Write** bauen wir eine kleine Korrektur ein. Es ist zwar nirgends geschrieben, dass die Puffer nur gradzahlig sein dürfen, aber ich habe sehr lange damit verbracht diesen Fehler zu finden. Die Fehlermeldung war dabei nicht sehr hilfreich. Das Gemeine daran, er taucht nur zur Laufzeit auf. Natürlich sichern wir das nun ab. Das Ereignis dafür ist bei beiden Textboxen **Leave**, welches auch die Datenbank aktualisiert.

```
Private Sub TB_Read_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Read.Leave
    Dim Korrektur As Integer
    Korrektur = Math.Round(Val(TB_Read.Text) / 2) * 2
    TB_Read.Text = Str(Korrektur)
    SerialPort1.ReadBufferSize = Korrektur
    If Tb_Projekt_Id.Text <> "-1" Then
        Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
```

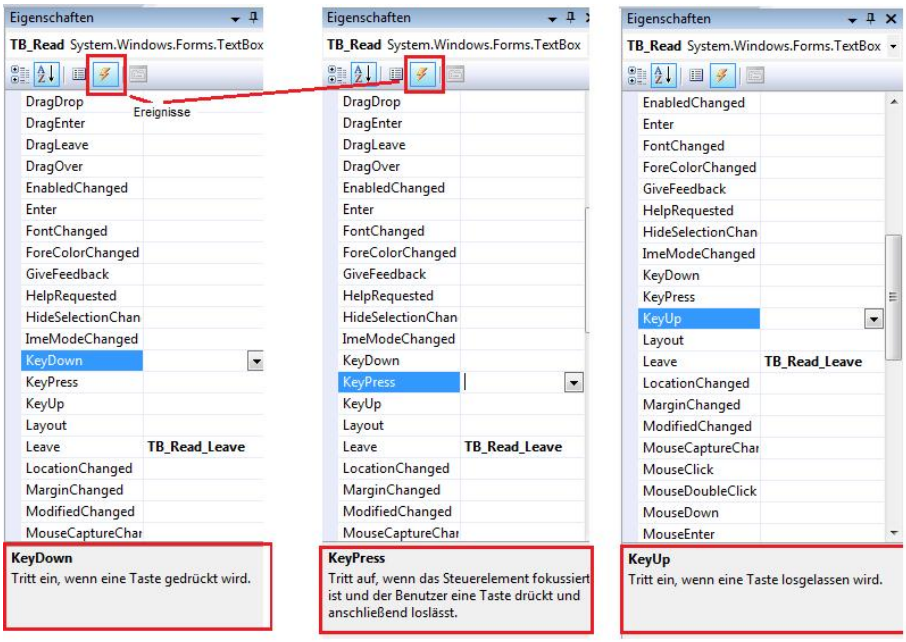
```
    End If
End Sub
```

```
Private Sub TB_Write_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Write.Leave
    Dim Korrektur As Integer
    Korrektur = Math.Round(Val(TB_Write.Text) / 2) * 2
    TB_Write.Text = Str(Korrektur)
    SerialPort1.WriteBufferSize = Korrektur
    If Tb_Projekt_Id.Text <> "-1" Then
        Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    End If
End Sub
```

Eine solche Korrektur ist praktisch, denn nun braucht bei der Eingabe nichts mehr beachtet werden. Wirklich? Dann gebt mal abc, also Buchstaben ein und verlasst die Textbox. Ja, da schlägt ein Laufzeitfehler zu, denn Buchstaben lassen sich nun mal nicht zu Zahlen wandeln. Auf eine Aussage, das macht doch keiner, sollten wir keinen Wert legen, sondern überlegen, wie zu verhindern ist, das keine Ziffern eingegeben werden.

### *1.4.1.5 Textboxeingaben kontrollieren*

Natürlich muss es ein Ereignis sein, welches ein Zeichen in die Textbox schreibt. Vielleicht gibt es ja ein Ereignis, in dem die Zeichen vor übernahme geprüft werden können. Sehen wir uns einmal die Ereignisse an, die von der Tastatur kommen.




*Ereignis KeyDown*

Eines der drei Ereignisse ist für unseren Zweck geeignet, Zeichen abzufangen und eventuell zu verändern. Die Entscheidung fällt auf **KeyPress**. Was können wir dort unternehmen, um zu verhindern das Buchstaben in eine Textbox geschrieben werden? Diese Übung hatten wir bereits auf Seite 54 beschrieben.

Geben wir einen Doppelklick auf das Ereignis **KeyPress**, um einen Rahmen für die Ereignisroutine zu bekommen.

```
Private Sub TB_Read_KeyPress(ByVal sender As System.Object, ByVal e As System.Windows.Forms.KeyPressEventArgs) Handles TB_Read.KeyPress
```

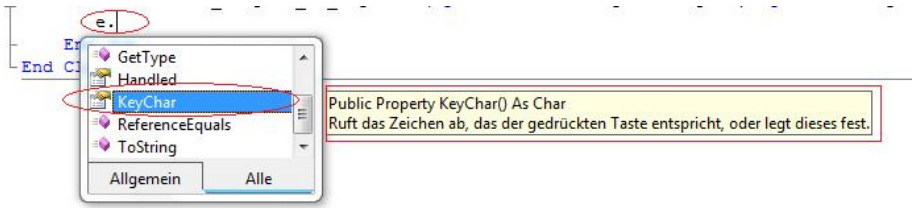
```
End Sub
```

Bisher ist den Parametereinträgen in einer automatisch erstellten Subroutine wenig Aufmerksamkeit geschenkt worden. Hier ist es erstmalig wichtig, diese Parameter genauer anzusehen.

**ByVal Sender as System.Objekt** gibt dem Unterprogramm die Info, wer dieses Ereignis ausgelöst hat. Eine weitere Variable in der Parameterliste heißt einfach nur **e** und ist vom Typ **PressEventArgs**. Mit ein wenig Kombinationsgabe und den bisherigen Erfahrungen werden wir ein paar Versuche anstellen.

**e** allein lässt sich nicht zuweisen. Wenn wir **e=““** schreiben, meckert der Compiler, das er **System.Windows.Form.KeyPressEventArgs** das nicht zuweisen kann. Also muss da ja noch etwas sein. Setzen wir hinter **e** mal einen Punkt

Na, da sag einer was gegen Visual Basic




*Der Ereignisparameter e*

Man wird also direkt darauf hingewiesen: es gibt ein Argrument **KeyChar**. Wenn wir nun einfach eine Abfrage nach Gültigkeit machen, dann können wir auch **KeyChar**, wenn es keine Ziffer ist, mit einem leeren **Char** überschreiben. Einfach zwei aufeinanderfolgende Textzeichen und der **Char** ist leer. Das probieren wir einmal aus.

Da die Zeichen 0 -9 hintereinander stehen, geht auch die Abfrage einfach. Es ist nur daran zu denken, das die Ziffern als Zeichen und keine Zahlen abgefragt werden müssen

```
If e.KeyChar < "0" Or e.KeyChar > "9" Then e.KeyChar = ""
```

Ein Test bestätigt nun unsere Vermutung. Die Textbox lässt nur noch Ziffern zu. Buchstaben werden nicht angenommen. Ja, aber nicht nur Buchstaben, auch Cursorsteuerung und Zeichen löschen geht nicht. Das

ist so nicht praktikabel. Nur, wie erfahren wir, welche **KeyChar**-Inhalte noch zugelassen werden müssen. Dazu nehmen wir die Textbox **Tb_Write** zu hilfe. So, wie wir ein eingegebenes Zeichen verändern können, ist es auch möglich die Textbox zu beschreiben. Wenn wir uns nun über die Funktion **Asc(<Char>)** den **ASCII**-Code liefern lassen und in der Textbox ausgeben, wissen wir, welche **Chr(<ASCII-Code>)** zugelassen werden müssen.

Probieren wir es gleich aus.

```
Private Sub TB_Write_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TB_Write.KeyPress
    TB_Write.Text = Asc(e.KeyChar)
    e.KeyChar = ""  'inhalt leeren, sonst wird Anzeige verfälscht.
End Sub
```

Dabei stellen wir fest, das die Entfernen Taste noch funktioniert. Bei der Taste **Backspace** erhalten wir den **Code 8**. Das sollte reichen. Nun soll auch höchstens eine 3 stellige Zahl eingegeben werden. Auch das können wir in dieser Routine unterbringen. Die Anzahl der in der Textbox bereits enthaltenen Zeichen erhalten wir über die Funktion **Len(<Text>)**.

```
Private Sub TB_Read_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TB_Read.KeyPress
  If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
    e.KeyChar = ""
  End If
  If Asc(e.KeyChar) <> 8 And Len(TB_Read.Text) = 3 Then
    e.KeyChar = ""
  End If
End Sub
```

Dabei ist auf dei Klammersetzung zu achten. Eine boolsche Klausel wird, wie in der Mathematik **Punkt vor Strich,** mit **Und vor Oder** bearbeitet. Ist etwas anderes gewünscht, sind Klammern erforderlich.

Nun übertragen wir diesen Code auch in das Ereignis **Tb_Write_KeyPress**

```
Private Sub TB_Write_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TB_Write.KeyPress
```

```
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(TB_Write.Text) = 2 Then
        e.KeyChar = ""
    End If
  End Sub
```

Die Eingabelänge wird auf 2 begrenzt.

Es ist nun möglich, das vor dem Eintrag in der Textbox ein Leerzeichen steht, das von der Datenbank bei der Konvertierung der Zahl zum Text eingebaut wird. Das verkürzt natürlich dann unerwünscht die Eingabe. Abhilfe bringt die Funktion **Trim**. Sie löscht alle führenden und abschließenden Leerzeichen. Wir haben sie auch schon benutzt. In der Subroutine **Load_Projekt** ändern wir die entsprechenden Zuweisungen. Hier das Beispiel an **Tb_Read** und **Tb_Write**

```
TB_Read.Text = Trim(Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).RPuffer))
```

```
TB_Write.Text = Trim(Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).WPuffer))
```

Damit sind fehlerhafte Eingaben in den Textboxen nicht mehr möglich.

#### *1.4.1.6 Darstellung empfangener Werte*

Auf der Rechten Seite haben wir zwei **RichTextBoxen** eingesetzt, die uns die empfangenen Daten anzeigen sollen. Dabei ist das obere Fenster, welches die Zahlenwerte liefert kein Problem, aber bitte, was ist eine Hex-Darstellung. Lasst mich hier mal ein klein wenig ausholen. Eine Zahlendarstellung kann in verschiedener Form dargestellt werden. Als Dezimalzahl, das wissen wir, denn damit sind wir groß geworden. Als Binärzahl, das ist uns erst mit der Erfindung des Computers bewusst und nun Hexzahl? Eine Hexzahl ist bei einem Byte immer 2 Zeichen groß, bei einer Zweibytezahl, einem Word, wie diese in der Computerwelt genannt wird, entsprechend 4 Zeichen. Das macht eine Zahlendarstellung übersichtlicher, wenn man sich eine ganze Liste davon ansehen soll. Klar, man könnte auch nur Texte schicken, die lassen sich auch hervorragend lesen. Na ja, wo es nicht um Zahlenwerte, sondern ASCII – Codes handelt ist das auch korrekt, aber wenn es Zahlenwerte sind, muss auch die Zahl 0 eine lesbare Form haben. Es gibt ASCII Codes, die bringen einen ganzen Texteditor durcheinander und von Übersichtlichkeit kann dann keine Rede sein. Eine Binärzahlendarstellung ist ungeeignet, da eine Liste Binärzahlen bald vor den Augen veschwimmt. Hexzahlen lassen auch große Zahlen gut lesbar in einer ordentlichen Struktur erscheinen. Aber wie stellt sich diese dar? Greifen wir auf eine grundlegende mathematische Basis unseres Zahlensystems. Das Binärsystem hat die Basis 2. Gut, es gibt nur zwei Zustände einer Ziffer 0 oder 1, alles was größer ist, landet in der nächsten Stelle. Somit ist die erste Stelle 2^0, die zweite Stelle 2^1, die dritte Stelle 2^2 usw. bewertet. Im Dezimalsystem haben wir die Basis 10. Damit sind die 10 Ziffern von 0 bis 9 darstellbar, jede größere Zahl landet in der nächsten Stelle. Auch hier die Stellenbewertung erste Stelle 10^0, die zweite Stelle 10^1, die dritte 10^2 usw. Klar, jetzt kommt Hexadezimal (Hexa griech.6+ dezimal 10) . Hier haben die Zahlen die Basis 16. Somit sind Ziffern von 0 bis 15 darstellbar. Hoppla, Zahlen von 0 bis 9 kennen wir, aber bitteschön, wie geht es dann weiter? Dann ist die Zahl doch 2stellig. Nein, denn hier wird einfach mit dem Alphabet ergänzt und die Ziffern A-F kommen hinzu. F hat dann den Wert 15. Somit ergibt sich die Stellenbewertung erste Stelle 16^0, zweite Stelle 16^1, dritte Stelle 16^3 usw. Man könnte, um nun diese Mathematik ausführlicher zu beschreiben schon eine kleine Abhandlung von mehreren Seiten verfassen. Hier kommt es mir aber nur darauf an, das Prinzip dieser Zahlen erklärt zu haben. Unsere Aufgabe wird nun sein, eine Dezimalzahl mit der Basis 10 in eine Hexadezimalzahl mit der Basis 16 zu wandeln und das ist gar nicht so schwer, wie es scheint.

Eine Dezimalzahl wird ganzzahlig durch 16 geteilt. Zum beispiel 34251. Das ergibt 2140 Rest 11, also B. Damit haben wir die unterste Ziffer B*16^0. Die verbleibenden 2140 werden wieder durch 16 geteilt. Das ergibt 133 Rest 12 für die zweite Ziffer also C*16^1 Teilen wir die verbliebenen 133 durch 16 und erhalten 8 Rest 5. Somit ist die dritte Ziffer 5*16^2 und die 8 brauchen wir gar nicht mehr teilen, denn 8/ 16 ist 0 rest 8. Somit ist die vierte Ziffer 8*16^3.

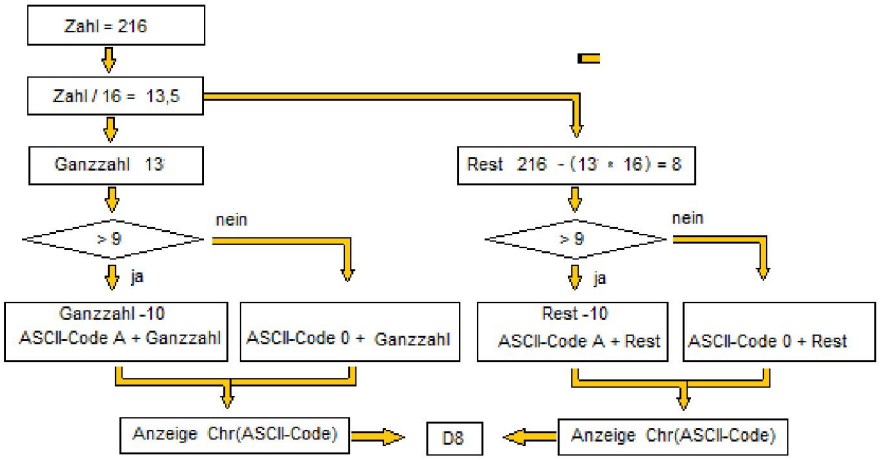
Dann rechnen wir einmal umgekehrt.

```
8*16^3=32768
5*16^2=1280
12*16^1=192
11*16^0=11
```

In einem Programm läuft das dann so ab:

Wir teilen wir eine Dezimalzahl durch 16 und multiplizieren den ganzzahligen Anteil wieder mit 16. Rest ist dann Ausgangswert minus dem Ergebnis der Multiplikation der Ganzzahl. Nun brauchen wir nur nachsehen, ob die Zahl <10 ist, wenn größer, dann wird der ASCII-Code von A genommen und von der Zahl 10 abgezogen. Der Rest zum ASCII_Code hinzu gezählt und wir haben einen ASCII Code zwischen A und F. Liegt die Zahl unter Zehn, wird diese zu ASCII Code = hinzugezählt und so erhalten wir ein ASCII Zeichen wischen 0 und 9. Das ging zu schnell?

Schauen wir uns mal den Ablauf in einer Grafik an



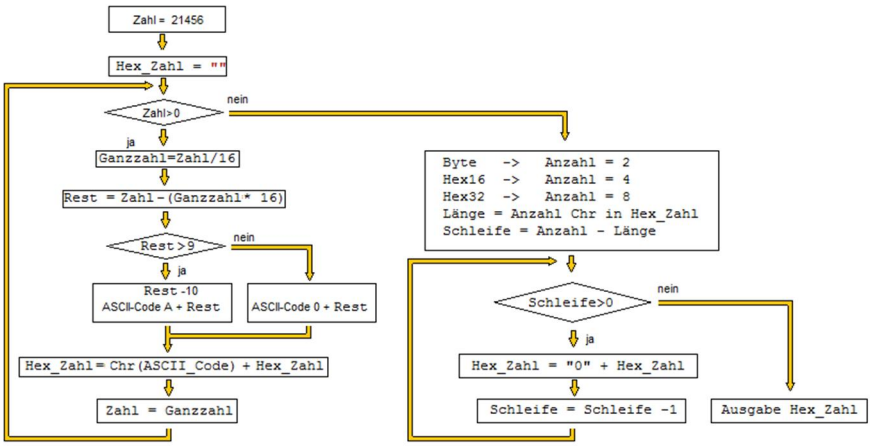

*PAP Hexadezimalzahl*

So wird der Ablauf deutlich. Auch die Mathematik stimmt, wie es die Gegenprobe zeigt

```
  16 ^ 0   * 8        =   8
+ 16 ^ 1   * D (13)   = 208
-----------------------------
           D 8        = 216
```

*Gegenprobe Hexadezimalzahl*

Nun können wir diese Aufgabe dem Programm übergeben. Der Vorgang ist wie folgt: Eine Zahl wird solange durch 16 geteilt, bis das Ergebnis 0 ist. Bei einer Zahl mit dem Format Byte ist klar, das es nur eine Teilung gibt und der obige Aufbau kann in zwei Schritten erledigt sein. Größere Zahlen lassen sich auch so erstellen, aber es geht auch mit einem kleinen Code und sehr elegant in einer Schleife. Übergeben wir dieser Routine noch das Format Byte, Word oder DWord, bzw. Hex16 oder Hex32, wie wir es in unserer Auswahlliste genannt hatten, können die Stellen nach oben mit 0 aufgefüllt werden. So ist eine Hex-Zahl nicht 8, sondern 08. Auch hierzu erst einen Ablaufplan




*PAP Hexadezimalzahl mit Schleife*

Nun fällt es auch nicht besonders schwer, diesen Ablauf in einem Programm abzubilden. Dies erfolgt in einer Funktion **IntToHex** mit den Übergabeparametern **Wert** und **Format**, da wir einen Wert zurück bekommen wollen. Da dieser Wert in einer Textbox angezeigt werden soll, ist der Datentyp String.

```
Public Function IntToHex(ByVal Wert As Byte, ByVal Format As String) As String
    Dim Hex_Zahl As String
    Dim Zahl As Integer
    Dim Temp As Integer
    Dim Rest As Integer
    Dim Laenge As Integer
    Dim i As Integer
    Dim CharCode As Integer
    Laenge = 2                    'Formatlänge 2 vorgeben
    Hex_Zahl = ""                 ' Hex_Zahl leeren
    Zahl = Wert                   ' Wert nach Zahl übertragen
    While Zahl > 0                ' verbleibenden Zahlenwert prüfen
      Temp = Math.Truncate(Zahl / 16) ' Zwischenwert bilden
      Rest = Zahl - (Temp * 16)     ' Rest aus Division
      If Rest > 9 Then              ' Bereich 0 - 9 oder A-F
        CharCode = Asc("A")
        Rest = Rest - 10
        CharCode = CharCode + Rest
      Else
        CharCode = Asc("0") + Rest
      End If
      Hex_Zahl = Chr(CharCode) + Hex_Zahl   ' Hex_Zahl zusammenstellen
      Trim(Hex_Zahl)                        ' von hinten nach vorn
      Zahl = Temp                           ' Zahl für nächsten Durchlauf setzen
    End While
    If Format = "Hex16" Then Laenge = 4     ' Formatlänge evtl. korrigieren
    If Format = "Hex32" Then Laenge = 8
    While Len(Hex_Zahl) < Laenge
      Hex_Zahl = "0" + Hex_Zahl             ' Hex_Zahl mit 0 auffüllen
    End While
    Return (Hex_Zahl)
  End Function
```

Um diese Funktion zu testen, und so sollten wir immer verfahren, wenn eine neue Aufgabe umgesetzt wurde, behelfen wir uns mit der bisher nicht genutzen Seite Visualisierung.

Dort bauen wir einfach ein Button ein zwei Textboxen und eine Combobox die mit den **Items** Byte **Hex16** und **Hex32** gefüllt wird. Das Textfeld der Combobox wird in **Eigenschaft** mit dem Text **Byte** vorbesetzt.

Das Ereignis Click des Button wird nun mit dem Aufruf der Funktion versehen und das Ergebnis der zweiten Textbox zugewiesen. Da wir hier keine Namen vergeben haben, müssen wir uns aus der Eigenschaft der beiden Texboxen sowie auch von der Combobox und dem Button die automatisch vergebenen Namen merken. Auf Absicherungen durch Fehleingaben verzichten wir, da diese Installation einzig der Prüfung der Funktion **IntToHex** dient. Somit ergibt sich ein primitiver Funktionsaufruf im Ereignis Click.

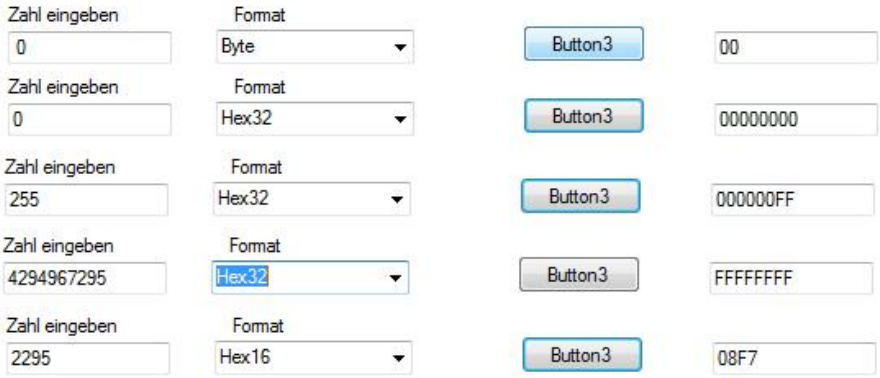
```
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button3.Click
    TextBox4.Text = IntToHex(Val(TextBox3.Text), ComboBox1.Text)
  End Sub
```

Wir können nun verschiedene Zahlen und Zahlenformate testen. Schnell stellen wir fest, das der Zahlenbereich Integer bei weitem nicht ausreicht. Klar, wir brauchen **Int32**, da ein Doppelwort auch 32 Bit hat. Bei den Tests meldet das Programmbei zahlen > 2147483647 einen Laufzeitfehler und stürzt ab. Somit ist die größte Hexzahl 7FFFFFFF, also nur 31 Bit groß. Das liegt daran, das der Zahlenbereich bei einer 32 Bit integer von –2147483646 bis +2147483647 geht. Das 32. Bit ist das Vorzeichen und alle Hexzahlen > 7FFFFFFF sind negativ. Um die vollen acht Byte mit FFFFFFFF anzuzeigen, sind wir gezwungen, mit **Int64** zu arbeiten. Das ergibt bei 4294967295 eine Anzeige von FFFFFFFF. Für die Vorgabe der Formate ist das aber unerheblich. Da wir die vollen 32 Bit Werte sehen wollen, übergeben wir auch eine Int64 und auch die lokalen Variablen für die Berechnungen werden mit Int64 deklariert. Wir können nun verschiedene Berechnungen mit den eingebauten Taschenrechner überprüfen.

Passen wir nun die Zahlenformate in der Übergabe und der Variablendeklaration an.

```
Public Function IntToHex(ByVal Wert As Int64, ByVal Format As String) As String
    Dim Hex_Zahl As String
    Dim Zahl As Int64
    Dim Temp As Int64
    Dim Rest As Int64
```

Nun sollten alle Formate bis zur 32Bit Zahl ordentlich und auch mit führenden Nullen dargestellt werden. Der zusammengestellte Screenschot zeigt das Ergebnis verschiedener Formatierung.

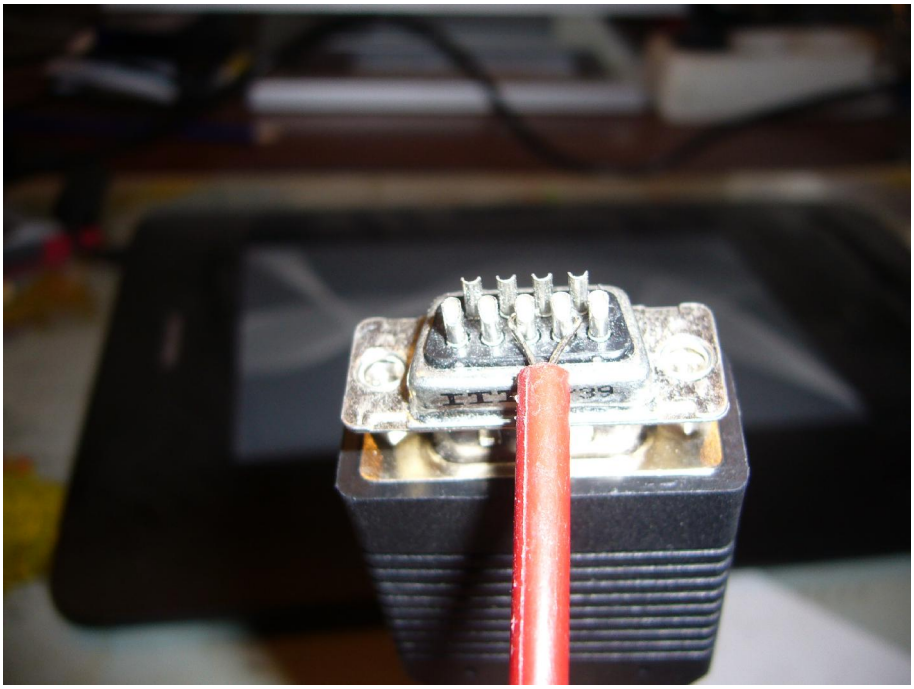



*Test mit Zahlen*

Damit ist diese Funktion geprüft und wir können uns den nächsten Aufgaben widmen. Die provisorischen Elemente können wieder entfernt werden.

### *1.4.2 Daten senden und empfangen*

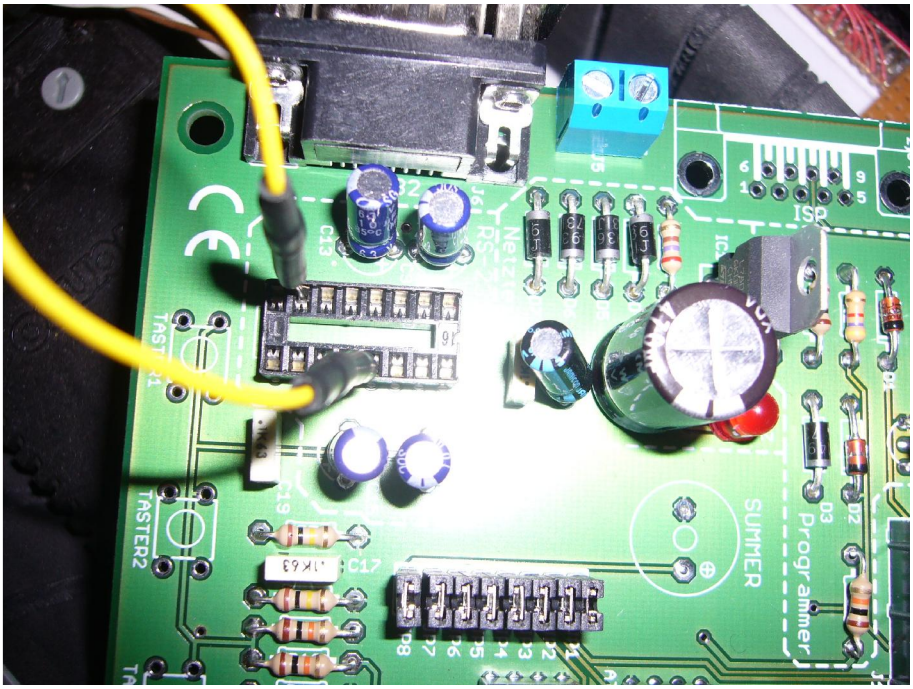
Das, worauf wir sicherlich schon gespannt gewartet haben, ist die Kommunikation zwischen PC und Controller. Die vorbereitenden Arbeiten sind geleistet und es steht eigentlich einer Verbindung nichts mehr im Weg. Außer, das wir noch keinen Controller entsprechend vorbereitet haben. Trotzdem können wir schon einen Versuch durchführen. Dazu brauchen wir nur eine 9 polige Sub-D Buchse. Die Pins 2 und 3 werden einfach kurzgeschlossen. Wer ein Fliegenbein besitzt, hat es ganz einfach.



*RS232 mit Hilfe einer Sub-D Buchse gebrückt*

Dazu steckt man einfach eine 9 pol. Dub-D Buchse auf den USB-Seriell-Wandler und kann so die Pins 2 und 3 (RxD und TxD) leicht verbinden.

Bei einem Pollin-Board braucht nur den Max 232 aus der Fassung entfernt werden und eine Drahtbrücke von 7 nach 13 auf der Fassung stecken. Dann empfängt der PC seine eigene Sendung.




*RS232 auf Fassung gebrückt bei entferntem MAX232 Baustein*

Anschließend wird entweder eine direkte RS232 Verbindung oder eine USB Verbindung mit dem Umsetzer hergestellt..

### *1.4.2.1 Eine Verbindung herstellen*

Wenn wir in unserer Liste der Ports einen Eintrag finden, können wir auf das Button **Verbinden** zugreifen. Mit dem Ereignis **Bt_Connect_Click** prüfen wir zuerst die Überschrift des Button. Steht im Text **Verbinden** ist klar, die Schnittstelle soll geöffnet werden. Deshalb wird diesmal mit einer Fehlerbehandlung in einer selbstgeschriebenen Funktion **Chk_Is_Connect** der Zugriff zum Öffnen der Schnittstelle geprüft.

```
Private Sub Bt_Connect_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Connect.Click
    If Bt_Connect.Text = "Verbinden" Then
      If Chk_Is_Connect() Then
        Bt_Connect.Text = "Trennen"
      End If
    Else
      Bt_Connect.Text = "Verbinden"
      SerialPort1.Close()
    End If
    Chk_Connect_Status()
  End Sub
```

Ist die Antwort positiv wird die Überschrift in **Trennen** geändert. Wird nun das Button erneut betätigt, erkennt die Ereignisroutine, das nun die Schnittstelle geschlossen werden soll. Abschließend wird in der Subroutine **Chk_Connect_Status** geprüft, ob der Zugriff auf die Schnittstellenparameter erlaubt oder gesperrt werden muss. Außerdem ist je nach Status auch das Button **Bt_Test** freigegeben oder blockiert.

Betrachten wir zuerst die Funktion **Chk_Is_Connect**. Sie liefert das Ergebnis Wahr oder Falsch. Untergebracht wird sie im Bereich der Public Functions.

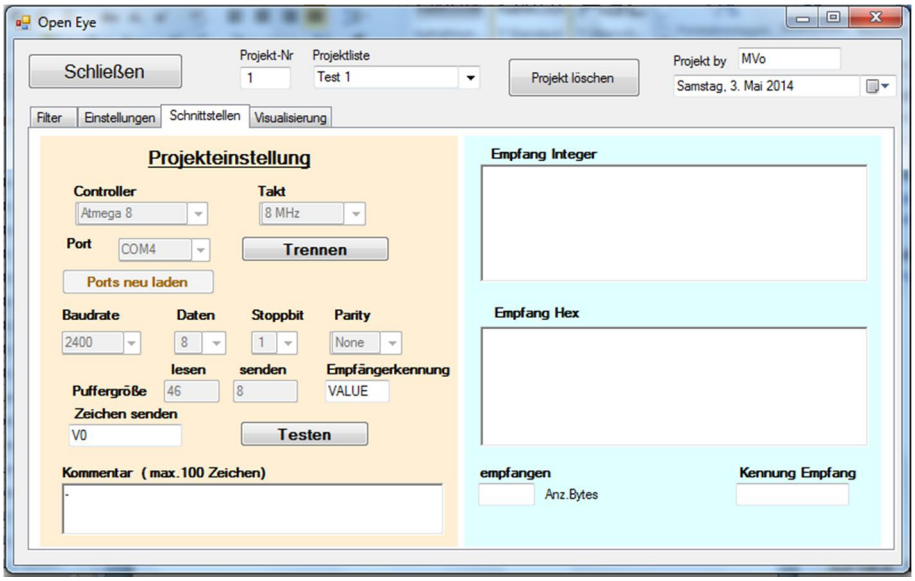
```
Public Function Chk_Is_Connect() As Boolean
    Dim Is_Connect As Boolean
    Is_Connect = True          ' Status vorbesetzen
    Try                         ' Fehlerbehandlung einschalten
      SerialPort1.Open()       ' Port öffnen
    Catch ex As Exception      ' Wenn fehler aufgetreten, Meldung absetzen
      MsgBox("Schnittstelle belegt", MsgBoxStyle.OkOnly, AcceptButton)
      Is_Connect = False       ' und Status auf geschlossen setzen
    Finally                    ' Ende der Bearbeitung Fehler
    End Try                    ' Ende der Fehlerkontrolle
    Return Is_Connect          ' Status zurückliefern
```

```
End Function
```

Ist diese Funktion ohne Fehler durchlaufen, ist auch der Port geöffnet. Nun sind die Objekte der Parametereinstellung zu sperren, da eine geöffnete Schnittstelle nicht geändert werden darf. Es folgt nun die Subroutine **Chk_Connect_Status**

```
Public Sub Chk_Connect_Status()
    Dim is_Connect As Boolean
    is_Connect = SerialPort1.IsOpen
    CB_Baud.Enabled = Not is_Connect
    CB_Controller.Enabled = Not is_Connect
    CB_Daten.Enabled = Not is_Connect
    CB_Port.Enabled = Not is_Connect
    CB_Parity.Enabled = Not is_Connect
    CB_Stop.Enabled = Not is_Connect
    CB_Takt.Enabled = Not is_Connect
    TB_Controller.Enabled = Not is_Connect
    TB_Read.Enabled = Not is_Connect
    TB_Takt.Enabled = Not is_Connect
    TB_Write.Enabled = Not is_Connect
    Bt_Port_New.Enabled = Not is_Connect
    Bt_Test.Enabled = is_Connect
End Sub
```

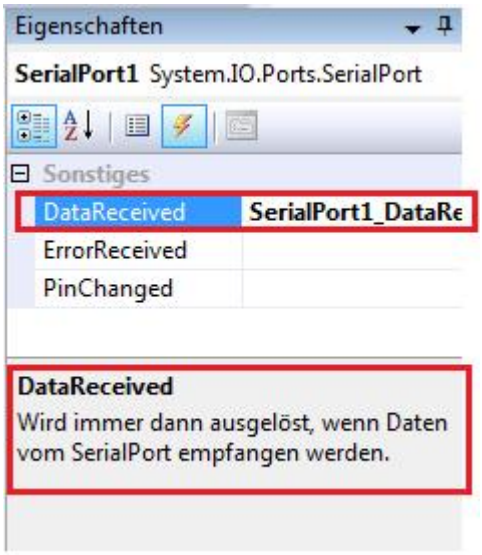
Nach der Installation der Programmblöcke kann die Auswirkung getestet werden. Bei jedem Betätigen des Buton **Verbinden**/ **Trennen** wird die Überschrift und die Freigabe der Eingabeobjekte geändert.




*Port ist verbunden*

### *1.4.2.2 Daten empfangen*

Langsam nähern wir uns unserer ersten Datenübertragung. Zuerst sollten wir uns darauf einrichten, Daten auch zu empfangen. Der Grund wird schnell ersichtlich. Wenn wir mit der beschriebenen Brücke zwischen Pin 2 und 3 bzw. **RxD** und **TxD** auf der seriellen Schnittstelle arbeiten, kommt normalerweise auch der Datenstrom, was immer wir auch senden, wieder zurück. Das ist einfach zu testen, wenn wir uns die Ereignisse vom Objekt **SerialPort1** ansehen.




*Ereignis Datenempfang*

Die Ereignisroutine **DataReceived** trifft mit der untenstehenden Beschreibung genau das, was wir brauchen. Also, einen Doppelklick drauf und schon kann es losgehen.

```
Private Sub SerialPort1_DataReceived(ByVal sender As System.Object, ByVal e As
System.IO.Ports.SerialDataReceivedEventArgs) Handles SerialPort1.DataReceived
    Dim I As Integer            ' Schleifenvariable zur Übernahme empfangener Daten
    Dim Read_Count As Integer     ' Anzahl eingetroffener Daten
    Dim Rec_Data As Byte          ' Einzelner Wert aus Portpuffer
    Read_Count = SerialPort1.BytesToRead
  End Sub
```

Auf die Zeile **End Sub** setzen wir einen Haltepunkt. Nun starten wir einmal das Programm, verbinden den Port und betätigen den Button Testen

Auch wenn es keinen sichtbaren Aufruf für **DataReceived** gibt, das Programm durchläuft diese Routine. Also wurde ein Datentransfer empfangen. Schreiben wir mal in die Textbox **Befehl** einen langen Eintrag und senden ihn ab. Das Programm wird angehalten und **Read_Cnt** liefert eine Zahl. Möglicherweise nur eine 1. Also sind noch nicht alle Daten empfangen und wir führen das Programm mit dem **Startbutton** in der **Menüleiste** fort.

```
Private Sub SerialPort1_DataReceived(ByVal sender As System.Object, ByVal e As System.IO
    Dim I As Integer                ' Schleifenvariable zur Übernahme empfangener Daten
    Dim Read_Count As Integer       ' Anzahl eingetroffener Daten
    Dim Rec_Data As Byte            ' Einzelner Wert aus Portpuffer
    Read_Count = SerialPort1.BytesToRead
End Sub
```

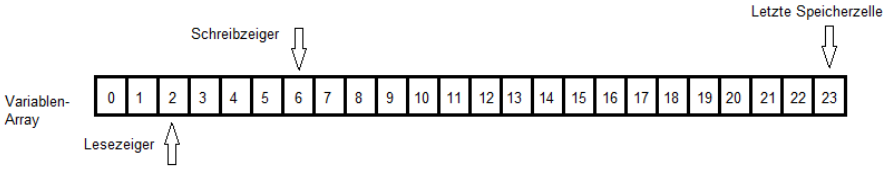
*Haltepunkt Read_Count*

Nun wird wieder angehalten und möglicherweise nun der Rest der Daten angezeigt. Vielleicht aber auch nur wieder ein Teil. Trotzdem, erst, wenn alle gesendeten Daten auch wieder Empfangen sind landet das Programm nicht mehr beim Haltepunkt. Es passiert folgendes:

Ein Empfang wird erkannt und in den internen Puffer der Schnittstelle gepackt, den wir übrigends in der Größe über **Tb_Read** und **Tb_Write** definieren. Manchmal ist das Ereignis so schnell, das gerade ein Zeichen eingetroffen ist, Beim nächsten Durchlauf werden dann die anderen Zeichen im internen Puffer gemeldet. Nun ist klar, wofür die Variable **Read_Count** benötigt wird und auch die Schleifenvariable i ist klar. Die Variable **Rec_Data** ist nun dazu da, die Werte aus dem Portpuffer zu holen und unserem Programm mitzuteilen. Doch das erweist sich etwas schwierig. Die Schnittstelle läßt es nicht zu, das die Daten direkt die **RichTextBoxen** im Programm übertragen werden. Also bauen wir uns einen Ringpuffer, der erst einmal alle Daten einsammelt.

### *1.4.2.3 Der Ringpuffer*

Das Prinzip eines Ringpuffers ist ein Speicher mit einer vorgegebenen Anzahl Speicherzellen. Diese werden nach und nach mit Werten belegt. Wird die letzte Speicherzelle erreicht, geht man davon aus, das die Werte in den ersten Speicherzellen bereits verarbeitet sind und überschreibt die Werte mit den neuen Informationen, indem die zeiger wieder den Anfangswert des Speichers zugewiesen bekommen.




*Schema Ringpuffer*

So werden die Daten für eine kurze Zeit gehalten. Der Speicherbedarf ergibt sich daraus, wie schnell die Daten ankommen und welcher Aufwand für die Verarbeitung erforderlich ist. So schätzt man ab, ob 2, 3 oder gar 5 Datenpakete abgelegt werden müssen, bevor der Speicheranfang wieder überschrieben werden kann. Da auch ein gut ausgerüsteter PC Speichergrenzen hat, darf das natürlich nicht ewig so weitergehen. Aber, die zuerst eingetroffenen Daten sollten längst verarbeitet sein, also kann dieser Speicherbereich wieder genutzt werden.

Deklarieren wir am Anfang unseres Programmes eine Variable mit der Aufnahmekapazität von 1000 Bytes. Dazu zwei Zeigervariablen **Read_pointer** und **Write_Pointer**.

```
Public Rec_Feld(999) As Byte
Public Read_Pointer As Integer
Public Write_Pointer As Integer
```

Beim Start des Programms werden beide Pointer- Variablen auf 0 gesetzt.

Nun läuft folgendes ab:

Kommt ein Datenpaket an und wird in **SerialPort1_DataReceived** detektiert, stoßen wir eine Schleife an, die nun genau so viele Zeichen aus dem Portpuffer holt, wie eingetroffen sind und in den Ringpuffer überträgt. Nach jedem Übertrag wird der Schreibzeiger **Write_Pointer** um

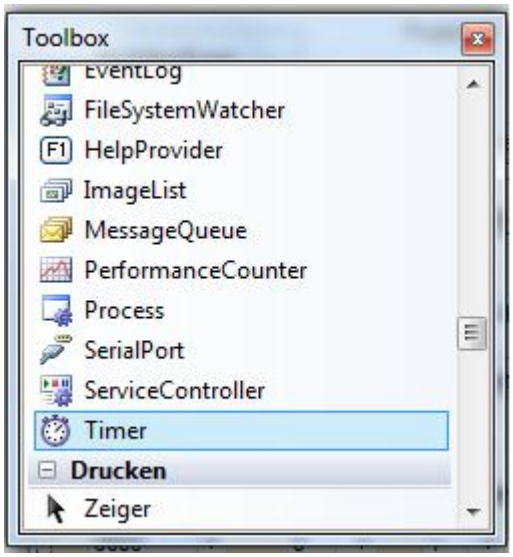
1 erhöht und die nächste Speicherzelle im Ringpuffer adressiert. Erreicht der Schreibzeiger das Ende des Ringpuffers, wird er wieder auf 0 gesetzt. Das ist nun die Aufgabe, die in der Ereignisroutine eingebaut wird.

```
Private Sub SerialPort1_DataReceived(ByVal sender As System.Object, ByVal e As
System.IO.Ports.SerialDataReceivedEventArgs) Handles SerialPort1.DataReceived
    Dim I As Integer              ' Schleifenvariable zur Übernahme empfangener Daten
    Dim Read_Count As Integer     ' Anzahl eingetroffener Daten
    Dim Rec_Data As Byte          ' Einzelner Wert aus Portpuffer
    Read_Count = SerialPort1.BytesToRead
    For I = 0 To Read_Count - 1
      Rec_Data = SerialPort1.ReadByte                 '  Byte vom Portpuffer holen
      Rec_Feld(Write_Pointer) = Rec_Data             ' mit Schreibzeiger adressieren
      Write_Pointer = Write_Pointer + 1              ' Schreibzeiger erhöhen
      If Write_Pointer > 999 Then Write_Pointer = 0 ' Ende Puffer, beginn von vorn
    Next
  End Sub
```

Beim Abholen der Daten von der Schnittstelle wird dort der belegte Speicher wieder freigegeben.

### *1.4.2.4 Ringpuffer auslesen*

Da wir die Daten nicht direkt in unser Programm mit dem Ereignis der Schnittstelle übertragen können, muss nun ein anderes Ereignis die Daten in die beiden **RichTextBoxen** übertragen. Ein Button fällt aus, da wir ja nicht wissen, wann ein Datenpaket angekommen ist und dauernd auf ein Button klicken kann auch nicht gewollt sein. Ein anderes Ereignis der bisher verwendeten Objekte kommt auch nicht in Frage. Aber es muss etwas geben, was zumindest in einem akzeptablen Zeitfenster den Ringpuffer auf neue Daten überprüft und die Daten zur Auswertung abholt. Zeit ist das Stichwort. Was bietet die **Toolbox**? Werfen wir doch mal einen Blick hinein und suchen ein Objekt, das mit Zeit in Verbindung gebracht werden kann. So werden wir in der **Toolbox** denn auch fündig.

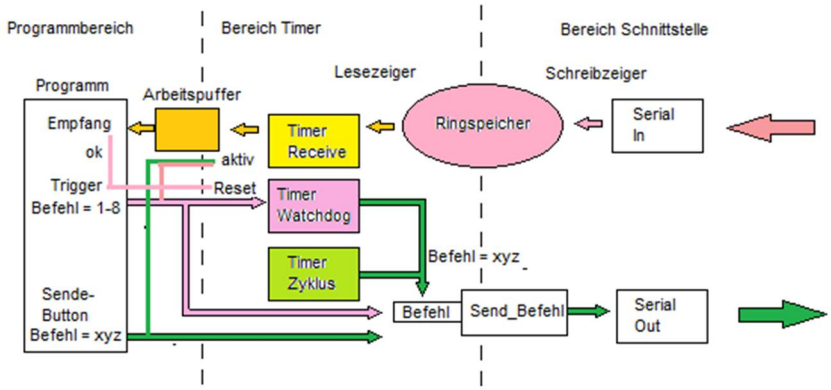



*Timer*

Der Hilfstext, den wir erhalten, wenn wir mit dem Mauszeiger über den Eintrag gehen liefert die Aussage User defiertes Zeit –Event. Ja, das probieren wir. Es reicht ja, alle Sekunden mal zum Ringpuffer schauen und zu prüfen, ob Schreib- und Lesezeiger unterschiedlich sind. So erkennen wir den Dateneingang.

### *1.4.2.5 Der Timer*

Ziehen wir uns den **Timer** nun auf unsere Anwendung. Der **Timer** ist ein Objekt, das in gewünschten Intervallen den Programmablauf unterbricht und vorgegebene Aufgaben erledigt. Solche Objekte sind immer dann interessant, wenn es um Ereignisse in verschiedenen Prozessen gibt die Daten miteinander austauschen müssen So einen Fall haben wir mit dem **SerialPort** vorliegen. Dieses Objekt arbeitet in einem eigenen Prozess und der direkte Weg, Daten an Objekte in unserer Applikation beim Eintreffen direkt weiterzuleiten ist nicht möglich. Daher nutzen wir den Ringpuffer als Bindeglied zwischen **SerialPort** und unserem Programm. Vielleicht verschafft hier eine kleine Skizze einen Überblick.




*Funktion Ringpuffer*

In der **Timer** Ereignisroutine werden Lese und Schreibzähler verglichen und bei Ungleichheit wird der Lesezeiger dem Schreibzeiger nachgeführt. Dabei werden die Daten aus der adressierten Speicherzelle gelesen und in die **RichTextBoxen** übertragen. Es reicht, wenn dieser Vorgang einmal in der Sekunde ausgeführt wird.

```
Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Timer1.Tick
    Dim AnzByte As Integer
    AnzByte = Write_Pointer - Read_Pointer
    While Write_Pointer <> Read_Pointer
        RT_Integer.Text = RT_Integer.Text + Str(Rec_Feld(Read_Pointer)) + ";"
        RT_Hex.Text = RT_Hex.Text + IntToHex(Rec_Feld(Read_Pointer), "Byte") + ";"
        Read_Pointer = Read_Pointer + 1
        If Read_Pointer > 999 Then Read_Pointer = 0
    End While
```

```
End Sub
```

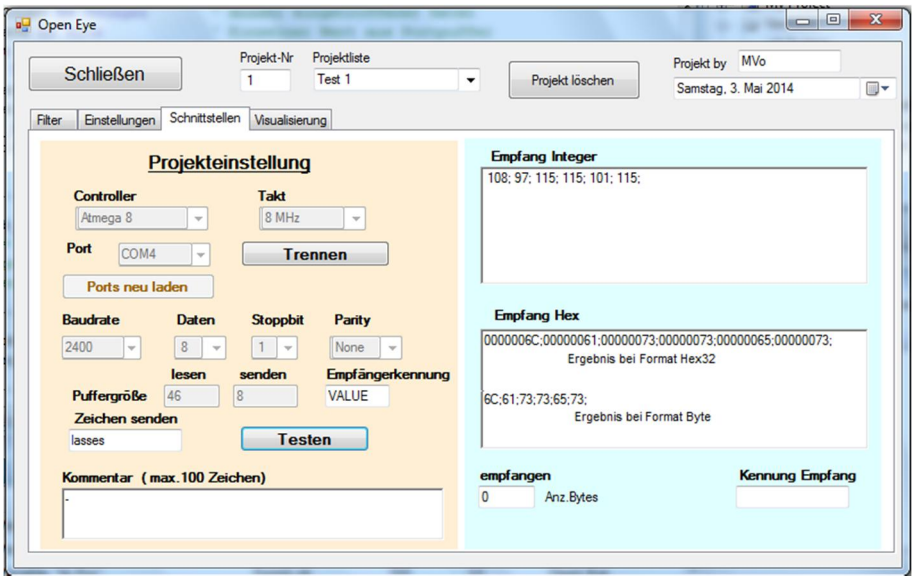
Diesmal setzen wir eine **While** Schleife ein. Solange Schreibzeiger und Lesezeiger unterschiedlich sind wird die Schleife durchlaufen, die Daten herauskopiert, im Format angepasst und an die **RichtextBoxen RT_Integer** sowie **Rt_Hex** und übertragen. Der Schreibzeiger wird dabei nachgeführt. Sehen wir uns das Ergebnis einmal an und testen diesen Progeammabschnitt. Diesmal sind auch lange und kurze Befehle im Ergebnis zu sehen. Obwohl normal das Format **Byte** ist, kann auch ruhig mal **Hex16** oder **Hex32** ausprobiert werden.

```
RT_Hex.Text = RT_Hex.Text + IntToHex(Rec_Feld(Read_Pointer), "Hex16") + ";"
```

Oder

```
RT_Hex.Text = RT_Hex.Text + IntToHex(Rec_Feld(Read_Pointer), "Hex32") + ";"
```

Mit dem Ergebnis können wir zufrieden sein.

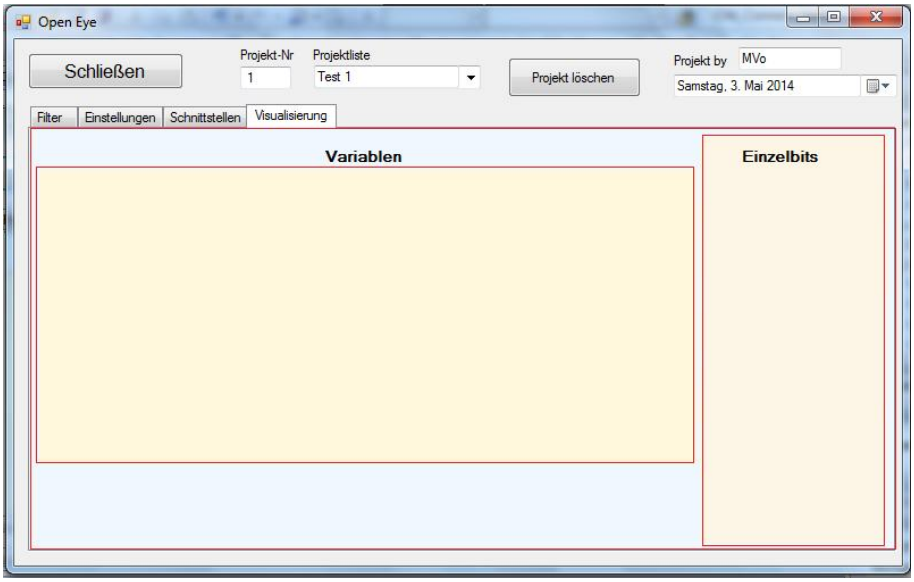



*Ergebnis Datenempfang*

## 1.5 Anzeigen der Controllerdaten

Auf Seite Einstellungen haben wir ein Button **Anzeige generieren** installiert und bisher nicht weiter beachtet. Nun werden wir ihm einen Sinn geben. Damit wir uns diesem Button zuwenden können, muss die Seite **Visualisierung** vorbereitet werden. Bis jetzt ist noch kein einziges Objekt auf dieser Seite angebracht. Das werden wir nun ändern. Durch anklicken des Tabulators bekommen wir die Seite aufgeschlagen. Jetzt überlegen wir, welche Objekte aus der Toolbox notwendig sind.

Verteilen wir erst einmal ein paar Panel auf der Seite und fügen die Überschriften ein.

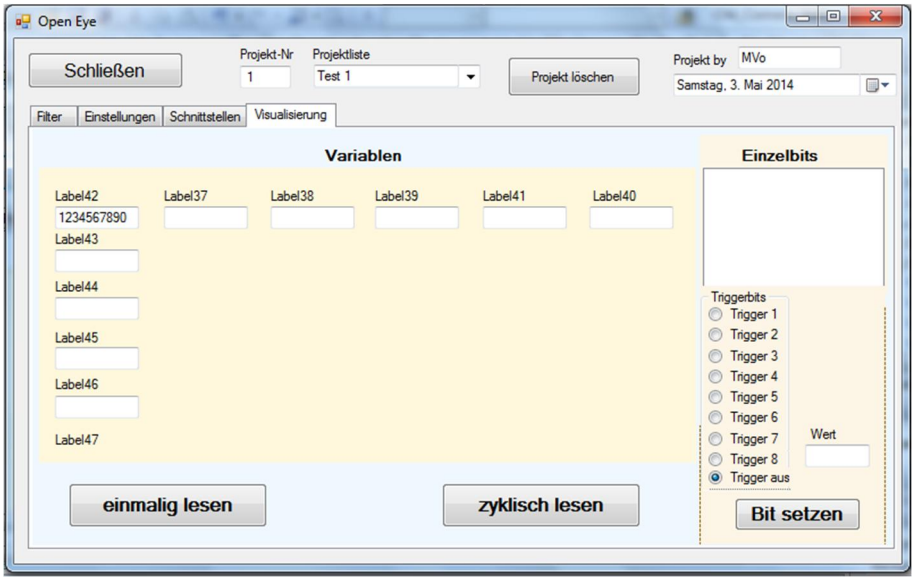



*Visualisierung vorbereiten*

Die drei Panels sind mit ihrer Eigenschaft **Backcolor** etwas eingefärbt und hier mit roten Rahmen etwas deutlicher gekennzeichnet. Hellblau ist die Seite **Tp_Visu**. Der Bereich mit der Überschrift **Variablen** ist der Platz zur **Werteanzeige** und ein Panel mit Namen **Pn_Variablen** und ganz rechts ist der Platz für die **Einzelbitanzeige** und die **Triggervorgabe**. Dieses Panel bekommt den Namen **Pn_Trigger.**

Die für diese Seite erforderlichen Labels werden im Einzelnen nicht beschrieben. Es ist aus den Screenshots zu sehen, mit welchem Text sie parametriert werden. Namen sind nicht erforderlich.

Im unteren blauen Bereich kommen noch zwei Button, um einmalig Daten zu holen oder über einen Timer eine zyklische Leseanforderung zu senden. Die Namen **Bt_Single** und **Bt_Zyklus** sollten die Funktion treffend bezeichnen. Um den Platzbedarf auszuloten, fügen wir in den Variablenbereich **Textboxen** ein. Dabei beschreiben wir eine **Textbox** mit 10 Ziffern und gestalten sie in der Größe so, das 6 in der Reihe und fünf untereinander passen. So haben wir exakt 30 Variablen direkt im Blick.




*Anzeigefeld einrichten*

Auf die Seite Einzelbit kommt eine **CheckedListbox**, da die einzelnen Bits mit ihrer Beschreibung so am besten angezeigt werden. Diese Listenart liefert **Checkboxen** als Einträge und zeigt einen booleschen Status zu den Einträgen. Der Name wird mit **Clb_Bit_Info** festgelegt. Darunter fügen wir eine **Groupbox** auf das Panel für **Radioboxen** und vergeben den Namen **Gb_Trigger**. Der Vorteil einer **Groupbox** ist eine Verbindung der darauf befindlichen **Radiobutton**. Es kann immer nur eines betätigt sein. Die Radiobuttons werden Rb_Bit_n benannt, wobei n die Bitnummer von 0 bis 7 ist. Die letzte Checkbox wird mit **Cb_TriggerOff** benannt und bekommt die Überschrift **Trigger aus**. Dieses Radiobutton wird in der Sub Routine **Set_Default** auf **Checked =True** gesetzt. Dies ist erforderlich, da sonst das eine Checkbox möglicherweise nicht reagiert, wenn **alle** mit dem Status **Checked =False** definiert sind. Natürlich könnte man auch Checkboxen einsetzen und den **Triggerbits** verschiedene Farben zuordnen. Aber diese Abwandlung

überlasse ich euch. Sicherlich werdet ihr nach der Fertigstellung von **Open_Eye** in der Lage sein, ein paar persönliche Anpassungen vorzunehmen.

Eine Textbox wird mit dem Wert besetzt, der durch die Radiobuttons erzeugt wird. Hier vergebe ich den Namen **Tb_Wert**. Das Button **Bit setzen** bekommt später die gleiche Funktion wie das Button **Testen** der Seite Schnittstellen, wobei hier die Sendedaten festgeschrieben werden. Das erste Zeichen ist **V** für Value**.** Darauf folgt der Bytewert aus der Textbox **TB_Wert**, der das Triggerbit Trigger_In im Controller setzt. Der Name wird mit **Bt_Set_Trigger** besetzt.

Sehen wir uns nun die weitere Gestaltung der vierten Seite an. Durch die Ansicht mit den provisorischen Textboxen kann ermittelt werden, wie viele Ausgabeelemente auf dem Panel **Pn_Variablen** im sichtbaren Bereich angezeigt werden können. Auch der Abstand untereinander wird so erst einmal geprüft und in etwa eingestellt. Dabei erhalten wir wichtige Informationen. Einmal ist es die Größe der Textbox, die in der Eigenschaft **Size** hinterlegt ist. Eine weitere wichtige Information betrifft den Abstand der Textboxen zueinander. Diese Information ist in der Eigenschaft **Location** enthalten und bezieht sich auf die Position auf dem Panel. Mit diesen Werten lässt sich die Position der Anzeigeelemente berechnen.

Die eingegebenen zehn Ziffern lassen sich in einer Textbox mit einer Länge von **74** Pixel darstellen. Bei einer Schrittweite von Textbox zu Textbox horizontal mit **94** Pixel und einem vertikalen Abstand von **50** Pixel ergibt sich eine gute Verteilung.

### *1.5.1 Ein Objekt erstellen*

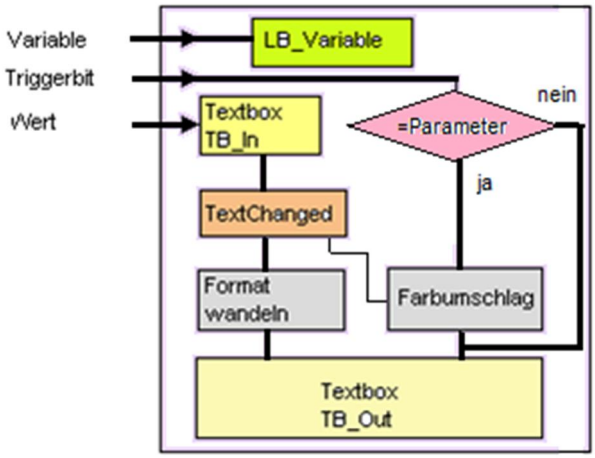
Wir kommen nun zu einem ganz besonderen Teil der Programmierung. Wir haben Werte mit unterschiedlichen Formaten darzustellen und obendrein soll die Textbox auch farblich markiert werden, wenn der Wert mit einem Trigger erzeugt wurde. Die Textboxen müssen wir zur Laufzeit erstellen und auf dem Panel verteilen, aber wie weisen wir nun diesen Textboxen die Werte zu. Schließlich können wir nicht ständig immer wieder neu die Textboxen anlegen. Außerdem ist die Darstellungsart abhängig vom Datentyp. Das bedeutet, die Werte müssen unterschiedlich übertragen werden. Wenn wir aber Eigenschaften von Objekten nutzen könnten und sie sich selbst die Datenstruktur zurechtbasteln, brauchen wir lediglich die Werte übergeben. Bringen wir erst einmal ein wenig Übersicht in die vielen offenen Punkte und schreiben auf eine Liste, was wir von der Anzeige erwarten.

Darstellung Integerwerte von 8 Bit, 16 Bit und 32 Bit
Darstellung Hexadezimal von 8 Bit, 16 Bit und 32 Bit
Darstellung von Byte 8 Bit binär in 0 und 1
Darstellung ASCII-Code 1 Byte
Signalisieren , wenn Anzeige ein Triggerereignis ist
Anzeigen, welche Werte zu dem Triggerereignis gehören.

Wie kann eine solche Übergabe an Objekte erfolgen, die gar nicht existieren. Nun wir bauen uns ein passendes eigenes Objekt, welches in der Lage ist, Formate zu erkennen und die Datenaufbereitung entsprechend vornimmt. Die Werte übergebe ich einfach in einem String dem Objekt und teile nur mit, welche Formatdarstellung ich erwarte. Klar ist, dass es Textboxen sein werden und klar ist auch, dass die Information Wert und Trigger übergeben werden muss. Das Objekt selbst bekommt bei der Erstellung das Format aufgedrückt. Nun muss aber so ein Objekt bei der Werteübergabe ein Ereignis auslösen, um die Darstellung zu aktualisieren. Textboxen können auf das Ereignis **Textchanged** reagieren. Das ist möglicherweise ein Ansatz, um beliebige Werte anzuzeigen. Allerdings nicht in der gleichen Textbox, denn das würde erneut ein Ereignis bedeuten. Somit braucht das Objekt außer ein paar **Parameter** zwei **Textboxen**. Damit es sich vernünftig darstellt, packen wir diese beiden Textboxen zusammen mit einer Überschrift auf ein **Panel**. Die Zusammenstellung erfolgt in einem neuen **UserControlObject**

Dieses Objekt soll in der Lage sein, Daten in gewünschte Formate zu wandeln und auszugeben. Es soll kontrollieren, ob die Daten durch ein

Triggerereignis empfangen wurden oder ob es ein normaler Datenempfang war. Immer bei solchen Gedankengängen hilft eine Skizze die unklaren Formulierungen zu ordnen. Es muss nicht perfekt sein, aber jede Skizzierung einer Aufgabe hilft die geeigneten Werkzeuge auszuwählen.



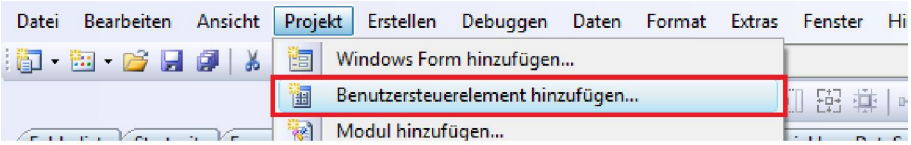

*Struktur Anzeigeobjekt*

Das Label bekommt den Namen der Variablen schon bei der Erstellung. Wenn nach einem Datenempfang die Werte in der Textbox Tb_In abgelegt werden, soll der Trigger, sofern er mit dem internen Parameter übereinstimmt, einen Farbumschlag der Textbox TB_Out veranlassen. Der Wert wird in das hinterlegte Format gewandelt und ausgegeben. Das Ganze wird durch das Ereignis TextChanged der Textbox Tb_In ausgelöst.

Die Feinheiten sind noch einzubringen, doch ist die Funktion des UserControl-Objektes mit dieser Skizze grob beschrieben.

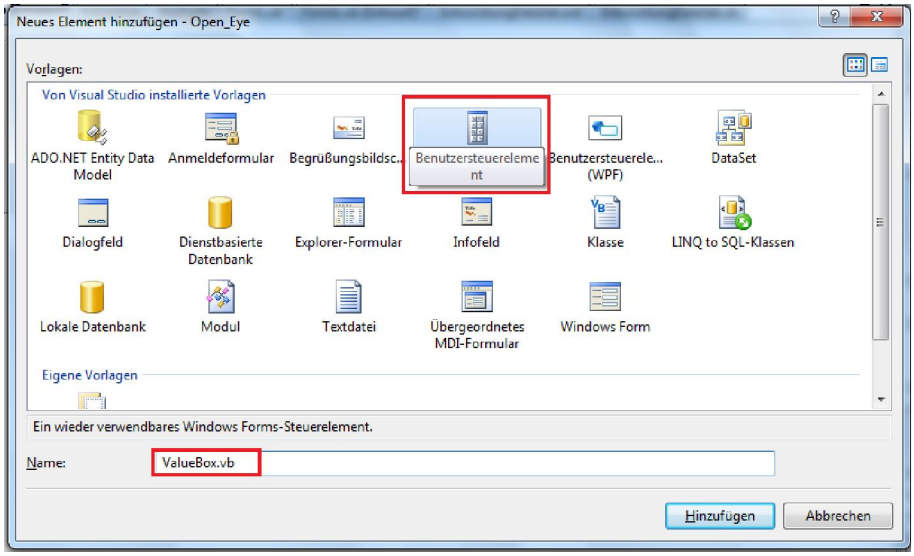
### *1.5.2 Der Aufbau des Anzeigeobjektes*

Über die Menüleiste **Projekt** gelangen wir zum Untermenü **Benutzersteuerelement hinzufügen**.



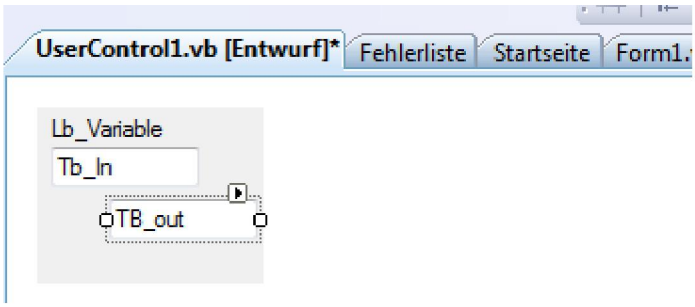

*Benutzersteuerelement hinzufügen*

Im folgenden Fenster wählen wir das **Benutzersteuerelement** aus und vergeben den Namen **ValueBox**.



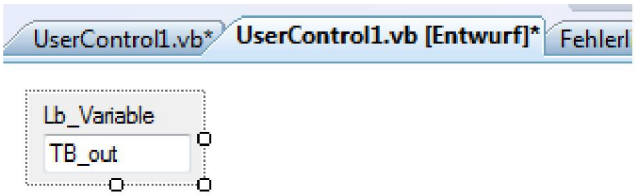

*Auswahl Benutzersteuerelement*

Es öffnet sich die Entwurfsansicht vom **UserControl**. Hier ziehen wir nun zwei Textboxen  und ein Label auf die Oberfläche, Das Label bekommt hier den Namen **Lb_Variable**, die erste Textbox den Namen **Tb_In**, die zweite **Tb_Out**. Die Größe der Textboxen stellen wir auf **Size 74:20** ein.



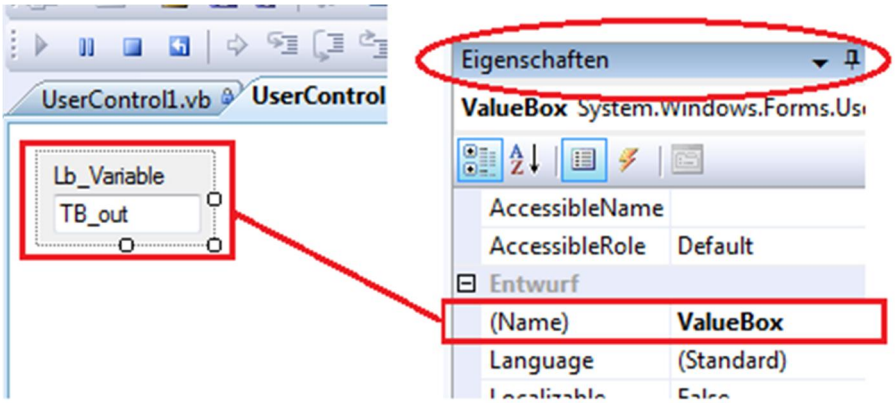

*Aufbau Anzeigeobjekt*

Die Textfelder werden nun übereinandergelegt. Tb_In wird unsichtbar geschaltet und so bleibt Tb_Out sichtbar. Dem Label muss hier ein Name vergeben werden, da es später den Variablennamen zugewiesen bekommt.




*Anzeigen zusammenstellen*

Schließlich wird dem gestalteten Objekt auch noch der Name **ValueBox** gegeben.




*Namen für Anzeigeobjekt vergeben*

Das gesamte Objekt bekommt jetzt noch eine Größenzuweisung in Size für Breite **86** (Width) und für Höhe **44** (height).

### *1.5.3 Die Aufgabe festlegen*

Die Textbox **Tb_In** bekommt nun ein Ereignis **TextChanged** zugewiesen. Dazu wird im Eigenschaftsfenster in der Rubrik Ereignisse das Ereignis doppelt angeklickt. Die Ansicht wechselt in den **Befehlseditor**.

```
Public Class UserControl1
   Public Format As String
   Public Triggerlevel As Byte
   Public Triggerwert As Byte
   Private Sub TB_In_TextChanged(ByVal sender As System.Object,
                  ByVal e As      System.EventArgs) Handles TB_In.TextChanged
   Dim Zahlenwert as Int64

   End Sub
End Class354
```

Bevor wir nun an das Ereignis der Textbox gehen, deklarieren wir für dieses Objekt ein paar Parameter. **Format** wird bei der Generierung besetzt. Das ist ja für die betreffende Variable bekannt und braucht nicht jedes Mal mit übergeben werden.

Die Variable **Trigger** wird zweimal. benötigt. **Triggerlevel** ist Bestandteil der Variableneinstellung, **Triggerwert** wird dem Objekt mit dem Wertesatz übergeben. In der Ereignisroutine benötige ich noch eine Variable für den **Zahlenwert**, der in das gewünschte Format gewandelt wird.

Nun wie geht es weiter. Der Wertesatz soll der Textbox **Tb_In** übergeben werden, damit diese das Ereignis **TextChanged** auslöst. Eine Textbox ist für Text und deshalb wird ihr auch ein String übergeben. Nur, wie bekomme ich bis zu vier Werte in diese Textbox, ohne andere zu überschreiben? Ich wende da einen kleinen Trick an. Das eigentliche Vorgehen entstammt dem Filter. Dort haben wir im Text nach Zeichen gesucht und entsprechende Texte herausgefiltert. Das setze ich hier auch ein. Der Übergabestring kann bis zu vier Byte beinhalten, getrennt durch ein Semikolon. Diesen String müssen wir zuerst in eine Zahl wandeln. Damit der Programmcode übersichtlich bleibt, schreibe ich eine separate Funktion, die den String bekommt und in einer Zahl zurückliefert.

```
Public Function Get_Wert(ByVal Eingang As String) As Integer
      Dim MarkCnt As Integer
      Dim MarkPos As Integer
      Dim CalcWert As Integer
      Dim RefText As String
```

```
    Dim WorkText As String
    WorkText = Eingang                    ' Zuerst Arbeitsstring besetzen
    CalcWert = 0                           ' und Defaultwerte setzen
    MarkCnt = 0
    MarkPos = InStr(WorkText, ";")        ' Semikolon im String?
    If MarkPos = 0 Then
       CalcWert = Val(WorkText)             ' ein Byte gleich Zahl
    Else
       RefText = WorkText                 ' ein Byte gleich Zahl
       While MarkPos>0 then
          RefText = Mid(WorkText, 1, MarkPos - 1)     ' Byte herauslösen
          WorkText = Mid(WorkText, MarkPos + 1, Len(WorkText) - MarkPos)
          ' Arbeitstext kürzen
          CalcWert = CalcWert + (Val(RefText) * 256 ^ MarkCnt) ' Zahlenwert aufaddieren
          MarkCnt = MarkCnt + 1                        ' Exponenten zähler erhöhen
          MarkPos = InStr(WorkText, ";")               ' Semikolon im String?
          If MarkPos = 0 Then  CalcWert = Val(WorkText)        ' letztes Byte
       End While
       CalcWert = CalcWert + (Val(RefText) * 256 ^ MarkCnt)     ' Letzes Byte aufaddieren
    End If
    Return (CalcWert)
  End Function
```

Zu dieser Routine eine kleine Beschreibung. Beginnen wir mit einem einzigen Byte. Im Übergabestring ist kein Semikolon und die **While** Schleife wird nicht ausgeführt. Der **CalcWert** wird gleich mit dem Zahlenwert im String besetzt.

Angenommen, es werden in einem String vier Bytes übergeben. Wichtig, **kein** abschließendes Semikolon.

<21;213;1;65>

Das niederwertige Byte muss dabei vorn stehen. Zuerst wird die 21 gefunden und mit 256^0 multipliziert. Dazu wieder einmal der einfache Satz aus der Mathematik: **Alle Zahlen ^0 ergeben 1**. Selbst die Zahl 0^0 ist 1. Glaubt ihr nicht? Probiert es aus, Taschenrechner mit wissenschaftlicher Ansicht, 0 eingeben Funktion x^y anwählen und 0 eingeben. Alles klar? Gut, dann kommt der nächste Schritt. Ich setze den Exponentenzähler eins hoch und addiere zum ersten Ergebnis den zweiten Stringteil hinzu, also 213*256^1. Der nächste Schritt ist mit dem Exponentenwert 2, also 1*256^2 und schließlich die letzte Zahl, das

höchstwertige Byte wird mit 65*256^3 berechnet und hinzuaddiert. Der Rückgabewert dieser Funktion entspricht dann der Zahl **1090639125.**

Den Aufruf **Get_Wert** setzen wir zuerst in die Ereignisroutine **Tb_In_TextChanged**.

```
Zahlenwert = Get_Wert(TB_In.Text)
```

Nun kommt die Zuweisung an die Textbox **Tb_Out**, denn das ist, was uns interessiert.

Beginnen wir mit dem Format **ASCII**

```
If (Format = ASCII) Then
        TB_Out.Text = Chr(Zahlenwert)
 End If
```

Das war einfach. Nehmen wir uns nun die **Int-Formate** vor.

```
If (Format = "Int8") Or (Format = "Int16") Or (Format = "Int32")
Then
    TB_Out.Text = Str(Zahlenwert)
End If
```

Das war auch nicht besonders schwer. Da wir uns mit dem hexadezimalen Format bereits beschäftigt haben, werde ich auf eine weitere Erklärung verzichten. Wir rufen einfach die Funktion unter angebe der Parameter Wert und Format auf, die wir bereits erstellt haben. Allerdings werden wir feststellen, dass diese **Function** nicht bekannt ist. Dabei haben wir sie doch **Public** angelegt. Warum geht es dann nicht? Nun, die Funktion gehört zu einem Objekt, welches zu diesem Zeitpunkt gar nicht bekannt ist. Unser Objekt **ValueBox** ist ja auch noch gar nicht in **Open_Eye** installiert. Nun, unter Angabe der Applikation **Frm_Open_Eye** und einem Punkt lässt sich die Funktion doch noch aufrufen.

```
If (Format = "Hex8") Or (Format = "Hex16") Or (Format = "Hex32") Then
        TB_Out.Text =   Frm_Open_Eye.IntToHex(Zahlenwert, Format)
End If
```

Bleibt die Darstellung **Byte**. Hier möchte ich das Bitmuster des Wertes sehen. Eine Wandlung existiert noch nicht, also ist es wieder an uns, eine

Funktion zu schreiben, die eine Zahl bekommt und einen String zurückliefert. Hier ist es auch ganz wichtig, dass alle acht Bits abgebildet sind. Ein kleiner Tipp zur Vorgehensweise. Binäre Darstellung hat was mit Basis 2 so wie die Dezimale Darstellung mit Basis 10 und die hexadezimale Darstellung mit Basis 16 zu tun. Ein Blick in die Funktion **IntToHex** bringt uns vielleicht die Erleuchtung. Dort haben wir eine Zahl ganzzahlig durch 16 geteilt und der Rest war immer der Wert der Stelle. Ist das auch bei binären Zahlen anwendbar, mit der Teilung 2? Ja natürlich. Auch im Dezimalsystem ist das Anwendbar und wir machen es schon unbewusst wenn wir eine Zahl **dreihundert** und **vier** und **fünfzig** nennen.

Und weil es ja fast passend ist kopieren wir die Routine **IntToHex** und nennen die Kopie **IntToBin**. Anschließend entfernen wir die unnützen Auswertungen, da nur numerische Ziffern zu erwarten sind. Auch das **Format** braucht nicht mit übergeben zu werden, da ein **Byte** immer die Länge 8 hat und die können wir fest eintragen. So entfällt auch die Variable **Laenge**.

```
Public Function IntToBin(ByVal Wert As Integer) As String
    Dim Bin_Zahl As String
    Dim Zahl As Integer
    Dim Temp As Integer
    Dim Rest As Integer
    Dim i As Integer
    Bin_Zahl = ""                    ' Bin_Zahl leeren
    Zahl = Wert                      ' Wert nach Zahl übertragen
    While Zahl > 0                    ' verbleibenden Zahlenwert prüfen
      Temp = Math.Truncate(Zahl / 2) ' Zwischenwert bilden
      Rest = Zahl - (Temp * 2)       ' Rest aus Division
      CharCode = Asc("0") + Rest
      Bin_Zahl = Chr(CharCode) + Bin_Zahl ' Bin_Zahl zusammenstellen
      Trim(Bin_Zahl)                  ' von hinten nach vorn
      Zahl = Temp                     ' Zahl für nächsten Durchlauf setzen
    End While
    While Len(Bin_Zahl) < 8
      Bin_Zahl = "0" + Bin_Zahl       ' Bin_Zahl mit 0 auffüllen
    End While
    Return (Bin_Zahl)
  End Function
```

Damit ist die Abfrage auf **Byte-Format** auch erledigt. Aber daran denken, die Wandlung ist im Editor von **Open_Eye** einzubringen und hier wieder mit der Angabe **Frm_Open_Eye** aufzurufen.

```
If (Format = "Byte") Then
        TB_Out.Text = Frm_Open_Eye.IntToBin(Zahlenwert)
End If
```

Bleibt noch die Einfärbung der Textboxen, wenn ein Triggerereignis die Daten geliefert hat und die Variable diesem Trigger zugeordnet ist.

```
If Triggerlevel = Triggerwert Then
        TB_Out.BackColor = Color.Pink
Else
        TB_Out.BackColor = TB_In.BackColor
End If
```

Zugegeben, über **Color.Pink** kann man streiten, doch finde ich den leicht rötlichen Farbton passend zum Triggerereignis.

Abschließend die Ereignisroutine im Ganzen

```
Private Sub TB_In_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TB_In.TextChanged
   Dim Zahlenwert As Int64
   If Tb_In.Text<>"" then
      Zahlenwert = Get_Wert(TB_In.Text)
      If (Format = "Hex8") Or (Format = "Hex16") Or (Format = "Hex32") Then
         TB_Out.Text = Frm_Open_Eye.IntToHex(Zahlenwert, Format)
      End If
      If (Format = "Byte") Then
         TB_Out.Text = Frm_Open_Eye.IntToBin(Zahlenwert)
      End If
      If (Format = "Int8") Or (Format = "Int16") Or (Format = "Int32") Then
         TB_Out.Text = Str(Zahlenwert)
      End If
      If (Format = "ASCII") Then
         TB_Out.Text = Chr(Zahlenwert)
      End If
      If Trigger = Is_Trigger Then
         TB_Out.BackColor = Color.Pink
      Else
         TB_Out.BackColor = TB_In.BackColor
```

```
    End If
  End If
End Sub
```

Ab jetzt besitzen wir ein spezielles Objekt, das wie eine Textbox beliebig oft in unsere Applikation eingebunden werden kann. Die Abfrage `If TB_In.Text<>""` ist wichtig. Es ist nicht immer gesagt, das sich die Daten ändern und wenn eine Markierung durch einen Trigger abgelegt ist, bleibt er möglicherweise längere Zeit stehen, obwohl kein Triggerereignis eingetroffen ist. Deshalb wird TB_In erst einmal mit einem Leerstring beschrieben. Klar, das ist keine Zahl und so würde die Wandlung zu einer Zahl auch gleich fehlschlagen. Eine 0 ist aber auch nicht sinnvoll, denn niemand gibt eine Garantie, das nicht auch ein 0 Wert schon längere Zeit auf der Variablen abgelegt ist. Der Leerstring aber erzwingt einen Wechsel und somit auch die Aktualisierung von Tb_Out.

### *1.5.4 Einbau einer Valuebox*

Wenn wir unseren Entwurf auf der Seite vier **Visualisierung** noch nicht bereinigt haben, müssen wir nun alle eingebauten Textboxen auf dem Panel **Pn_Variablen** entfernen. Dann wechseln wir zur Seite Einstellungen und geben einen Doppelclick auf das Button **Anzeige generieren**

Der Rahmen der Ereignismethode **Bt_Gen_Anzeige_Click** wird bereitgestellt. Nun wird es spannend. Beginnen wir ganz langsam mit ein paar Versuchen. Zuerst schreiben wir eine kleine Subroutine, die eine **Valuebox** parametrieren und auf unserem Panel darstellen kann. Zur Übergabe der Parameter verwenden wir wie bereits bei den Projektdaten an die Datenbank eine Datenstruktur. Was ist erforderlich, für eine Subroutine, die die Aufgabe erledigen soll, genau ein einziges Objekt auf einem Panel zu installieren.

Da ist der **Variablenname** für die Überschrift und zur Identifizierung über den Objektnamen. Hinzu kommt die **Formatinformation**, denn wir möchten ja auch unterschiedliche Werteanzeigen realisieren. Schließlich wird die Eigenschaft **Aktiv** mitgegeben, um das Objekt sichtbar oder unsichtbar darzustellen. Fehlt noch der Wert vom **Trigger**, der als Eigenschaft mit übernommen werden muss, um später mit den Daten der Schnittstelle verglichen zu werden.

Hinzu kommen die Positions- und Größeneigenschaften **Top**, **Left**, **Width** und **Height**.

Würden all diese Parameter über eigene Parameter übergeben, wäre der Kopf der Routine fast nicht lesbar. Mit einer angelegten Datenstruktur ist es Übersichtlich. Daher wird die folgende Struktur am Anfang unseres Programms deklariert.

```
Public Structure Var_Rec
    Dim Name As String
    Dim Format As String
    Dim Aktiv As Boolean
    Dim Trigger As Integer
    Dim Top As Integer
    Dim Left As Integer
    Dim Width As Integer
    Dim Height As Integer
  End Structure
```

Kommen wir nun zur Subroutine, die eine Valuebox auf dem Panel **Pn_Variable** ablegt. In der Übergabe wird der Parameter **Parameter_Rec** deklariert.

In der Subroutine wird eine lokal deklarierte Variablen angelegt. Allerdings ist hier eine Besonderheit gegeben. Diese Variable ist vom Typ **ValueBox**, einem **Object UserControl**. Ihr Zweck ist es, das Objekt zu erstellen und das hat Folgen. Aber dazu etwas später die Erklärung. Zuerst einmal die Deklaration für unser selbstgestricktes Objekt **ValueBox**.

```
Dim My_Textfeld as Valuebox
```

Diese Zeile deklariert eine Variable, die alle, aber auch wirklich alle Eigenschaften der von uns erzeugten **ValueBox** enthält. Dazu gehört das **Label**, die beiden **Textboxen** sowie die intern angelegten Variablen. Und über **My_Textfeld** haben wir auch Zugriff darauf.

```
Public Sub Generate_ValueBox(ByVal Var_Daten  as  As Var_Rec)
    Dim My_Textfeld As ValueBox            ' Die Variable My_Textfeld wird als Typ Valuebox
    My_Textfeld = New ValueBox            '  das Objekt wird erzeugt
    My_Textfeld.Parent = Pn_Variablen    ' und dem Panel “Pn_Anzeige” zugeordnet
    My_Textfeld.Name = "VB_" + Var_Daten.Variable       ' Name bezogen auf die Variable
    My_Textfeld.Lb_Variable.Text = Var_Daten.Variable   ' Labeltext ist Variablennamen
    My_Textfeld.Format = Var_Daten.Format              ' Die Eigenschaft Format wird besetzt
    My_Textfeld.Triggerwert = "0"                      ' Triggerwert  Default 0.
    If Var_Daten.Trigger>0 then                        ' Triggerlevel aus Tabelle
      My_Textfeld.Triggerlevel = 2^(Var_Daten.Trigger -1)          ' anpassen
    Else
      My_Textfeld.Triggerlevel =0
    End If
    My_Textfeld.Visible = Var_Daten.Aktiv                        ' Sichtbarkeit
    My_Textfeld.TB_In.Text = "0"                   ' Anzeige Default 0
    My_Textfeld.Top = Var_Daten.Top                            ' Position
    My_Textfeld.Left = Var_Daten.Left
    My_Textfeld.Width = Var_Daten.Width                        ' Größe
    My_Textfeld.Height = Var_Daten.height
  End Sub
```

Die Zuweisung

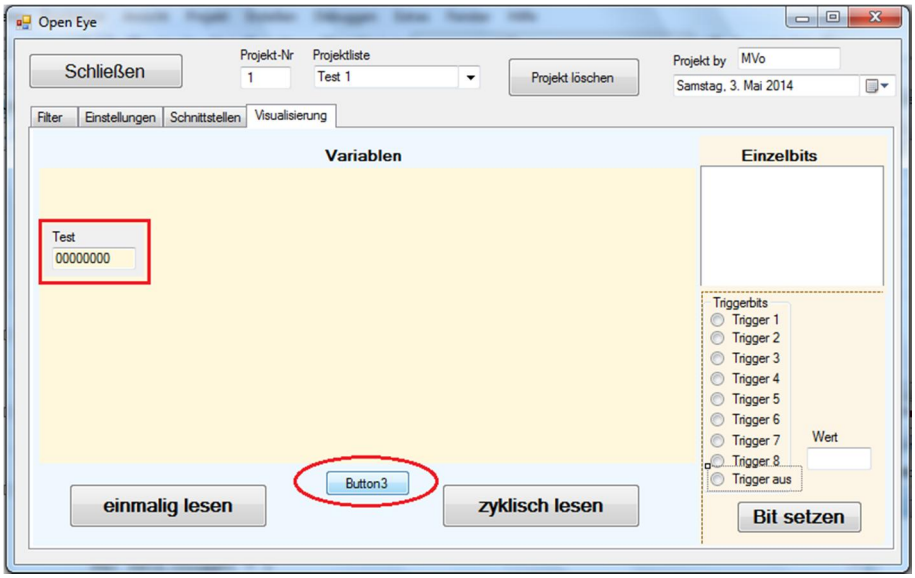
```
My_Textfeld = New ValueBox
```

erzeugt ein neues Objekt. Es ist eben nicht nur eine Variable, sondern ein Verweis auf das **UserControlObject**. Es wird im Speicher angelegt und bleibt auch nach dem Verlassen dieser Routine erhalten. Mit anderen Worten, bei jedem Aufruf wird der Speicher mit einem neuen **UserControl** gefüllt. Es stört uns erst einmal nicht, aber wenn keine Maßnahme ergriffen wird, kann ein Speicherüberlauf die Folge sein. Man spricht auch von dynamischer Ablage von Objekten. Wenn also das Programm nicht geschlossen und keine weitere Maßnahme ergriffen wird, kommen immer mehr von diesen selbst ernannten Objekten hinzu. Der Platz, wo sie installiert werden wird mit **Parent** angegeben. Sie sind praktisch wie Kinder ihren Eltern zugeordnet und wenn **Parent** das Panel-Objekt **Pn_Variablen** ist, sind die Positionsparameter auf den Bereich im Panel bezogen. Aber führen wir erst einmal einen Test durch, der uns die Wirkung dieser Subroutine zeigt. Dazu installieren wir auf der Seite Visualisierung einfach ein **Button** und rufen mit den **Click-Ereignis** diese Subroutine auf.

Dazu muss eine lokale Variable vom Typ **Var_Rec** deklariert und die Parameter in der Strukturvariablen mit Werten besetzt werden.

```
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button3.Click
    Dim Par_Satz As Var_Rec
    Par_Satz.Aktiv = True          ' sichtbar
    Par_Satz.Format = "Byte"       ' binäre Darstellung
    Par_Satz.Trigger = 0
    Par_Satz.Name = "Test"         ' Objektname und Überschrift
    Par_Satz.Left = 4              ' Position
    Par_Satz.Top = 50
    Par_Satz.Width = 88
    Par_Satz.Height = 46
    Generate_ValueBox(Par_Satz)
  End Sub
```

Anschließend starten wir das Programm und klicken auf den provisorischen Button. Es erscheint das **UserControl**, welches wir erstellt haben.




*Anzeigeobjekt einfügen*

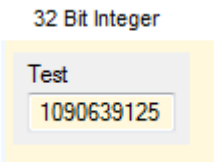
Doch egal wie oft wir nun den Button betätigen, es zeigt sich nichts Neues. Ist ja auch klar, die weiteren Objekte liegen exakt auf den Vorgängern. Wir können jetzt mal ein wenig mit den Parametern spielen und verschiedene Positionen angeben. Das Objekt folgt den Anweisungen. Verändern wir einmal die Formatangabe und auch in der Subroutine die Vorbesetzung von **Tb_In** in den Wert <123>. Auch mal einen Wert wie auf Seite 336 angegeben mit vier Byte und dem Format Int32 sowie Hex32.

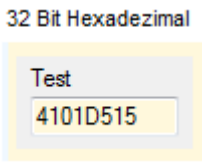
```
My_Textfeld.TB_In.Text = "21;213;1;65"          ' Anzeige Default 0
```

Und die Formatangabe

```
Par_Satz.Format = "Int32"    ' binäre Darstellung
```

Die Anzeigen liefern nun die exakten Werte. Das lässt sich leicht mit einem Taschenrechner überprüfen. Auch der angegebene Wert von Seite 336 sollte mit dem angezeigten Wert übereinstimmen




*Zahlentest*

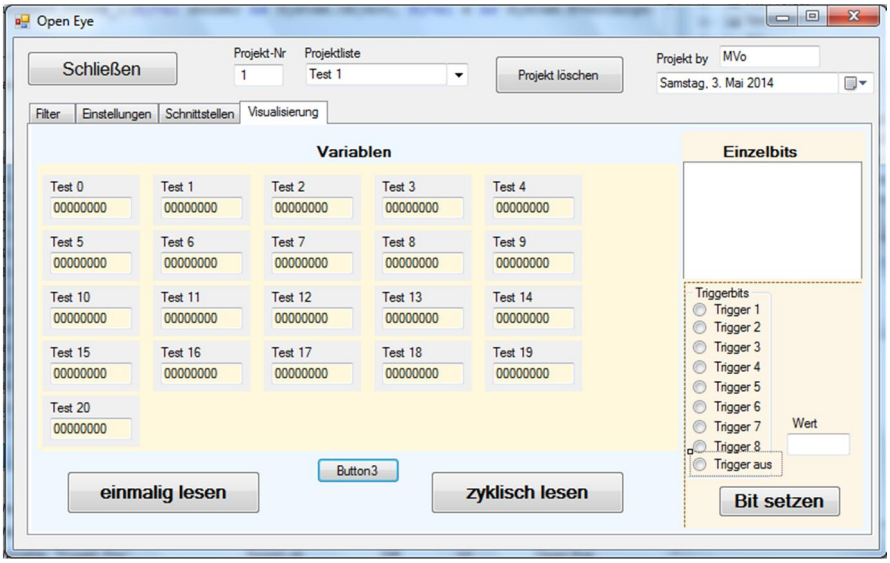
### *1.5.4.1 Eine große Anzahl der Objekte erzeugen*

Nun möchten wir aber viele solcher Anzeigeelemente auf dieser Seite erzeugen. Dafür erweitern wir erst einmal die Ereignisroutine unseres provisorischen Button, um den Erfolg auch gleich auf der Seite zu sehen.

Zuerst wird zusätzlich ein Schleifenzähler deklariert. Dann erfolgen die Zuweisungen der Startpositionen und der Größe außerhalb der Schleife. Innerhalb der Schleife werden die restlichen Parameter zugewiesen und die Subroutine **Generate_Valuebox** aufgerufen. Nach dem Aufruf schieben wir die horizontale Position **Left** um 100 Pixel weiter. Reicht der Platz zur Darstellung nicht, wird die vertikale Position **Top** 50 Pixel nach unten geschoben und horizontal wieder auf Anfang gestellt.

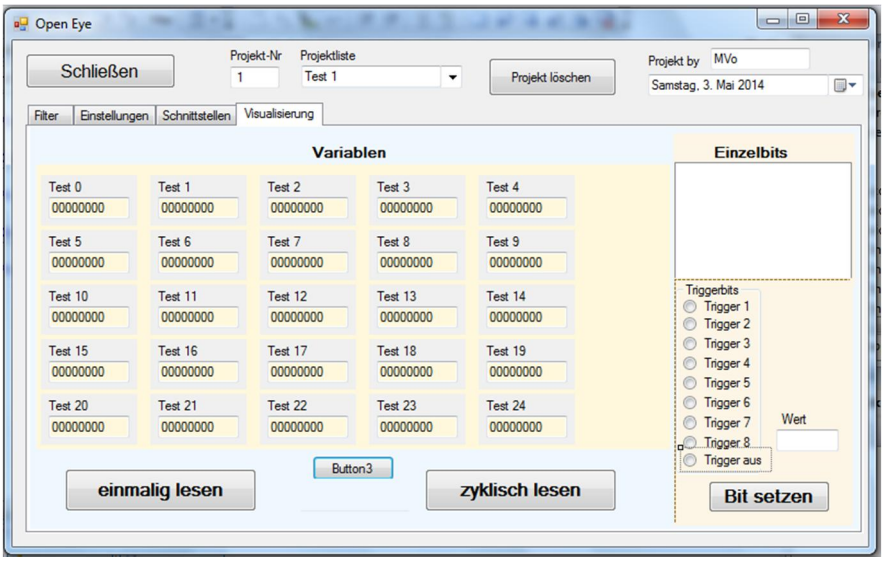
```
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button3.Click
    Dim Par_Satz As Var_Rec
    Dim i As Integer
    Par_Satz.Left = 4            ' Position
    Par_Satz.Top = 10
    Par_Satz.Width = 88
    Par_Satz.Height = 46
    For i = 0 To 20
      Par_Satz.Aktiv = True       ' sichtbar
      Par_Satz.Format = "Byte"    ' binäre Darstellung
      Par_Satz.Trigger = 0
      Par_Satz.Name = "Test" + Str(i)      ' Objektname und Überschrift
      Generate_ValueBox(Par_Satz)
      Par_Satz.Left = Par_Satz.Left + 100     ' nächste Position
      If Par_Satz.Left + 100 > Pn_Variablen.Width Then
        Par_Satz.Left = 4
        Par_Satz.Top = Par_Satz.Top + 50
      End If
    Next
  End Sub
```

Nach dem Test dieser Änderung erhalten wir nun 21 Objekte in dem Panel **Pn_Variablen** verteilt.



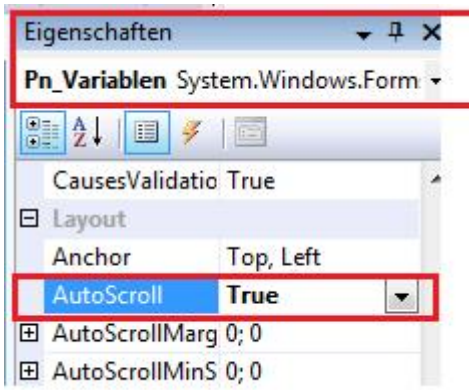

*alle Anzeigen passen*

Erhöhen wir einmal die Anzahl der Schleifendurchläufe auf 40. Die Seite ist zwar voll, aber nicht alle Objekte sind sichtbar, wie der Screenshot zeigt. Ein Scrollbalken wie bereits angekündigt ist auch nicht zu sehen.



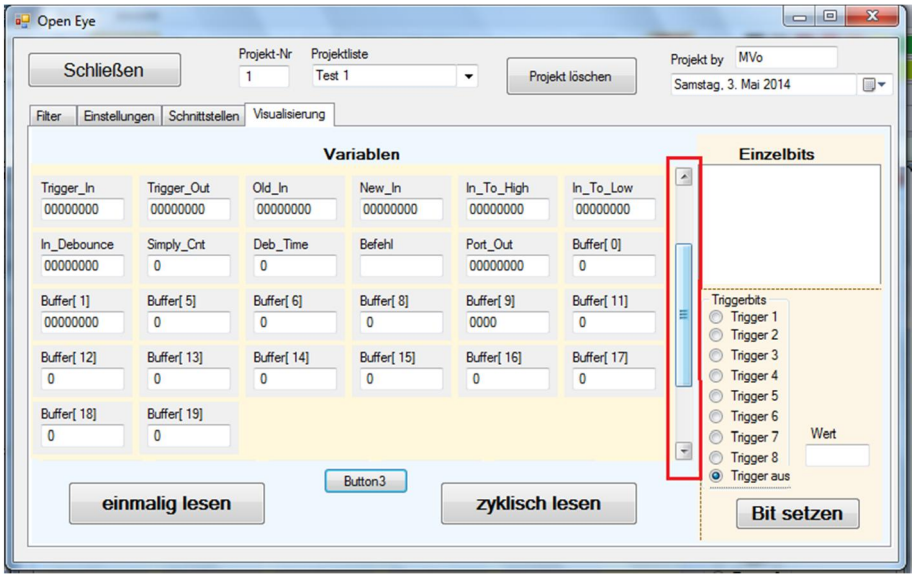

*Zu viele Anzeigen*

Wie ist nun die Darstellung der weiteren Objekte zu erlangen. Zuerst wird in der Eigenschaftsliste vom Panel **Pn_Variablen** die Eigenschaft **AutoScroll** auf **true** gesetzt.




*Anzeige Autoscroll ein*

Damit bekommen wir einen Scrollbalken und können nun alle Objekte in den sichtbaren Bereich schieben.




*Anzeige mit Scrollbalken*

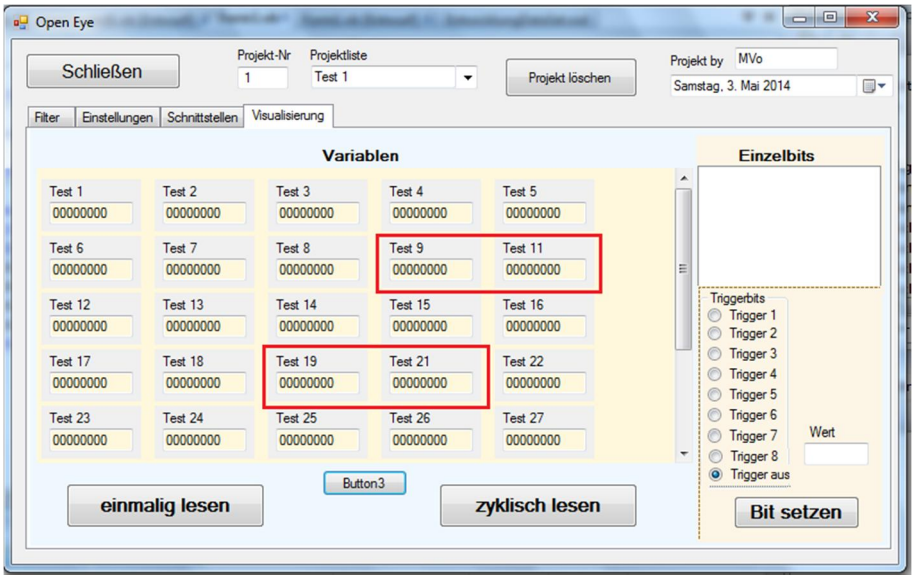
Eine weitere Möglichkeit ist durch die Eigenschaft **Aktiv** eingerichtet. Sie verhindert, dass die erzeugte **Valuebox** angezeigt wird, obwohl sie vorhanden ist. So kann man sich Projekte einrichten und verschiedene Ansichten wichtiger Variablen erzeugen, um sie in einer Gesamtansicht zu beobachten, Dazu werden wir jedes 10. Objekt in unserer Schleife einmal unterdrücken und die Positionsfortschaltung aussetzen.

```
For i = 0 To 40
   Par_Satz.Aktiv = (i / 10) <> Math.Truncate(i / 10)
   Par_Satz.Format = "Byte"     ' binäre Darstellung
   Par_Satz.Trigger = 0
   Par_Satz.Name = "Test" + Str(i)       ' Objektname und Überschrift
   Generate_ValueBox(Par_Satz)
   If Par_Satz.Aktiv Then
      Par_Satz.Left = Par_Satz.Left + 100     ' nächste Position
      If Par_Satz.Left + 100 > Pn_Variablen.Width Then
         Par_Satz.Left = 4
         Par_Satz.Top = Par_Satz.Top + 50
      End If
   End If
Next
```

Die Änderungen sind ziemlich einfach. Die Eigenschaft **Aktiv** wird wieder mit einer mathematischen Klausel geprüft. Das Kennen wir ja schon, aber trotzdem noch einmal die Erklärung zur Erinnerung.

Die Aussage X/10 ist gleich einem ganzzahligen Ergebnis Truncate(X/10) kann wahr sein oder auch nicht. Ebenso die umgekehrte Aussage X/10 ist ungleich einem ganzzahligen Ergebnis Truncate(X/10). Und dieses Wahr oder Unwahr wird dem Parameter **Aktiv** zugewiesen. Dieser wiederum kann dann den Bereich zuschalten, um die Positionierung des nächsten Objektes zu berechnen.

Das Ergebnis im Screenshot:




*Anzeigeobjekte ausblenden*

Die Objekte 10, 20 und höchstwahrscheinlich auch 30 und 40 fehlen in der Darstellung. Sie sind lediglich unsichtbar, da der Aufruf **Generate_ValueBox** erfolgt ist.

Damit sind die Experimente mit einem provisorischen Button erst einmal beendet und wir gehen zu unserer Variablentabelle auf der Seite Einrichten.

### *1.5.5 Objekte zur Anzeige der Variablenwerte erzeugen*

Da unsere Arbeit nicht ganz umsonst sein soll, kopieren wir den Inhalt der Ereignisroutine Click des provisorischen Button in das Ereignis Click des Button Bt_Gen_Anzeige. Dann werden die Zuweisungen geändert Diesmal kommen die Einträge in der Tabelle in den Parametersatz. Dabei wird die Schleifengrenze mit der Anzahl der Tabellenzeile festgelegt. In der Schleife werden die Einträge der mit i adressierten Tabellenzeilen in den Parametersatz geschrieben.

```
Private Sub Bt_Gen_Anzeige_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Gen_Anzeige.Click
    Dim Par_Satz As Var_Rec
    Dim i As Integer
    Par_Satz.Left = 4            ' Position
    Par_Satz.Top = 10
    Par_Satz.Width = 88
    Par_Satz.Height = 46
    For i = 0 To DG_Variablen.Rows.Count - 2
      Par_Satz.Aktiv = DG_Variablen.Item("CL_Aktiv", i).Value = "Ja"
     ' bildet einen boolschen Ausdruck aus “ja”.
      Par_Satz.Format = DG_Variablen.Item("CL_Format", i).Value
     ' kopiert das Format aus der Tabellenzeile.
     Par_Satz.Trigger = Val(DG_Variablen.Item("CL_Trigger", i).Value)
      If Par_Satz..Trigger > 0 Then  ' Bitnummer in Bit darstellen
          Par_Satz..Trigger = 2 ^ ( Par_Satz..Trigger – 1)  ' Bit setzen
      End If
     ' kopiert den Triggerwert aus der Tabellenzeile
      Par_Satz.Name = DG_Variablen.Item("CL_Variable", i).Value
     ' kopiert den Variablennamen aus der Tabellenzeile
      Generate_ValueBox(Par_Satz)
      If Par_Satz.Aktiv Then
         Par_Satz.Left = Par_Satz.Left + 100    ' nächste Position
         If Par_Satz.Left + 100 > Pn_Variablen.Width Then
            Par_Satz.Left = 4
            Par_Satz.Top = Par_Satz.Top + 50
         End If
      End If
    Next
  End Sub
```

Der Parameter Trigger wird dabei verändert. Warum? Nun, in der Tabelle adressiert der Eintrag ein **Item** in der Combobox **Cb_Trigger** dort steht von **Kein** bis **8.** Der Eintrag **Kein** steht also im **Item 0**, **Bit 0 in Item 1**

usw. Der Vergleichswert ist aber abhängig, ob im Byte **Trigger** das Bit 0, 1, 2, 3, 4, 5, 6 oder 7 gesetzt ist und das ist **Tabelleneintrag-1.** Das Bit zu setzen ist einfach. Wir haben gelernt, das die Stellen mit der **Basis 2^x** definiert sind. **Bit 0**, also das erste Bit im Byte hat die Stelle **2^0**, **Bit 1**, das zweite Bit im Byte **2^1** usw. Um dort eine 1 einzutragen muss dementsprechend einfach nur **2^x** gerechnet werden. Das entspricht den Werten 1, 2, 4, 8, 16, 32, 64 und 128. Ist **Kein** Bit gesetzt, braucht auch nichts verglichen werden. Demnach bleibt der **Trigger** 0, wenn in der Tabelle eine 0 steht, also die Referenz zu **Kein,** ansonsten ist der Tabelleneintrag-1 der Exponent von 2 an den Trigger weiter zu geben. Leider können wir das zur Zeit nicht testen, aber anzeigen. In der Routine **Tb_In_TextChanged**, also auf der Seite **UserControl.VB** setzen wir einfach als letzte Zeile

```
TB_Out.Text=Frm_Open_Eye.IntToBin(Trigger).
```

Damit überbügeln wir alle bisherigen Zuweisungen an **Tb_Out.Text**. Das Programm wird gestartet und die ersten Variablen bekommen Triggerereignisse zugewiesen. Wird nun die Anzeige generiert, ist in den Anzeigen das zugehörige **Bit** gesetzt.


*einen Triggerwert anzeigen*

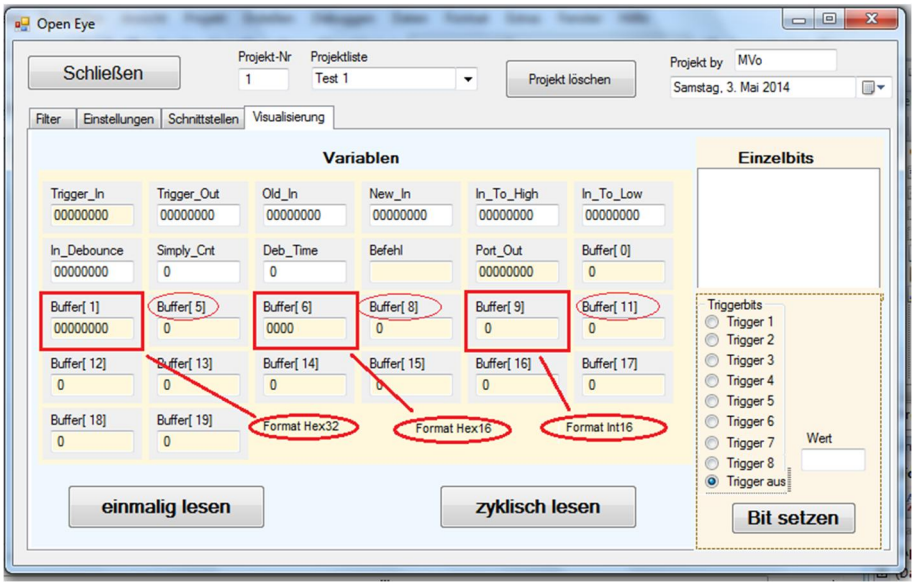
Die Zeile wird anschließend wieder entfernt. Sie diente nur der Kontrolle der einwandfreien Funktion der bisherigen Programmierung. Nun ist auch dieser Part geprüft und der nächste Schritt kann folgen.

### *1.5.6 Objekt ausschließen*

Die Erzeugung der Anzeigeobjekte aufgrund der Tabellendaten ist vom Prinzip her klar. Nun kommen die Feinheiten. Dass nicht jedes Objekt visualisiert werden muss, ist durch den Eintrag Aktiv bestimmt. Es wird, wenn nicht **Aktiv**, zwar erzeugt, aber nicht visualisiert. Das ist für die Wertezuweisung wichtig. Es gibt aber einen Eintrag in der Tabelle, der aufzeigt, dass zu einem Wert mehrere Bytes gehören. So sind die Formate **Int16**, **Int32**, **Hex16** und **Hex32** mit mehreren Tabellenzeilen verbunden. Die gesperrten Zeilen dürfen kein Anzeigeobjekt generieren, da für diese Objekte keine Werte vorliegen. Diese Aufgabe muss nun in die Schleife integriert werden. Also wird über die Erzeugung noch eine Abfrage gepackt, ob die Tabellenzeile freigegeben ist.

```
For i = 0 To DG_Variablen.Rows.Count - 2
        If DG_Variablen.Item("CL_Freigabe", i).Value = "Ja" Then
            Par_Satz.Aktiv = DG_Variablen.Item("CL_Aktiv", i).Value = "Ja"
            ' bildet einen boolschen Ausdruck aus “ja”.
            Par_Satz.Format = DG_Variablen.Item("CL_Format", i).Value
            ' kopiert das Format aus der Tabellenzeile.
            Par_Satz.Trigger = Val(DG_Variablen.Item("CL_Trigger", i).Value)
            If Par_Satz.Trigger > 0 Then  ' Bitnummer in Bit darstellen
                Par_Satz.Trigger = 2 ^ (Par_Satz.Trigger – 1)  ' Bit setzen
            End If
            ' kopiert den Triggerwert aus der Tabellenzeile
            Par_Satz.Name = DG_Variablen.Item("CL_Variable", i).Value
            ' kopiert den Variablennamen aus der Tabellenzeile
            Generate_ValueBox(Par_Satz)
            If Par_Satz.Aktiv Then
                Par_Satz.Left = Par_Satz.Left + 100     ' nächste Position
                If Par_Satz.Left + 100 > Pn_Variablen.Width Then
                    Par_Satz.Left = 4
                    Par_Satz.Top = Par_Satz.Top + 50
                End If
            End If
        End If
    Next
```

Diese Änderung wird sofort getestet. Das Ergebnis zeigt der Screenshot.

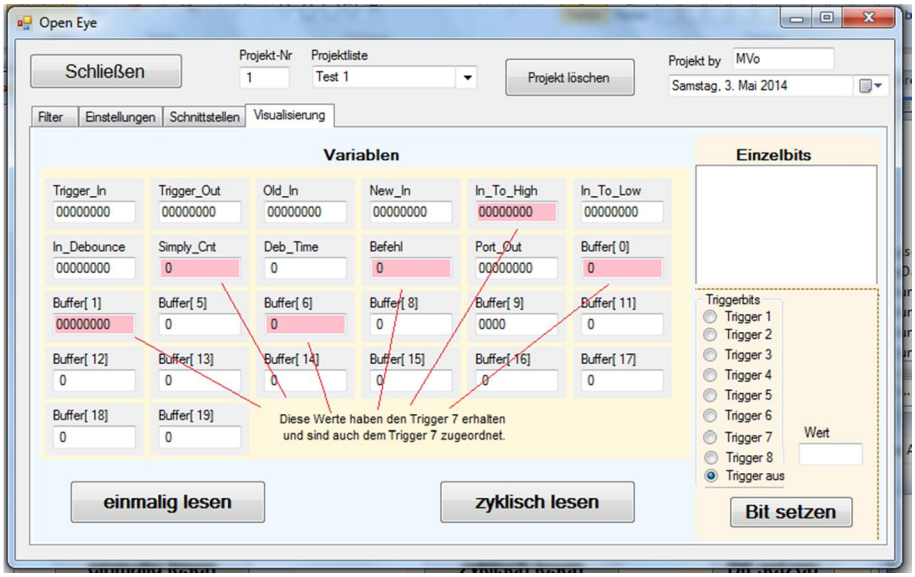



*Anzeige verschiedene Formate*

Obwohl ein **Byte**-Format von einem **Hex32**-Format auf den ersten Blick nicht zu unterscheiden ist, sind die Formate auch bei Null-Werten erkennbar. Durch den Nummernsprung ist schon auf einen größeren Wert als 255 zu schließen.
Damit ist der Bereich der Variablenanzeige soweit abgehandelt. Einen letzten Test führen wir aber noch durch, den Farbumschlag bei einem Triggerwert, der den Einstellungen in der Tabelle entspricht. Dazu setzen wir in der Subroutine **Generate_ValueBox** die Zuweisung an **Triggerwert** auf **64**

```
My_Textfeld.Triggerwert = "64"
```

Deutlich zeichnen sich nun die Felder ab, die dem empfangenen Triggerwert 64 zugeordnet sind. Das ist der Trigger 7.



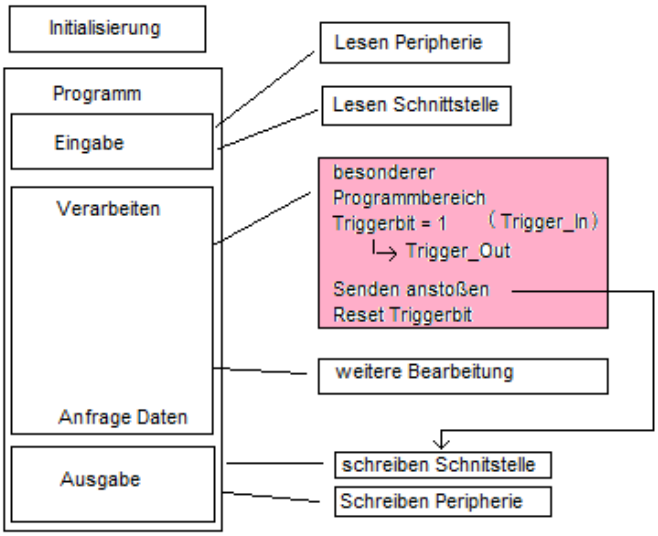

*Anzeige von Trigger*

Eine wichtige Aufgabe ist noch nicht erledigt. Bevor neue Ausgabeobjekte erzeugt werden, müssen eventuell bereits installierte Objekte entfernt werden. Sonst laufen wir Gefahr, bei intensiver Nutzung irgendwann einmal einen Systemabsturz zu bekommen, weil der Speicher nicht mehr ausreicht. Dazu sind zwei einfache Anweisungen in der Ereignismethode **Bt_Gen_Anzeige** gleich am Anfang hinter die Variablendeklaration einzufügen

```
Pn_Variablen.Controls.Clear()        ' Panel leeren
```

### *1.5.6.1 Funktionserklärung Trigger.*

Im Controllerprogramm ist eine Stelle, die bei einer Bearbeitung sofort ein paar Variablen anzeigen soll. Niemand kann den Zeitpunkt bestimmen, und daher wird diese Stelle mit einem Triggerbyte markiert. Da ich hier noch keinen Assemblercode einstellen will, zeige ich einen Programmablauf im klassischen EVA-Stil



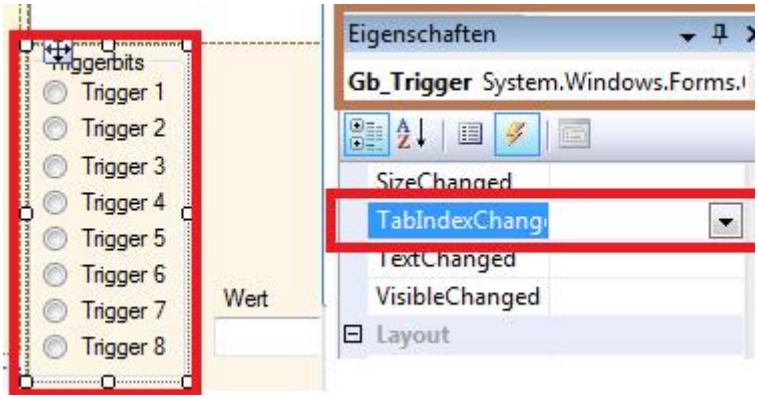

*Triggerereignis im Controller*

Über die Empfangsroutine bekommt der Controller einen Trigger mitgeteilt, eben dieses spezielle Bit. In der dafür vorgesehenen Bearbeitung prüft eine kleine Routine, ob ein solches Bit vorliegt und wenn, löst sie sofort über die Senderoutine den Datentransfer aller Variablen aus. Dabei wird der Trigger zurückgeliefert und dient der Visualisierung für die Farbmarkierung. Eine Datenanforderung ohne Trigger liefert die Werte erst, wenn der Controller im Programm die Befehle vom Controller auswertet. Der Unterschied in den Variablenwerten ist der Zeitpunkt. Mit Trigger kann direkt hinter einer Zuweisung der Transfer ausgelöst werden, um den aktuell zu diesem Zeitpunkt gültigen Status abzufragen, ein anderes Mal abhängig von der Auswertung eines Befehles vom PC am Ende der Programmbearbeitung. Die angezeigten Werte beziehen sich dann auf den Inhalt der Assemblervariablen nach Durchlauf des gesamten Programms.

### *1.5.7 Der Bereich Triggerbits*

Es bietet sich an, das Kapitel der Generierung und Behandlung von Triggerbits anzuschließen und die Einzelbitdarstellung etwas nach hinten zu schieben.

Die Triggerbits werden mit **Radiobutton** in einer **Groupbox** angelegt und zugewiesen. Auch hier ist ein Ereignis zu suchen, welches benutzt werden kann, aufgrund einer Änderung in der **Groupbox** und dem Status einer Radiobox den Wert des Triggerbytes zu berechnen. Beginnen wir also zuerst mit der Suche im Eigenschaftsfenster der **Groupbox** unter Ereignissen. Von allen Ereignissen ist **TabIndexChange** vielversprechend.




*Groupbox TabIndexChanged*

Probieren wir einmal aus, welche Möglichkeiten damit gegeben werden. Erstellen wir die Ereignisroutine durch einen Doppelclick auf den Eintrag und deklarieren Wert und Info als lokale Variablen. Eine **Integer**, die andere **String**.

```
Private Sub Gb_Trigger_TabIndexChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Gb_Trigger.TabIndexChanged
    Dim Wert As Integer
    Dim Info As String
    Info = Gb_Trigger.Text
    Wert = Rb_Bit_0.Checked
End Sub
```

Auf die Zeile **Wert =** setzen wir einen Haltepunkt und starten das Programm. Beim Start wird das Programm hier zwar einmal angehalten,

aber wenn wir dann **Radiobuttons** anklicken nicht mehr. Also, diese Routine ist nicht geeignet. Sie wird wieder gelöscht und weiter gesucht. Auch **TextChanged** bringt nicht den gewünschten Erfolg und wir brechen die Suche ab. Da wir wissen, das die Radiobuttons sich gegenseitig verriegeln, können wir vielleicht auf ein Ereignis **CheckedChanged** zurückgreifen. Einen Versuch ist es ja wert.

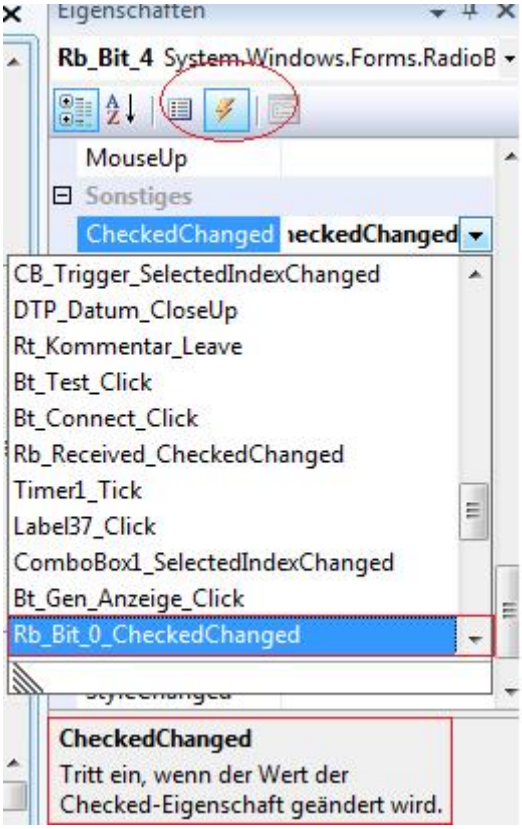
```
Private Sub Rb_Bit_0_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Rb_Bit_0.CheckedChanged
    If Rb_Bit_0.Checked Then Tb_Wert.Text = Str(2 ^ 0)
  End Sub
```

Diesmal lassen wir den Haltepunkt und weisen der Textbox **Tb_Wert** direkt das Ergebnis **2^0** zu. Das Ereignis holen wir von der Radiobox **Rb_Bit_0**.

Der Versuch zeigt einen Erfolg. Die Textbox bekommt einen Wert, wenn wir das Radiobutton von Bit 0 anklicken. Nun brauchen wir nicht jedem Radiobutton ein eigenes Ereignis zuweisen. Es reicht, wenn alle als Ereignis diese Ereignismethode zugewiesen bekommen und das Programm entsprechend erweitert wird. Dabei besetzen wir die Textbox mit 0 vor. Sollten keine Radiobutton angeklickt sein, soll auch in der Textbox eine 0 stehen.

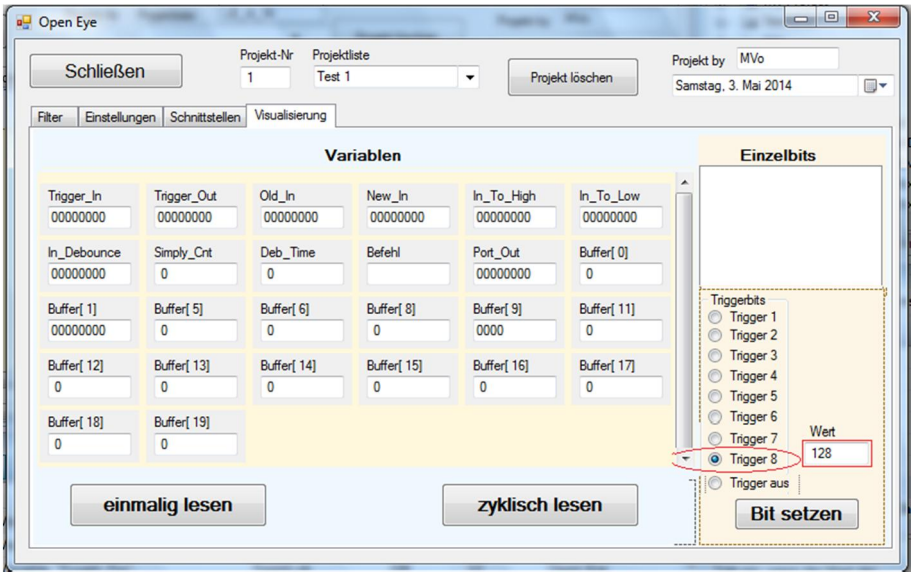
```
Private Sub Rb_Bit_0_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Rb_Bit_0.CheckedChanged
    TB_Wert.Text = "0"
    If Rb_Bit_0.Checked Then TB_Wert.Text = Str(2 ^ 0)
    If Rb_Bit_1.Checked Then TB_Wert.Text = Str(2 ^ 1)
    If Rb_Bit_2.Checked Then TB_Wert.Text = Str(2 ^ 2)
    If Rb_Bit_3.Checked Then TB_Wert.Text = Str(2 ^ 3)
    If Rb_Bit_4.Checked Then TB_Wert.Text = Str(2 ^ 4)
    If Rb_Bit_5.Checked Then TB_Wert.Text = Str(2 ^ 5)
    If Rb_Bit_6.Checked Then TB_Wert.Text = Str(2 ^ 6)
    If Rb_Bit_7.Checked Then TB_Wert.Text = Str(2 ^ 7)
 End Sub
```

Die Zuweisung der Ereignisroutine erfolgt nun bei jedem Radiobutton in der Eigenschaft unter Ereignisse **CheckedChanged**. Dazu wird die Liste der bereits erstellten Ereignismethoden mit dem **Dropdown**-Button geöffnet und **Rb_Bit_0_CheckedChanged** ausgewählt.




*RadioButton CheckedChanged*

Alle Radiobutton bekommen auf diese Weise das Ereignis zugewiesen. Anschließend wird das Programm gestartet und das Ergebnis geprüft.

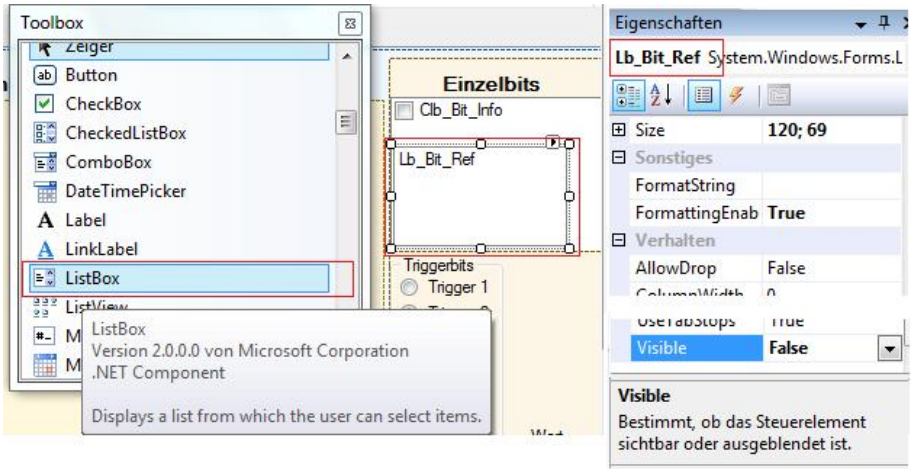



*Trigger setzen*

### *1.5.8 Erfassung der Einzelbits*

In diesem Abschnitt erfolgt die Zuordnung der kommentierten und abgespeicherten Einzelbits an die **Items** in der Checkboxliste. Die Einzelbits sind in einer Datenbanktabelle abgelegt und durch eine einfache Abfrage in die Checkboxliste einzubringen. Da wir in der Schleife auch die Formate der Variablen erfassen und wissen, das nur Byte-Formate für eine Einzelbitdarstellung in Frage kommen, kann eine kleine Subroutine aufgerufen werden, die uns die erforderlichen Daten zur Verfügung stellt und die Checkboxliste **Clb_Bit_Info** füllt.

Damit wir einen schnellen Zugriff auf die Einträge bekommen, wenn wir später die Werte übergeben wollen, brauchen wir zu den Einträgen eine Referenz. Etwas ähnliches haben wir bereits mit der Projektliste in der Combobox **Cb_Projekte** machen müssen, um die **Id_Nr** des Projektes zu erhalten. Hier ist es der Zugriff auf ein Bit einer Variablen und der Bitstelle. Mit einer unsichtbaren Listbox lässt sich diese Information ablegen, damit wir bei der Wertezuweisung über diese Referenz die Einträge wiederfinden. Deshalb installieren wir eine unsichtbare Listbox über der Checklistbox und vergeben den Namen **Lb_Bit_Ref**




*Listbox unsichtbar*

Bei jedem Aufruf der Ereignisroutine **Bt_Gen_Anzeige** sollen nun Daten in die Listen eingetragen werden. Damit auch nur die Daten dieser Auswahl der Variablen dort erscheinen, müssen die Listen wie auch das Panel **Pn_Variable** geleert werden.Das erledigen wir am Anfang der Routine **Bt_Gen_Anzeige**

```
Private Sub Bt_Gen_Anzeige_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Gen_Anzeige.Click
    Dim Par_Satz As Var_Rec
    Dim i As Integer
    Pn_Variablen.Controls.Clear()   ' Panel leeren
    Clb_Bit_Info.Items.Clear()      ' Checklistbox leeren
    Lb_Bit_Ref.Items.Clear()        ' Referenzliste leeren
    ……..
```

Nun können wir die Checklistbox und die Referenzliste füllen.

Am Anfang steht wieder ein Rumpf einer Subroutine und die Überlegung, welche Aufgabe zu erledigen ist und welche Informationen dieser Subroutine mitgeteilt werden muss.

Da eine Variable in der Datenbank eine eindeutige Zuordnung hat, kann auch über diese Zuordnung die Einzelbitinformation abgeholt werden. Deshalb wird diese Routine einen Übergabeparameter bekommen, der auf die **Id_Nr** der aktuellen Variable zeigt. Im Text muss ein Hinweis sein, zu welcher Variablen die Bits gehören und so übergeben wir auch den Variablennamen. Lokal erwarten wir mehrere Datensätze, die als Einträge in der CheckListbox **Clb_Bit_Info** landen. Dafür wird ein Schleifenzähler deklariert. Und natürlich ist auch die Anzahl der empfangenen Datensätze eine lokale Variable wert. Um eine übersichtliche Befehlsstruktur zu bekommen, setzen wir für die Information aus der Datenbanktabelle auch lokale Variablen. Einmal für die Bitnummer und dann für die Bitbeschreibung. Schließlich nehmen wir noch eine Stringvariable **Listeneintrag**. Sie bekommt aus allen selektierten Informationen eine Zusammenstellung.

Im ersten Schritt holen wir die Anzahl der von der Abfrage zurückgelieferten Datensätze und weisen dies der Variablen **Anzahl** zu.

Nun kann in einer Schleife ein Datensatz nach dem Anderen gelesen, die Information für den Listeneintrag zusammen gestellt und schließlich dort eingetragen werden. Diese **CheckListbox** funktioniert wie eine **ListBox**, hat aber zusätzlich eine **Checkbox** in jedem **Item**. Dies ist bei der Zuweisung zu beachten und der Wert **False** für den Status der **Checkbox** als Parameter hinzuzufügen.

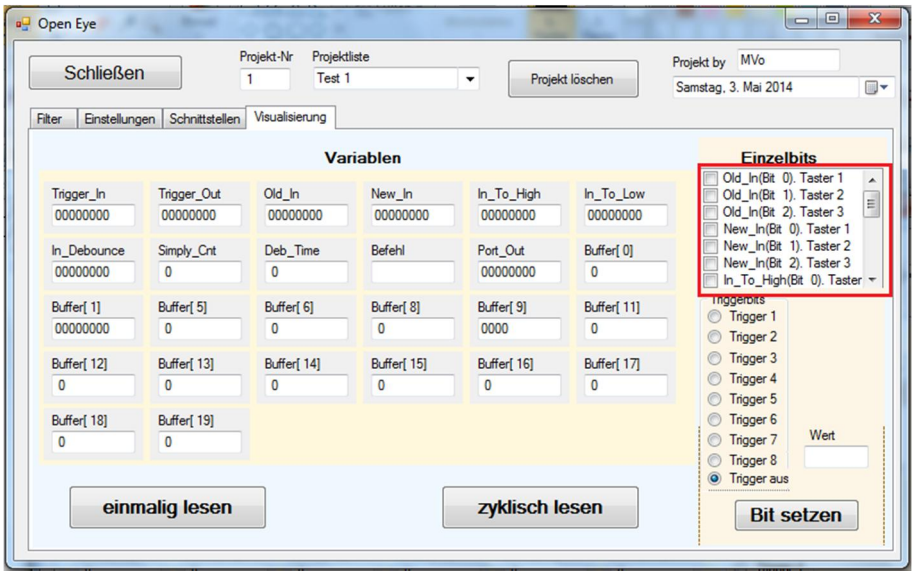
```
Public Sub Set_Bitliste(ByVal Zeile_Nr As Integer)
    Dim I As Integer
    Dim Id_Nr As Integer
    Dim Variable As String
    Dim Anzahl As Integer
    Dim Bit_Nr As Integer
    Dim BitInfo As String
    Dim Listeintrag As String
    Id_Nr = Val(DG_Variablen.Item("CL_Id_Nr", Zeile_Nr).Value)
    Variable = DG_Variablen.Item("CL_Variable", Zeile_Nr).Value
    Anzahl = EinzelbitTableAdapter.Get_VariablenBit(Id_Nr).Rows.Count
    If Anzahl > 0 Then
      For I = 0 To Anzahl - 1
        Bit_Nr = EinzelbitTableAdapter.Get_VariablenBit(Id_Nr).Item(I).Bit
        BitInfo = EinzelbitTableAdapter.Get_VariablenBit(Id_Nr).Item(I).Funktion
        Listeintrag = Variable + "(Bit " + Str(Bit_Nr) + ")." + BitInfo
        Clb_Bit_Info.Items.Add(Listeintrag, False)
        Lb_Bit_Ref.Items.Add(Str(Zeile_Nr) + "." + Str(Bit_Nr))    ' Referenz
      Next
    End If
 End Sub
```

Fügen wir nun diese Subroutine in unser Programm und rufen diese im Ereignis **Bt_Gen_Anzeige_Click** unter der Bedingung **Format = Byte** auf.

```
Private Sub Bt_Gen_Anzeige_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Gen_Anzeige.Click
    Dim Par_Satz As Var_Rec
    Dim i As Integer
    Pn_Variablen.Controls.Clear()        ' Panel leeren
    Clb_Bit_Info.Items.Clear()            ' Checklistbox leeren
    Par_Satz.Left = 4                       ' Position
    Par_Satz.Top = 10
    Par_Satz.Width = 88
    Par_Satz.Height = 46
    For i = 0 To DG_Variablen.Rows.Count - 2
      If DG_Variablen.Item("CL_Freigabe", i).Value = "Ja" Then
        Par_Satz.Aktiv = DG_Variablen.Item("CL_Aktiv", i).Value = "Ja"
        ' bildet einen boolschen Ausdruck aus “ja”.
        Par_Satz.Format = DG_Variablen.Item("CL_Format", i).Value
        ' kopiert das Format aus der Tabellenzeile.
        Par_Satz.Trigger = Val(DG_Variablen.Item("CL_Trigger", i).Value)
        If Par_Satz.Trigger > 0 Then  ' Bitnummer in Bit darstellen
```

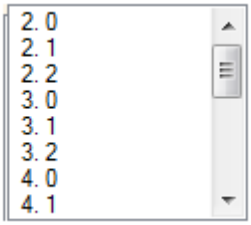
```
            Par_Satz.Trigger = 2 ^ Par_Satz.Trigger - 1  ' Bit setzen
        End If
        ' kopiert den Triggerwert aus der Tabellenzeile
        Par_Satz.Name = DG_Variablen.Item("CL_Variable", i).Value
        ' kopiert den Variablennamen aus der Tabellenzeile
        Generate_ValueBox(Par_Satz)
        If Par_Satz.Format = "Byte" Then   ' Einzelbits erfassen
            Set_Bitliste(i)
        End If
        If Par_Satz.Aktiv Then
            Par_Satz.Left = Par_Satz.Left + 100     ' nächste Position
            If Par_Satz.Left + 100 > Pn_Variablen.Width Then
                Par_Satz.Left = 4
                Par_Satz.Top = Par_Satz.Top + 50
            End If
        End If
    End If
  Next
  TC_Auswahl.SelectTab(3)              ' Weiterschalten zur Ansicht
End Sub
```

Ein Test der erweiterten Programmierung bringt das Ergebnis.




*Ergebnis Einzelbitanzeige*

Hier einmal die Inhalte der unsichtbaren Listbox **Lb_Bit_Ref**




*Inhalt unsichtbare Listbox*

Der Eintrag 2.0 zeigt auch auf das zweite Anzeigeelement und der Variablenansicht und 0 auf Bit 0 mit der Beschreibung Taster 1.

Die Beschreibung in der CheckListbox passt also zur Referenz.

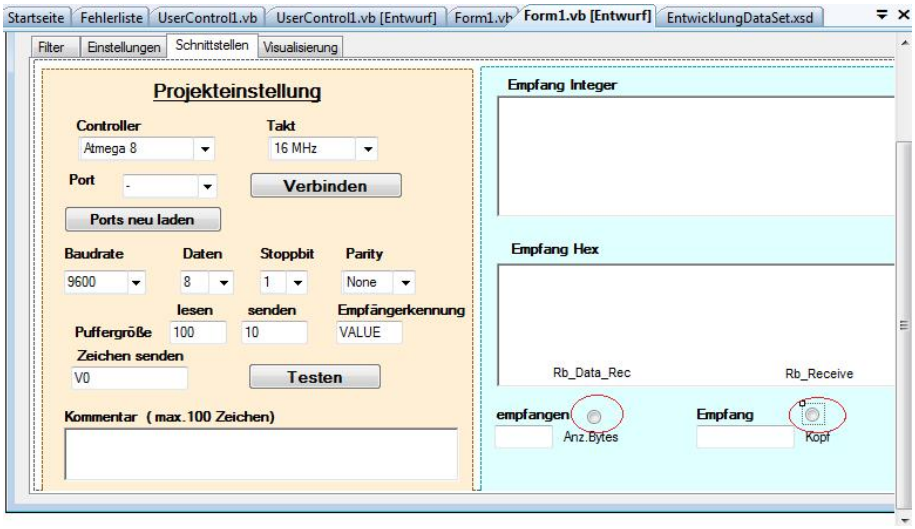
## 1.6 Empfangene Daten zuweisen

Mit der Schnittstelle und dem Datenempfang haben wir uns ja bereits befasst und auch den Empfang geprüft. An dieser Stelle wollen wir uns nun Gedanken machen, wie die eingehenden Werte in die Anzeigeobjekte übertragen werden können. Dabei ist hier schon ein wenig auf Effizienz zu achten, denn sobald Daten eingegangen sind, sollten sie möglichst schnell zu der Anzeige gelangen. Bekannt ist, dass der **Timer** den Ringspeicher abfragt. Also setzen wir dort an und schreiben die eingehenden Werte in ein separates globales **ByteArray**.

Es wird gleich hinter dem **Rec_Feld** am Anfang des Programms deklariert.

```
Public Werte_Feld(999) As Byte
```

Ob die Dimensionierung zu groß ist, spielt erst einmal keine Rolle. Dieses Array ist kein Ringpuffer, sondern nimmt genau eine Datenübertragung der Werte, also ohne Vorkopf auf. Diesen müssen wir vorher entfernen. Erhalten wir Daten, ist der Vorkopf ein String. Auf Seite drei **Schnittstellen** ist ein Textfeld **Empfängerkennung.** Dort ist der im Controller programmierte **Kopf** einer Datenübertragung eingetragen. Diesen gilt es nun im Datenstrom zu finden, denn nicht immer kommen die Daten so synchron wie erwartet an. Dazu wird mit zwei Radiobutton gearbeitet, die unserer Seite drei **Schnittstelle** hinzugefügt werden. **Rb_Data_Rec** und **Rb_Receive** sind die Namen. **Rb_Receive** signalisiert den Empfang von der Telegrammkennung und **Rb_Data_Rec** den Empfang aller Datenwerte.




*Daten eintragen*

Beim Start unseres Programmes setzen wir in der Initialisierungsroutine **Set_Default** die Textbox **Tb_Kennung** auf **leer** und **Tb_Empfang** auf **0**, Die Checkbox **Cb_Empf_Daten**, die wir noch hinter den Textfeld **Tb_Kennung** einfügen, setzen wir auf False.

```
Public Sub Set_Default()
    Tb_Kennung.Text= ""
    Cb_Emp_Daten.Checked = False
    TB_Empfang.Text = "0"
    Rb_ = False
    Data_Cnt = 0
    Read_Pointer = 0
    Write_Pointer = 0
    Rb_TriggerOff.Checked =True
```

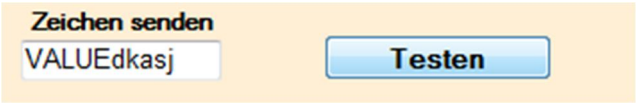
Nun erweitern wir die Timerroutine **Timer1_Tick**, um einen Empfang auszuwerten und gegebenenfalls den Datenstrom zu synchronisieren.

### *1.6.1 Programm testen*

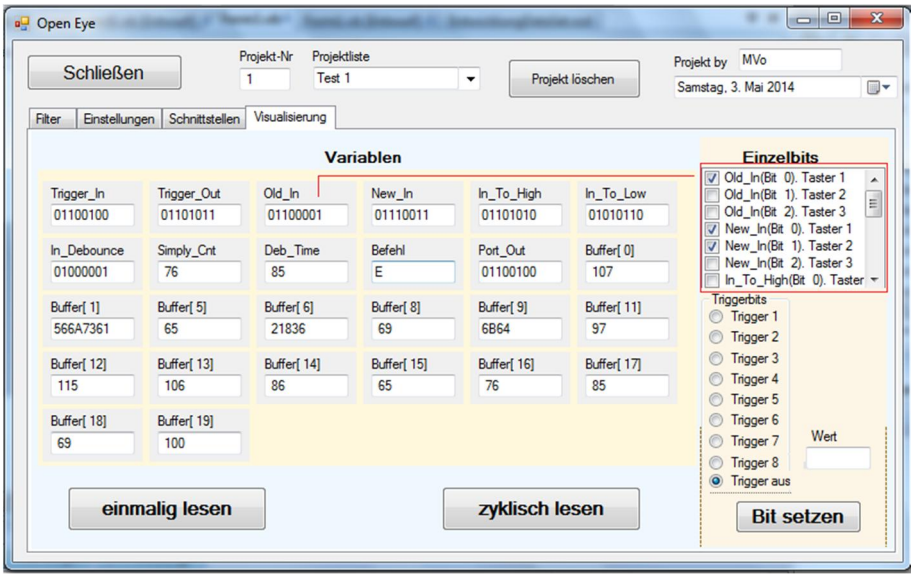
Bei allen folgenden Tests, die in den Anzeigeobjekten Werte darstellen sollen, ist nach dem Start **mit vorhandenen oder aus einem Assemblertext herausgefilterten und übernommenen Daten zu arbeiten**. Die folgenden Routinen greifen auf die Datenbank zu und benötigen die Information aus den Tableadaptern.

Wenn das Programm gestartet ist, kann auf der Seite zwei **Einstellungen** mit dem Button **Bt_Gen_Anzeige** der Aufbau in der Seite vier **Visualisierung** durchgeführt werden. Anschließend wird auf Seite drei **Schnittstellen** der Port aktiviert und bei gebrückten Kontakten 2 und 3 auf dem 9 pol. Sub-D Stecker solange das Button **Testen** betätigt, bis so viele Daten eingegangen sind, das vom Programm ein vollständiges Telegramm erkannt ist. Dies zeigt sich daran, dass die Anzeige **Empfangen** auf 0 gesetzt ist, obwohl noch scheinbar Daten eingetroffen sind. Keine Sorge, der Ringpuffer verliert keine Information. Der Text in der Textbox **Zeichen senden** sollte beginnend mit VALUE mit vielen beliebigen Zeichen gefüllt werden.

Nun können wir zur Seite vier **Visualisierung** wechseln und das Ergebnis prüfen.




*Test Empfang*




*Ergebnis Daten aus Übertragung*

### *1.6.2 Setzen des Wertefeldes*

Zuerst bekommt die Ereignisroutine **Timer1_Tick** eine lokale Variable. **Anz_Byte**. Zwei weitere Variablen Telekopf und Cnt_Data werden global am Anfang des Programms vereinbart, da sie für mehrere Ereignisse des Timers verfügbar sein müssen.

```
Public  Cnt_Data as Integer
Public Telekopf as String
```

Das Timerereignis Tick schalten wir erst einmal mit dem Befehl **Timer1.Enabled = false** ab, da diese neuen Programmschritte sonst nicht zu kontrollieren wären. Nun beginnt die bereits erstellte **While-Schleife**. Hier wird zuerst der Status vom Radiobutton **Rb_Received** abgefragt. Ist er gesetzt, kommen Daten an. Also werden die Daten in das ByteArray **Werte_Feld** geschrieben und der Datenzähler **Cnt_Data** erhöht. Ist die Anzahl erreicht, die in der Textbox **Tb_Anz_Variablen** hinterlegt ist, wird der Status vom RadioButton **Rb_Received** zurückgenommen und der Status der Checkbox **Cb_Empf_Daten** gesetzt. Auch der Datenzähler **Cnt_Data** wird zurückgesetzt und auch die Variable Telekopf mit einem Leerstring gelöscht.

Ist der Kopf noch nicht erkannt, ist der Status vom RadioButton **Rb_Received** nicht gesetzt und der empfangene Wert wird dem **Telekopf** zugefügt. Nun kommt ein Vergleich, ob die Zeichenfolge in der Textbox **Tb_Kopf** enthalten ist. Wenn der Vergleich mit 0, also nicht enthalten, beantwortet wird, löschen wir den Inhalt der Variablen **Kennung** und beginnen von vorn. Ist der Vergleich positiv, prüfen wir, ob die Vorgabe in der Textbox **Tb_Kopf** erfüllt ist. Dann wird das Radiobutton **Rb_Received** gesetzt und der Inhalt von **Kennung** gelöscht. Nach diesen Schritten wird die Variable der Textbox **Tb_Kennung** wieder zugewiesen.

```
Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Timer1.Tick
    Dim AnzByte As Integer
    Timer1.Enabled = False
    AnzByte = Write_Pointer - Read_Pointer
    While Write_Pointer <> Read_Pointer
      If Rb_Received.Checked Then
        Werte_Feld(Cnt_Data) = Rec_Feld(Read_Pointer)
        Cnt_Data = Cnt_Data + 1
        If Cnt_Data = Val(TB_Anzahl_Var.Text)-1 Then
```

```
            Cnt_Data = 0
            Telekopf = ""
            Rb_Received.Checked = False
            Cb_Emp_Daten.Checked = True
         End If
      Else
         Telekopf = Telekopf + Chr(Rec_Feld(Read_Pointer))
         If InStr(TB_Kopf.Text, Telekopf) = 0 Then
            Telekopf = ""
         Else
            Rb_Received.Checked = (Telekopf = TB_Kopf.Text)
         End If
         TB_Kennung.Text = Telekopf
      End If
      RT_Integer.Text = RT_Integer.Text + Str(Rec_Feld(Read_Pointer)) + ";"
      RT_Hex.Text = RT_Hex.Text + IntToHex(Rec_Feld(Read_Pointer), "Byte") + ";"
      Read_Pointer = Read_Pointer + 1
      If Read_Pointer > 999 Then Read_Pointer = 0
   End While
   TB_Empfang.Text = Str(Cnt_Data)
   Timer1.Enabled = True
End Sub
```

Nun ist nach dem Durchlauf der Routine **Timer1_Tick** der Wertepuffer aufbereitet und der Datenstrom synchronisiert. Wir können das Testen indem wir das Programm starten und auf die Radiobutton achten. Am Ende der Routine geben wir den **Timer** wieder frei.

### *1.6.3 Werte in Anzeige übertragen*

Jetzt wird es richtig spannend. Hat sich die Arbeit gelohnt und waren unsere Überlegungen richtig? Wir werden es gleich wissen, denn wir haben die Bestätigung, dass der Datenempfang mit dem **Timer** ganz gut funktioniert. Und wir haben, wenn auch vielleicht gar nicht bemerkt, ein Ereignis, das uns signalisiert, die Daten können in die Anzeige übertragen werden. Es ist das Radiobutton, welches nach Erhalt aller Werte auf **Checked= True** gesetzt wird. Und das löst ein **CheckedChange**-Ereignis aus. Deshalb werden wir die Wertezuweisung aus dem Timerereignis herausnehmen und das **CheckedChanged**-Ereignis bearbeiten.

Zuerst wieder die erforderliche Deklaration lokaler Variablen. Da ist eine Variable, die die Werteablage adressiert. Da die Anzahl der Anzeigeobjekte durch die Formate der 16 und 32 Bit-Variablen nicht übereinstimmt, müssen wir für die Anzeigen einen eigenen Zähler haben. Für die Wertezuweisung an die Textbox **Tb_In** haben wir einen String vorgesehen, deshalb ist auch hier eine Variable für den Wertestring erforderlich. Schließlich brauchen wir eine Variable vom Typ **Valuebox**, die dem **Control** des Panel **Pn_Variablen** die Werte übergibt.

Beginnen wir wie immer, wenn etwas Neues auf uns zukommt, mit dem Gerüst der Subroutine. In diesem Fall ist es die Ereignismethode **Rb_Data_Rec_CeckedChange**. Das erhalten wir direkt mit einem Doppelklick auf das Radiobutton. Dann fügen wir auch gleich die Variablendeklaration hinein und die Abfrage, ob der Status gesetzt ist, denn das Ereignis tritt auch auf, wenn der Status gelöscht wird.

```
Private Sub Rb_Data_Rec_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Rb_Data_Rec.CheckedChanged
  Dim Data_Cnt As Integer          ' Zähler für Wertearray
  Dim Feld_Cnt As Integer          ' Zähler für Anzeigeobjekte
  Dim Anzahl As String            ' Anzahl für Vergleichswert
  Dim Format As String            ' Format aus der Tabelle
  Dim Wertsatz As String          ' Zusammenstellung Werte für das Anzeigeobjekt
  Dim Trigger As Byte             ' Trigger_Out, der zweiten empfangenen Wert
  Dim Feld As ValueBox            ' Referenzvariable für das akt. Anzeigeobjekt
  If Rb_Data_Rec_Checked then

  End if
End Sub
```

Nun beginnen wir mit den Startwerten der beiden Zähler und setzen sie auf 0.

```
Data_Cnt = 0                    ' Zähler beginnen bei 0
Feld_Cnt = 0
```

Es folgt die Zuweisung des Grenzwertes aus der Textbox **Tb_Anzahl_Var** an die Variable Anzahl. Schließlich wird der zweite Wert aus dem **Werte_Feld** der Variablen **Trigger** zugeordnet. Das ist der Wert von **Trigger_Out**, den der Controller bei Bearbeitung eines Triggerereignisses zurückliefert. Die Variable ist nicht unbedingt erforderlich, da die Zuweisung von **Werte_Feld(1)** direkt erfolgen kann. Zum Verständnis ist die gesonderte Zuweisung einfacher. Dann erfolgt der Start einer **While-Schleife**. Eine For-Next kommt hier nicht in Frage, da die Werte zwar nacheinander aus dem Wertearray geholt werden, aber die einzelnen Anzeigen unterschiedliche Anzahl von Variablenwerten aufnehmen. Anschließend werden die Formatbereiche bearbeitet. Das Format holen wir aus der Tabelle entsprechend mit der Variablen **Data_Cnt**. Die gesperrten Zeilen in der Tabelle werden nicht berücksichtigt.

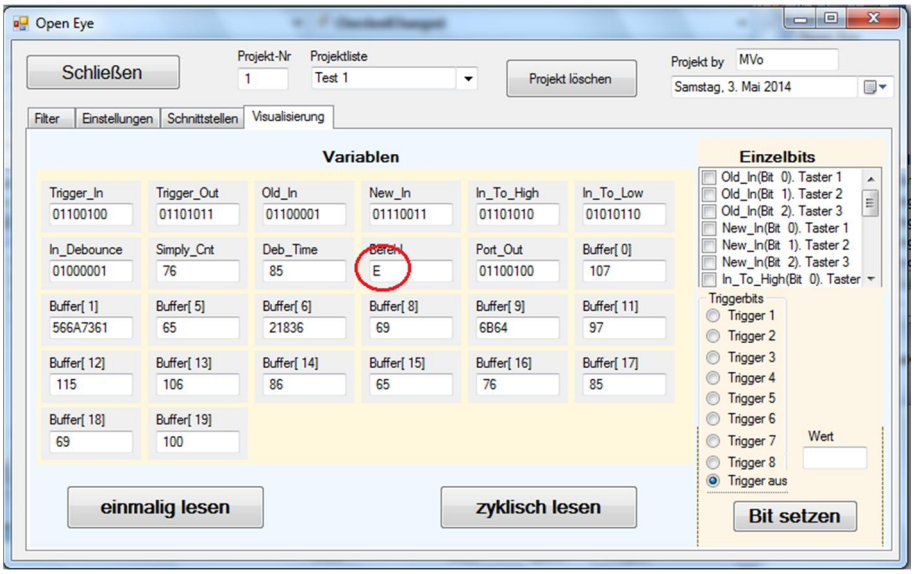
Das aktuelle Anzeigeelement ist in den **Controls** vom Panel **Pn_Variablen** abgelegt und kann der Variablen **Feld** einfach mit dem Zählerwert von **Feld_Cnt** zugewiesen werden. **Feld_Cnt** wird nach jedem Schleifendurchlauf erhöht und **Data_Cnt** bei jedem Zugriff auf **Werte_Feld**.

```
Anzahl = Val(TB_Anzahl_Var.Text)    ' Genzwert für Schleife
Trigger = Werte_Feld(1)     ' Wert von Trigger_Out
' keine for next, da Data_Cnt unterschiedliche Schrittweite
While Data_Cnt < Anzahl
  Wertsatz = ""                 ' beginnen mit leerem Wertesatz
  Format = DG_Variablen.Item("CL_Format", Data_Cnt).Value
  If Format = "Byte" Or Format = "Int8" Or Format = "Hex8" Or Format = "ASCII" Then
    Wertsatz = Str(Werte_Feld(Data_Cnt))    ' Wertesatz nur ein Byte
    Feld = Pn_Variablen.Controls(Feld_Cnt)  ' Variable ist aktuelles Anzeigeobjekt
   Feld.Triggerwert = Trigger
  Feld.TB_In.Text = ""                    ' Änderung erzwingen
   Feld.TB_In.Text = Wertsatz              ' dann  Wertesatz
   Data_Cnt = Data_Cnt + 1                 ' Datenzähler hochsetzen
 End If
 If Format = "Int16" Or Format = "Hex16" Then
   Wertsatz = Str(Werte_Feld(Data_Cnt))
```

```
    Wertsatz = Wertsatz + ";" + Str(Werte_Feld(Data_Cnt + 1)) ' Wertesatz 2 Byte
    Feld = Pn_Variablen.Controls(Feld_Cnt)
    Feld.Triggerwert = Trigger
   Feld.TB_In.Text = ""                    ' Änderung erzwingen
    Feld.TB_In.Text = Wertsatz
    Data_Cnt = Data_Cnt + 2                 ' Datenzähler zwei hochsetzen
  End If
  If Format = "Int32" Or Format = "Hex32" Then
    Wertsatz = Str(Werte_Feld(Data_Cnt))
    Wertsatz = Wertsatz + ";" + Str(Werte_Feld(Data_Cnt + 1))
    Wertsatz = Wertsatz + ";" + Str(Werte_Feld(Data_Cnt + 2))
    Wertsatz = Wertsatz + ";" + Str(Werte_Feld(Data_Cnt + 3))  ' Wertesatz 4 Byte
    Feld = Pn_Variablen.Controls(Feld_Cnt)
    Feld.Triggerwert = Trigger
    Feld.TB_In.Text = ""                    ' Änderung erzwingen
    Feld.TB_In.Text = Wertsatz
    Data_Cnt = Data_Cnt + 4                 ' Datenzähler vier hochsetzen
  End If
  Feld_Cnt = Feld_Cnt + 1                   ' Anzeigezähler nachführen
End While
Rb_Data_Rec.Checked=False
```

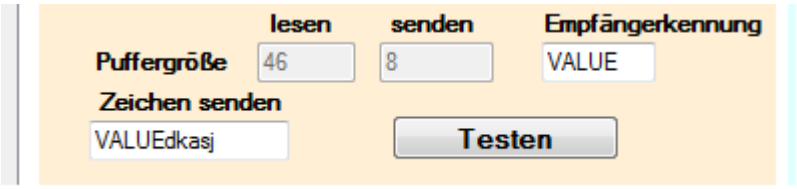
Schließlich, wenn die Bearbeitung erfolgt ist, wird der Status von **Rb_Data_Rec** zurückgesetzt.

Nun wird wieder getestet. Sind die Werte auch richtig? Speziell bei den zwei- und vierbyte Formaten muss genau hingesehen werden. Da ist eine Übertragung vom Controller, der von Speicherzelle 0 bis zur letzten Variable die Daten verschickt. Das bedeutet, die Speicherzellen mit den niederen Adressen kommen zuerst an. Der Wertesatz wird auch von unten nach oben aufgebaut, also 1.Wert, 2. Wert, 3. Wert und 4. Wert. Im Ereignis **Tb_In_TextChanged** ist 1.Wert*256^0, 2. Wert*256^1 usw. Somit wird auch das Ergebnis richtig berechnet. Theoretisch jedenfalls. Die Bytefolge haben wir ja bereits überprüft. Sehen wir uns nun das Ergebnis im Screenshot an




*Daten überprüfen*

.Ich habe ein Feld markiert. Es zeigt ein großes E an. Das können wir leicht überprüfen, ob es stimmt. Das ist der Befehl, den ich gesendet habe:




*Zeichen gesendet*

Das erste VALUE wird entfernt, weil es die Kopfkennung ist. Also werden dkasjVALUEdkasj usw. Empfangen, wenn ich senden mehrmals betätige. Da ein vollständiger Empfang mit den Variablen vom Controller, bzw. den Tabelleneinträgen in Dg_Variablen abgeglichen wird, ist die Zeichenfolge Bestandteil von Daten und kein neuer Kopf. Zählen wir nun nach und vergleichen, so ist im 10. Feld auch das zehnte, dem Datenempfang zugeordnete Zeichen ein E. Die Aufgabe scheint gelöst. Zumindest die Werte in den Ausgabeboxen passen. Würden wir alle mit dem Format ASCII belegen, passten auch die Zeichen.

Am Anfang des Kapitels habe ich die Effizienz angesprochen. Allerdings ist diese Bearbeitung nicht besonders effizient. Seht euch mal die Bytezählerei an. Klar, hier werden lokalen Variablen eingesetzt und wir wissen, dass diese nach Verlassen der Subroutine nicht mehr gültig sind. Aber irgendwie müssen die Werte erhalten bleiben, denn beim nächsten Aufruf der Subroutinen soll es ja mit der Datenzählerei weitergehen. So ist Data_Cnt im Timer typisch. Oder auch die Variable Kennung. Niemand kann sagen, ob ein gesamtes Telegramm komplett in einem Timer-Ereignis abgearbeitet werden kann, oder ob evtl. mehrere Aufrufe erfolgen. Also müssen die Inhalte aktueller Zähl- und Vergleichswerte erhalten bleiben, bis der Datensatz komplett ist. Bei genauer Betrachtung ist nur die TextBox Tb_Kennung betroffen. Sie wird bei jedem Schleifendurchlauf gelesen und wieder beschrieben. Data_Cnt wird nur am Anfang der Timer-Routine gelesen und nach der While Schleife wieder beschrieben. Damit kann man leben. Wenn wir nun auf eine globale Variable Kennung zugreifen, dann können wir uns die lokale sparen und die Textbox Tb_Kennung dann beschreiben, wenn der Kopf komplett zusammengestellt wird. Das dürfte ein wenig die Geschwindigkeit der Bearbeitung erhöhen. Ändern wir also das Programm entsprechend ab.

Deklarieren der globalen Variable

```
'*****************************************************************************
'**          globale Variablen und Variablentypen
'*****************************************************************************
Public Rec_Feld(999) As Byte
Public Werte_Feld(999) As Byte
Public Read_Pointer As Integer
Public Write_Pointer As Integer
Public Telekopf As String
```

Das Timer1_Tick Ereignis

```
Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Timer1.Tick
    Dim AnzByte As Integer
    Dim Data_Cnt As Integer
    If Not Rb_Data_Rec.Checked Then
 ' nur bearbeiten, wenn keine Werte zur Anzeige kopiert werden
      Timer1.Enabled = False
      AnzByte = Write_Pointer - Read_Pointer
      Data_Cnt = Val(TB_Empfang.Text)
      While Write_Pointer <> Read_Pointer
        If Rb_Received.Checked Then
          Werte_Feld(Data_Cnt) = Rec_Feld(Read_Pointer)
          Data_Cnt = Data_Cnt + 1
          If Data_Cnt = Val(TB_Anzahl_Var.Text)-1 Then
            Data_Cnt = 0
            Rb_Received.Checked = False
            Rb_Data_Rec.Checked = True
          End If
        Else
          Telekopf = Telekopf + Chr(Rec_Feld(Read_Pointer))  'ist globale Variable
          If InStr(TB_Kopf.Text, Telekopf) = 0 Then
            Telekopf = ""
          Else
            If Telekopf = TB_Kopf.Text Then
              Rb_Received.Checked = True
              TB_Kennung.Text = Telekopf
            End If
          End If
        End If
        RT_Integer.Text = RT_Integer.Text + Str(Rec_Feld(Read_Pointer)) + ";"
        RT_Hex.Text = RT_Hex.Text + IntToHex(Rec_Feld(Read_Pointer), "Byte") + ";"
        Read_Pointer = Read_Pointer + 1
        If Read_Pointer > 999 Then Read_Pointer = 0
      End While
      TB_Empfang.Text = Str(Data_Cnt)
      Timer1.Enabled = True
    End If
  End Sub
```

Dabei ist zu beachten, dass keine weiteren Daten aus dem Ringpuffer bearbeitet werden, solange die Ereignisroutine **RB_Data_Cnt_Checked** damit beschäftigt ist, die Werte zur Anzeige zu bringen. Deshalb wird für diese Zeit die Bearbeitung von **Timer1_Tick** ausgesetzt.

Und schließlich wird noch das Radiobutton **Rb_Received** in der Ereignisroutine **Rb_Data_Rec** nach Bearbeitung zurückgesetzt.

```
While Data_Cnt < Anzahl
    ......
End While
Rb_Data_Rec.Checked = False
Rb_Received.Checked = False
```

### *1.6.4 Status Einzelbit anzeigen*

Nun müssen noch die Einzelbits aktualisiert und ihr Status sichtbar werden. Der Einbau dieser Aufgabe erfolgt in der Ereignisroutine **Rb_Data_Rec_CheckedChanged.** Hier haben wir die Auswertung des Formates in der Wertezuweisung und können direkt in diesem Bereich eine Subroutine aufrufen, die die Checkboxen in der Checkboxliste entsprechend bedient. Was brauchen wir für Informationen? Nun, ein Byte mit den Bits wäre nicht schlecht, am besten gleich als String. Den haben wir nach der Zuweisung an **Tb_In** der Variable **Feld** in der **Tb_Out** vorliegen. Dann brauchen wir noch die Zeilennummer der Tabelle **Dg_Variablen**. Diese Information liegt in der Variablen **Data_Cnt** vor. Auch wissen wir, dass es bis zu acht Checkboxen für eine Variable gibt. Dafür benötigen wir einen Schleifenzähler.

Damit können wir einen Rahmen für eine Subroutine **Set_CheckBit** schreiben.

```
Public Sub Set_CheckBit(ByVal Zeile_Nr As Integer, ByVal ByteWert As String)
   Dim i As Integer

 End Sub
```

Um der Checklistbox einen Status zuweisen zu können ist eine boolesche Variable sinnvoll. Für die Referenz der Listen eine Variable als String, um den Eintrag Zeilennummer und Bitnummer zu finden und eine als Integer, um den gefundenen Index auf die CheckListbox zu übertragen. Fügen wir also weitere lokale Variablen ein und bauen die Schleife auf. Hier ist eine feste Schleife angebracht, weil ja grundsätzlich alle 8 Bits geprüft werden müssen. Aufgrund unserer Struktur in der Ablage kann ja sein, das Bit 2, 4, 6 und 7 spezifiziert und die anderen nicht in der Datenbanktabelle enthalten sind. Auch eine While-Schleife müsste alle acht Bits prüfen. Alternativ könnte mit der Datenbankabfrage natürlich die Bitliste geholt werden, aber das würde wesentlich mehr Zeit in Anspruch nehmen. Deklarieren wir die weiteren lokalen Variablen und schreiben den Rahmen der Schleife von 0 bis 7, da die Bits auch von 0 bis 7 nummeriert werden.

```
   Dim Ref_Str As String
   Dim Ref_Index As Integer
   Dim Set_Chk As Boolean
   For i = 0 To 7

  Next
```

Bei der weiteren Programmierung ist etwas besonders zu beachten. Die Bits im Byte fangen bei 0 an zu zählen. So stellt sich auch der String mit dem Bitmuster dar, aber, das erste Zeichen im String ist links und nicht rechts und die Zählung beginnt mit 1. Doch mit einem kleinen mathematischen Trick lässt sich auch dieses Problem beheben. Zuerst prüfen wir, ob in der Referenzliste ein Eintrag mit Zeilennummer und Bitnummer gegeben ist. Dazu haben wir der Subroutine die Zeilennummer der Variablen mitgegeben, die Bitnummer entstammt dem Schleifenzähler. Mit diesen beiden Werten schaffen wir uns einen String, der als Suchbegriff auf die unsichtbare Referenzliste **Lb_Bit_Ref** mit **IndexOf** zugreift und einen Index zurückliefert. Nicht negativ, also >= 0 entspricht einem gefundenen Referenzwert.

```
Ref_Str = Str(Zeile_Nr) + "." + Str(i)
Ref_Index = Lb_Bit_Ref.Items.IndexOf(Ref_Str)
If Ref_Index >= 0 Then

End If
```

Es ergibt sich nun die Aufgabe, bei einem gefundenen Referenzwert den Status **True** oder **False** entsprechend dem Bit im Übergabewert zu setzen. Eine **1** bedeutet, die Checkbox in der Liste soll gesetzt sein. Also müssen wir das Ergebnis für unsere boolesche Variable vom Wert des adressierten Zeichens abhängig machen. Das Zeichen können wir mit **Mid(<Text>,<Anfang>,<Anzahl>)** adressieren. Da wir wissen, das der übergebene String acht Zeichen lang ist, können wir mit **Anfang = 8 - i** den Text von hinten her absuchen.

Somit ist die Zuweisung an die boolesche Variable eigentlich schon klar

```
Set_Chk = Mid(ByteWert, 8 - i, 1) = "1"
```

Der Vergleich **Mid(ByteWert, 8-I,1) = 1** liefert eine Antwort. Wenn es stimmt, ist die Antwort **Wahr** oder wenn nicht dann halt **Falsch**. Damit hat die boolesche Variable ihren Wert und kann nun der Checklistbox zugewiesen werden

```
Clb_Bit_Info.SetItemChecked(Ref_Index, Set_Chk)
```

Zusammenhängend und mit ein paar Kommentarzeilen ergänzt ist die Subroutine übersichtlich.

```
Public Sub Set_CheckBit(ByVal Zeile_Nr As Integer, ByVal ByteWert As String)
    Dim i As Integer
    Dim Ref_Str As String          ' Referentext für Index der Liste
    Dim Ref_Index As Integer       ' Index Listeneintrag
    Dim Set_Chk As Boolean         ' Hilfsvariable für Checkbox
    For i = 0 To 7
      Ref_Str = Str(Zeile_Nr) + "." + Str(i)                ' Referenztext bilden
      Ref_Index = Lb_Bit_Ref.Items.IndexOf(Ref_Str)  ' Index holen
      If Ref_Index >= 0 Then                        ' wenn Index gültig
        Set_Chk = Mid(ByteWert, 8 - i, 1) = "1"    ' Text von rechs nach Links prüfen
        Clb_Bit_Info.SetItemChecked(Ref_Index, Set_Chk) ' Status Checkbox zuweisen
      End If
    Next
End Sub
```

Schließlich wird noch der Aufruf dieser Subroutine in das Ereignis des Radiobutton **Rb_Data_Rec_CheckedChanged** im Bereich der Byte-Formate eingebunden.

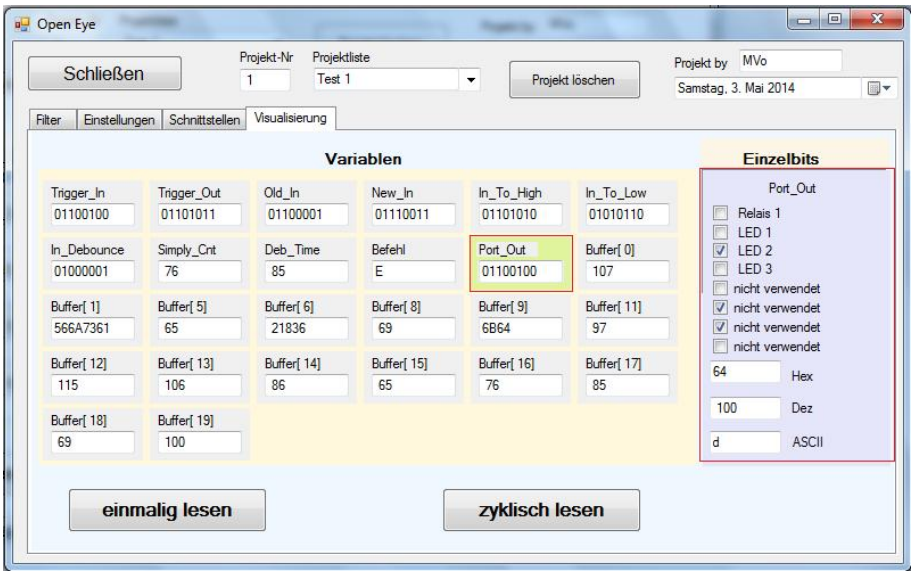
```
        If Format = "Byte" Or Format = "Int8" Or Format = "Hex8" Or Format = "ASCII" Then
            Wertsatz = Str(Werte_Feld(Data_Cnt))    ' Wertesatz nur ein Byte
            Feld = Pn_Variablen.Controls(Feld_Cnt)  ' Variable ist aktuelles Anzeigeobjekt
            Feld.Triggerwert = Trigger
           Feld.TB_In.Text = ""                     ' Änderung erzwingen
            Feld.TB_In.Text = Wertsatz
            If Format = "Byte" Then
              Set_CheckBit(Data_Cnt, Feld.TB_Out.Text)     ' Status Einzelbits setzen
            End If
            Data_Cnt = Data_Cnt + 1              ' Datenzähler hochsetzen
        End If
```

Nun können wir wieder den Fortschritt testen und prüfen, ob unsere Erwartungen an das Programm erfüllt werden.

### *1.6.5 Gezielte Bitinformation*

Ein kleines Schmankerl hab ich noch für diese Seite. Beim Durchblick der Ereignisse von Objekten findet man einen Eintrag, der **MouseHover** heißt und einen **MouseLeave**. Mit diesen beiden Ereignissen möchte ich erreichen, das, wenn der Mauszeiger über einem Anzeigeobjekt bleibt, ein Fenster aufgeblendet wird und nur diese Bits im Status und mit Beschreibung anzeigt und zusätzlich vielleicht noch die anderen Formatdarstellung.

So, wie ich es jetzt einmal im Screenshot vorstelle.




*Anzeige Einzelbit spezial*

Kommt der Mauszeiger in den grünlich markierten Bereich, soll das fliederfarbene Objekt geöffnet und den Wert der Variablen Port_Out mit allen Einzelbits sowie in den Formaten Hex, Int und ASCII anzeigen. Verlässt der Mauszeiger das Anzeigefeld, soll diese Information wieder ausgeblendet werden. Um das zu erreichen, müssen wir unserem selbstgestrickten Objekt noch eine Variable verpassen, damit die Information aus der Datenbank geholt werden kann. Hier handelt es sich um eine manuelle Tätigkeit, und da ist die Verarbeitungsgeschwindigkeit nicht das Maß der Dinge. Der Datenbakzugriff die einfachste Art, alle Informationen in das UserControl zu holen.

Auch der Parametersatz zur Installation des UserControl muss um eine Variable erweitert werden. Beginnen wir beim Parametersatz.

```
Public Structure Var_Rec
    Dim Var_Id As Integer      ' Neu hinzugekommen
    Dim Name As String         ' diese und  alle weiteren bleiben
```

Nun ist in der Ereignisroutine vom Button Bt_GenAnzeige_Click in der Generierungsschleife die Id_Nr der Variablen mit zu übergeben.

```
For i = 0 To DG_Variablen.Rows.Count - 2
 If DG_Variablen.Item("CL_Freigabe", i).Value = "Ja" Then
   Par_Satz.Var_Id = DG_Variablen.Item("Cl_Id_Nr", i).Value   ' Übergabe ID-Nr der Variablen
   Par_Satz.Aktiv = DG_Variablen.Item("CL_Aktiv", i).Value = "Ja"
            ' bildet einen boolschen Ausdruck aus “ja”.
   Par_Satz.Format = DG_Variablen.Item("CL_Format", i).Value
            ' kopiert das Format aus der Tabellenzeile.
   Par_Satz.Trigger = Val(DG_Variablen.Item("CL_Trigger", i).Value)
   If Par_Satz.Trigger > 0 Then  ' Bitnummer in Bit darstellen
     Par_Satz.Trigger = 2 ^ Par_Satz.Trigger - 1  ' Bit setzen
   End If
            ' kopiert den Triggerwert aus der Tabellenzeile
   Par_Satz.Name = DG_Variablen.Item("CL_Variable", i).Value
            ' kopiert den Variablennamen aus der Tabellenzeile
   Generate_ValueBox(Par_Satz)    ' jetzt mit der Id_Nr der  Variablen
```

In der Subroutine Generate_ValueBox weisen wir dann der generierten Valuebox diesen Wert zu.

```
Public Sub Generate_ValueBox(ByVal Var_Daten As Var_Rec)
    Dim My_Textfeld As ValueBox            ' Die Variable My_Textfeld wird als Typ Valuebox
    My_Textfeld = New ValueBox             ' lokal deklariert. Mit “New Valuebox” wird das
Objekt erzeugt
    My_Textfeld.Parent = Pn_Variablen      ' und Panel “Pn_Anzeige” zugeordnet
    My_Textfeld.Variable_Id = Var_Daten.Var_Id '+++++ neu ++++++
    My_Textfeld.Name = "VB_" + Var_Daten.Name      ' Der Name des Objektes wird
vergeben, bezogen auf die Variable
    My_Textfeld.Lb_Variable.Text = Var_Daten.Name   ' Der Labeltext bekommt den
Variablennamen zugewiesen
```

Da dieser Parameter aber noch nicht existiert, wird hier keine Hilfe angeboten und der eingefügte Text mit einem blauen Unterstrich markiert. Das ist immer ein Zeichen, da stimmt etwas nicht. Ist ja auch klar, den Parameter müssen wir auch erst einrichten.

Bleiben wir aber erst einmal im Bereich von **Open_Eye** und ziehen uns auf die Seite **Visualisierung** ein Panel und vergeben den Namen **Pn_Bitanzeige**. Die Farbe für **Backcolor** habe ich auf **Lavender** gestellt, ist aber nicht zwingend vorgegeben. Hier ist der eigene Geschmack gefragt. Allerdings sollte es sich schon ein wenig von den anderen Elementen abheben. Damit es nicht im Weg ist, setzen wir das Panel ganz nach rechts über den Bereich der Einzelbitliste. Aber zum Einrichten können wir es erst einmal auf der Mitte der Seite belassen. Oben kommt ein Label mit dem Namen **Lb_Name** hinein. Es bekommt später den Variablennamen zugewiesen. Dann folgen 8 Checkboxen., die die Namen **Cb_Bit0** bis **Cb_Bit7** erhalten. Schließlich setzen wir drei Textboxen mit den Namen **Tb_WertHex**, **Tb_WertInt** und **Tb_WertASCII** ein und beschriften mit drei Labels die Textboxen. Diese Labels werden nicht verändert und brauchen deshalb auch keinen Namen. Die Größe des Panels ist etwa 260 Pixel hoch und 165 Pixel breit. (Eigenschaft **Size Height** und **Width**)

Damit ist das Panel eingerichtet und wir wechseln nun in den Editor von **UserControl**. Dort fügen wir zuerst die Variable **Variable_Id** ein, die auf die **Id_Nr** der Datenbankvariablen bezogen ist.

```
Public Class ValueBox
   Public Format As String
   Public Variable_Id As Integer              ' neu hinzugekommen
   Public Triggerwert As Byte
   Public Triggerlevel As Byte
```

Anschließend schaffen wir uns gleich die Rahmen für die beiden Ereignisroutinen **MouseHover** und **MouseLeave**

```
Private Sub ValueBox_MouseHover(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.MouseHover
      If Format = "Byte" Then
      End If
 End Sub
```

```
Private Sub ValueBox_MouseLeave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.MouseLeave


End Sub
```

Auch wenn wir noch gar nicht wissen, wie wir die Information in das Panel **Pn_Bitanzeige** bekommen, klar ist, beim Verlassen soll es wieder ausgeblendet werden. Die Zuweisung erfolgt wieder mit dem Namen der Anwendungsoberfläche **Frm_Open_Eye**. Damit ist die Ereignisroutine **MouseLeave** erledigt.

```
Private Sub ValueBox_MouseLeave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.MouseLeave
    Frm_Open_Eye.PN_BitAnzeige.Visible = False
    Frm_Open_Eye.Pn_Trigger.Visible = True
   ' wenn Ansicht stört und in  MouseHover  ausgeblendet wird
End Sub
```

Genau so klar ist, dass dieses Panel sichtbar werden soll, wenn das Ereignis MouseHover eintritt und das Format des Anzeigeobjektes Byte ist.. Also kommt in diese Ereignismethode auch die Zuschaltung mit

```
    Frm_Open_Eye.PN_BitAnzeige.Visible = True
```

In die Subroutine. Dies testen wir erst einmal und starten das Programm. Das Panel sollte sichtbar und auch wieder unsichtbar werden, oder? Nun, bei mir war nicht gleich der Erfolg da und es ist durchaus möglich, dass es euch ähnlich ergeht, aber keine Panik. Geben wir dem Panel eine Chance, sich ganz oben auf den vielen Objekte auch zu zeigen. Zuerst ordnen wir es mit der Eigenschaft Parent der Seite Tp_Visu .zu

```
Frm_Open_Eye.PN_BitAnzeige.Parent = Frm_Open_Eye.Tp_Visu
```

Dann stellen wir es mit BringToFront ganz nach oben

```
     Frm_Open_Eye.PN_BitAnzeige.BringToFront()
```

Und schließlich schicken wir es an den rechten Rand, damit es uns nicht die Sicht und den Zugang zu den anderen Anzeige-Objekten versperrt

```
        Frm_Open_Eye.PN_BitAnzeige.Left = 590
        Frm_Open_Eye.PN_BitAnzeige.Top = 34
```

Die bisherige Zusammenstellung der Befehle

```
Private Sub ValueBox_MouseHover(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.MouseHover
    If Format = "Byte" Then
        Frm_Open_Eye.PN_BitAnzeige.Left = 590
        Frm_Open_Eye.PN_BitAnzeige.Top = 34
        Frm_Open_Eye.PN_BitAnzeige.Parent = Frm_Open_Eye.Tp_Visu
        Frm_Open_Eye.PN_BitAnzeige.BringToFront()
    End If
End Sub
```

Diesmal sollte der Test auch das erwartete Ergebnis liefern.

Nun müssen wir zuerst ein Array mit **nicht verwendet** füllen. Der Grund: Die Datenbankabfrage liefert uns nur Einträge, die eben nicht den Inhalt haben **nicht verwendet**. Diese Bits sind nicht gespeichert. Aber, die Datenbank liefert die Bitnummer zu den geführten Informationen und damit werden wir in einer zweiten Schleife die Zellen im Array adressieren und überschreiben.

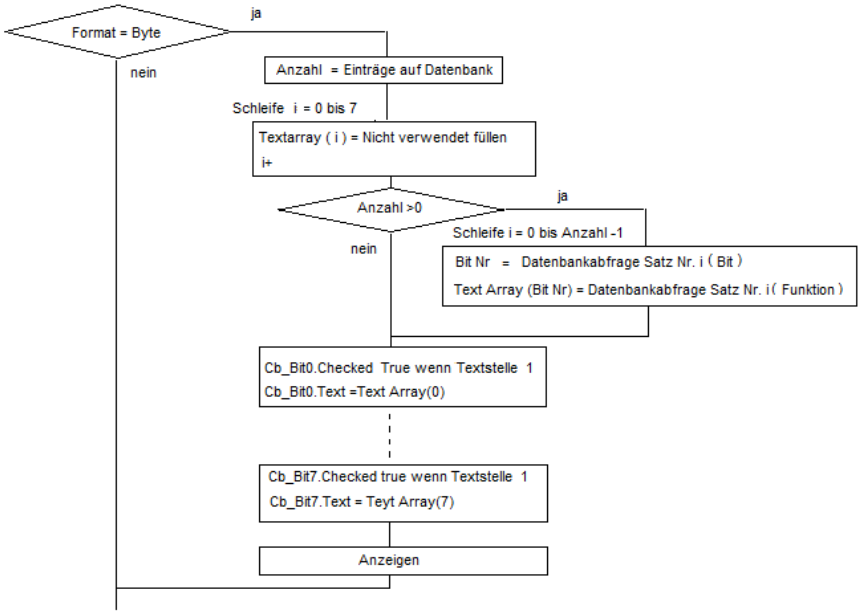
Legen wir nun erst einmal die lokalen Variablen fest. Wie immer bei einer Aktion, die mehrere Durchläufe erfordert kommt eine Schleife in Betracht und wir brauchen einen Schleifenzähler. Dann haben wir eine Datenbankabfrage und für das Ergebnis ist die Variable Anzahl schon mehrfach aufgetaucht.

Dann liefert uns die Datenbank auch noch die Bitnummer bei vorhandenen Einträgen, die wir zur Adressierung brauchen und schließlich noch ein Textarray für die Bitbeschreibung.

```
Dim Anzahl As Integer
Dim i As Integer
Dim InfoNr As Integer
Dim InfoText(7) As String
```

Mit diesen vier Variablen sollte nun die Programmstruktur aufzubauen sein. Bevor wir zur uns der Programmierung zuwenden, sollten wir erst einmal einen Programmablauf skizzieren

.




*PAP Einzelbits bewerten*

Ein solcher Ablaufplan oder PAP, wie er manchmal auch genannt wird ordnet die Sinne. Er ist nicht immer notwendig, aber wenn mal eine ziemlich komplizierte Sache angegangen wird, ist ein PAP immer ein gutes Mittel, Strukturen herauszuarbeiten und Aufgaben kleinen Abschnitten, hier sind es Kästen, zuzuordnen. Entscheidungswege werden sichtbar und die erforderlichen Aufgaben. Nun, diesen PAP habe ich einfach mal so dahingeschludert. Er ist für mich, nicht für eine Programmdokumentation, die ich aus der Hand gebe. Dann müsste ich mich nach den Regularien und Vorgaben halten, genormte Symbole benutzen. Klar, dafür gibt es auch Software, aber für jemand, der sich hier ein Hobby gestaltet, sicherlich zu weit gegriffen. Und da reichen eben schnelle Skizzen völlig aus, um sich Durchblick zu verschaffen.

Gut, der linke Strang ist mit der Abfrage **If Format =Byte** abgehakt. Wenden wir uns dem Rechten Teil zu und laden zuerst einmal die **Anzahl** der Einträge an Einzelbits, die dieser Variable zugeordnet sind. Dann füllen wir mit einer **For .. Next** Schleife das Textarray **Infotext** mit **nicht verwendet.**

Auf eine Abfrage**,** ob Anzahl der gefundenen Datensätze >0 ist verzichte ich, da die zweite Schleife bei Anzahl = 0 sowieso nicht durchlaufen wird.

```
Anzahl =
 Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Rows.Count
 For i = 0 To 7
    InfoText(i) = "nicht verwendet"
  Next
  For i = 0 To Anzahl - 1
      InfoNr = Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Item(i).Bit
      InfoText(InfoNr) =
     Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Item(i).Funktion
  Next
```

Innerhalb der Schleife erfolgt der Zugriff auf den Datenbanksatz, der zuerst die Bitnummer erfragt und dann im zweiten Befehl die Funktion an die Stelle im Array InfoText schreibt, die mit der Bitnummer adressiert ist. Danach kann der Rest dann auch zugewiesen werden. Die Checkboxen erhalten als Text den Inhalt der jeweiligen Arrayzelle und der Status wird wieder mit einer Vergleichsoperation auf den String in **TB_Out** erledigt. Diese Textbox hat ja mittlerweile das Bitmuster zugewiesen bekommen.

Klar, hab ich fast vergessen, das Label. Auch der Name der Variablen liegt im Objekt vor und braucht nur dem Label **Lb_Name.Text** zugewiesen werden.

```
Frm_Open_Eye.LB_Name.Text = Lb_Variable.Text
Frm_Open_Eye.Cb_Bit0.Checked = Mid(TB_Out.Text, 8, 1) = "1"
Frm_Open_Eye.Cb_Bit0.Text = InfoText(0)
Frm_Open_Eye.Cb_Bit1.Checked = Mid(TB_Out.Text, 7, 1) = "1"
Frm_Open_Eye.Cb_Bit1.Text = InfoText(1)
Frm_Open_Eye.Cb_Bit2.Checked = Mid(TB_Out.Text, 6, 1) = "1"
Frm_Open_Eye.Cb_Bit2.Text = InfoText(2)
Frm_Open_Eye.Cb_Bit3.Checked = Mid(TB_Out.Text, 5, 1) = "1"
Frm_Open_Eye.Cb_Bit3.Text = InfoText(3)
Frm_Open_Eye.Cb_Bit4.Checked = Mid(TB_Out.Text, 4, 1) = "1"
Frm_Open_Eye.Cb_Bit4.Text = InfoText(4)
Frm_Open_Eye.Cb_Bit5.Checked = Mid(TB_Out.Text, 3, 1) = "1"
Frm_Open_Eye.Cb_Bit5.Text = InfoText(5)
Frm_Open_Eye.Cb_Bit6.Checked = Mid(TB_Out.Text, 2, 1) = "1"
Frm_Open_Eye.Cb_Bit6.Text = InfoText(6)
Frm_Open_Eye.Cb_Bit7.Checked = Mid(TB_Out.Text, 1, 1) = "1"
Frm_Open_Eye.Cb_Bit7.Text = InfoText(7)
```

Schließlich werden den drei Textboxen **Tb_WertHex**, **Tb_WertInt** und **Tb_WertASCII** noch der Wert der Variablen entsprechend gewandelt zugewiesen

```
Frm_Open_Eye.Tb_WertHex.Text = Frm_Open_Eye.IntToHex(Val(TB_In.Text), Format)
Frm_Open_Eye.Tb_WertInt.Text = TB_In.Text
Frm_Open_Eye.Tb_WertASCII.Text = Chr(Val(TB_In.Text))
        Frm_Open_Eye.PN_BitAnzeige.Parent = Frm_Open_Eye.Tp_Visu
        Frm_Open_Eye.PN_BitAnzeige.Left = 590
        Frm_Open_Eye.PN_BitAnzeige.Top = 34
        Frm_Open_Eye.PN_BitAnzeige.BringToFront()
        Frm_Open_Eye.Pn_Trigger.Visible = False
        Frm_Open_Eye.PN_BitAnzeige.Visible = True
```

Das Ergebnis sollte nun der Vorstellung im Screenshot von Seite 372 entsprechen.

Hier noch einmal die gesamte Ereignisroutine MouseHover

```
Private Sub ValueBox_MouseHover(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.MouseHover
```

```
    Dim Anzahl As Integer
    Dim i As Integer
    Dim InfoNr As Integer
    Dim InfoText(7) As String
    If Format = "Byte" Then        ' Format ist Objektvariable, nicht lokal
      Anzahl =
Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Rows.Count
      For i = 0 To 7
        InfoText(i) = "nicht verwendet"
      Next
      For i = 0 To Anzahl - 1
        InfoNr =
Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Item(i).Bit
        InfoText(InfoNr) =
Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Item(i).Funktion
      Next
      Frm_Open_Eye.LB_Name.Text = Lb_Variable.Text
      Frm_Open_Eye.Cb_Bit0.Checked = Mid(TB_Out.Text, 8, 1) = "1"
      Frm_Open_Eye.Cb_Bit0.Text = InfoText(0)
      Frm_Open_Eye.Cb_Bit1.Checked = Mid(TB_Out.Text, 7, 1) = "1"
      Frm_Open_Eye.Cb_Bit1.Text = InfoText(1)
      Frm_Open_Eye.Cb_Bit2.Checked = Mid(TB_Out.Text, 6, 1) = "1"
      Frm_Open_Eye.Cb_Bit2.Text = InfoText(2)
      Frm_Open_Eye.Cb_Bit3.Checked = Mid(TB_Out.Text, 5, 1) = "1"
      Frm_Open_Eye.Cb_Bit3.Text = InfoText(3)
      Frm_Open_Eye.Cb_Bit4.Checked = Mid(TB_Out.Text, 4, 1) = "1"
      Frm_Open_Eye.Cb_Bit4.Text = InfoText(4)
      Frm_Open_Eye.Cb_Bit5.Checked = Mid(TB_Out.Text, 3, 1) = "1"
      Frm_Open_Eye.Cb_Bit5.Text = InfoText(5)
      Frm_Open_Eye.Cb_Bit6.Checked = Mid(TB_Out.Text, 2, 1) = "1"
      Frm_Open_Eye.Cb_Bit6.Text = InfoText(6)
      Frm_Open_Eye.Cb_Bit7.Checked = Mid(TB_Out.Text, 1, 1) = "1"
      Frm_Open_Eye.Cb_Bit7.Text = InfoText(7)
      Frm_Open_Eye.Tb_WertHex.Text =
                                Frm_Open_Eye.IntToHex(Val(TB_In.Text), Format)
      Frm_Open_Eye.Tb_WertInt.Text = TB_In.Text
      Frm_Open_Eye.Tb_WertASCII.Text = Chr(Val(TB_In.Text))
      Frm_Open_Eye.PN_BitAnzeige.Parent = Frm_Open_Eye.Tp_Visu
      Frm_Open_Eye.PN_BitAnzeige.Left = 590
      Frm_Open_Eye.PN_BitAnzeige.Top = 34
      Frm_Open_Eye.PN_BitAnzeige.BringToFront()
      Frm_Open_Eye.Pn_Trigger.Visible = False
    ' wenn Ansicht stört, in MouseLeave wieder sichtbar schalten
```

```
        Frm_Open_Eye.PN_BitAnzeige.Visible = True
    End If
End Sub
```

### *1.6.6 Daten vom Controller anfordern*

Die Wertedarstellung ist abgeschlossen und nun ist es an der Zeit, die Arbeit mit einem Controller vorzubereiten. Betrachten wir einmal, welche Möglichkeiten gegeben sind. Eine kennen wir bereits. Das Button **Testen**. Es sendet alles, was im Textfeld **Zeichen senden** steht. Für unseren Controller möglicherweise nett, aber er wird damit so erst einmal nichts anfangen können. Wir werden vom ankommenden Datenstrom gar nicht so viele Bytes auswerten müssen, um über Befehle Aktionen auf einem Controller auszulösen. So reicht es **R1** und **r1** zu senden, um Relais 1 ein- und auszuschalten. Ein **V0** könnte einen Transfer der Variablen auslösen und ein **V+<Trigger>** den internen Trigger_In setzen, um in einem bestimmten Abschnitt im Programm darauf zu reagieren. Was auch immer für Aktionen im Controller geplant sind, der Auftrag muss beiden klar sein, dem PC wie auch dem Controller. So ist auch die Telegrammkennung ein wichtiger Bestandteil und wird auch aus diesem Grund auf der Datenbank in den Projektdaten abgelegt. Kommen wir nun zu den Button **Bt_Single** und **Bt_Zyklus**.

Mit diesen Button werden auch die Zeichen in dem Textfeld versendet, doch sie sollen nun eindeutig dem Controller sagen, liefere mir deine Variableninhalte. Also zum Beispiel **V0**. Deshalb muss nach Abschluss von den Tests im Textfeld **Zeichen senden** ein Eintrag stehen, den der Controller entsprechend deuten kann. Auch dieser Wert gehört zu den Projektdaten. Es ist zwar frei, den Sendeinhalt zu gestalten, doch die Datenbank speichert den aktuellen Inhalt der Textbox. Wenn das nicht erwünscht ist, muss nachgearbeitet werden.

Die **Bt_Single_Click** Ereignismethode ist daher einfach ein Duplikat von Button **Testen**, aber eben auf der Beobachtungsseite.

```
Private Sub Bt_Single_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Bt_Single.Click
    Send_Befehl(TB_Befehl.Text)
End Sub
```

### *1.6.7 Daten automatisch anfordern*

Beim Button **Bt_Zyklus** gehen wir einen anderen Weg. Zuerst wird, wie bei der Portverbindung auch der Text verändert. Somit bekommen wir zwei Zustände zur Auswertung.

```
Private Sub Bt_Zyklus_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Zyklus.Click
    If Bt_Zyklus.Text = "zyklisch lesen" Then
      Bt_Zyklus.Text = "Zyklus Stoppen"
    Else
      Bt_Zyklus.Text = "zyklisch lesen"
    End If
End Sub
```

Das Verhalten testen wir gleich und wir sehen, dass sich die Beschriftung vom Button bei jedem Klick ändert. Um nun einen automatischen Auftrag an den Controller zu senden, Variablenwerte zu liefern, brauchen wir einen Timer, der in festen Zeitabständen den Befehl aus dem Ereignis **Bt_Single_Click** ausführt. Also laden wir uns einen weiteren Timer in unsere Applikation aus der Toolbox. In der Ereignisroutine **Bt_Zyklus** wird nun bei der Textzuweisung **Zyklus Stoppen** der Timer2 mit der Eigenschaft **Enabled =True** aktiviert und bei der anderen Textzuweisung **zyklisch lesen** mit **Enabled = False** abgeschaltet**.**

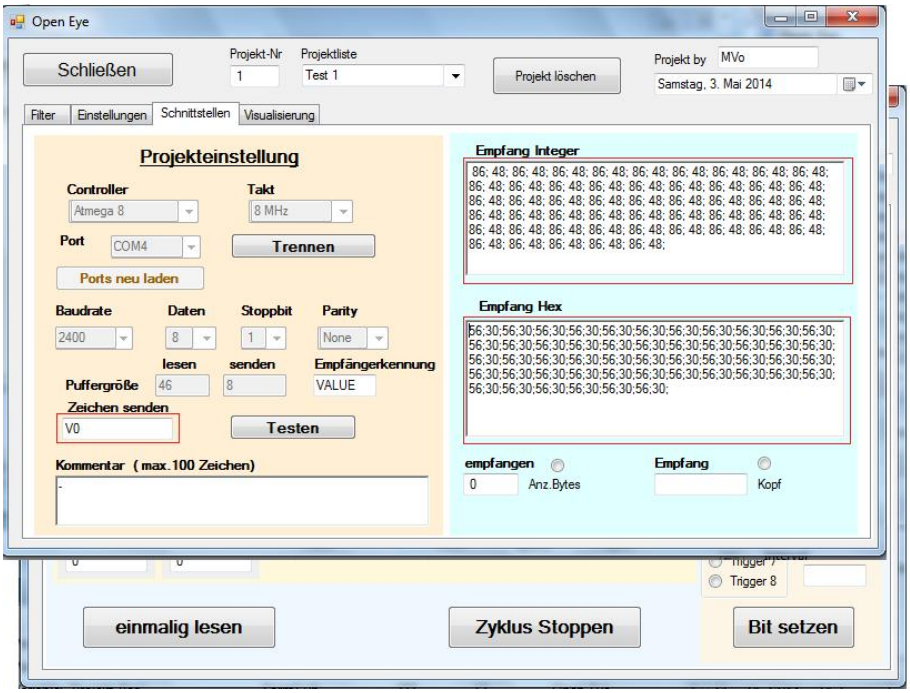
```
Private Sub Bt_Zyklus_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Zyklus.Click
    If Bt_Zyklus.Text = "zyklisch lesen" Then
      Bt_Zyklus.Text = "Zyklus Stoppen"
      Timer2.Enabled = True
    Else
      Bt_Zyklus.Text = "zyklisch lesen"
      Timer2.Enabled = False
    End If
  End Sub
```

Den Intervall setzen wir auf eine krumme Zahl, beispielsweise 879 . Kleinere Zeitintervalle machen keinen Sinn, da das Auge ja auch Zeit braucht, Änderungen zu erkennen. So ist aber ein Sekundentakt im Controller gut zu beobachten. Mit einem Doppelklick auf den Timer bekommen wir den Rahmen für das **Tick-Ereignis**. Dort tragen wir den Sendeanstoß ein.

```
Private Sub Timer2_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Timer2.Tick
    Send_Befehl(TB_Befehl.Text)
  End Sub
```

In der Parametrierung wird dieser Timer2 mit **Enabled = false** beim Programmstart abgeschaltet. Den **Timer1** setzen wir im Intervall auf 100. Er soll schneller reagieren, wenn Daten eintreffen.

Mit ein paar einfachen Schritten lässt sich das Ergebnis prüfen. Das Programm starten und. auf Seite zwei die Anzeige erzeugen. Auf Seite drei den Port verbinden und auf Seite vier den Zyklus einschalten. Zurück auf Seite drei und den Dateneingang in den **RichTextBox** verfolgen.



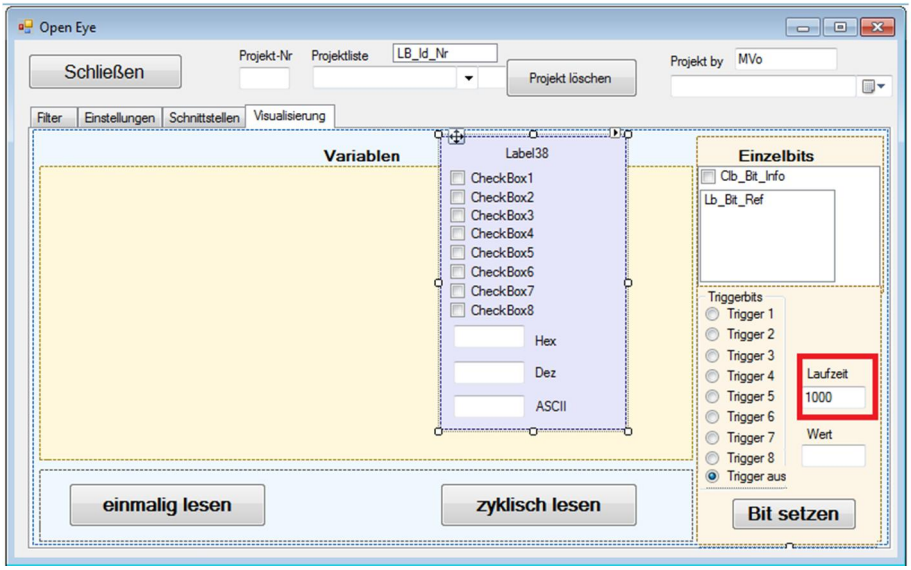

*Automatik Daten holen*

Dabei fällt auf, auch wenn der Port nicht verbunden ist, kann ein Sendeauftrag über die Button vergeben werden. Deshalb binden wir die Freigabe von **Bt_Single** und **Bt_Zykus** mit in die Subroutine **Chk_Connect_Status** ein.

### *1.6.8 Einen Trigger absenden*

Im Gegensatz zu der Anforderung der gesamten Variablen ist der zweite gesendete Wert von Bedeutung. Dieser soll die Bitmuster 00000001 für Trigger 1, 00000010 für Trigger 2, 00000100 für Trigger 3 usw. enthalten. Diese Bits lassen sich auf den Controller einfach abfragen. Da diese Bitmuster aber keine Zeichen sind, die in der Textbox Tb_Befehl so einfach hingeschrieben werden können, müssen wir uns überlegen, wie wir vorgehen wollen. Bleiben wir bei den ASCII Zeichen 0 bis 8 müssen wir im Controller die Bits aufbereiten. Leicht zu erfahren ist die Tatsache, das ein ASCII-Zeichen 0 hexadezimal 30 ist oder Dezimal 48. Anders gesehen können wir die Aufbereitung aber auch im PC vornehmen und nach einem ASCII Zeichen V das folgende Byte als Bitmuster im Controller auswerten. Dann ist die Textbox aber nicht Quelle für den Sendeauftrag. Mir scheint die zweite Lösung trotzdem besser, denn der Controller muss ja nicht mit solchen Aufgaben belastet werden. Der Befehl für den Trigger ist immer der gleiche Aufbau und von daher muss er nicht über die Textbox laufen. Steht noch eine weitere Entscheidung an, da nun klar ist, dass sowieso ein separater Sendeauftrag generiert wird. Nehmen wir V für Value oder setzen wir T für Trigger ein. Auch hier gilt zu beachten, wenn es Unterschiedliche Befehle sind müssen sie auch unterschiedlich bewertet werden. Das wird im Buchteil Assembler deutlich. Hier entscheide ich mich, es bei V zu lassen und beim Eintreffen dieses Zeichens das zweite Byte immer auszuwerten.

Nun ist beim Trigger ein weiteres Problem zu bedenken. Wie soll sich das Programm verhalten, wenn der Trigger nicht ausgelöst wird. Es ist Durchaus möglich, dass der Programmabschnitt mit dem eingerichteten Trigger gar nicht ausgeführt wird. In einem solchen Fall muss die Zeit überwacht werden, in der die Bearbeitung im Controller erwartet wird. Danach ist der Trigger zurück zu setzen, damit er weitere Bearbeitung nicht stört. Hier kommt mal wieder v<Trigger> zum Einsatz. Wie die Befehle zum Schalten von Relais mit R<n> und r<n> für Ein und Aus wird auch das V für Trigger Ein und v für Trigger Aus definiert. Soweit ist nun die Vorgehensweise klar. Lediglich die Zeitüberwachung fehlt. Dafür setzen wir eine Textbox auf Seite vier neben das Triggerbutton und benennen sie Tb_Watchdog. In diesem Fenster lassen wir nun wieder nur Ziffern zu, wie wir es in den Textboxen Tb_Read und Tb_Write schon durchgeführt haben.




*Trigger Watchdog*

Zu dieser Textbox benötigen wir natürlich auch einen weiteren **Timer**. Bisher habe ich darauf verzichtet, den Timern Namen zu geben, vielleicht sollten wir das nun nachholen. Zur Erinnerung, **Timer1** hat den Ringpuffer auf eingegangene Daten kontrolliert und ist mit 100 mSek. eingestellt. **Timer2** ist für das zyklische Senden der Variablenabfrage zuständig und hatte eine krumme Zeitparametrierung. **Timer3** ist nun der Wachhund für die Laufzeit eines Triggers. Deshalb vergeben wir nun folgende Namen

Tigger1 Ist **Tr_Chk_RS232**

Trigger2 ist **Tr_Zyklus**

Trigger3 ist **Tr_Watchdog**

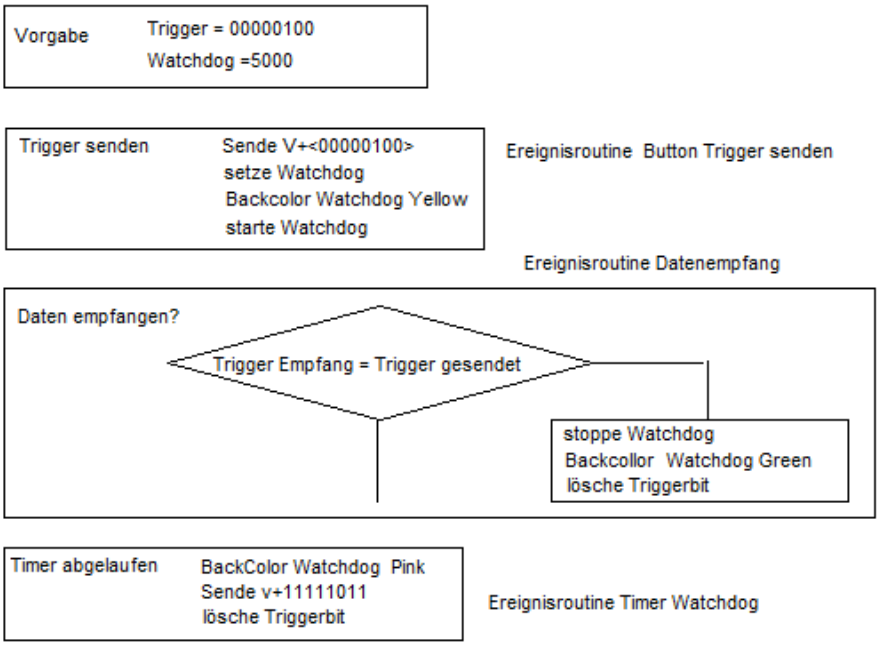
Ändern wir nun unter Eigenschaften die Namen der Trigger.

Die Textbox **Tb_Watchdog** bekommt in der Ereignisroutine **KeyPress** die Kopie von **Tb_Read_KeyPress**. Die maximale Anzahl der Zeichen setzen wir auf 9. Der Grund, der Intervallparameter ist ein **Integer** und damit anders als beim Mikrocontroller mit 16 Bit unter Windows eine **32 Bit Zahl**. Das entspricht einer 10 stelligen Zahl, allerdings nicht für 9999999999. Wenn wir 9 Stellen zulassen, reicht das für eine Zeit von 999 999 999 mSek oder 999 999 Sekunden. Das entspricht einer Zeit von rund 277 Stunden. Die Regel wird hier eher bei 5 Sekunden liegen und so reicht das allemal. Die Grenze ist also nur dazu da, den Bereich des Typ **Integer** nicht zu verletzen.

```
Private Sub TB_Watchdog_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TB_Watchdog.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
      e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(TB_Watchdog.Text) = 9 Then
      e.KeyChar = ""
    End If
  End Sub
```

Diese Funktion wird auch gleich geprüft.

Nun gilt es den Wert in den Parameter **Intervall** des Timers **Tr_Watchdog** zu schreiben. Das erledigen wir in der Ereignisroutine **Bt_Set_Trigger**. Hier stellen wir auch den Sendeauftrag zusammen und signalisieren einen aktiven Triggerauftrag mit Einfärbung der Textbox **Tb_Watchdog**. Der Timer wird nach dem Sendeauftrag gesetzt. Das wird wieder mit einer Skizze vom Ablauf etwas besser deutlich.




*PAP Wachhund für Trigger*

Beim Aufbau der Skizze erkennen wir schnell, dass es vier getrennt laufende Programmblöcke sind, die hier den Job erledigen. Alles sind Ereignisse. Zuerst interessieren nur die beiden oberen Blöcke. Voraussetzung ist die Wahl eines Triggers und die Zeitvorgabe für die Überwachungszeit. Das bedeutet, der Text im Wertefeld muss >0 sein, ansonsten sollte kein Sendeauftrag stattfinden. Mit dem Ereignis **Tb_Wert_TextChanged** geben wir das Button **Bit setzen** bei Inhalt <> 0 frei. Die Textbox selbst sollte nur durch ein Ereignis der Radiobutton **Rb_Bit_0** bis **Rb_Bit_7** sowie **Trigger löschen** verändert werden. Eine manuelle Eingabe könnte fehlerhafte Programmausführung bedeuten. Deshalb wird diese Textbox auch in den Eigenschaften mit **Enabled** auf **false** gesperrt. So funktioniert auch das **TextChanged** Ereignis wie erwartet.

```
Private Sub TB_Wert_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TB_Wert.TextChanged
  If TB_Wert.Text <> "0" Then
     Bt_Set_Trigger.Enabled = True
     TB_Watchdog.BackColor = Color.White
  Else
     Bt_Set_Trigger.Enabled = False
  End If
End Sub
```

Mit der Freigabe des Button **Bit Setzen** signalisieren wir gleich in der Textbox **Tb_Watchdog** farblich die Freigabe und die Bereitschaft des Timers **Tr_Watchdog**. Nach dem Senden der Triggeranforderung sollte kein weiterer Trigger möglich sein. Das bedeutet, die Radiobutton und auch das Button **Bt_Set_Trigger** müssen bis zur Antwort durch den Trigger oder den Ablauf der Überwachungszeit gesperrt werden. Im Prinzip könnten wir das Panel **Pn_Trigger** sperren. Damit wären dann alle darauf befindlichen Objekte für den Zugriff gesperrt. Allerdings müssen wir uns für die Prüfung etwas einfallen lassen. Ok, ein provisorisches Button zwischen **Bt_Single** und **Bt_Zyklus** könnte für einen Test erst einmal herhalten.

Setzen wir nun alle erforderlichen Befehle in das Ereignis **Bt_Set_Trigger_Click**

```
Private Sub Bt_Set_Trigger_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Set_Trigger.Click
    Dim Trigger_on(2) As Byte
    Trigger_on(0) = Asc("V")
    Trigger_on(1) = Val(TB_Wert.Text)
    Tr_Watchdog.Interval = Val(TB_Watchdog.Text)
    Tr_Watchdog.Enabled = True
    TB_Watchdog.BackColor = Color.Yellow
    SerialPort1.Write(Trigger_on, 0, 2)
    Pn_Trigger.Enabled = False
  End Sub
```

Theoretisch können wir jetzt eine Triggerantwort anfordern. Aber, und ich denke, hier werden alle einmal stolpern, es muss auch die Schnittstelle zugeschaltet sein. Es macht ja auch keinen Sinn, einen Trigger abzusenden, ohne einen Verbindungspartner zu haben. Wird auf eine geschlossene Verbindung zugegriffen, gibt es einen Laufzeitfehler. Deshalb werden wir die Scharfschaltung und die Freigabe in die Statusabfrage der Schnittstelle einbinden.

```
Public Sub Chk_Connect_Status()
  Dim is_Connect As Boolean
  is_Connect = SerialPort1.IsOpen          ' Port ist verbunden
  CB_Baud.Enabled = Not is_Connect         ' Zugriff nur Offline
  CB_Controller.Enabled = Not is_Connect
  CB_Daten.Enabled = Not is_Connect
  CB_Port.Enabled = Not is_Connect
  CB_Parity.Enabled = Not is_Connect
  CB_Stop.Enabled = Not is_Connect
  CB_Takt.Enabled = Not is_Connect
  TB_Controller.Enabled = Not is_Connect
  TB_Read.Enabled = Not is_Connect
  TB_Takt.Enabled = Not is_Connect
  TB_Write.Enabled = Not is_Connect
  Bt_Port_New.Enabled = Not is_Connect
  Bt_Test.Enabled = is_Connect             ' Zugriff nur Online
  Bt_Single.Enabled = is_Connect
  Bt_Zyklus.Enabled = is_Connect
  '*  Trigger nur bei verbundener Schnittstelle
  Pn_Trigger.Enabled = Is_Connect
  If Not Is_Connect Then
    Bt_Zyklus.Text= "zyklisch lesen"
    Tr_Zyklus.Enabled = False
    Rb_TriggerOff.Checked = True
    Tr_Watchdog.Enabled = False
    Bt_Set_Trigger.Enabled = False
  End If
End Sub
```

Die Button **Bt_Single** und **Bt_Zyklus** könnten nach einer Triggeranforderung aktiv bleiben, aber folgt gleich nach einem Trigger eine normale Anforderung, sind die Informationen weg. Deshalb sollte ein zyklisches Lesen unterbrochen werden. Manuell kann jederzeit eine Anforderung erfolgen.

```
Private Sub Bt_Set_Trigger_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Set_Trigger.Click
    Dim Trigger_on(2) As Byte
    Bt_Zyklus.Enabled = False
    Bt_Zyklus.Text = "zyklisch lesen"
    Tr_Zyklus.Enabled = False
    Trigger_on(0) = Asc("V")
    Trigger_on(1) = Val(TB_Wert.Text)
    Tr_Watchdog.Interval = Val(TB_Watchdog.Text)
    Tr_Watchdog.Enabled = True
    TB_Watchdog.BackColor = Color.Yellow
    SerialPort1.Write(Trigger_on, 0, 2)
    Pn_Trigger.Enabled = False
  End Sub
```

Das Verhalten und den Datentransfer können wir auch hier mit den **RichtextBoxen** und einem gebrückten Sub_D Stecker ( Pin 2 und 3 ) leicht prüfen. Dazu setzen wir die Überwachungszeit auf 10000 (10 Sekunden) und können nach einer Triggeranforderung den Dateneingang kontrollieren.

.

### *1.6.4 Den Empfang Trigger quittieren*

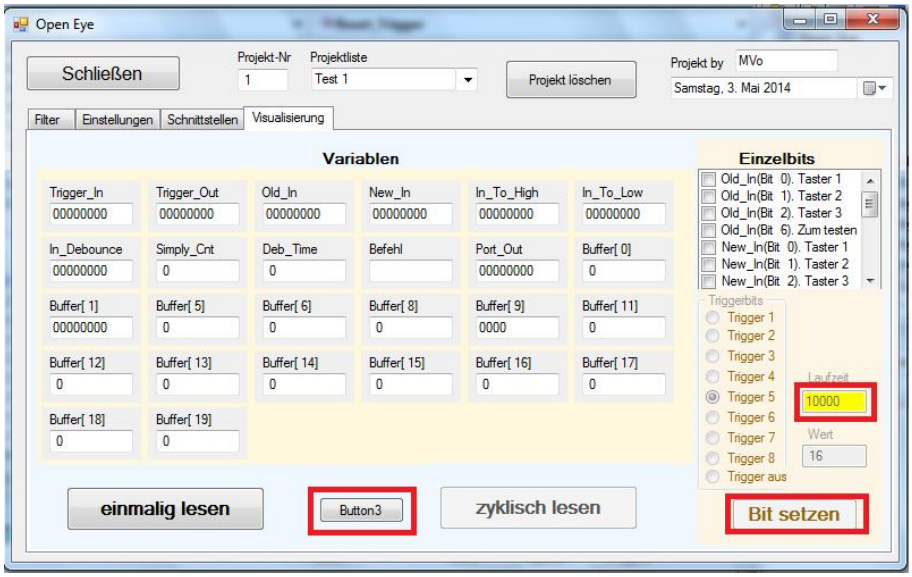
Zu diesem Zweck nehmen wir einfach ein **Button** und setzen in der **Ereignismethode Click** den Wert aus **Tb_Wert.Text** in den Aufruf von **Reset_Trigger.**

```
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button3.Click
    Reset_Trigger(Val(TB_Wert.Text))
 End Sub
```

Hier prüfen wir, ob der Triggerwert dem übergebenen Wert entspricht. Klar, wenn wir mit dem Button diese Routine aufrufen passt das, aber später wird dieser Routine die Variable Trigger_Out übergeben. So können wir unterscheiden, ob die Antwort aus einer normalen Anfrage oder aus dem Triggerauftrag stammt.

```
Public Sub Reset_Trigger(ByVal Trigger As Integer)
    If (Trigger = 1) And (Rb_Bit_0.Checked) Then Rb_TriggerOff.Checked = True
    If (Trigger = 2) And (Rb_Bit_1.Checked) Then Rb_TriggerOff.Checked = True
    If (Trigger = 4) And (Rb_Bit_2.Checked) Then Rb_TriggerOff.Checked = True
    If (Trigger = 8) And (Rb_Bit_3.Checked) Then Rb_TriggerOff.Checked = True
    If (Trigger = 16) And (Rb_Bit_4.Checked) Then Rb_TriggerOff.Checked = True
    If (Trigger = 32) And (Rb_Bit_5.Checked) Then Rb_TriggerOff.Checked = True
    If (Trigger = 64) And (Rb_Bit_6.Checked) Then Rb_TriggerOff.Checked = True
    If (Trigger = 128) And (Rb_Bit_7.Checked) Then Rb_TriggerOff.Checked = True
    If Rb_TriggerOff.Checked Then
       TB_Watchdog.BackColor = Color.LawnGreen   ' Textbox grün Empfang Trigger
       Pn_Trigger.Enabled = True                  ' Bereich Trigger freigeben
       Tr_Watchdog.Enabled = False                ' Wachhund stoppen
       Bt_Zyklus.Enabled = True                    ' zyklische bearbeitung freigeben
    End If
```

Wenn wir nun einen Trigger vorwählen und den Trigger absenden, wird anhand der Farbgebung von Tb_Watchdog deutlich, welchen Zustand das Programm gerade hat. Der Screenshot zeigt einen versendeten Trigger und das die Überwachungszeit läuft. Das Button **zyklisch lesen** ist ausgeblendet sowie eine weitere Triggeranforderung gesperrt.




*Trigger quittieren*

Wird mit dem provisorischen Button ein **Reset_Trigger** durchgeführt, sollte das Textfeld grün werden und die Bedienung bis auf den Button **Bit setzen** für alle Objekte wieder möglich sein.

Das provisorische Button kann nach Abschluss aller Prüfungen wieder entfernt werden. Jetzt müssen wir den Aufruf durch den Button mit der empfangenen Antwort ersetzen.

Weitere Maßnahmen sind bei einer Triggerantwort nicht erforderlich, da der Trigger im Controller bei der Bearbeitung gelöscht wird.

Dazu haben wir in der Ereignisroutine **Rb_Data_Rec_CheckedCanged** bereits den zurückgelieferten Trigger aus dem Datenstrom geholt und den Anzeigeobjekten verfügbar gemacht. Nun fügen wir einfach noch den Aufruf Reset_Trigger mit dem Parameter Trigger ein.

Nachfolgend der Abschnitt in **Rb_Data_Rec_CheckedCanged**.

```
Private Sub Rb_Data_Rec_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Rb_Data_Rec.CheckedChanged
        ……..
    Trigger = Werte_Feld(1)              ' bereits getestet, jetzt Reset Trigger
    Reset_Trigger(Trigger)               '  gesendeten Trigger quittieren
    While Data_Cnt < Anzahl
```

### *1.6.9 Zeitüberschreitung Trigger*

Bei einer Überschreitung der Überwachungszeit ist der gesetzte Trigger noch im Controller unverändert. Im Timerereignis muss nun signalisiert werden, dass die Anfrage abgebrochen wurde. Das geschieht mit einem Farbwechsel der Textbox **Tb_Watchdog** nach rot. So wissen wir immer, ob noch eine Zeit läuft, der Vorgang abgeschlossen oder fehlgeschlagen ist. Bei einer Zeitüberschreitung müssen wir nicht nur die Bedienung der Trigger freigeben sondern auch im Controller das Triggerbit löschen. Dafür senden wir ein kleines v mit der Triggernummer. Der Rest muss im Controller bearbeitet werden..

```
Private Sub Tr_Watchdog_Tick(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Tr_Watchdog.Tick
    Dim Trigger_on(2) As Byte
    TB_Watchdog.BackColor = Color.Red
    Pn_Trigger.Enabled = True
    Rb_TriggerOff.Checked = True
    Tr_Watchdog.Enabled = False
    Trigger_on(0) = Asc("v")
    Trigger_on(1) = Val(TB_Wert.Text)
    SerialPort1.Write(Trigger_on, 0, 2)
    Bt_Zyklus.Enabled = True
  End Sub
```

Wichtig ist auch, den Timer **Tr_Watchdog** wieder zu sperren, sonst löst er immer wieder aus.

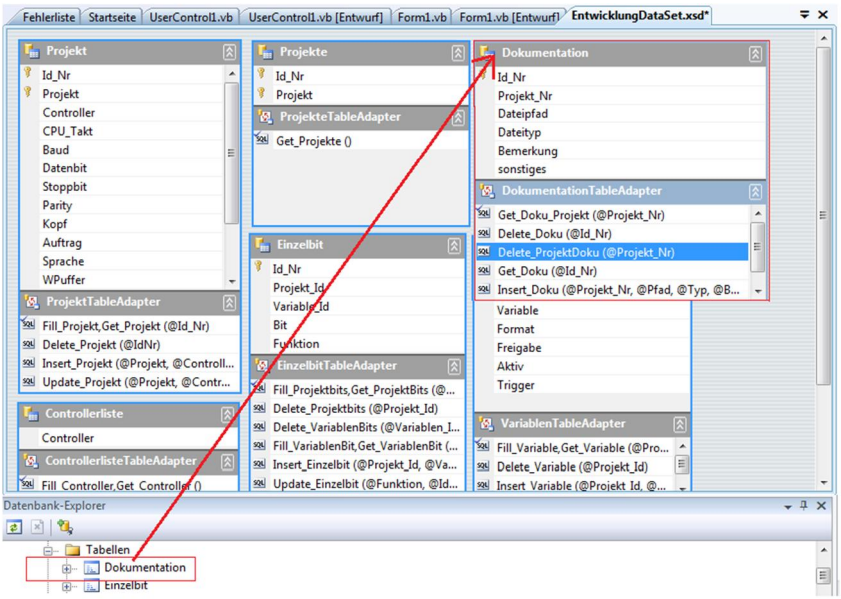
Auch diese Funktion testen wir mit allen möglichen Bedingungen. Eine Fehlbedienung sollte ausgeschlossen sein. Wenn uns etwas auffällt oder stört, ist es immer wichtig, Änderungen gleich einzubauen und erneut zu testen. Ist das Programm erst einmal gewachsen und klemmt an allen Ecken und Kanten, kann man gleich von vorn beginnen.

.

## 1.7 Dokumentation

In unserer Aufgabenbeschreibung ist noch ein Punkt unbearbeitet. Die Dokumentation. Werfen wir noch einmal auf die Seite, wo wir unsere TableAdapter eingerichtet haben. Bis auf die Tabelle Dokumentation haben wir alle Tableadapter eingerichtet und bereits erfolgreich eingesetzt. Bevor die fünfte Seite für Zeichnungen und Skizzen sowie weiterer Dokumentenverwaltung gestaltet wird, werden wir uns noch einmal den Datenbankabfragen zuwenden und den Tableadapter für die bereits erstellte Tabelle Dokumentation die Abfragen zusammenstellen.

Dazu ziehen wir die Tabelle aus dem Datenbankexplorer auf die

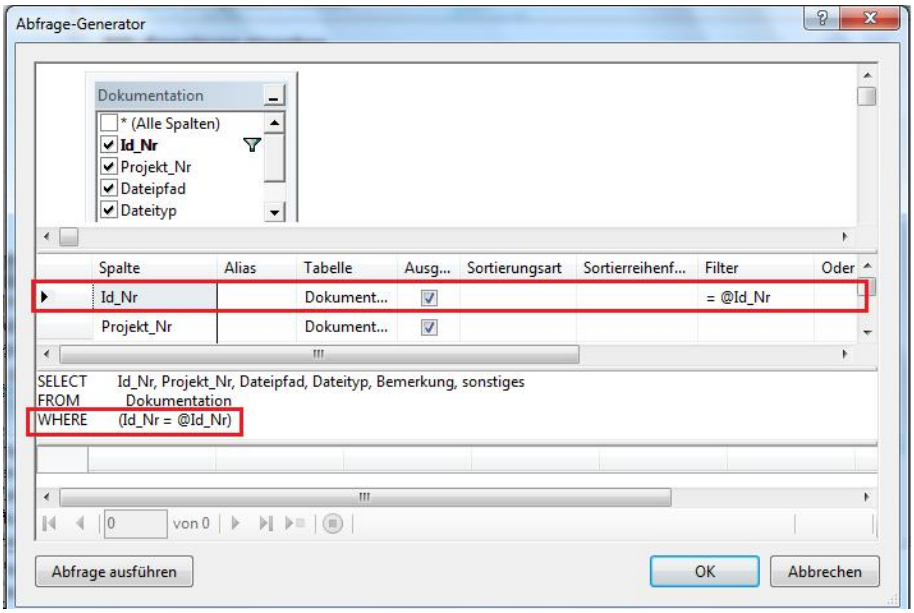



*Tableadapter Dokumentation*

Nun überlegen wir, welche Information wir von der Datenbank erhalten wollen. Klar, Einfügen, Ändern und Löschen sind erst einmal die Basisanweisungen, wobei die Delete einmal für einen einzelnen Eintrag und einmal für das Projekt bezogen wird. Aber keine Angst, es werden dabei keine Dateien gelöscht, sondern nur die Pfadangaben. Es ist ja durchaus möglich, Zeichnungen in verschiedenen Projekten zu hinterlegen.

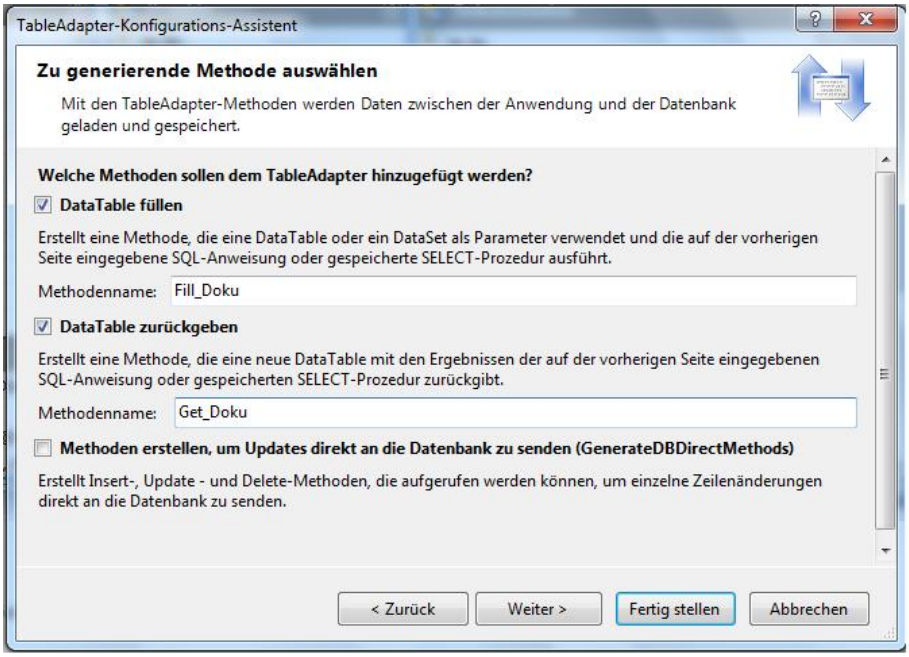
### *1.7.1 Einrichten Tableadapter*

Beginnen wir mit der ersten Anweisung, die bereits vorgeschlagen wird. Zuerst wird wieder in **Erweiterte Optionen** die **Insert**, **Update** und **Delete**-Anweisungen abgewählt und dann eine Abfrage auf die **Id_Nr** generiert.




*Abfrage Dokumente*

Schließlich noch die Namensvergabe für diese erste Abfrage




*Name Fill und Get Dokument*

### *1.7.1.1 Dokumentation bearbeiten*

Nun folgen die Standartanweisungen **Insert**, **Update** und **Delete**. Der Weg ist bereits mehrfach beschrieben. Beginnen wir mit Insert.

| Spalte | Neuer Wert |
|---|---|
| Projekt_Nr | @Projekt_Nr |
| Dateipfad | @Pfad |
| Dateityp | @Typ |

```
INSERT INTO Dokumentation
                (Projekt_Nr, Dateipfad, Dateityp, Bemerkung, sonstiges)
VALUES        (@Projekt_Nr,@Pfad,@Typ,@Bemerkung,@Sonstiges)
```

*SQL Anweisung Insert Dokumentation*

Der Name wird mit **Insert_Doku** vergeben.

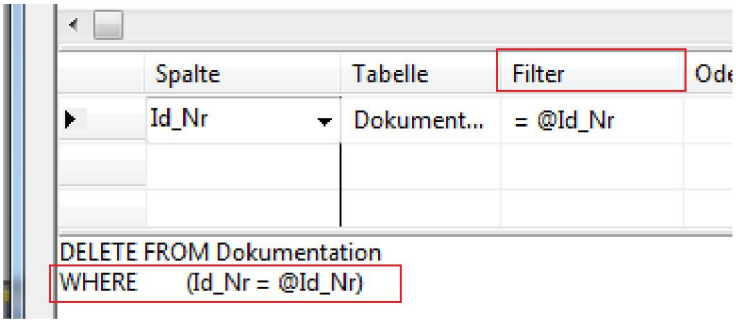
Die Update-Anweisung erhält den Filter der eigenen Id_Nr. Der name ist mit Update_Doku bezeichnend

| Spalte | Tabelle | Set | Neuer Wert | Filter | Oder... | Oder... |
|---|---|---|---|---|---|---|
| sonstiges | Dokument... | ☑ | @Sonstiges | | | |
| Id_Nr | Dokument... | ■ | | = @Id_Nr | | |

```
UPDATE    Dokumentation
SET       Dateipfad = @Pfad, Dateityp = @Typ, Bemerkung = @Bemerkung, sonstiges = @Sonstiges
WHERE     (Id_Nr = @Id_Nr)
```

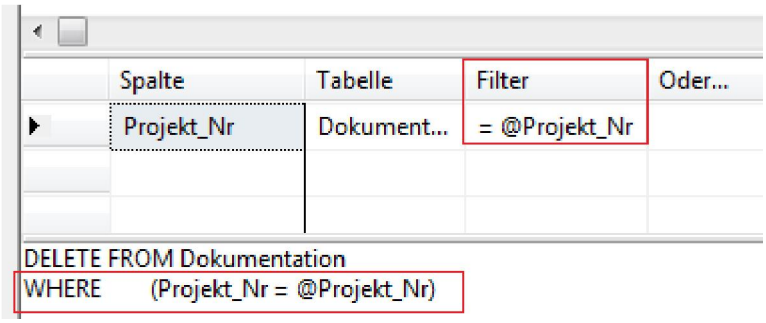
*SQL Anweisung Update Dokumentation*

Von den Delete-Anweisungen brauchen wir zwei. Die erste setzen wir auch auf den eigenen Schlüssel.




*SQL Anweisung Delete Dokumentation*

Der Name wird wieder einfach mit **Delete_Doku** vergeben und die zweite Löschanweisung wird mit **Delete_ProjektDoku** benannt. Der grund ist, das auch einzelne Dokumente aus einem Projekt heraus gelöscht werden sollen. Wird ein Projekt gelöscht, ist auch die gesamte Dokumentation nicht mehr erforderlich. Beachtte: Es werden nur die Verzeichnisangaben gelöscht, nicht die zeichnunggen, Bilder oder andere Dateien.



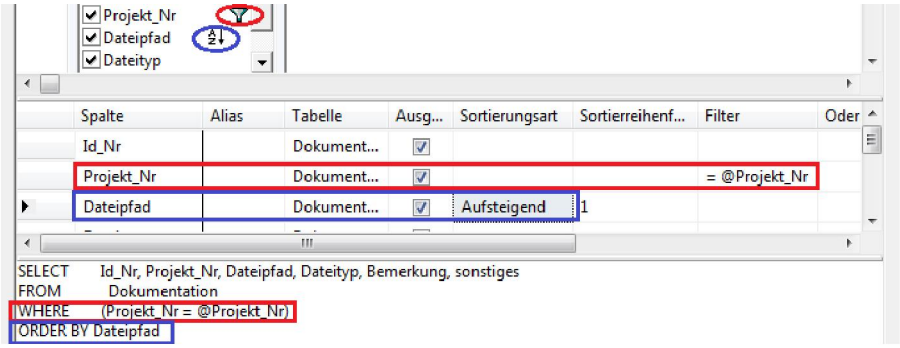

*SQL Anweisung Delete DokuProjekt*

Diese Anweisungen sollten kein Problem bereiten. Sie sind bereits mehrfach ausführlich im Kapitel Datenbank erklärt und so beschränke ich die Erklärungen auf die Einstellung **im Abfrage-Generator.**

### *1.7.1.2 Dokumentation gezielt laden*

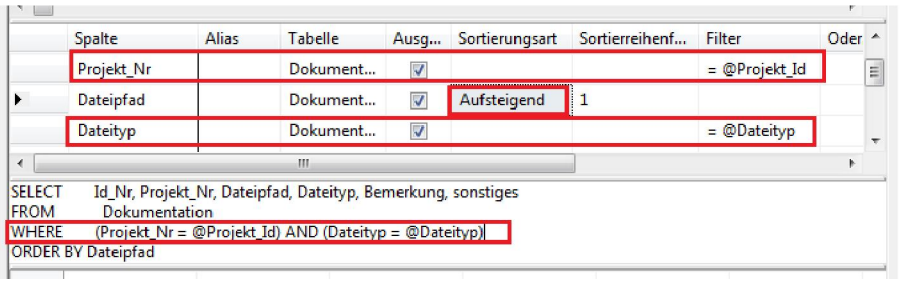
Bleibt noch die Anfrage zum Füllen der Dateiliste zu erstellen. Wenn wir alle Dateien in einer Pfadliste aufnehmen wollen, nicht nur um auch eine Referenz zu haben, macht es Sinn, den Dateityp mitzuführen und die Listen entsprechend den Dateitypen zuzuordnen. So lassen sich Bilder und Skizzen in PictureBoxen darstellen und eine Textdokumentation in einer RichTextBox. Letztere muss im RTF-Format abgelegt sein. Bei allen anderen Formaten aber bleibt die Referenz.

Über den Dateityp lassen sich auch Dateien gezielt laden. Dafür haben wir zwei zusätzliche Abfragen anzulegen. In der ersten Anweisung werden alle Dateien zum Projekt in der Pfadliste angezeigt. Hier wird der Filter nur auf die Projekt_Nr gesetzt.




*Liste Dokumentation zum Projekt*

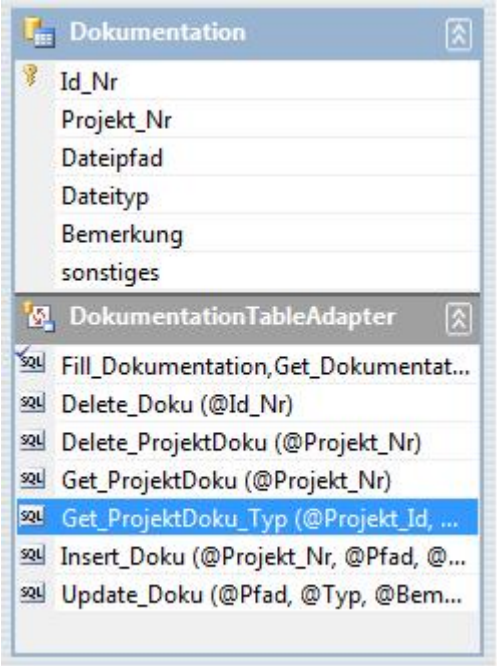
Der Name für diese Abfrage wird mit Get_ProjektDoku vergeben. Bleibt noch die Abfrage für weitere Unterscheidung. Zum Beispiel nur Grafikdateien oder Richtextformate, aber auch PDF, selbst wenn diese mit den vorhandenen Objekten in der Toolbox nicht dargestellt werden können.




*Liste Doku Projekt und Typ*

Diese letzte Abfrage der Tabelle Dokumentation bekommt den Namen Get_ProjektDoku_Typ.

Der Tableadapter stellt sich auf dem Dataset dann so dar




*Tableadapter Dokumentation fertig*

### *1.7.2 Tabelle Teileliste*

Vervollständigen wir unsere Datenbank noch mit dem Anlegen einer teioleliste. So haben wir dann später immer noch die Möglichkeit, eine Seite zu gestalten und die Teiletabelle zu verwalten. Fügen wir also erst einmal eine neue Tabelle ein

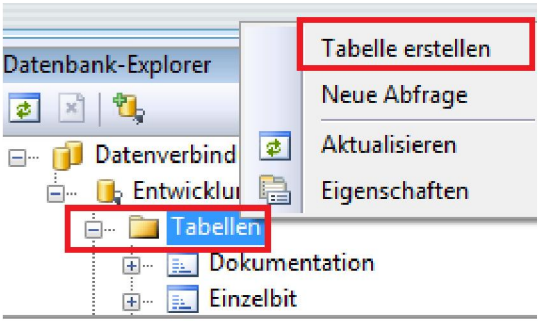


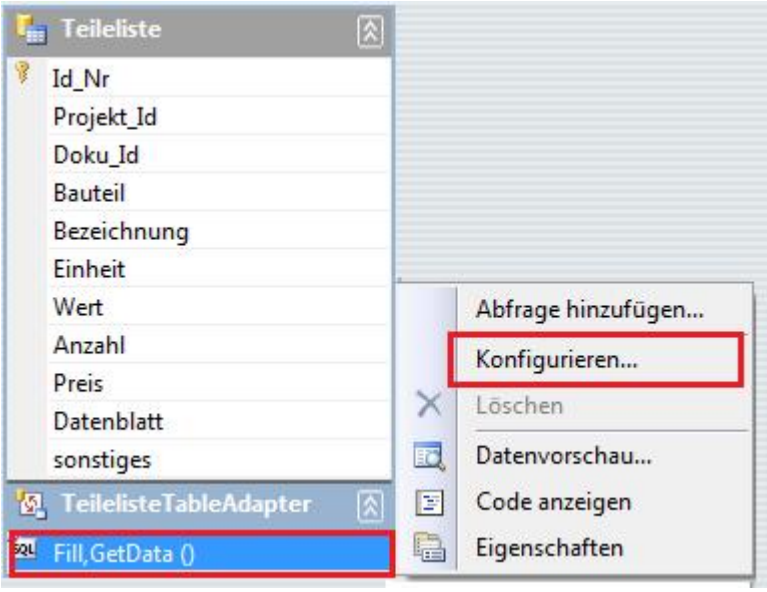

*Tabelle hinzufügen*

und definieren die Spalten. Der Vorschlag sollte für eine abschließende Datenhaltung vollkommen ausreichen.

| Spaltenname | Datentyp | Länge | NULL-We... | Eindeutig | Primärer .. |
|---|---|---|---|---|---|
| Id_Nr | int | 4 | Nein | Ja | Ja |
| Projekt_Id | int | 4 | Nein | Nein | Nein |
| Doku_Id | int | 4 | Nein | Nein | Nein |
| Bauteil | nvarchar | 40 | Nein | Nein | Nein |
| Bezeichnung | nvarchar | 40 | Nein | Nein | Nein |
| Einheit | nvarchar | 20 | Nein | Nein | Nein |
| Wert | int | 4 | Nein | Nein | Nein |
| Anzahl | tinyint | 1 | Nein | Nein | Nein |
| Preis | float | 8 | Nein | Nein | Nein |
| Datenblatt | nvarchar | 100 | Nein | Nein | Nein |
| sonstiges | int | 4 | Nein | Nein | Nein |

*Tabelle Teileliste*

Diese Tabelle ziehen wir wieder auf die Ansicht **EntwicklungDataSet** und beginnen mit der Konfiguration der **Fill**- und **Get** Methoden




*Tableadapter Teileliste*

Die Schritte sind den vorhergehenden gleich. Es gibt eine Abfrage für einen Datensatz, eine Abfrage für Datensätze zur Dokumentation, beispielsweise einer Zeichnung und eine Abfrage zum Projekt. Es müssen weiterhin eine Löschanweisung auf ein Bauteil, Bauteile einer Dokumentation und Löschanweisung auf das Projekt bezogen eingerichtet werden. Schließlich bleibt die **Insert** und **Update** – Anweisung. Update bezieht sich immer auf die eigene Id_Nr.

Zuerst einmal die Abfragen, um Datensätze zu holen. Die Namen vergeben wir mit **Fill_Teileliste** und **Get_Teileliste** bezogen auf die **Id_Nr.**

Dann folgen **Get_TeilelisteDoku** bezogen auf die **Doku_Id** und schließlich noch **Get_TeilelisteProjekt** bezogen auf die **Projekt_Id**

| Spalte | Alias | Tabelle | Ausg... | Sortierungsart | Sortierreihenf... | Filter | Oder |
|---|---|---|---|---|---|---|---|
| Id_Nr | | Teileliste | ☑ | | | = @Id_Nr | |
| Projekt_Id | | Teileliste | ☑ | | | | |
| Doku_Id | | Teileliste | ☑ | | | | |

```
SELECT    Id_Nr, Projekt_Id, Doku_Id, Bauteil, Bezeichnung, Einheit, Wert, Anzahl, Preis, Datenblatt, sonstiges
FROM      Teileliste
WHERE     (Id_Nr = @Id_Nr)
```

Name : Get_Teileliste

| Spalte | Alias | Tabelle | Ausg... | Sortierungsart | Sortierreihenf... | Filter | Oder |
|---|---|---|---|---|---|---|---|
| Id_Nr | | Teileliste | ☑ | | | | |
| Projekt_Id | | Teileliste | ☑ | | | | |
| Doku_Id | | Teileliste | ☑ | | | = @Doku_Id | |
| Bauteil | | Teileliste | ☑ | Aufsteigend | 1 | | |

```
SELECT    Id_Nr, Projekt_Id, Doku_Id, Bauteil, Bezeichnung, Einheit, Wert, Anzahl, Preis, Datenblatt, sonstiges
FROM      Teileliste
WHERE     (Doku_Id = @Doku_Id)
ORDER BY Bauteil
```

Name : Get_DokuListe

| Spalte | Alias | Tabelle | Ausg... | Sortierungsart | Sortierreihenf... | Filter | Oder |
|---|---|---|---|---|---|---|---|
| Projekt_Id | | Teileliste | ☑ | | | = @Projekt_Id | |
| Doku_Id | | Teileliste | ☑ | | | | |
| Bauteil | | Teileliste | ☑ | Aufsteigend | 1 | | |

```
SELECT    Id_Nr, Projekt_Id, Doku_Id, Bauteil, Bezeichnung, Einheit, Wert, Anzahl, Preis, Datenblatt, sonstiges
FROM      Teileliste
WHERE     (Projekt_Id = @Projekt_Id)
ORDER BY Bauteil
```

Name : Get_ProjektListe

*Teilelisten laden*

Dann folgen die Delete- Anweisungen. Zuerst die auf die eigene Id_Nr bezogene, da auch einzelne Einträge entfernt werden sollen. Der Name ist mit Delete_Bauteil etwas abweichend, trifft aber den Kern. Eine Teileliste ist eine Liste von Bauteilen auf eine Zeichnung bezogen.

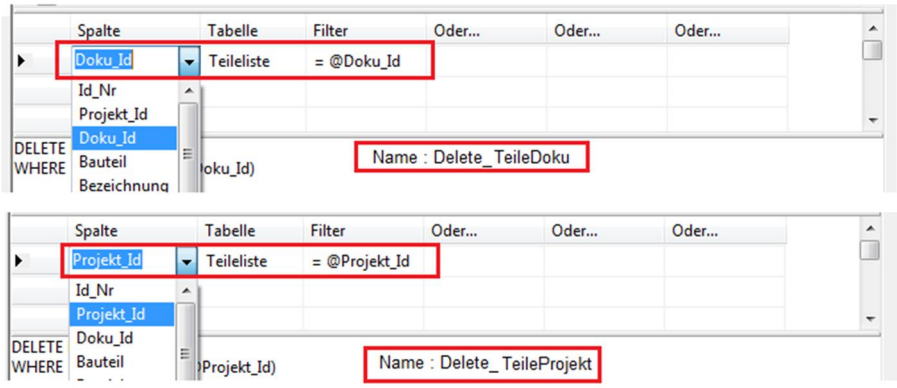
| Spalte | Tabelle | Filter | Oder... | Oder... | Oder... |
|---|---|---|---|---|---|
| Id_Nr | Teileliste | = @Id_Nr | | | |

Id_Nr
Projekt_Nr
Doku_Id
Bauteil
Bezeichnung

```
DELETE
WHERE
```

*Bauteil löschen*

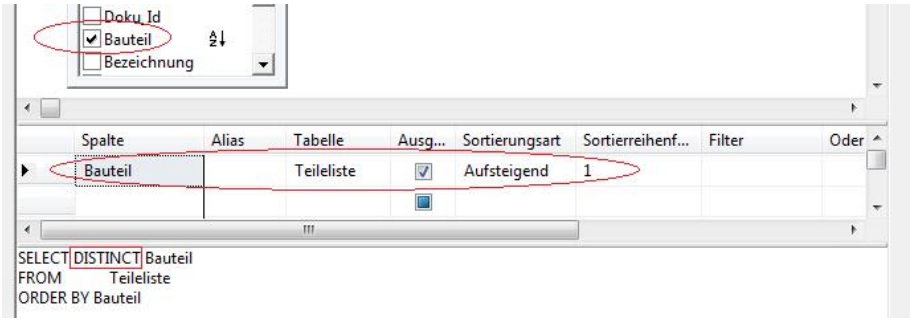
Das Löschen eines einzelnen Bauteiles ist aber erst der Anfang. Wird eine Dokumentation gelöscht, könnte auch eine Bauteileliste betroffen sein. Daher werden auch die Bauteile einer Zeichnung, also Delete_TeileDoku und die weitere Löschvorgabe beim Löschen eines Projektes mit Delete_TeileProjekt aufgenommen.




*SQL Anweisung Teileliste Delete*

Schließlich kommen die zwei Anweisungen Update und Insert, die hier nicht noch einmal ausgeführt sind. Die Anweisung sollte hinreichend klar sein.

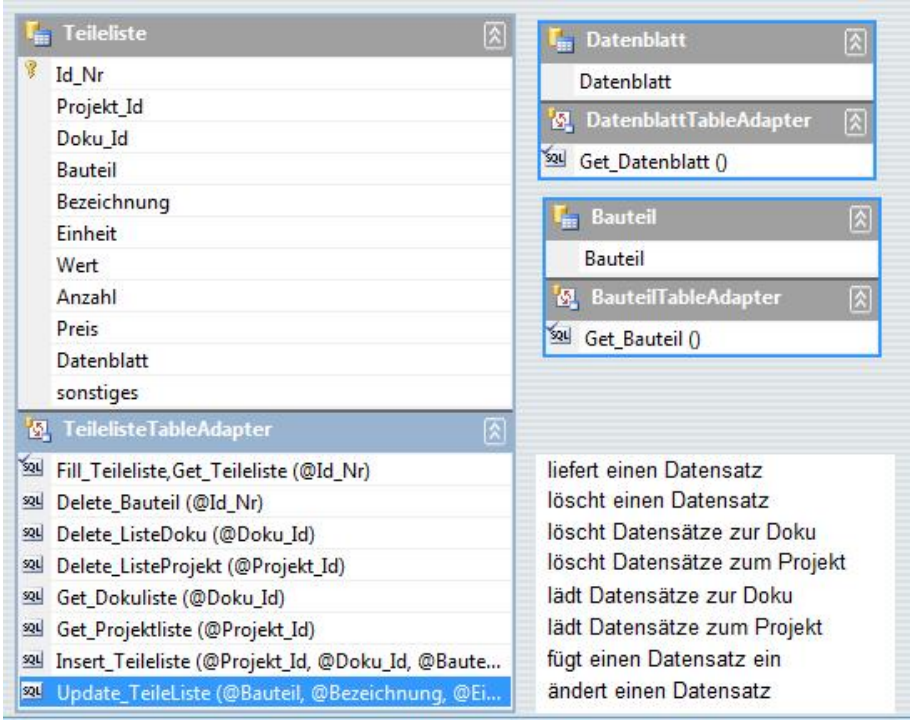
Dann ziehen wir uns von dieser Tabelle noch zwei Tableadapter auf die Dataset-Ansicht. Hier wird jeweils nur ein Eintrag herausgeholt. Einmal ist es das Bauteil und einmal der Pfad zur Datenblattdatei. In den Tableadaptern wird nur die Methode Get benötigt. Die Namensvergabe fällt auch simpel aus: **Get_Bauteil** und **Get_Dateipfad**




*Abfrage Bauteile*

Auch hier müssen wir von Hand das Wort Disinct in die Abfrage einfügen, damit keine doppelten Bauteile zurückgeliefert werden. Eine Eingrenzung durch einen Filter auf das Projekt oder gar auf die Zeichnung ist nicht sinnvoll, da diese Einträge in Comboboxen zur Auswahl stehen können. Diese Tabelle wird mit dem Namen **Bauteile** belegt und eine zweite nach gleichem Schema erstellte Tabelle **Datenblatt**. So haben wir später die Möglichkeit zur schnellen Auffindung von Datenblättern und anderen Dokumenten.

Auf der Ansicht des Datasets sind nun diese Tableadapter hinzugekommen.



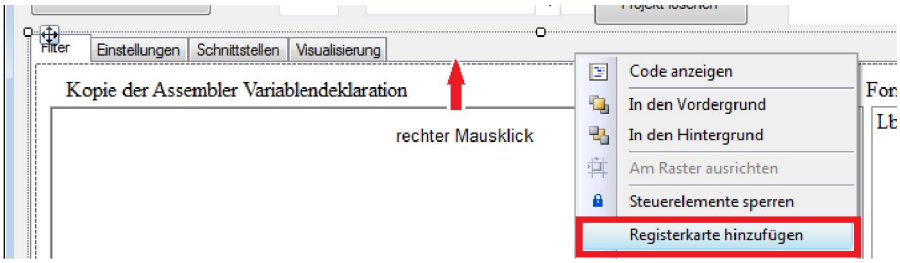

*Bauteile Tableadapter*

Nun sollten genug Tabellen und Tableadaper vorbereitet sein, um auch den letzten Wunsch zu erfüllen.

### *1.7.3 Die Seite Dokumentation*

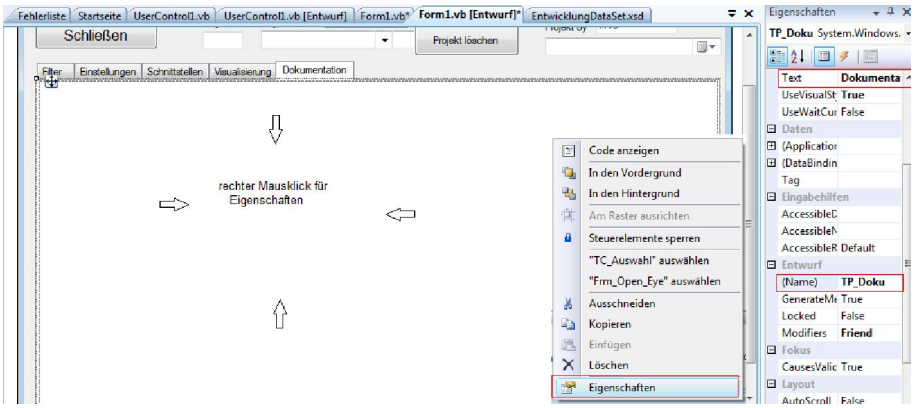
Damit wir mit den angelegten Datenbanktabellen auch etwas anfangen können müssen wir eine weitere Seite in das **TabControlObjekt** bringen.

Der Vorgang sollte uns noch geläufig sein. Um eine Seite hinzuzufügen, müssen wir den Focus auf das TabControl bekommen. Dazu klicken wir mit der rechten Maustaste in den freien Reiterbereich.




*Focus TabControl*

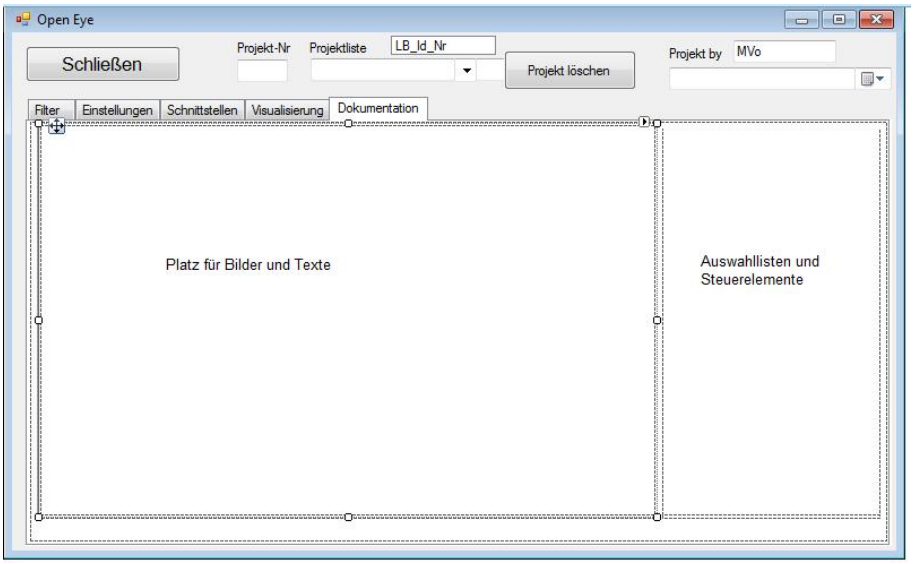
Es wird ein kleines Menü aufgeschlagen, wo wir in der Auswahl das Hinzufügen einer Registerkarte bekommen. Das ist genau das, was wir brauchen. Anschließend wird die eingefügte Seite parametriert. Sie bekommt eine Überschrift und den Namen **Tp_Doku**. Tp steht wieder für **TabPage**




*Einrichten Seite Dokumentation*

Für das weitere Vorgehen ist wieder ein kurzer Brainstorm angesagt. Wie soll die Information von dieser Seite geliefert werden. Ein Objekt für Bilder wäre schön. So bekommt man auch gleich eine Schaltung zu sehen.

Natürlich werden die Einträge aus der Datenbank in Comboboxen zur Auswahl gestellt und die Einstellung der Filter übernimmt ebenfalls eine Combobox. Setzen wir für eine grobe Orientierung zwei Panel auf die



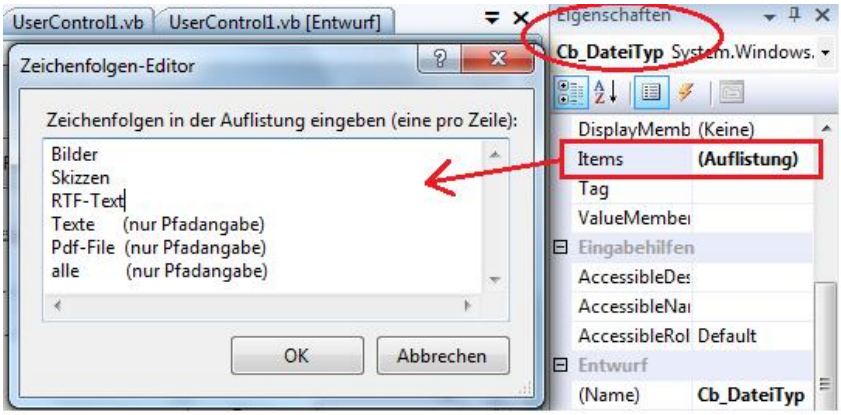

Seite. Ein großes links und ein etwas kleineres rechts.

*Einteilung Seite Dokumentation*

### *1.7.3.1 Objektauswahl zur Tabelle*

Beginnen wir mit den Objekten der rechten Seite, denn möglicherweise reicht der kleine Platz nicht aus und muss noch erweitert werden. Ganz oben wird ein Button eingebaut für **Dokumente laden**. Entsprechend bekommt es auch den Namen **Bt_LoadDoku**. Darunter wird eine Combobox gesetzt und über die Combobox eine Textbox. Zwischen Button und Combobox muss noch Platz für ein Label für die Überschrift **Dokumente** sein. Entsprechend erhält auch die Combobox den Namen **CB_DokuPfad** und die Textbox den Namen **Tb_DokuPfad**.

Um eine Sammlung von Dokumenten etwas zu ordnen, besitzt die Datenbanktabelle eine Spalte **DateiTyp**. Dafür setzen wir ebenfalls eine Combobox mit dem Namen **Cb_DateiTyp** und darüber wieder eine Textbox mit dem Namen **Tb_DateiTyp**. Die Combobox bekommt auch über ihre Eigenschaft **Items** die Einträge vorgegeben.




*Itemliste Dateitypen*

Nicht jede Datei lässt sich in unserem Programm darstellen. Bei **Rich-Text-Formaten** z.B. **WordPad** ist dies mit einer RichTextBox machbar. Bilder und Skizzen sind mit einer PictureBox darstellbar. Damit aber auch alle zum Projekt gehörenden Dateiverweise aufgenommen werden können, sind sie mit dem Zusatz **nur Pfadangabe** gekennzeichnet.

Zu diesen beiden Objekten gehört natürlich wieder ein Label mit der Beschriftung **Dateityp**

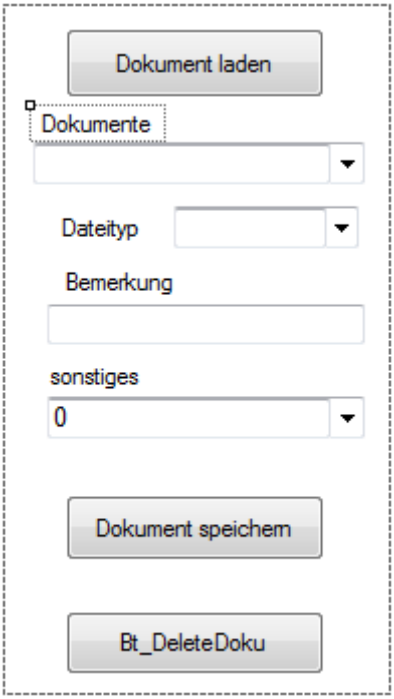
Eine weitere Textbox und ein Label ist für die Bemerkung fällig. Dem entsprechend ist die Beschriftung des Labels Bemerkung und der Name der Textbox Tb_Bemerkung.

Zusätzlich noch eine Kombination aus Textbox und Combobox für sonstiges. Die Spalte ist vom Typ Integer und als Referenz für mögliche Erweiterungen gedacht. Da diese Information noch keine Bedeutung hat, wird die Textbox mit einer 0 vorbesetzt und für den Zugriffgesperrt. Ebenfalls die Combobox. Trotzdem werden die Namen mit Tb_Sonstiges und Cb_Sonstiges vergeben.

Um die Tabelle Dokumentation der Datenbank mit Inhalten zu versorgen fehlen noch zwei Button. Diesmal müssen wir explizit die Information übergeben, also das **Dokument speichern**. Dafür richten wir ein Button **Bt_Store_Doku** ein.

Auch für die Aufgabe **Dokument löschen** benutzen wir ein Button **Bt_Delete-Doku**.

Diese Objekte bauen wir nun auf das rechte Panel der Seite Dokumentation ein.




*Bedienung Datenbanktabelle*

### *1.7.3.2 Auswahl Objekte für Bilder und Dokumente*

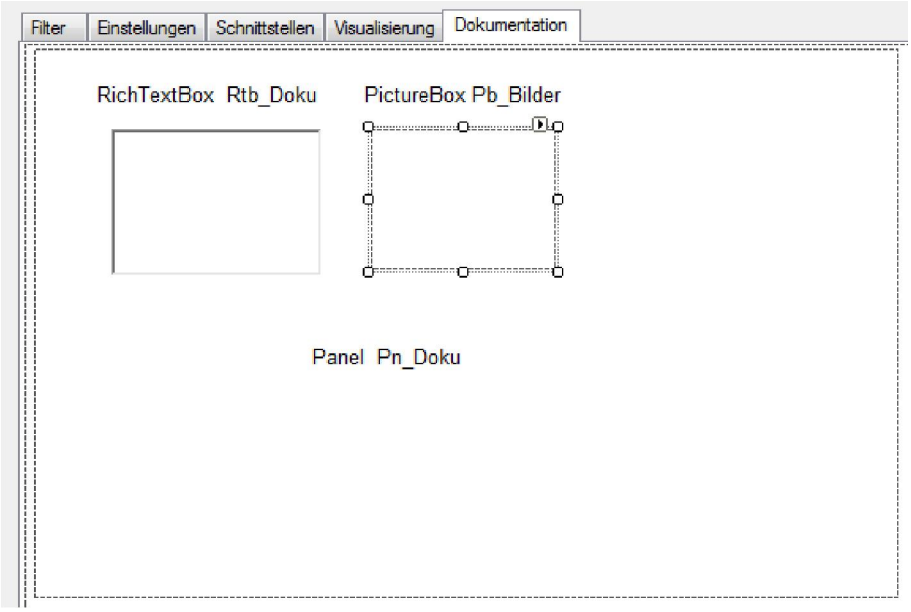
Wenden wir uns nun der linken Seite zu. Ich sagte bereits, dass dort eine RichTextBox eine direkte Ausgabe von einfachen Texten aus WordPad oder aus dem Editor möglich ist. Also bauen wir dort auch eine ein. Die Größe vergeben wir bei der Initialisierung der Anwendung. Dafür haben wir die Subroutine **Set_Defaults** geschrieben. Auch die Eigenschaft Parent setzen wir dort auf das Panel **Pn_Doku**. Wenn wir eine RichTextBox gleich über das ganze Panel ziehen, ist ein Zugriff auf weitere dort installierte Objekte nur sehr umständlich möglich. So wird diese RichTextbox einfach klein erstellt. Zur Laufzeit bekommt sie dann die volle Größe. Die RichTextBox bekommt den Namen **Rtb_Doku** und die PictureBox **Pb_Bilder.**

In der Sub-Routine **Set_Default** fügen wir nun die folgenden Befehle hinzu

```
Public Sub Set_Default()
    PB_Bilder.Parent = Pn_Doku
    PB_Bilder.Width = Pn_Doku.Width
    PB_Bilder.Height = Pn_Doku.Height
    PB_Bilder.Top = 0
    PB_Bilder.Left = 0
    Rtb_Doku.Parent = Pn_Doku
    Rtb_Doku.Width = Pn_Doku.Width
    Rtb_Doku.Height = Pn_Doku.Height
    Rtb_Doku.Top = 0
    Rtb_Doku.Left = 0
    .....
```

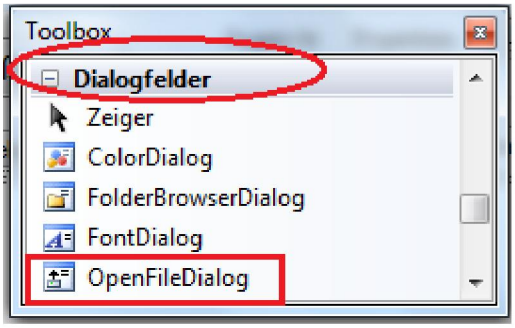
### *1.7.3.3 Dialogobjekt Dateiexplorer*

Wenn wir nun das Programm starten werden wir von den beiden Objekten nichts sehen. Damit etwas sichtbar wird, muss ein Text in die Richtextbox oder ein Bild in die Picturebox geladen werden. Soweit sind wir aber noch nicht und deshalb sehen wir uns erst einmal den Entwurf an.




*Objekte zur Ansicht der Dokumenten*

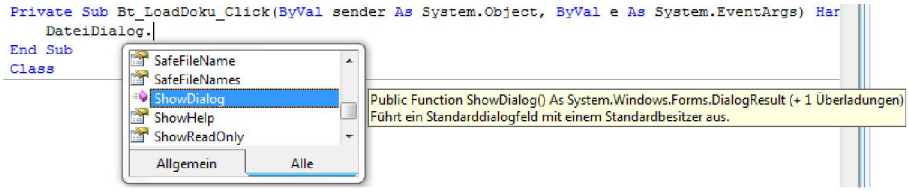
Nun suchen wir uns ein geeignetes Objekt in der Toolbox, um einen Dateizugriff zu bekommen.



*Dialog Dateizugriff*

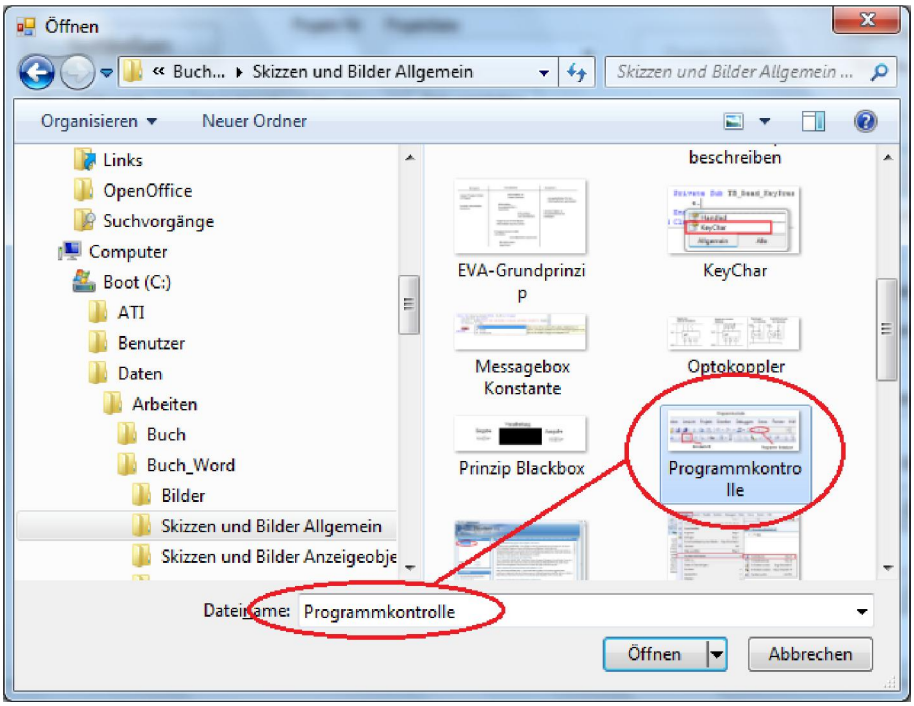
In der Rubrik Dialogfelder finden wir den Eintrag OpenFileDialog. Diesen ziehen wir auf unsere Anwendung und vergeben den Namen DateiDialog. Hier lass ich mal den Präfix weg, denn es ist auch der einzige installierte Dialog in dieser Anwendung.

Damit wir wissen, wie er arbeitet, rufen wir ihn mit dem Button **Bt_LoadDoku** einfach mal auf. Dazu wird das Ereignis **Bt_LoadDoku_Click** erzeugt und der Name des OpenfileDialog-Objektes eingegeben. Wieder ist ein besonderes Auge für die Hilfe und die angebotenen Funktionen erforderlich.




*Aufruf ShowDialog*

Dieser Eintrag scheint vielversprechend und so fügen wir ihn hier ein. Anschließend starten wir das Programm und betrachten nach einem Klick auf das Button Dokument laden des Ergebnis.



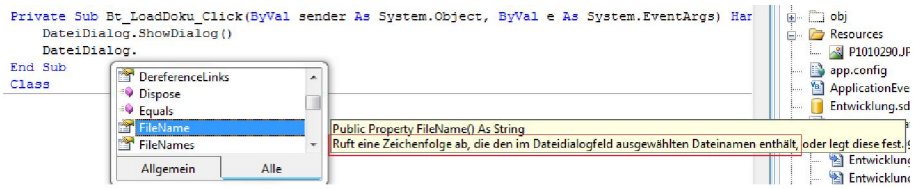
*Dateidialog*

Diesen Anblick kennen wir vom Windows-Explorer. Wenn wir eine Datei anklicken, erscheint ihr Name im Feld **Dateiname**. Schön eigentlich, aber ist das auch verwendbar für unsere Anwendung? Schauen wir einmal, welche Möglichkeiten sich da noch so bieten. Welche Information hält der Dateidialog noch bereit.

### *1.7.3.4 Dateinamen holen*

Wieder beginnen wir eine Befehlszeile mit DateiDialog und einem Punkt.

Bei der Suche durch die Liste findet man den Eintrag FileName. Werfen wir dort mal einen Blick auf die Information, die beim Anklicken des Eintrags geliefert wird.

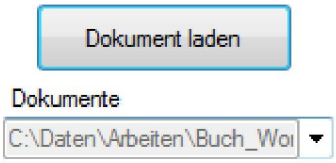



*Onlinehilfe Filename*

Vielleicht können wir diese Information der Textbox Tb_Doku_Pfad zuweisen. Ein Test bestätigt diesen Gedanken.

```
Private Sub Bt_LoadDoku_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_LoadDoku.Click
    DateiDialog.ShowDialog()
    Tb_Doku_Pfad.Text = DateiDialog.FileName
End Sub
```

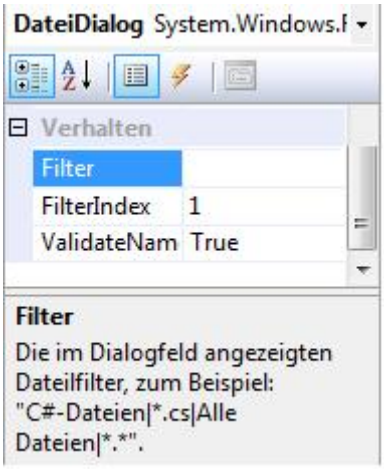
Diese Befehlszeilen öffnen den Dateidialog und bei Betätigen des Button Öffnen wird der Dateiname mit allen Pfadangaben in die Textbox kopiert, wie ein Ausschnitt vom Screenshot zeigt.




*Übernahme Dateiname*

### *1.7.3.5 Dateifilter setzen*

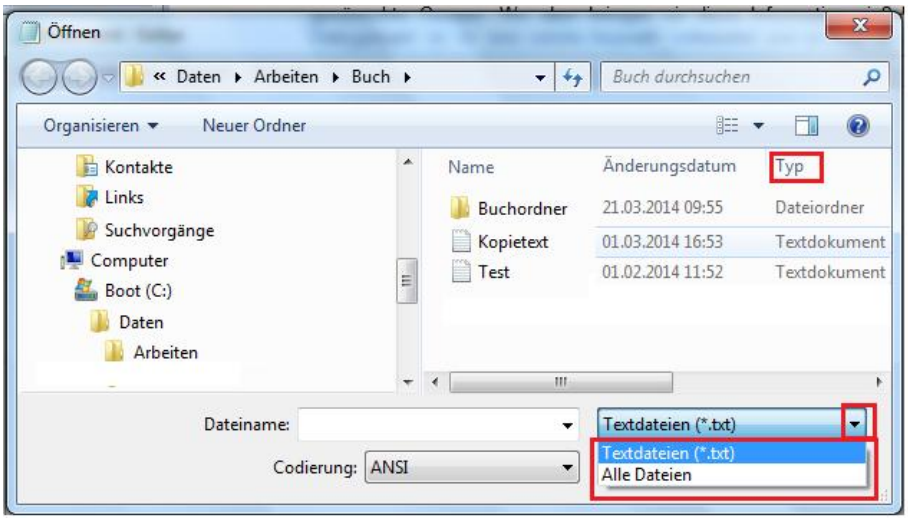
Es wird nun Zeit die Dateitypen auszuwählen, die wir in unserer Combobox hinterlegt haben. Dafür gibt es zwei Gründe. Einmal möchte ich wissen, ob ein Bild oder ein Text geladen wird um die Richtextbox oder die Picturebox sichtbar zu schalten. Zum anderen ist es auch schön übersichtlich, wenn nicht alle Dateien angezeigt werden, sondern nur die gewünschte Gruppe. Wo aber bringen wir diese Information ein? Das Dialogobjekt ist für eine solche Auswahl vorbereitet und es liegt an uns, die Eigenschaft zu finden, wo der Dateifilter untergebracht ist. Nachdem das Wort bereits gefallen ist, dürfte es nicht schwer sein. Filter ist das Stichwort und in den Eigenschaften finden wir es wieder mit einer kurzen Erklärung.




*Eigenschaft Dateifilter*

Das ist nicht ganz einfach zu verstehen deshalb gehe ich hier etwas genauer darauf ein. Wie so ein Filter aussieht, erkennt man Beispielsweise, wenn eine Textdatei in den Editor geladen werden soll.

Es kann nicht schaden, sich solch eine Dateiauswahl von einem Programm einmal anzusehen. Alle Programme verwenden Filter. Der Aufbau ist folgender: in der Liste steht der Name des Dateityps und das Kürzel, welches der Datei angehangen ist.




*Ansicht Dateiexplorer*

Wir sehen hier eine Combobox mit Textinhalten. Das ist in der Hilfe vergleichbar der Textteil vor dem senkrechten Strich. Der Teil dahinter ist der Suchschlüssel für die Datei. Also **Textdateien (*.txt)** ist der sichtbare Teil in der Liste. Nicht sichtbar ist der Suchfilter ***.txt**. Ein ganzer Eintrag im Filter ist also **Textdateien (*.txt)|*.txt**

Wollen wir einen Filter für PDF-Files setzen muss der Filter entsprechend **PDF-Files (*.pdf)|*.pdf** sein. Übrigens das **|** -Zeichen bekommt man mit **AltGr und <.**

Nun ist unser Filter abhängig von dem ausgewählten Dateityp in der Combobox **Cb_DateiTyp**. Die Itemliste hatten wir bereits vorbesetzt. Somit ist der erste Schritt die Auswahl auch sichtbar zu machen. Dazu wird die Ereignismethode **Cb_Dateityp_SelectedIndexChange** erzeugt und eine Kopie von **Cb_DateiTyp.Text** in **Tb_Dateityp.Text** eingetragen. Anschließend wird der Dateifilter des Dateidialogs beschrieben.

```
Private Sub Cb_DateiTyp_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_DateiTyp.SelectedIndexChanged
    Tb_Dateityp.Text = Cb_DateiTyp.Text
    DateiDialog.Filter = "alle Dateien (*.*)|*.*"
    If Tb_Dateityp.Text = "Bilder" Then
      DateiDialog.Filter = "Bilder (*.jpg;*.jpeg;*.jpe)|*.jpg;*.jpeg;*.jpe|Bilder (*.Gif)|*.gif|Bilder
(*.Tif;*.Tiff)|*.tif;tiff"
    End If
    If Tb_Dateityp.Text = "Skizzen" Then
      DateiDialog.Filter = "Grafikdateien (*.png)|*.png|Paint Dateien (*.bmp)|*.bmp"
    End If
    If Tb_Dateityp.Text = "PDF-Files" Then DateiDialog.Filter = "Pdf Dateien (*.pdf)|*.pdf"
    If Tb_Dateityp.Text = "RTF-Text" Then DateiDialog.Filter = "RichTextdatei (*.rtf)|*.rtf"
    If Tb_Dateityp.Text = "Text" Then DateiDialog.Filter = "Textdatei (*.txt)|*.txt"
  End Sub
```

Sehen wir uns einmal den Filter für Grafikdateien an

```
      DateiDialog.Filter = "Grafikdateien (*.png)|*.png|Paint Dateien (*.bmp)|*.bmp"
```

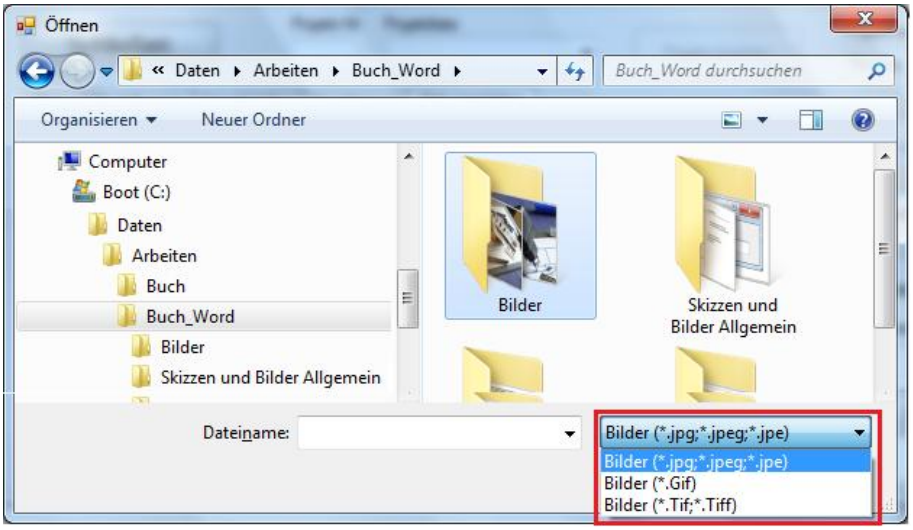
Dabei ist der erste Eintrag in der Auswahlliste

```
"Grafikdateien (*.png)
```

Und der zweite Listeneintrag

```
Paint Dateien (*.bmp)
```

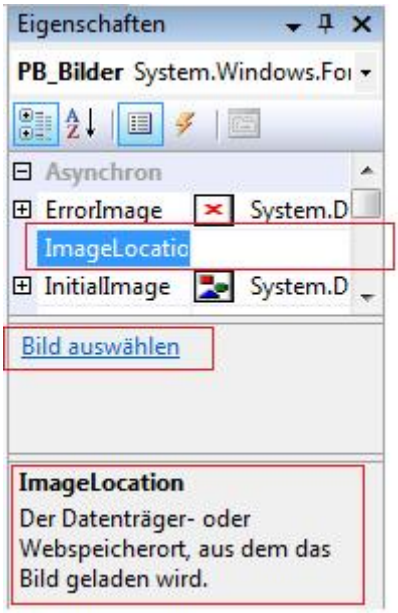
Die Funktion des Filters können wir nun testen und sehen, ob unsere Erwartungen erfüllt werden. Der Screenshot zeigt das Ergebnis.




*Auswahl Dateityp Bilder*

### *1.7.3.6 Bilddateien laden*

Auch wenn nicht alles sichtbar, so ist doch der gesamte Pfad mit der Datei in der Textbox enthalten. Suchen wir uns doch einmal eine Bilddatei und öffnen sie. Nun soll sie aber auch gleich in der Picturebox Pb_Bilder angezeigt werden. Der Blick in die Eigenschaften gibt nicht sofort die Lösung. So ist wieder das Gefühl für Eigenschaften gefragt. Was kommt in Frage, was kann ausgeschlossen werden. Ist man sich nicht sicher, probiert man es einfach aus. Wichtig ist der Hinweistext, der beim Anklicken der Eigenschaft im unteren Bereich erscheint.

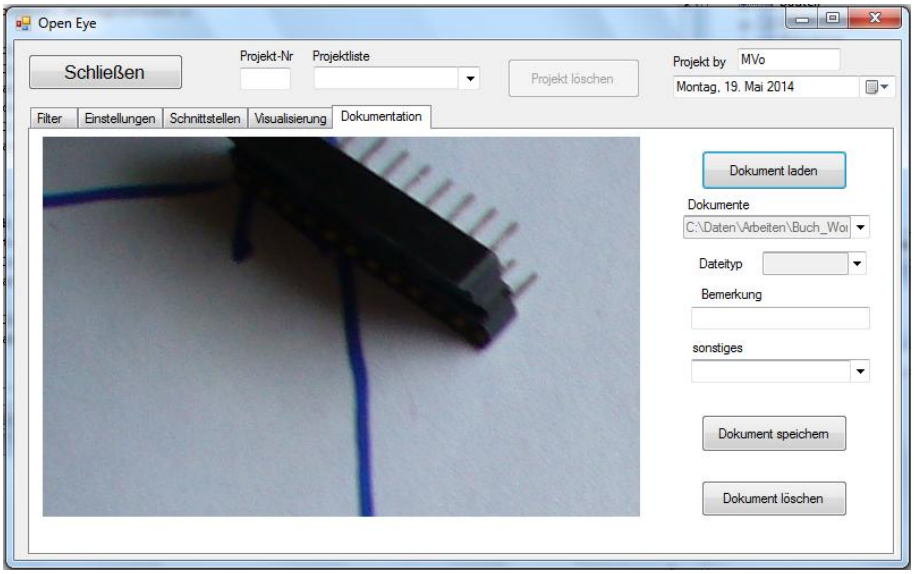



*Picturebox Bild laden*

ImageLocation – Bildposition – Nicht sofort ist der Bezug zur Datei gegeben. Egal, ausprobieren und den Text aus der Textbox Tb_Doku_Pfad dort eintragen.

```
Private Sub Bt_LoadDoku_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_LoadDoku.Click
    DateiDialog.ShowDialog()
    Tb_Doku_Pfad.Text = DateiDialog.FileName
    PB_Bilder.ImageLocation = Tb_Doku_Pfad.Text
End Sub
```

Das Ergebnis ist zwar positive, aber damit ist nichts anzufangen.



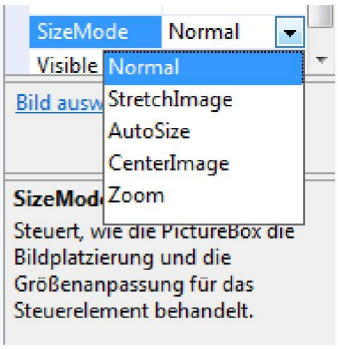

*Picturebox Bildausschnitt*

Es ist nur ein kleiner Ausschnitt des Bildes erkennbar. Ok, also, das Bild passt nicht in diese Picturebox. Bei den RichTextboxen wurden bei größeren Datenmengen immer Scrollbalken sichtbar und selbst unser Panel von der Variablenansicht konnte welche erzeugen, wenn nicht alle Anzeigeobjekte im sichtbaren Bereich waren. Also wird wieder ein Blick in die Eigenschaften erforderlich, ob es dort eine Zuschaltung von Scrollbalken gibt.

### *1.7.3.7 Picturebox anpassen*

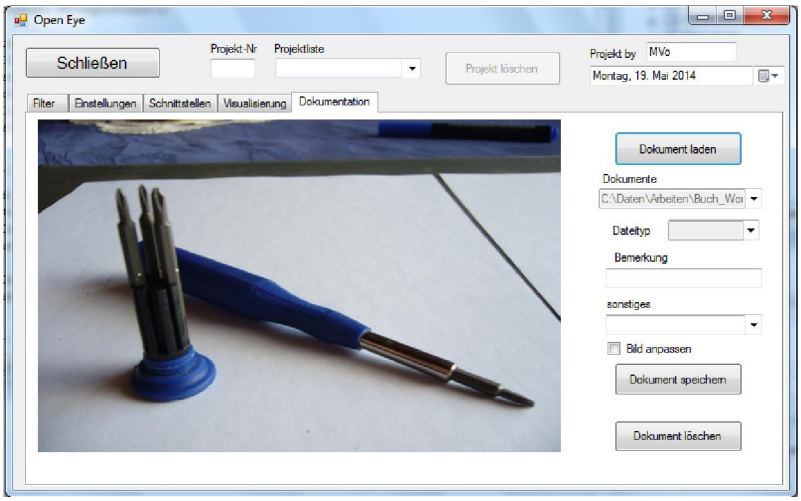
Trotz genauem Hinsehen, nichts deutet direkt auf einen Scrollbalken hin und so ist wieder mal das Gefühl gefragt. Klicken wir uns also durch die Eigenschaften und lesen die Information.

Bei SizeMode klingt die Information nicht schlecht und wir werfen einen Blick in die DropDown-Liste.



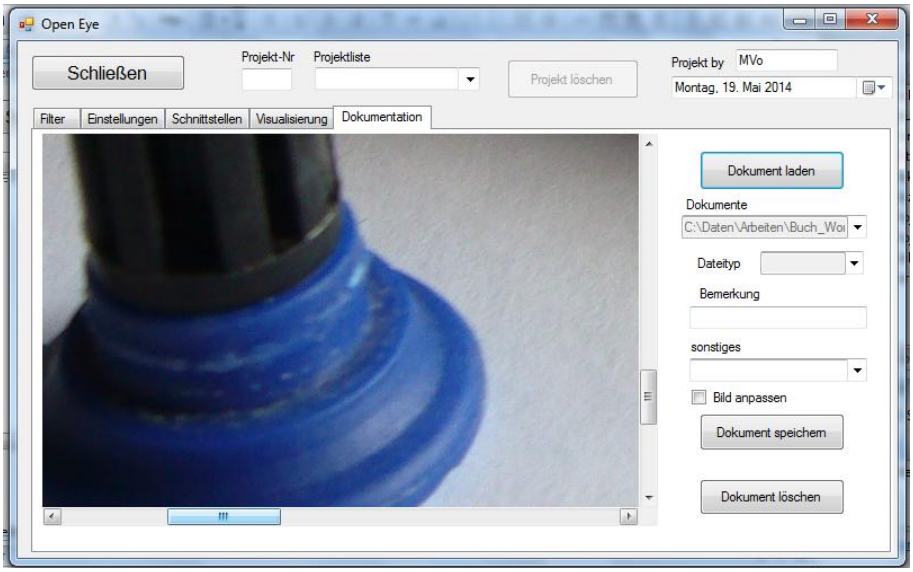

*Picturebox Bildanpassung*

Ein Versuch mit StretchImage kann nicht schaden.




*PictureBox ganzes Bild*

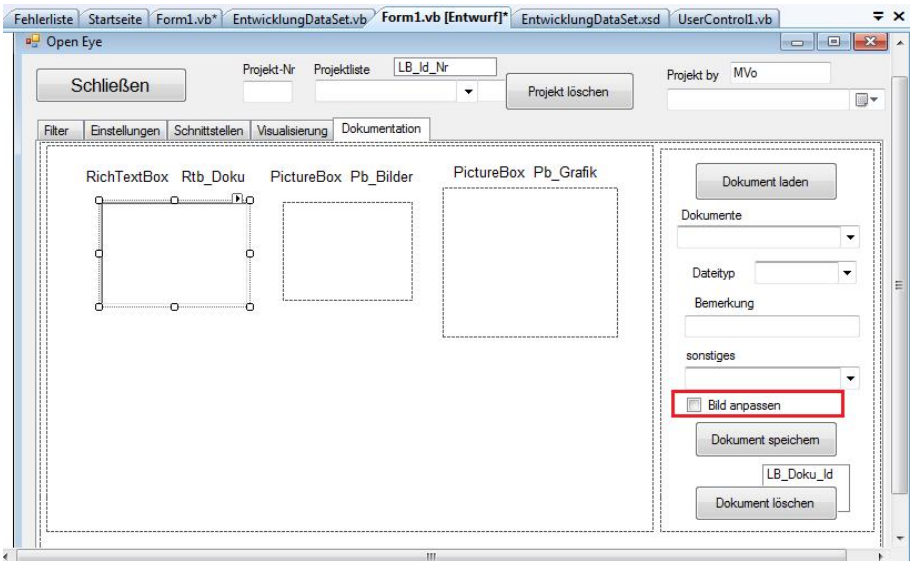
Na, das war ein Volltreffer. Aber was ist mit AutoSize? Den Versuch ist es Wert und es ist ja auch kein großer Aufwand. Es könnte ja sein, das größere Bilder sonst nicht richtig dargestellt werden und die Einstellung nur zufällig das ganze Bild zeigt. Mal sehen, wie dieses Bild mit der Funktion AutoSize dargestellt wird.




*PictureBox AutoSize*

Also hier sind die Scrollbalken versteckt. Das bringt mich auf eine Idee. Vielleicht ist eine komplette Darstellung nicht immer so deutlich und man möchte Details besser erkennen. Wenn man nun zwei Pictureboxen einbaut, beiden das gleiche Bild zuordnet und zwischen den beiden Darstellungen umschalten kann ist eine Gesamtansicht und eine Detailansicht möglich. Also legen wir eine zweite Picturebox auf das Panel und verfahren in der Subroutine **Set_Default** für die Größe der PictureBox nach dem bekannten Muster.

Zusätzlich fügen wir eine Checkbox ein. Sie soll die Umschaltung der beiden Pictureboxen erledigen. Die zweite Picturebox erhält den Namen **Pb_Grafik** und die Checkbox **Cb_Picture_Scale**




*Aufbau Seite Doku erweitert*

Die Eigenschaft **SizeMode** für die PictureBox **Pb_Bilder** wird mit **Zoom** besetzt. **StrechImage** könnte das Bild verfälschen, weil es in die Picturebox eingepasst wird. **Zoom** dagegen behält das Seitenverhältnis bei.

Für die Picturebox **Pb_Grafik** ist Die Eigenschaft **SizeMode AutoSize** richtig. Hier bekommen wir die Möglichkeit, eine Detailansicht über die Scrollbalken anzufahren.

### *1.7.3.8 Textbox oder Picturebox*

Durch den Dateityp wissen wir, welches Dokument geladen wird. Abhängig davon werden wir nun die Richtextbox oder eine der beiden Pictureboxen sichtbar schalten. Diesmal nehmen wir das Ereignis vom Button Dokument laden.

Um den Dateityp zu bestimmen genügt es die Position in der Combobox abzufragen und den Wert in eine Integervariable zu legen. Mit diesem Wert ist über eine **Select Case** Abfrage einfach möglich. **Select Case**, das ist mal wieder etwas Neues. Natürlich hätten es auch eine Anzahl von If-Anweisungen getan, aber wir wollen ja Basic lernen und **Select Case** ist eine elegante Art, über den Wert einer Variablen Programmblöcke zu bearbeiten. Doch da sind wir noch nicht. Zusätzlich zu den bisherigen Parametern setzen wir das Startverzeichnis **InitialDirectory** des Dateidialogs auf das Standardlaufwerk **C**. Natürlich darf es auch ein anderes Laufwerk sein, wenn die Daten woanders liegen. Dann rufen wir den Dialog auf. Wird er geschlossen, ist der gewählte Dateiname verfügbar. Soweit waren wir ja schon. Nun geben wir im ersten Schritt die Checkbox **Cb_Picture_Scale** frei. Sie wird bei Textdateien gesperrt um nicht ungewollt ein Bild über den Text zu legen. Nachdem wir die Position des Eintrags in der Liste abgefragt haben, steht der bereits erwähnten **Select Case** Anweisung nichts im Weg. Bis hierher erst einmal der Code.

```
Private Sub Bt_LoadDoku_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_LoadDoku.Click
    Dim Typ As Integer
     DateiDialog.InitialDirectory = "c:\"
    'DateiDialog.RestoreDirectory = True
    DateiDialog.ShowDialog()
    Tb_Doku_Pfad.Text = DateiDialog.FileName
     Cb_Picture_Scale.Enabled = True
    Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
    Cb_Picture_Scale.Enabled = True
    Select Case Typ
       Case 0                    ' Bilder
       Case 1                    ' Skizzen
       Case 2                    ' RichText (WordPad)
       Case 3                     ' Text ( Editor )
       Case 4                     ' Pdf-Files
    End Select
  End Sub
```

Zuerst wird der Fall 0 bearbeitet. Da der Dateidialog Bilddateien zurückliefert, ist es sinnvoll, die Picturebox mit der Einstellung **Zoom** sichtbar zu schalten und evtl. auch die Checkbox zurück zu setzen.

```
Case 0                          ' Bilder
        Cb_ Picture_Scale .Checked = False
        PB_Bilder.Visible = Not Cb_Picture_Scale.Checked
        Pb_Grafik.Visible = Cb_Picture_Scale.Checked
        Rtb_Doku.Visible = False
        PB_Bilder.ImageLocation = Tb_Doku_Pfad.Text
        Pb_Grafik.ImageLocation = Tb_Doku_Pfad.Text
```

Diesen Schritt testen wir erst einmal aus und sehen uns das Ergebnis an. Da beide Pictureboxen das Bild zugewiesen bekommen, sollten wir nun an die Umschaltung denken. Dazu wird die Ereignismethode der **Cb_Picture_Scale_CeckedChange** programmiert. Es genügt die beiden Zeilen **.Visible** der Pictureboxen dort einzusetzen.

```
Private Sub Cb_Picture_Scale_CheckedChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Picture_Scale.CheckedChanged
      PB_Bilder.Visible = Not Cb_Picture_Scale.Checked
      Pb_Grafik.Visible = Cb_Picture_Scale.Checked
   End Sub
```

Mit einem erneuten Test sollte nun eine Umschaltung der beiden Darstellungen überprüft werden.

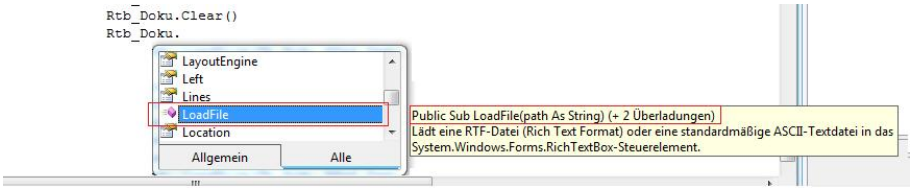
Kommen wir zu den Skizzen. Entgegen der ersten Ansicht bei Bildern, wo ein Gesamtbild zuerst aufgeschlagen wird, kommt hier das Detailbild zur ersten Ansicht. Aber selbstverständlich ist auch eine Umschaltung zwischen den Darstellungen möglich.

```
Case 1                          ' Skizzen
            Cb_Picture_Scale.Checked = True
            Rtb_Doku.Visible = False
            PB_Bilder.Visible = Not Cb_Picture_Scale.Checked
            Pb_Grafik.Visible = Cb_Picture_Scale.Checked
            PB_Bilder.ImageLocation = Tb_Doku_Pfad.Text
            Pb_Grafik.ImageLocation = Tb_Doku_Pfad.Text
```

Nun stehen uns auch die mit Paint erstellten Zeichnungen zur Verfügung und mit einem weiteren Test wird das Ergebnis überprüft.

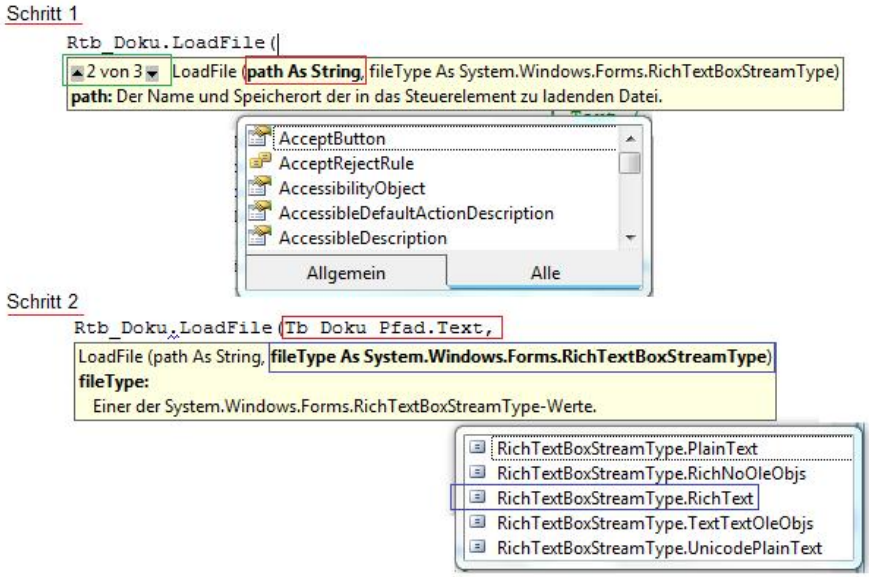
Kommen wir im nächsten Schritt zu den Texten. Hier ist es erforderlich, das die Pictureboxen ausgeblendet werden und auch bleiben. Deshalb wird die Checkbox **Cb_Picture_Scale** auch gesperrt, die Pictureboxen ausgeblendet und die Richtextbox **Rtb_Doku** sichtbar gemacht. Die nächste Aktion ist das entfernen eventuell vorhandener Texte. Anschließend wird die Datei über **LoadFile** eingetragen.

Nun, **LoadFile** ist keine Eigenschaft der Picturebox, die im Eigenschaftskatalog aufgeführt ist. Um darauf zu kommen, muss halt wieder mächtig kombiniert werden. Es ist kein Text, der dieser **RichTextBox** zugeführt wird sondern eine Datei oder ein File. Wenn der Eigenschaftskatalog nichts dafür anbietet, muss die Online-Hilfe herhalten.




*RichTextBox LoadFile*

Um diesen Eintrag zu finden wird einfach die Onlinehilfe durchgeblättert bis ein Eintrag zu passen scheint. Klickt man diesen dann an, wird weitere Information und Auskunft über die Parameter geliefert. Im rot eingerahmten Hinweistext steht etwas von zwei Überladungen. Das schauen wir uns noch einmal genauer an.

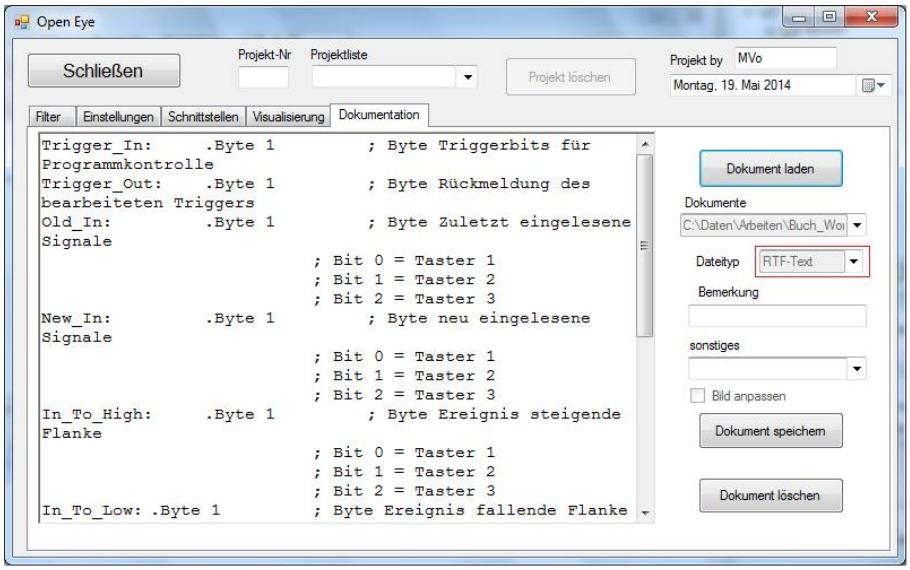



*RichTextBox Onlinehilfe LoadFile*

Mit dem Öffnen der Klammer hinter **LoadFile** wird in der Online-Hilfe mit einem kleinen Auswahlmenü auf die drei Überladungen hingewiesen. Ich habe den Bereich grün umrahmt. Blättern wir einmal durch und sehen uns die Überlagerungen an. Path ist klar, das ist der Text in der Textbox **Tb_Doku_Pfad**. FileType könnte ein brauchbarer Hinweis sein und so erscheint bei der zweiten Überlagerung in Schritt zwei auch eine weitere Auswahl von Filetypen. Setzen wir einmal den blau eingerahmten Text **RichTextBoxStreamType.RichText** ein.

```
Case 2                          ' RichText (WordPad)
        Cb_Picture_Scale.Enabled = False
        PB_Bilder.Visible = False
        Pb_Grafik.Visible = False
        Rtb_Doku.Visible = True
        Rtb_Doku.Clear()
        Rtb_Doku.LoadFile(Tb_Doku_Pfad.Text, RichTextBoxStreamType.RichText)
```

Ein **Stream**, nur um das mal kurz zu erklären, ist wie ein Datenstrom zu betrachten, der in die **RichTextBox** fließen soll. Und ein anschließender Test gibt der Annahme Recht. Dazu brauchen wir nur eine Datei mit WordPad anlegen und diese dann mit der Auswahl Dateityp **RTF-Text** aufrufen.




*Ansicht WordPad Datei*

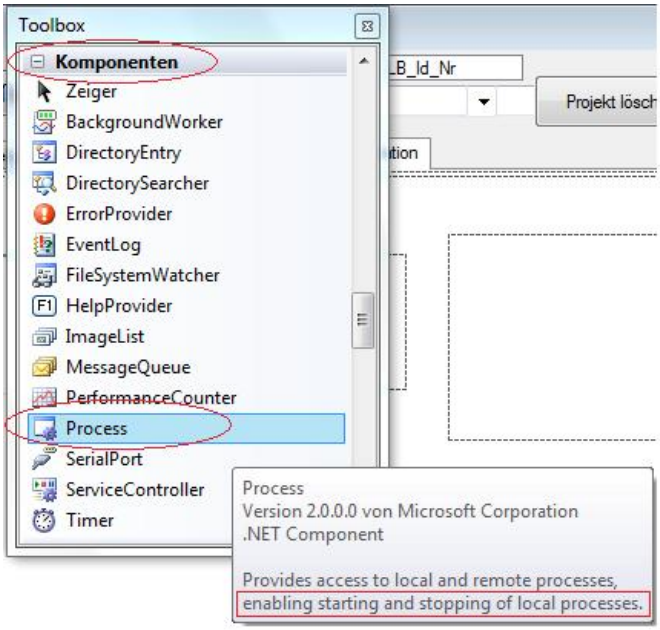
Nun hätten wir aber auch ganz gern einfache Editor-Texte im Format **txt** angezeigt. Das erledigen wir mit **Case 3**. Der Weg ist ja bekannt, lediglich der **RichTextBoxStreamType** wird mit **PlainText** ausgewählt.

```
Case 3                          ' Text ( Editor )
        Cb_Picture_Scale.Enabled = False
        PB_Bilder.Visible = False
        Pb_Grafik.Visible = False
        Rtb_Doku.Visible = True
        Rtb_Doku.Clear()
        Rtb_Doku.LoadFile(Tb_Doku_Pfad.Text, RichTextBoxStreamType.PlainText)
```

### *1.7.3.9 Eine fremde Anwendung starten*

Bleiben noch Pdf-Files, wie sie vielfach für Datenblätter benutzt werden. Hier gibt es kein Objekt, welches diesen Dateitypen ausgeben kann. Wie bekommen wir aus unserer Anwendung heraus ein Pdf-File zu Gesicht. Klar, es geht auch ohne, aber ein Programmierer sollte erst einmal einen Standpunkt haben: **Geht nicht gibt's nicht**. Es geht vielleicht über Umwege, aber es geht. Was passiert eigentlich, wenn man im Windows Explorer einen Doppelklick auf eine Datei macht. Es wird die Anwendung mit der ausgewählten Datei geöffnet. Genauso auch, wenn eine ausgewählte **Pdf_Datei** mit dem Button **Öffnen** aktiviert wird. Geht das auch mit unserem Programm? Nun, der Dateidialog öffnet nicht eine Pdf-Datei, was wir sehr schnell herausbekommen. Da tut sich einfach nichts, also ist etwas mehr programmaufwand erforderlich. Vielleicht ist in der Toolbox so etwas wie Task oder Prozess. Auch Execute wäre vorstellbar.

Und tatsächlich. In der Rubrik Komponenten taucht das Objekt Prozess auf. Wird der Eintrag mit der Maus berührt erscheint der Hinweis, dass hier Prozesse gestartet und gestoppt werden können. Das probieren wir einfach mal aus und ziehen einen **Process** auf unsere Anwendung.




*Prozess aufrufen*

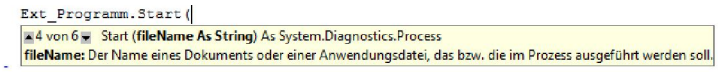
Das Objekt landet nicht in unserer Anwendung, sondern darunter, wo auch die Timer, der Dateidialog und die ganzen Tableadapter stehen. Den Namen **Prozess1** ändern wir in **Ext_Progamm.**


*Externer Prozess*

Nun wenden wir uns der letzten Case-Anweisung zu.

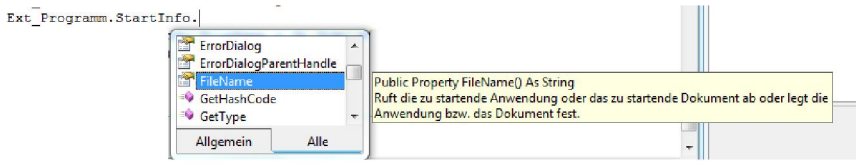
```
Case 4                          ' Pdf-Files
        PB_Bilder.Visible = False
        Pb_Grafik.Visible = False
        Rtb_Doku.Visible = True
        Rtb_Doku.Clear()
        Rtb_Doku.Text = "PDF Datei wird geladen"
        Ext_Programm.Start(Tb_Doku_Pfad.Text)
```

Die Richtextbox benutzen wir für eine Meldung, dass ein PDF-File geladen wird. Das kann schon mal ein bisschen dauern, denn manche Dateien sind nicht gerade klein. Zum Beispiel Datenblätter von Controllern. Auch hier führt der Weg über die Online Hilfe zum Ergebnis




*Programm starten*

Nun kann es möglich sein, das dieser Zugriff nicht funktioniert. Man kann diese Funktion auch anders aufrufen. Dazu wird die Startinfo benötigt. Auch hier der Blick auf die Online Hilfe




*StartInfo Process*

Wird Start ohne einen Parameter aufgerufen, so wird die Datei geöffnet, die in der StartInfo hinterlegt ist. Die beiden folgenden Zeilen

```
Ext_Programm.StartInfo.FileName = Tb_Doku_Pfad.Text
Ext_Programm.Start()
```

ersetzen die Zeile

```
Ext_Programm.Start(Tb_Doku_Pfad.Text)
```

Wird nun das Programm gestartet und ein PDF-File geladen wird auch ein Reader geöffnet, der die Datei darstellen kann. Allerdings bleibt der Text in der RichTextBox stehen. Das ist unschön und nicht gewollt. Direkt hinter dem Startbefehl können wir ihn nicht löschen, da ein anderes Programm gestartet wird. Visual Basic leitet diesen Vorgang nur ein, aber wartet nicht auf Vollzug. Allerdings hilft ein kleiner Trick.

Unser Programm wird zum Start kurz verlassen. Das löst ein Ereignis Activated bei Rückkehr aus. Dort löschen wir den Text in der RichTextBox.

```
Private Sub Frm_Open_Eye_Activated(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Activated
    Rtb_Doku.Clear()
  End Sub
```

Allerdings hat das einen unschönen Nebeneffekt. Verlässt man Open_Eye um ein anderes Programm zu starten oder dort etwas nachzuschlagen sind die Informationen in der RichTextBox gelöscht. Aber wir sind erfinderisch. Es ist ja nur ein Objekt erforderlich, das zum Starten eingeblendet wird und diesen Vorgang signalisiert. Es soll auch ein bisschen nett sein, mal sehen, was die Toolbox bietet.

### *1.7.3.10 Eine Fortschritsanzeige*

In der Toolbox finden wir ein Objekt ProgressBar. Wenn wir dieses in unserer Anwendung einbauen und zum Start des externen Programms sichtbar schalten, zusätzlich einen Timer starten, der in einem Intervall von 50 msek. den Wert in der Progressbar erhöht und bei Erreichen des Maximalwertes wieder bei 0 beginnt, sollte dies eine fürs Auge gut sichtbare Einrichtung werden.

Dazu wird eine Progressbar auf Seite 5 eingebaut. Der Name Pb_Fortschritt ist auch bezeichnend. Anfangs ist das Objekt unsichtbar. Die Eigenschaft Step setzen wir auf 1 und Style auf Continius

Ein weiterer Timer Tr_Fortschritt wird mit einem Intervall von 50 besetzt und ist anfangs auch abgeschaltet.

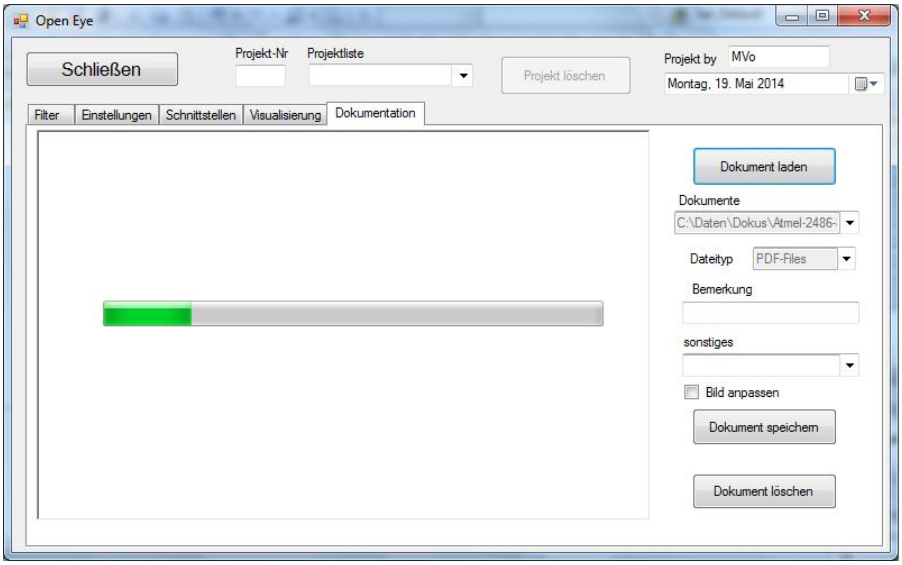
Die Ereignismethode Activate ändern wir ab.

```
Private Sub Frm_Open_Eye_Activated(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Activated
    Pb_Fortschritt.Value = 0
    Pb_Fortschritt.Visible = False
    Tr_Fortschritt.Enabled = False
End Sub
```

Und auch in case 4 wird das Programm entsprechend geändert. Die RichTextBox bekommt nun keinen Text mehr zugewiesen, statt dessen visualisieren wir die ProgressBar und geben den Timer frei.

```
    Pb_Fortschritt.Visible = True
    Tr_Fortschritt.Enabled = True
    Ext_Programm.Start(Tb_Doku_Pfad.Text)
```

Schließlich testen wir das Ergebnis.




*Fortschrittsanzeige*

### *1.7.3.11 Datei zum Projekt hinzufügen*

Wir können nun schon einen kleinen Erfolg feiern, denn die restlichen Aufgaben erledigen wir doch fast mit links. Trotzdem sollten wir es langsam angehen lassen. Beginnen wir mit der Aufnahme der Dokumentation in unsere Projektdatei.

Dazu wird in bereits geübter Manier eine Datei geladen. Mit dem Button Dokument speichern soll der Dateipfad, die Information aus Sonstiges und Bemerkung in der Datenbanktabelle abgelegt werden. Eine separate Subroutine ist hier nicht erforderlich, da die Daten alle vorliegen und die Insert-Methode vom Tableadapter auch nur einmal aufgerufen wird.

Die Struktur ist relativ einfach. Für die Übergabeparameter definieren wir zuerst die Variablen, damit nicht in den Übergabeparametern noch Daten zurechtgelegt werden müssen. Außerdem bleibt die Parameterliste übersichtlich. Die Routine beginnt mit dem Laden der aktuellen Projekt_Nr. Dann wird geprüft, ob ein Projekt angelegt ist. Dies ist der Fall, wenn die Projektnummer größer 0 ist. Wird eine Datei gespeichert rufen wir die Routine Load_Doku auf, um die Combobox Cb_Doku_Pfad hinter der Textbox Tb_Doku_Pfad mit vorhandenen Einträgen zu füllen. Dabei wird der Dateityp der Subroutine mit übergeben. In einer Stringvariablen wird das Ergebnis dieser Routine aufbereitet und am Ende ausgegeben.

```
Private Sub Bt_Store_Doku_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Store_Doku.Click
    Dim Proj_Nr As Integer
    Dim Datei As String
    Dim Typ As Integer
    Dim Bem As String
    Dim sonst As Integer
    Dim Info As String
    Proj_Nr = Val(Tb_Projekt_Id.Text)
    If Proj_Nr > 0 Then
      Datei = Tb_Doku_Pfad.Text
      Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
      Bem = Tb_Bemerkung.Text
      sonst = Val(Tb_Sonstiges.Text)
      DokumentationTableAdapter.Insert_Doku(Proj_Nr, Datei, Typ, Bem, sonst)
      Load_Doku(Proj_Nr, Typ)
      Info = "Der Dateipfad ist auf der Datenbank gespeichert" + Chr(13)
    Else
      Info = "Es ist noch kein Projekt angelegt" + Chr(13)
      Info = Info + "Dateipfad konnte nicht gespeichert werden"
```

```
    End If
    MsgBox(Info, MsgBoxStyle.OkOnly, AcceptButton)
  End If
End Sub
```

Nun ist die Subroutine **Load_Doku** noch gar nicht vorhanden, aber das kennen wir ja schon. Im Bereich der anderen Load-Routinen ist diese schnell angelegt.

### *1.7.3.12 Projektdokumentation laden*

Diese Subroutine deckt gleich mehrere Bedingungen ab. So soll beim Laden eines Projektes eine Liste aller zum Projekt gehörenden Dokumente in der Combobox **Cb_Doku_Pfad** angelegt werden. Wird ein Dateityp gewählt, soll die Liste auf diesen Dateitypen begrenzt werden. Für uns bedeutet das verschiedene Aufrufstellen im Programm. Hier ist schon mal die erste angegeben hinter der **Insert**-Anweisung, um die Liste neu zu laden. Einen weiteren Aufruf packen wir in die Routine **Load_Projekt** die von der Combobox **Cb_Projekte** oder beim Start aufgerufen wird. Doch zuerst einmal die Subroutine **Load_Doku**. Als Parameter werden Projektnummer und Dateityp mit übergeben.

```
Public Sub Load_Doku(ByVal Projekt As Integer, ByVal DateiTyp As Integer)
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Id_Pos As Integer
    Cb_DokuPfad.Items.Clear()
    Pn_TeileKorr.Enabled = False
    Tb_Doku_Nr.Text = "-1"          ' keine Freigabe Teileliste
    If DateiTyp = 5 Then            ' alle Dateien
      Anzahl = DokumentationTableAdapter.Get_ProjektDoku(Projekt).Rows.Count
      If Anzahl > 0 Then
        For i = 0 To Anzahl - 1

Cb_DokuPfad.Items.Add(DokumentationTableAdapter.Get_ProjektDoku(Projekt).Item(i).Dat
eipfad)

LB_Doku_Id.Items.Add(DokumentationTableAdapter.Get_ProjektDoku(Projekt).Item(i).Id_Nr)
        Next
        ' Bei Neuer Datei bleibt der Eintrag im Textfeld erhalten
        If Tb_Doku_Pfad.Text = "" Then Tb_Doku_Pfad.Text = Cb_DokuPfad.Items(0)
      End If
    Else
      Anzahl = DokumentationTableAdapter.Get_ProjektDoku_Typ(Projekt,
DateiTyp).Rows.Count
      If Anzahl > 0 Then
        For i = 0 To Anzahl - 1

Cb_DokuPfad.Items.Add(DokumentationTableAdapter.Get_ProjektDoku_Typ(Projekt,
DateiTyp).Item(i).Dateipfad)

LB_Doku_Id.Items.Add(DokumentationTableAdapter.Get_ProjektDoku_Typ(Projekt,
DateiTyp).Item(i).Id_Nr)
        Next
```

```
            ' Bei Neuer Datei bleibt der Eintrag im Textfeld erhalten
            If Tb_Doku_Pfad.Text = "" Then Tb_Doku_Pfad.Text = Cb_DokuPfad.Items(0)
        End If
    End If
    If Anzahl > 0 Then
        Id_Pos = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
        If Id_Pos >= 0 Then
            Tb_Doku_Nr.Text = LB_Doku_Id.Items(Id_Pos)        ' Dokument-Id ermitteln
            Load_Dokudaten(Val(Tb_Doku_Nr.Text), DateiTyp)
        End If
    Else
        Cb_DokuPfad.Enabled = False
        Rtb_Bemerkung.Enabled = False
        Cb_Sonstiges.Enabled = False
        Tb_Sonstiges.Enabled = False
    End If
  End If
End Sub
```

Liegen aus der Abfrage keine Ergebnisse vor, wird die Bedienung der Comboboxen **Cb_Doku_Pfad** und **Cb_Sonstiges** sowie die Textboxen **Tb_Sonstiges** und **Rtb_Bemerkung** gesperrt. Im anderen Fall muss der angezeigte Datensatz in die zugeordneten Objekte eingetragen werden. So soll die Richtextbox **Rtb_Bemerkung** die entsprechende Information erhalten. Die Spalte **sonstiges** wird zwar auch mit eingetragen, ist aber ohne Bedeutung. Diese neue Subroutine bekommt den Namen **Load_Dokudaten.** Der Aufruf dieser Subroutine ist bereits in **Load_Doku** eingefügt, muss aber noch erstellt werden.

Die Parameter sind **Id_Nr** des Dokumentes und der **Dateityp**.

```
Public Sub Load_Dokudaten(ByVal Doku_Nr As Integer, ByVal Doku_typ As Integer)
Dim Anzahl As Integer
  Anzahl=
DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Rows.Count
 If Anzahl > 0 Then
   Rtb_Bemerkung.Text=
DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Item(0).Bemerku
ng
   Tb_Sonstiges.Text=
Str(DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Item(0).son
stiges)
   If Doku_typ = 1 Then
      Load_Teileliste(Val(Tb_Doku_Nr.Text))'Teileliste laden
      Pn_TeileKorr.Enabled = True           ' Dateifunktion freigeben
    End If
    Cb_DokuPfad.Enabled = True
    Rtb_Bemerkung.Enabled = True
    Cb_Sonstiges.Enabled = True
    Tb_Sonstiges.Enabled = True
  End If
 End Sub
```

Wird nun über die Combobox **Cb_Doku_Pfad** eine andere Dokumentation ausgewählt, kann über den Aufruf diese Subroutine auch gleich die zusätzliche Information geladen werden.

```
Private Sub Cb_DokuPfad_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_DokuPfad.SelectedIndexChanged
     Dim Id_Nr As Integer
     Dim Pos_Nr As Integer
     Dim Anzahl As Integer
     Tb_Doku_Pfad.Text = Cb_DokuPfad.Text
     Pos_Nr = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
     If Pos_Nr >= 0 Then
        Id_Nr = Val(LB_Doku_Id.Items(Pos_Nr))
        Tb_Doku_Nr.Text = LB_Doku_Id.Items(Pos_Nr)
     End If
     Pos_Nr = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
     Show_Dokument()
     Load_Dokudaten(Id_Nr, Pos_Nr)
   End Sub
```

Auch wenn der Dateityp verändert wird, muss die Liste neu geladen werden. Dazu fügen wir in der Ereignisroutine **Cb_Dateityp_SelectedItemChanged** den Aufruf der Subroutine **Load_Doku** ein. Allerding ist vorher der Dateityp abzufragen und der Routine mit zu übergeben. Die **Projekt_Nr** wird direkt aus der Textbox entnommen. Zusätzlich wird die Textbox **Tb_Doku_Pfad** geleert, damit die Laderoutine eine passende Datei eintragen kann.

```
Private Sub Cb_DateiTyp_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_DateiTyp.SelectedIndexChanged
    Dim Typ As Integer
    Tb_Dateityp.Text = Cb_DateiTyp.Text
    Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
    DateiDialog.Filter = "alle Dateien (*.*)|*.*"
    If Tb_Dateityp.Text = "Bilder" Then
      DateiDialog.Filter = "Bilder (*.jpg;*.jpeg;*.jpe)|*.jpg;*.jpeg;*.jpe|Bilder (*.Gif)|*.gif|Bilder
(*.Tif;*.Tiff)|*.tif;tiff"
    End If
    If Tb_Dateityp.Text = "Skizzen" Then
      DateiDialog.Filter = "Grafikdateien (*.png)|*.png|Paint Dateien (*.bmp)|*.bmp"
    End If
    If Tb_Dateityp.Text = "PDF-Files" Then DateiDialog.Filter = "Pdf Dateien (*.pdf)|*.pdf"
    If Tb_Dateityp.Text = "RTF-Text" Then DateiDialog.Filter = "RichTextdatei (*.rtf)|*.rtf"
    If Tb_Dateityp.Text = "Text" Then DateiDialog.Filter = "Textdatei (*.txt)|*.txt"
    Tb_Bemerkung.Text = "-"
    Tb_Doku_Pfad.Text= ""
    Load_Doku(Val(Tb_Projekt_Id.Text), Typ)
  End Sub
```

Das dabei der Filter möglicherweise auf alle Dateien gesetzt wird, ist kein Problem. Es ermöglicht uns auch nicht vorgesehene Dateien zu erfassen und später vielleicht einmal zu zu ordnen.

### *1.7.3.13 Dokumentation ändern*

Mittlerweile sollte das schon wie von selbst gehen. Was kann verändert werden? Nun, sonstiges ist nicht verwendet, also bleibt nur die Bemerkung. Alle anderen Veränderungen an dem Datensatz dieses Dokumentes machen keinen Sinn. Ein manueller Schreibzugriff auf die Pfaddatei kann gar nicht so gesichert werden, dass eine fehlerhafte Eingabe auszuschließen ist. Den Typ einer Datei verändern? Wozu? Auch die Projektzuordnung muss bleiben und so bleibt nur die Bemerkung übrig. Eine Änderung daran können wir mit dem **Leave**-Ereignis erkennen und die **Update**-Anweisung direkt ausführen. Zu einer Korrektur ist aber die **Id_Nr** des Datensatzes erforderlich. Diese finden wir in der unsichtbaren Listbox **Lb_Doku_id.**

```
Private Sub Rtb_Bemerkung_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Rtb_Bemerkung.Leave
    Dim Id_Nr As Integer
    Dim Pos_Nr As Integer
    Dim Typ As Integer
    If Bt_Store_Doku.Enabled = False Then
      Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
      Pos_Nr = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
      Id_Nr = LB_Doku_Id.Items(Pos_Nr)
      DokumentationTableAdapter.Update_Doku(Tb_Doku_Pfad.Text,                    Typ,
Rtb_Bemerkung.Text, Val(Tb_Sonstiges.Text), Id_Nr)
    End If
  End Sub
```

### *1.7.3.14 Dokument löschen*

Bleibt noch der Aufruf, wenn ein Eintrag gelöscht wird. Auch hier wird die **Id_Nr** aus der unsichtbaren Listbox **Lb_Doku_Id** entnommen, um den aktuell vorliegenden Datensatz auch zu referenzieren. Dieser Delete-Aufruf kann in der Ereignisroutine des Button **Bt_DeleteDoku_Click** bleiben. Bevor aber ein Eintrag entfernt wird, vorsichtshalber noch einmal mit einer MsgBox nachfragen. Wird der Löschvorgang bestätigt, wird auch der Eintrag aus der Textbox **Tb_Doku_Pfad** gelöscht.

```
Private Sub Bt_Delete_Doku_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Delete_Doku.Click
    Dim Id_Nr As Integer
    Dim Typ As Integer
    Dim Eintrag_Nr As Integer
    Dim Antwort As Integer
    Dim Info As String
    Antwort = MsgBox("Soll die Dokumentation wirklich entfernt werden?",
MsgBoxStyle.YesNo, "Warnung")
    If Antwort = vbYes Then
      Eintrag_Nr = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
      If Eintrag_Nr < 0 Then
        Info = "Eintrag nicht gefunden. Löschen fehlgeschlagen"
        MsgBox(Info, MsgBoxStyle.OkOnly, AcceptButton)
      Else
        Id_Nr = Val(LB_Doku_Id.Items(Eintrag_Nr))
        DokumentationTableAdapter.Delete_Doku(Id_Nr)
        TeilelisteTableAdapter.Delete_ListeDoku(Id_Nr)
        Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
        Tb_Doku_Pfad.Text = ""
        Load_Doku(Val(Tb_Projekt_Id.Text), Typ)
        Info = "Eintrag ist entfernt"
      End If
      MsgBox(Info, MsgBoxStyle.OkOnly, AcceptButton)
    End If
  End Sub
```

Diese Programmierung sollte auch getestet werden, bevor wir den nächsten Schritt beginnen.

Wenn dieser Test erfolgreich war, bleibt noch das Löschen der Dokumente mit dem Löschen des Projektes. Hier braucht keine explizite

Frage, ob es gewünscht ist, denn wenn das Löschen eines Projekts bereits beschlossen ist, kann das auch auf die Dokumente bezogen werden.

```
If Antwort = vbYes Then
        ' Nachtragen, Variablen zum aktuellen Projekt löschen
        VariablenTableAdapter.Delete_Variable(Val(Tb_Projekt_Id.Text))
        ' Nachtragen, Einzelbits zum aktuellen Projekt löschen
        EinzelbitTableAdapter.Delete_Projektbits(Val(Tb_Projekt_Id.Text))
        ' Nachtragen, Dokumente zum aktuellen Projekt löschen
        DokumentationTableAdapter.Delete_ProjektDoku(Val(Tb_Projekt_Id.Text))
        ' ****************************************************
        ProjektTableAdapter.Delete_Projekt(Id_Nr)
```

Damit sind die Dateizugriffe für die Dokumentation abgeschlossen. Es ist aber noch etwas zu erledigen, das ich jetzt gern ansprechen möchte. Die Funktion Dokument speichern ist in keinster Weise gesichert. So ist es möglich, xmal die gleiche Datei abzulegen. Das werden wir jetzt ändern. Bevor ein Datensatz abgespeichert wird, werden wir prüfen, ob er bereits Bestandteil der Liste ist. Da wir mit dem Wechsel des Dateityps auch die Liste entsprechend laden, dürfte es ausreichen, den Eintrag der Datei zu prüfen, ob er in der Liste steht. Ist das der Fall, sperren wir einfach den Zugriff auf das Button Dokument speichern.

```
Private Sub Bt_LoadDoku_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_LoadDoku.Click
    Dim Typ As Integer
    Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
    DateiDialog.InitialDirectory = "c:\"
    'DateiDialog.RestoreDirectory = True
    DateiDialog.ShowDialog()
    Tb_Doku_Pfad.Text = DateiDialog.FileName
    ' Freischaltung Button zum Speichern der Dokumentation
    Bt_Store_Doku.Enabled = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text) < 0
   .......
```

Diese zusätzliche Zeile verhindert einen doppelten Eintrag in der Datenbank.

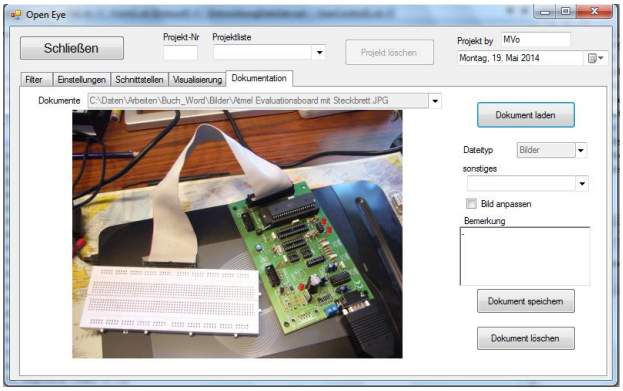
Nun sind noch die beiden Textboxen **Tb_DokuPfad** und **Tb_Bemerkung** für ihren Inhalt viel zu klein. Da hilft nichts, das muss geändert werden, damit die Information auch lesbar ist. Die Combobox **Cb_Dokupfad** betrifft das ebenso. Für den Dateipfad verschieben wir die beiden Objekte

**Cb_DokuPfad** und **Tb_DokuPfad** auf die linke Seite. So kann man zumindest ca. 70 Zeichen darstellen, was für die meisten Dateipfade ausreichen wird. Damit die Subroutine **Set_Default** diese beiden Elemente nicht durch die Größenzuweisung an die RichTextbox und Pictureboxen verdeckt, werden die Parameter angepasst. **Top** wird auf 26 gesetzt und **Height** um 26 reduziert.

```
PB_Bilder.Height = Pn_Doku.Height - 26
    PB_Bilder.Top = 26
```

So wie hier für die Picturebox **Pb_Bilder** werden die Werte auch bei **Pb_Grafik** und **Rtb_Doku** gesetzt.

Den frei gewordenen Platz nutzen wir, um die Textbox **Tb_Bemerkung** gegen eine Richtextbox zu tauschen. Mit dieser Ansicht bin ich zufrieden.




*Neuaufteilung Seite Doku*

So langsam kommen wir zum Abschluss. Es fehlt noch der Aufruf eines anderen Bildes oder Textes, wenn wir einen anderen Eintrag in der Combobox Dokumente auswählen. Das machen wir uns nun ganz einfach.

Es wird eine Subroutine **Show_Doku** geschrieben und der Inhalt hinter der Zuweisung des Pfades an die Textbox **Tb_Doku_Pfad** der Ereignisroutine **Bt_LoadDoku** dort hinein kopiert. Dann wird der Aufruf dieser Routine einmal vom Button und einmal aus der Ereignisroutine **Cb_DokuPfad** aufgerufen.

Das ergibt dann folgende Korrektur

```
Public Sub Show_Dokument()
    Dim Typ As Integer
    Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
    ' Freischaltung Button zum Speichern der Dokumentation
    Bt_Store_Doku.Enabled = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text) < 0
    Cb_Picture_Scale.Enabled = True
    Select Case Typ
      Case 0                          ' Bilder
        Cb_Picture_Scale.Checked = False
        PB_Bilder.Visible = Not Cb_Picture_Scale.Checked
        Pb_Grafik.Visible = Cb_Picture_Scale.Checked
        Rtb_Doku.Visible = False
        PB_Bilder.ImageLocation = Tb_Doku_Pfad.Text
        Pb_Grafik.ImageLocation = Tb_Doku_Pfad.Text
      Case 1                          ' Skizzen
        Cb_Picture_Scale.Checked = True
        Rtb_Doku.Visible = False
        PB_Bilder.Visible = Not Cb_Picture_Scale.Checked
        Pb_Grafik.Visible = Cb_Picture_Scale.Checked
        PB_Bilder.ImageLocation = Tb_Doku_Pfad.Text
        Pb_Grafik.ImageLocation = Tb_Doku_Pfad.Text
      Case 2                          ' RichText (WordPad)
        Cb_Picture_Scale.Enabled = False
        PB_Bilder.Visible = False
        Pb_Grafik.Visible = False
        Rtb_Doku.Visible = True
        Rtb_Doku.Clear()
        Rtb_Doku.LoadFile(Tb_Doku_Pfad.Text, RichTextBoxStreamType.RichText)
      Case 3                           ' Text ( Editor )
        Cb_Picture_Scale.Enabled = False
        PB_Bilder.Visible = False
        Pb_Grafik.Visible = False
        Rtb_Doku.Visible = True
        Rtb_Doku.Clear()
        Rtb_Doku.LoadFile(Tb_Doku_Pfad.Text, RichTextBoxStreamType.PlainText)
      Case 4                           ' Pdf-Files
        PB_Bilder.Visible = False
        Pb_Grafik.Visible = False
        Rtb_Doku.Visible = True
        Rtb_Doku.Clear()
        Pb_Fortschritt.Visible = True
        Tr_Fortschritt.Enabled = True
        Ext_Programm.Start(Tb_Doku_Pfad.Text)
```

```
            'Ext_Programm.StartInfo.FileName = Tb_Doku_Pfad.Text
            'Ext_Programm.Start()
        End Select
    End Sub
```

Nun die geänderte **Bt_LoadDoku_Click**

```
Private Sub Bt_LoadDoku_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_LoadDoku.Click
        DateiDialog.InitialDirectory = "c:\"
        DateiDialog.ShowDialog()
        Tb_Doku_Pfad.Text = DateiDialog.FileName
        Show_Dokument()
    End Sub
```

Und die erweiterte **Cb_DokuPfad_SelectedItemChanged**

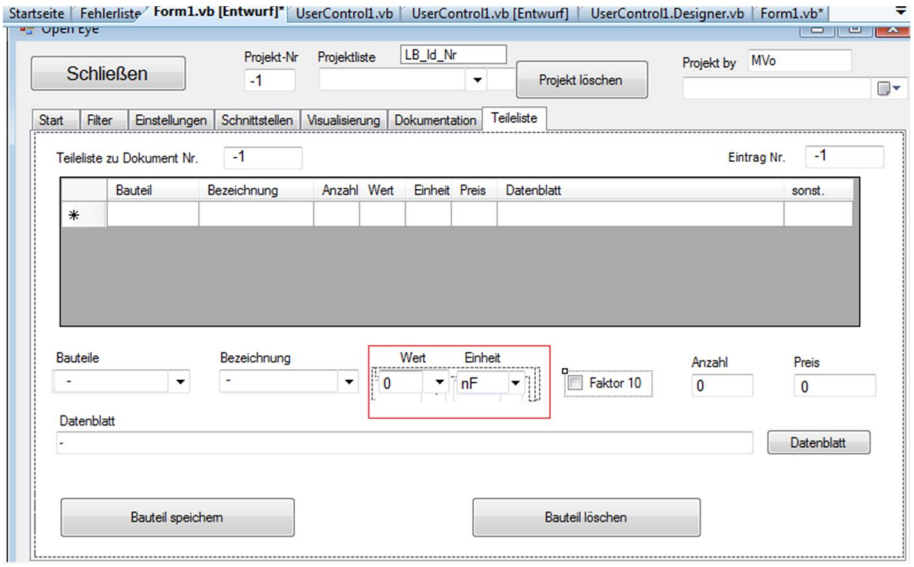
```
Private Sub Cb_DokuPfad_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Cb_DokuPfad.SelectedIndexChanged
        Tb_Doku_Pfad.Text = Cb_DokuPfad.Text
        Show_Dokument()
 End Sub
```

Bis auf die Seite der Teileliste scheint ja alles fertig zu sein

Auch, wenn ich es nicht vorgesehen habe, werde ich dennoch einen Vorschlag für die Teileliste auf einer weiteren Seite präsentieren.

### *1.7.4 Option Bauteile erfassen*

Den Aufbau dieser Seite werde ich nicht detailliert erklären. Zuerst einmal einen Screenshot mit ein paar Hinweisen.

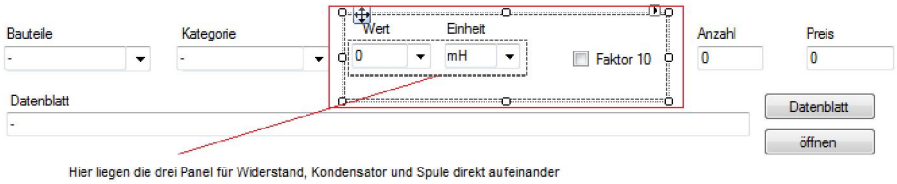



*Verwaltung Bauteile*

Der rot umrandete Bereich besteht aus drei Panels mit jeweils 2 Comboboxen und 2 Textboxen.

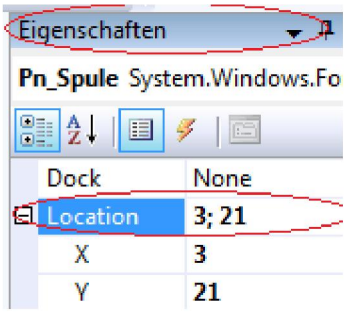

*Panel für Wert und Einheit*

Sie sollen in Abhängigkeit vom Bauteil Widerstand, Kondensator oder Induktivität zugeschaltet werden. Die Checkbox **Cb_Faktor10** hilft den vorgegebenen Wertebereich in den Comboboxen klein zu halten. Damit auch die Überschriften und die Checkbox **Cb_Faktor10** ausgeblendet werden können, ziehen wir diese drei Panels, die beiden Label und die Checkbox auf ein weiteres Panel. Dieses bekommt den Namen **Pn_Einheit**.




*Bauteilen mit Einheit*

Damit die untergeordneten Panels nicht in einander angelegt werden, wird das Hauptpanel **Pn_Einheit** etwas größer angelegt und die kleinen Panels **Pn_Widerstand**, **Pn_Kondensator** und **Pn_Spule** verteilt. Anschließend werden sie mit ihrer Eigenschaft **Location** direkt aufeinander gelegt und die Größe des Panels **Pn_Einheit** angepasst




*Eigenschaft Location*

Nun wird das Ereignis Cb_Kategorie_SelectedItemChanged der Combobox Kategorie entsprechend programmiert, damit dieses Panel nur bei Widerstand, Kondensator oder Spule sichtbar ist.

```
Private Sub Cb_Kategorie_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Kategorie.SelectedIndexChanged
    Dim Ref_Text As String
    Pn_Spule.Visible = False
    Pn_Widerstand.Visible = False
    Pn_Kondensator.Visible = False
    TB_Kategorie.Text = Cb_Kategorie.Text
    Ref_Text = TB_Kategorie.Text
    If (Ref_Text = "Widerstand") Or (Ref_Text = "Kondensator") Or (Ref_Text = "Spule")
Then
      Pn_Einheit.Visible = True
      If TB_Kategorie.Text = "Widerstand" Then
        Pn_Widerstand.Visible = True
      End If
      If TB_Kategorie.Text = "Kondensator" Then
        Pn_Kondensator.Visible = True
      End If
      If TB_Kategorie.Text = "Spule" Then
        Pn_Spule.Visible = True
      End If
    Else
      Pn_Einheit.Visible = False
    End If
  End Sub
```

Die Tabelle Teileliste ist angelegt und zwei weitere Tableadapter für Bauteil und Bezeichnung. Mit diesen Inhalten werden die Vorgaben in den Comboboxen ergänzt. Das bedeutet, Beim Start des Programms muss auch eine **Load_Bauteile** und eine **Load_Bezeichnung** Subroutine aufgerufen werden. Ich werde dies jetzt nicht im Einzelnen aufführen, lediglich den Aufbau der beiden Unterprogramme.

Zuerst **Load_Bauteil**

```
Public Sub Load_Bauteil()
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Bauteil As String
    Anzahl = BauteilTableAdapter.Get_Bauteil.Rows.Count
    If Anzahl > 0 Then
      For i = 0 To Anzahl - 1
        Bauteil = BauteilTableAdapter.Get_Bauteil.Item(i).Bauteil
        If Cb_Bauteil.Items.IndexOf(Bauteil) < 0 Then
          Cb_Bauteil.Items.Add(Bauteil)
        End If
      Next
    End If
  End Sub
```

Dann **Load_Bezeichnung**

```
  Public Sub Load_Bezeichnung()
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Bauteiltyp As String
    Anzahl = KategorieTableAdapter.Get_Kategorie.Rows.Count
    If Anzahl > 0 Then
      For i = 0 To Anzahl - 1
        Bauteiltyp = KategorieTableAdapter.Get_Kategorie.Item(i).Bezeichnung
        If CB_Bezeichnung.Items.IndexOf(Bauteiltyp) < 0 Then
          CB_Bezeichnung.Items.Add(Bauteiltyp)
        End If
      Next
    End If
  End Sub
```

Nun die Routine, um ein Bauteil in die Teileliste einzutragen. Dafür gibt es das Button **Bt_Store_Bauteil_Click** Ereignis

```
Private Sub Bt_Store_Bauteil_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Store_Bauteil.Click
    Dim Projekt_Nr As Integer
    Dim Eintrag_Nr As Integer
    Dim Doku_Nr As Integer
    Dim Bauteil As String
    Dim Kategorie As String
    Dim Wert As Integer
    Dim Einheit As String
    Dim Anzahl As Integer
    Dim Preis As Single
    Dim Doku As String
    Projekt_Nr = Val(Tb_Projekt_Id.Text)
    Eintrag_Nr = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
    Doku_Nr = Val(LB_Doku_Id.Items(Eintrag_Nr))
    Bauteil = Tb_Bauteil.Text
    Kategorie = TB_Kategorie.Text
    Wert = 0
    Einheit = "-"
    If Pn_Kondensator.Visible Then
      Einheit = TB_Einheit_Farad.Text
      Wert = Val(TB_Wert_Farad.Text)
    End If
    If Pn_Widerstand.Visible Then
      Einheit = Tb_Einheit_Ohm.Text
      Wert = Val(Tb_Wert_Ohm.Text)
    End If
    If Pn_Spule.Visible Then
      Einheit = Tb_Einheit_Spule.Text
      Wert = Tb_Wert_Spule.Text
    End If
   If Cb_Faktor10.Checked Then
       Wert = Wert * 10
    End If
    Anzahl = Val(Tb_Anzahl.Text)
    Preis = Val(Tb_Preis.Text)
    Doku = TB_Bauteildoku.Text
    TeilelisteTableAdapter.Insert_Teileliste(Projekt_Nr, Doku_Nr, Bauteil, Kategorie, Einheit, Wert, Anzahl, Preis, Doku, 0)
  End Sub
```

Gleich nach dem Ablegen eines Bauteils wird die Bauteilliste wieder gelesen, damit die Werte in die Tabelle eingetragen werden und die Liste aktualisiert wird. Dafür habe ich die Subroutine **Load_Teileliste**

```
Public Sub Load_Teileliste(ByVal Doku_Id As Integer)
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Id_Nr As Integer
    Dim Projekt_Nr As Integer
    Dim Doku_Nr As Integer
    Dim Bauteil As String
    Dim Kategorie As String
    Dim Wert As Integer
    Dim Einheit As String
    Dim Anz As Integer
    Dim Preis As Single
    Dim Datenblatt As String
    Dim Sonst As Integer
    Dg_Bauteile.Rows.Clear()
   Tb_Eintrag_Nr.Text ="-1"
    Anzahl = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Rows.Count
    If Anzahl > 0 Then
      For i = 0 To Anzahl - 1
        Id_Nr = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Id_Nr
        Projekt_Nr =
TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Projekt_Id
        Doku_Nr = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Doku_Id
        Bauteil = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Bauteil
        Kategorie =
TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Bezeichnung
        Wert = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Wert
        Einheit = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Einheit
        Anz = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Anzahl
        Preis = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Preis
        Datenblatt =
TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Datenblatt
        Sonst = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).sonstiges
        Dg_Bauteile.Rows.Add()
        Dg_Bauteile.Item("Cl_Id_Bauteil", i).Value = Str(Id_Nr)
        Dg_Bauteile.Item("Cl_Projekt_Nr", i).Value = Str(Projekt_Nr)
        Dg_Bauteile.Item("Cl_Doku_Nr", i).Value = Str(Doku_Nr)
        Dg_Bauteile.Item("Cl_Bauteil", i).Value = Bauteil
        Dg_Bauteile.Item("Cl_Bezeichnung", i).Value = Kategorie
```

```
            Dg_Bauteile.Item("Cl_Wert", i).Value = Str(Wert)
            Dg_Bauteile.Item("Cl_Einheit", i).Value = Einheit
            Dg_Bauteile.Item("Cl_Anzahl", i).Value = Str(Anz)
            Dg_Bauteile.Item("Cl_Preis", i).Value = Str(Preis)
            Dg_Bauteile.Item("Cl_Datenblatt", i).Value = Datenblatt
            Dg_Bauteile.Item("Cl_sonstiges", i).Value = Str(Sonst)
            Tb_Eintrag_Nr.Text = Str(Id_Nr)
        Next
    End If
End Sub
```

Das Löschen führe ich direkt in der Button **Bt_del_Bauteil** durch und auch hier wieder das Lesen der Teileliste nach dem erfolgten Löschvorgang.

```
Private Sub Bt_del_Bauteil_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_del_Bautteil.Click
    Dim Id_Nr As Integer
    Dim Antwort As Integer
    Id_Nr = Val(Tb_Eintrag_Nr.Text)
    If Id_Nr >= 0 Then
        Antwort = MsgBox("Soll die Dokumentation wirklich entfernt werden?",
MsgBoxStyle.YesNo, "Warnung")
        If Antwort = vbYes Then
            TeilelisteTableAdapter.Delete_Bauteil(Id_Nr)
            Load_Teileliste(Val(Tb_Doku_Nr.Text))
            Load_Bauteil()
            Load_Kategorie()
        End If
    End If
End Sub
```

Hier muss allerdings nachgearbeitet werden. So ist das Löschen einer Bauteileliste beim Entfernen der zugehörigen Dokumentation sowie das Löschen aller Bauteile dieses Projektes beim Löschen Projekt erforderlich, da die Referenzen sowieso nicht mehr stimmen.

So ist in der Ereignismethode **Bt_Delete_Doku_Click** der Nachtrag

```
TeilelisteTableAdapter.Delete_ListeDoku(Id_Nr)
```

erforderlich und auch in der Subroutine **Delete_Projekt** muss die vorbereitete Delete-Anweisung eingetragen werden

```
TeilelisteTableAdapter.Delete_ListeProjekt(Val(Tb_Projekt_Id.Text))
```

Kommen wir zur Korrektur der Teileliste. Ein Bauteil kann verändert werden, also aus einem Transistor BC326 könnte ein BC 325 werden. Ein Widerstandswert kann sich ändern, die Anzahl und auch ein Preis. Auch die Pfadangabe eines Datenblattes beziehen wir mit ein. Diesmal kann kein **Leave**-Ereigniss herhalten, denn es sind sowohl Neueingaben als auch Änderungen erforderlich. Aus diesem Grund wird noch ein Button **Korrektur** eintragen zusätzlich eingebaut. Hier mache ich es jetzt ganz einfach. Wieder schreibe ich eine extra Subroutine und kopiere dort die Anweisungen hinein. Übergeben wird nur die **Id_Nr** des Bauteileeintrags. Entweder **-1** für **Insert** oder ein Wert **>0** für **Update**. Es kommt noch eine Variable Info hinzu, die den Erfolg oder Misserfolg in einer MsgBox anzeigt. Somit kann diese Routine **Insert** und **Update**-Anweisungen bedienen. Selbst wenn bei einer negativen **Id_Nr** das Button **Korrektur** bedient wird, stört es nicht. Dann wird der Datensatz eben neu eingetragen.

```
Public Sub Store_Bauteil(ByVal Id_Nr As Integer)
    Dim Info As String
    Dim Projekt_Nr As Integer
    Dim Doku_Nr As Integer
    Dim Bauteil As String
    Dim Kategorie As String
    Dim Wert As Integer
    Dim Einheit As String
    Dim Anzahl As Integer
    Dim Preis As Single
    Dim Doku As String
    Projekt_Nr = Val(Tb_Projekt_Id.Text)
    Doku_Nr = Val(Tb_Doku_Nr.Text)
    If Doku_Nr < 0 Then
      Info = "Zeichnung noch nicht gespeichert. Teileliste kann nicht übernommen werden "
    Else
      If Id_Nr < 0 Then
        Info = "Bauteil zur Liste hinzugefügt“
      Else
        Info = "Eintrag wurde korrigiert"
      End If
```

```
        Bauteil = Tb_Bauteil.Text
        Kategorie = TB_Kategorie.Text
        Wert = 0
        Einheit = "-"
        If Pn_Kondensator.Visible Then
            Einheit = TB_Einheit_Farad.Text
            Wert = Val(TB_Wert_Farad.Text)
        End If
        If Pn_Widerstand.Visible Then
            Einheit = Tb_Einheit_Ohm.Text
            Wert = Val(Tb_Wert_Ohm.Text)
        End If
        If Pn_Spule.Visible Then
            Einheit = Tb_Einheit_Spule.Text
            Wert = Tb_Wert_Spule.Text
        End If
        If Cb_Faktor10.Checked Then
            Wert = Wert * 10
        End If
        Anzahl = Val(Tb_Anzahl.Text)
        Preis = Val(Tb_Preis.Text)
        Doku = TB_Bauteildoku.Text
        If Id_Nr < 0 Then
            TeilelisteTableAdapter.Insert_Teileliste
            (Projekt_Nr, Doku_Nr, Bauteil, Kategorie, Einheit, Wert, Anzahl, Preis, Doku, 0)
        Else
            TeilelisteTableAdapter.Update_TeileListe
                    (Bauteil, Kategorie, Einheit, Wert, Anzahl, Preis, Doku, 0, Id_Nr)
        End If
        Load_Teileliste(Doku_Nr)
    End If
    MsgBox(Info, MsgBoxStyle.OkOnly, AcceptButton)
End Sub
```

Die Ereignisroutine **Bt_Store_Bauteil_Click** ist jetzt kurz und bündig

```
Private Sub Bt_Store_Bauteil_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Store_Bauteil.Click
    Tb_Eintrag_Nr.Text = "-1"
    Store_Bauteil(-1)
    Load_Bauteil()
    Load_Kategorie()
End Sub
```

Und die Routine von **Bt_Korr_Click** ist auch nicht viel größer

```
Private Sub Bt_Korr_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Bt_Korr.Click
    Dim Id_Nr As Integer
    Id_Nr = Val(Tb_Eintrag_Nr.Text)
    Store_Bauteil(Id_Nr)
  End Sub
```

Nun soll eine Korrektur oder eine Eingabe nur möglich sein, wenn auch eine Zeichnung vorliegt. Dazu setzen wir die drei Button für **Speichern**, **Korrektur** und **Löschen** auf ein Panel, welches wir erst einmal mit

**Enabled =False** sperren. Es bekommt den Namen **Pn_Teile Korr**. Zur Aktivierung gibt es zwei Bedingungen:

**Es muss eine Zeichnung (Skizze) vorliegen und diese muss auf der Datenbank eingetragen sein.**

Diese Information bekommen wir in der Subroutine **Load_Dokudaten**. Immer, wenn eine neue Datei abgelegt oder eine alte Datei gelöscht wird, rufen wir diese Routine auf, um den Zugriff über die Combobox zu aktualisieren.

Hier ist es ein Leichtes, erst einmal das Panel mit den Button zur Datensicherung zu sperren.

```
Public Sub Load_Dokudaten(ByVal Doku_Nr As Integer, ByVal Doku_typ As Integer)
    Dim Anzahl As Integer
    Pn_TeileKorr.Enabled = False
    Anzahl = DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Rows.Count
    If Anzahl > 0 Then
      Rtb_Bemerkung.Text                                                    =
DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Item(0).Bemerkung
      Tb_Sonstiges.Text                                                     =
Str(DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Item(0).sonstiges)
      If Doku_typ = 1 Then
        Load_Teileliste(Val(Tb_Doku_Nr.Text))        ' Teileliste laden, wenn vorhanden
        Pn_TeileKorr.Enabled = True                  ' Dateifunktionen freigeben
      End If
      Cb_DokuPfad.Enabled = True
      Rtb_Bemerkung.Enabled = True
      Cb_Sonstiges.Enabled = True
      Tb_Sonstiges.Enabled = True
    End If
  End Sub
```

Da die Combobox Cb_Dokupfad ebenfalls diese Routine aufruft, haben wir ein Großteil der Arbeit schon erledigt.

Jetzt muss nur noch mit einem Button **Datenblatt** ein Dateipfad eingetragen und mit einem weiteren **Lesen** dieses PDF gestartet werden. Um den Dateipfad zu bekommen, nutzen wir wieder den **Dateidialog**. Der Button bekommt den Namen **Bt_Datenblatt** und darunter setzen wir noch das Button **Bt_Lesen**, um das Datenblatt zu öffnen. Natürlich sollen nur PDF-Files gezeigt werden und so wird auch der Dateifilter vorher gesetzt.

```
Private Sub Bt_Datenblatt_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Datenblatt.Click
    DateiDialog.Filter = "Pdf Dateien (*.pdf)|*.pdf"
    DateiDialog.InitialDirectory = "c:\"
    DateiDialog.ShowDialog()
    TB_Bauteildoku.Text = DateiDialog.FileName
    TB_Bauteildoku.Enabled = True
    Bt_Lesen.Enabled = True
  End Sub
```

Die Bedienung des Button **Bt_Lesen** lassen wir nur zu, wenn auch ein Dateipfad in der Textbox **Tb_BauteileDoku** eingetragen ist. Um diesen Eintrag wieder zu löschen, kann ich in der Tabelle einen Eintrag zur Korrektur auswählen oder in die Textbox klicken.

In der folgenden Ereignisroutine **TB_Bauteildoku_Enter** wird sie dann sofort für den Zugriff gesperrt, der Inhalt entfernt und ein Bindestrich eingefügt. Auch das Button **Bt_Lesen** wird wieder gesperrt. Da ich eine Pfadangabe von Hand eingetragen nicht will, kann ich so vorgehen.

```
Private Sub TB_Bauteildoku_Enter(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles TB_Bauteildoku.Enter
        TB_Bauteildoku.Text = "-"
        TB_Bauteildoku.Enabled = False
        Bt_Lesen.Enabled = False
 End Sub
```

Nun noch das externe Programm starten. Funktioniert wie auf der Seite Dokumentation nur hier mit dem Button **Bt_Lesen**

```
Private Sub Bt_Lesen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Bt_Lesen.Click
    Ext_Programm.Start(TB_Bauteildoku.Text)
  End Sub
```

Jetzt haben wir ein paar Textboxen**,** die wir direkt beschreiben können und die nur Zahlen enthalten sollen. Das erledigen wir mit **KeyPress** der jeweiligen Textbox. Bei den Bauteilewerten begrenzen wir die Eingabelänge auf 3. Das ergibt einen Maximalwert von 999 und mit der Checkbox **Cb_Faktor10** können wir die Werte entsprechend hoch setze, Auch das hat einen Grund. So gibt es bei Widerständen genormte Werte. Z.B. 15 Ohm, 150 Ohm 1,5 KOhm 15 KOhm usw. Da ich keine Realzahlen verwenden will, wird 1,5 KOhm mit 150 *10 Ohm eingegeben. Dasselbe gilt für die Kapazitäten und Induktivitäten. Die gängigsten Werte werden in der Combobox hinterlegt und sollte doch mal ein Exot dabei sein, kann immer noch von Hand ergänzt werden.

Da bis auf den Vergleich der Textlänge in der Textbox der Code in den drei Wertefeldern gleich behandelt wird, zeige ich hier nur eine KeyPress Routine

```
Private Sub Tb_Wert_Ohm_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles Tb_Wert_Ohm.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
      e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(Tb_Wert_Ohm.Text) = 3 Then
      e.KeyChar = ""
    End If
  End Sub
```

Die Textbox **Tb_Preis** macht einen kleinen Unterschied- sie darf Realzahlen haben. Ist ja auch klar, 3 € 50 sind nun mal 3.50 €.

Dafür muss der Textfilter das auch berücksichtigen und noch etwas kommt hinzu - hier wird kein Komma sondern ein Punkt erwartet.

Für den Filter Keypress ist das aber kein Problem. Taucht ein Komma auf, setzen wir einen Punkt dafür. Dann erweitern wir die Abfrage und lassen den Punkt auch mit durchrutschen. Aber nur, wenn noch keiner im Text ist!

```
Private Sub Tb_Preis_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles Tb_Preis.KeyPress
    If (e.KeyChar = ",") Then e.KeyChar = "."
    If (InStr(Tb_Preis.Text, ".") > 0) And (e.KeyChar = ".") Then
      e.KeyChar = ""
    End If
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 And e.KeyChar <>
"." Then
      e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(Tb_Preis.Text) = 8 Then
      e.KeyChar = ""
    End If
  End Sub
```

Kommen wir zur Tabelle. Um einen Wert zu editieren, muss die Tabellenzeile in den Editbereich kopiert werden. Das ist auch so richtig, aber bei einem neuen Eintrag wäre es schön, wenn die Eingabefelder frei wären. Dazu benutze ich die freie Tabellenzeile. Wird diese angeklickt, werden alle Eingabefelder geleert und der Focus auf die Textbox Tb_Bauteil gesetzt. Hier nun die Ereignisroutine **Dg_Bauteile_CellClick**

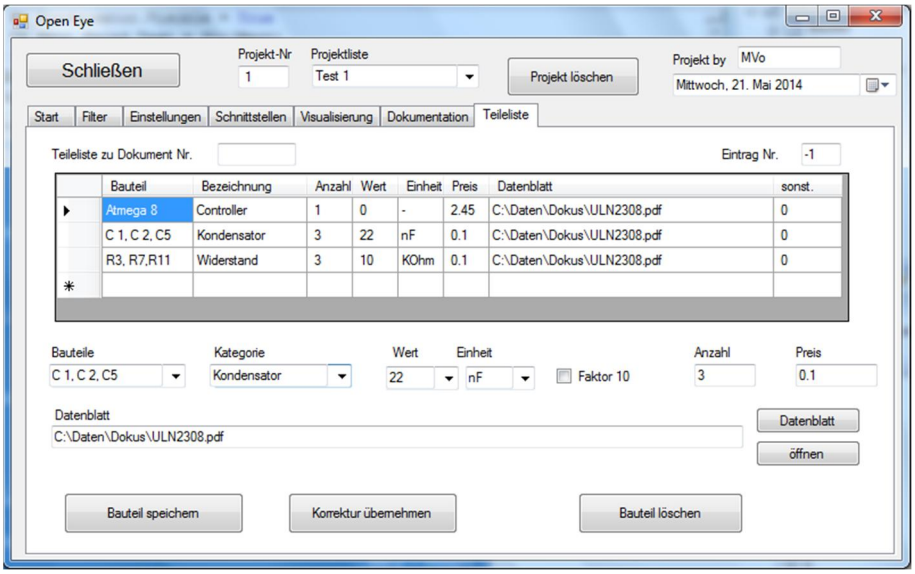
```
Private Sub Dg_Bauteile_CellClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DataGridViewCellEventArgs) Handles Dg_Bauteile.CellClick
    Dim Wert As Integer
    Dim Row_Nr As Integer
    Row_Nr = Dg_Bauteile.CurrentCell.RowIndex
    If Dg_Bauteile.Item("CL_Id_Bauteil", Row_Nr).Value <> Nothing Then          ' Zelle
muss Wert enthalten
      Tb_Eintrag_Nr.Text = Str(Dg_Bauteile.Item("CL_Id_Bauteil", Row_Nr).Value)
      If Val(Tb_Eintrag_Nr.Text) > 0 Then
        Tb_Bauteil.Text = Dg_Bauteile.Item("CL_Bauteil", Row_Nr).Value
        TB_Kategorie.Text = Trim(Dg_Bauteile.Item("CL_Bezeichnung", Row_Nr).Value)
        Pn_Kondensator.Visible = False
        Pn_Widerstand.Visible = False
        Pn_Spule.Visible = False
        Pn_Einheit.Visible = False
        Wert = Val(Dg_Bauteile.Item("CL_Wert", Row_Nr).Value)
        If Wert > 99 Then
          Wert = Wert / 10
          Cb_Faktor10.Checked = True
        else
          Cb_Faktor10.Checked = False
        end If
       If (Ref_Text = "Widerstand") Or (Ref_Text = "Kondensator") Or (Ref_Text = "Spule")
Then
          Pn_Einheit.Visible = True
          If Ref_Text = "Kondensator" Then
            Pn_Kondensator.Visible = True
            TB_Wert_Farad.Text = Str(Wert)
            TB_Einheit_Farad.Text = Dg_Bauteile.Item("CL_Einheit", Row_Nr).Value
          end If
          If Ref_Text = "Spule" Then
            Pn_Spule.Visible = True
            Tb_Wert_Spule.Text = Str(Wert)
            Tb_Einheit_Spule.Text = Dg_Bauteile.Item("CL_Einheit", Row_Nr).Value
          end If
          If Ref_Text = "Widerstand" Then
```

```
                Pn_Widerstand.Visible = True
                Tb_Wert_Ohm.Text = Str(Wert)
                Tb_Einheit_Ohm.Text = Dg_Bauteile.Item("CL_Einheit", Row_Nr).Value
              end If
            else
              Pn_Einheit.Visible = False
            end If
            Tb_Anzahl.Text = Dg_Bauteile.Item("CL_Anzahl", Row_Nr).Value
            Tb_Preis.Text = Dg_Bauteile.Item("Cl_Preis", Row_Nr).Value
            TB_Bauteildoku.Text = Dg_Bauteile.Item("Cl_Datenblatt", Row_Nr).Value
            Bt_Lesen.Enabled = TB_Bauteildoku.Text <> "-" ' Sperrt Button, um Datei zu
öffnen oder gibt es frei
          End If
        Else
          'wenn leere Tabellenspalte dann neuer Datensatz
          Tb_Eintrag_Nr.Text = "-1"
          Tb_Bauteil.Text = "-"
          TB_Kategorie.Text = "-"
          Cb_Faktor10.Checked = False
          TB_Wert_Farad.Text = "0"
          TB_Einheit_Farad.Text = "µF"
          Tb_Wert_Spule.Text = "0"
          Tb_Einheit_Spule.Text = "mH"
          Tb_Wert_Ohm.Text = "0"
          Tb_Einheit_Ohm.Text = "Ohm"
          Tb_Anzahl.Text = "0"
          Tb_Preis.Text = "0"
          TB_Bauteildoku.Text = "-"
          Tb_Bauteil.Select()
        End If
      End Sub
```

Die Vorgehensweise ist von den Variablen bekannt. Einzig die Zuweisung an **Bt_Lesen.enabled** ist vielleicht etwas ungewöhnlich. Aber die Zeile **Bt_BauteilDoku** hat entweder ein **–** Zeichen oder einen Pfad. Und ist nicht der Strich, sondern ein Pfad enthalten ist die Bedingung **<> -** richtig und somit das Button freigegeben.

Um die volle Funktionalität zu bekommen, sind noch ein paar Grundeinstellungen in der Sub Routine **Set_Default** oder **Frm_Open_Eye_Load** durchzuführen. Auch ist diese Seite noch nicht perfekt gestaltet aber es ist möglich, damit zu arbeiten.

Der Screenschot zeigt den komplett fertigen Aufbau der Seite Teileverwaltung



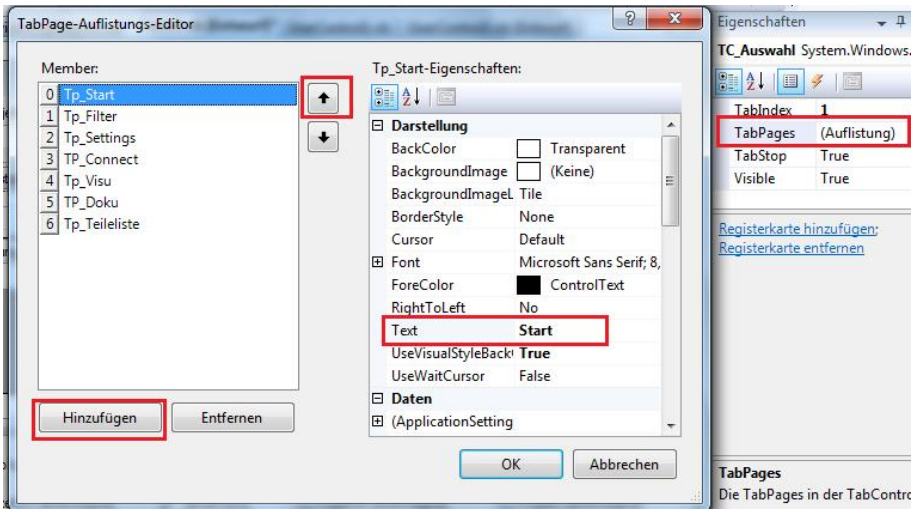

*Ansicht Bauteileliste*

Dieses Bild weicht vom ersten Aufbau ab. Hier sind zusätzlich die Button Korrektur übernehmen und öffnen, um ein Datenblatt anzuzeigen.

## 1.8 Abschluss

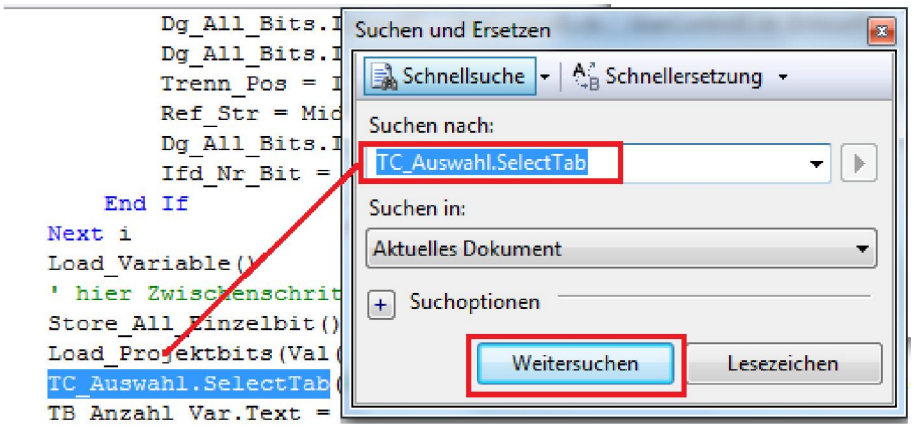
Nun ist das Programm vollständig und sollte ein guter Begleiter bei den vielen Projekten sein. Vielleicht gefällt es euch, wenn sich das Programm mit einer Startseite meldet. Dazu wird eine weitere Seite im TabControl eingebunden.

Über Eigenschaften bekommen wir den Eintrag Auflistung und über diesen Eintrag das linke Fenster.



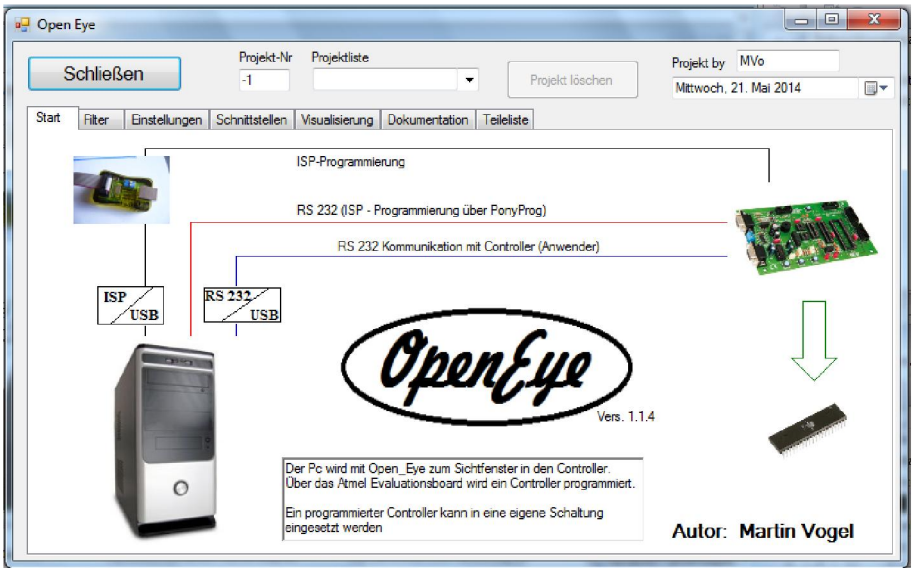
*Hinzufügen einer Startseite*

Zuerst wird eine Seite hinzugefügt, der Name **Tp_Start** vergeben und der Text besetzt, anschließend mit Pfeilbutton nach oben geschoben. Nun ist die erste Seite das Startfenster. Bei diesem Vorgang verändern sich auch die Seitennummern, Das fällt auf, wenn gefilterte Daten übernommen werden. Anfangs wurde anschließend die Seite Einstellungen aufgerufen. Nun passiert nichts mehr. Aber kein Problem. Mit der Schnellsuche den Aufruf **TC_Auswahl.SelectTab** suchen und anpassen.



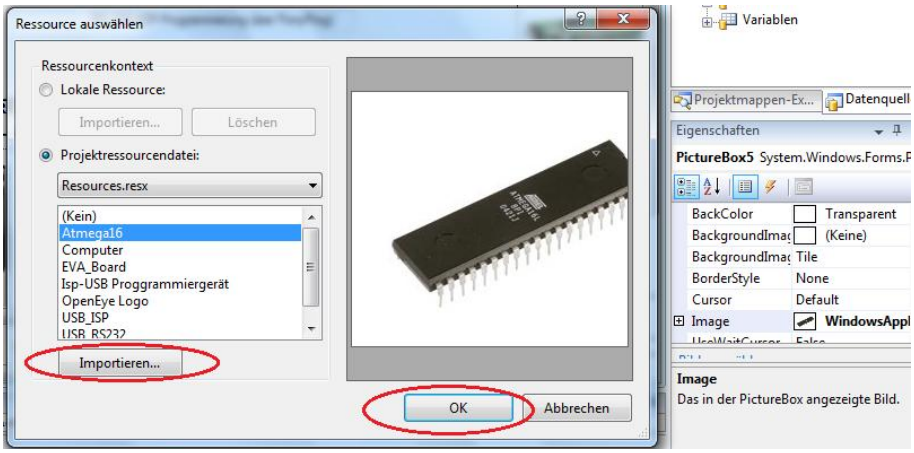

*Seitenweiterschaltung korrigieren*

Nun können wir uns dem Inhalt widmen. Ein schönes Logo. Ein paar Fotos und ein paar Skizzen. Auch noch ein paar erklärende Texte und fertig.




*Startseite*

Die Bilder sind in PictureBoxen abgelegt. Dafür stellt Visual Basic eine Resourcendatei zur Verfügung. Wird in der Eigenschaft der Picturebox Image angeklickt, öffnet sich die Resourcendatei. Über das Button Importieren kann eine Bilddatei ausgewählt und mit dem Button OK der Picturebox zugewiesen werden. Diese Bilder werden nicht zur Laufzeit zugewiesen und sind nun ein Bestandteil der Applikation.




*Resourcendatei für Startseite*

Die schönen Verbindungslinien sind manuell einzubringen. Das Ereignis Paint der Seite lässt diese Gestaltung zu.

```
Private Sub Tp_Start_Paint(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.PaintEventArgs) Handles Tp_Start.Paint
    ' Schwarze Linie ist der USB-ISP Weg
    e.Graphics.DrawLine(Pens.Black, 100, 15, 100, 210)
    e.Graphics.DrawLine(Pens.Black, 100, 15, 650, 15)
    e.Graphics.DrawLine(Pens.Black, 650, 15, 650, 80)

    ' rote Linie ist Programmieren über RS 232 zum Evaluationsboard  mit PonyProg
    e.Graphics.DrawLine(Pens.Red, 140, 80, 140, 210)
    e.Graphics.DrawLine(Pens.Red, 140, 80, 630, 80)

    ' blaue Linie ist die USB-RS232 Verbindung zum Evaluationsboard Datenleitung
    e.Graphics.DrawLine(Pens.Blue, 180, 110, 180, 220)
    e.Graphics.DrawLine(Pens.Blue, 180, 110, 630, 110)

    'Grüne Linien bauen den Pfeil auf
    e.Graphics.DrawLine(Pens.Green, 680, 150, 680, 200)
    e.Graphics.DrawLine(Pens.Green, 700, 150, 700, 200)
    e.Graphics.DrawLine(Pens.Green, 690, 220, 670, 200)
    e.Graphics.DrawLine(Pens.Green, 690, 220, 710, 200)
    e.Graphics.DrawLine(Pens.Green, 680, 150, 700, 150)
    e.Graphics.DrawLine(Pens.Green, 670, 200, 680, 200)
    e.Graphics.DrawLine(Pens.Green, 700, 200, 710, 200)
  End Sub
```

Es mag etwas mühsam sein, sieht aber schon professionell aus. Die Texte werden in Label abgelegt. Es soll ja niemand daran Änderungen vornehmen.

# 1.9 Anhang Basic Listing

## *1.9.1 Globale Variablendeklaration*

```
'*****************************************************************************
'*          globale Variablen und Variablentypen
'*****************************************************************************
  Public Rec_Feld(999) As Byte
  Public Werte_Feld(999) As Byte
  Public Read_Pointer As Integer
  Public Write_Pointer As Integer
  Public Telekopf As String

  Public Structure ProjektRecord
     Dim Id_Nr As Integer        ' Schlüssel Zur referenzierung
     Dim Projekt As String       ' Projektname
     Dim Controller As String    ' Wert aus Textbox
     Dim Takt As String          ' Wert aus Textbox
     Dim Baud As Integer         ' Wert aus Index Combobox
     Dim Daten As Integer        ' Wert aus Index Combobox
     Dim Stopp As Integer        ' Wert aus Index Combobox
     Dim Parity As Integer       ' Wert aus Index Combobox
     Dim Kopf As String          ' Wert aus Textbox
     Dim Auftrag As String       ' Wert aus Textbox
     Dim WPuffer As Integer      ' Wert aus Textbox
     Dim RPuffer As Integer      ' Wert aus Textbox
     Dim Sprache As String       ' Wert aus Textbox
     Dim Autor As String         ' Wert aus Textbox
     Dim Datum As String         ' Wert vom System
     Dim Kommentar As String     ' Wert aus Textbox
  End Structure

  Public Structure Var_Rec
     Dim Var_Id As Integer       ' Neu hinzugekommen
     Dim Name As String          ' alle weiteren bleiben
     Dim Format As String
     Dim Aktiv As Boolean
     Dim Trigger As Integer
     Dim Top As Integer
     Dim Left As Integer
     Dim Width As Integer
     Dim Height As Integer
  End Structure
```

### 1.9.2 Funktionen

#### 1.9.2.1 Funktion Get_BitInfo

```
Public Function Get_Bit_Info(ByVal Akt_Zeile As String) As String
    Dim Bit_pos As Integer
    Dim Bit_Info As String
    Dim Info As String
    Bit_Info = ""                                   ' wenn Ergebnis ungültig
    Bit_pos = InStr(Akt_Zeile, "; Bit ")            ' Position Bitnummer auffinden
    If Bit_pos > 0 Then
        Bit_pos = Bit_pos + 5                       ' Bit-Nr. 5 Stellen weiter hinten
        Bit_Info = ".Bit " + Akt_Zeile(Bit_pos) + " : "   ' Rückgabe vorbereiten mit Bitnummer
        Bit_pos = InStr(Akt_Zeile, "=")             ' Information suchen
        Info = Mid(Akt_Zeile, Bit_pos + 1, Len(Akt_Zeile) - Bit_pos)   ' und herauskopieren
        Info = Trim(Info)                           ' Leerzeichen entfernen
        Bit_Info = Bit_Info + Info                  ' Information Rückgabe hinzufügen
        ' Ergebnis ist gültig
    End If
    Return Bit_Info
End Function
```

### *1.9.2.2 Funktion Get_Var_Name*

```
Public Function Get_Var_Name(ByVal Akt_Zeile As String, ByVal Akt_DP As Integer) As String
    Dim Akt_Name As String
    Dim Akt_Format As String
    Dim Byte_Cnt As String
    Dim Zeile As String
    Dim Ergebnis As String
    Dim Byte_Pos As Integer
    Dim I As Integer
    Zeile = Akt_Zeile
    Akt_Name = Mid(Zeile, 1, Akt_DP - 1)
    Akt_Format = "Int8"                                    'Defaultformat
    If InStr(Zeile, "; Byte") > 0 Then Akt_Format = "Byte"
    If InStr(Zeile, "; ASCII") > 0 Then Akt_Format = "ASCII"
    If InStr(Zeile, "; Hex") > 0 Then Akt_Format = "Hex"
    If InStr(Zeile, "; Hex15") > 0 Then Akt_Format = "Hex16"
    If InStr(Zeile, "; Hex32") > 0 Then Akt_Format = "Hex32"
    If InStr(Zeile, "; Int16") > 0 Then Akt_Format = "Int16"
    If InStr(Zeile, "; Int32") > 0 Then Akt_Format = "Int32"
    Byte_Cnt = ""
    Ergebnis = ""                                          '  Wenn Ergebnis ungültig
    Byte_Pos = InStr(Zeile, ".")
    If Byte_Pos > 0 Then
      Byte_Pos = Byte_Pos + 6
      While (Mid(Zeile, Byte_Pos, 1) >= "0") And (Mid(Zeile, Byte_Pos, 1) <= "9")
        Byte_Cnt = Byte_Cnt + Mid(Zeile, Byte_Pos, 1)          ' Stringaddition
        Byte_Pos = Byte_Pos + 1                            ' Zeiger nachführen
      End While
      If Byte_Cnt = "1" Then
        Lb_Variablen.Items.Add(Akt_Name)                   ' Variable in Liste einfügen
        Lb_Formate.Items.Add(Akt_Format)                   ' Format in Liste einfügen
      Else
        For I = 0 To Val(Byte_Cnt) - 1
          Lb_Variablen.Items.Add(Akt_Name + "[" + Str(I) + "]")   ' Variablenarray einfügen
          Lb_Formate.Items.Add(Akt_Format)                 ' und die zugehörigen Formate
        Next I
      End If
      Ergebnis = Akt_Name + ";" + Akt_Format   ' Name und Format  an Filter zurückliefern
    End If
    Return (Ergebnis)
  End Function
```

### *1.9.2.3 Funktion IntToHex*

```
Public Function IntToHex(ByVal Wert As Int64, ByVal Format As String) As String
    Dim Hex_Zahl As String
    Dim Zahl As Int64
    Dim Temp As Int64
    Dim Rest As Int64
    Dim Laenge As Integer
    Dim i As Integer
    Dim CharCode As Integer
    Laenge = 2                      'Formatlänge 2 vorgeben
    Hex_Zahl = ""                    ' Hex_Zahl leeren
    Zahl = Wert                     ' Wert nach Zahl übertragen
    While Zahl > 0                   ' verbleibenden Zahlenwert prüfen
        Temp = Math.Truncate(Zahl / 16) ' Zwischenwert bilden
        Rest = Zahl - (Temp * 16)       ' Rest aus Division
        If Rest > 9 Then                ' Bereich 0 - 9 oder A-F
            CharCode = Asc("A")
            Rest = Rest - 10
            CharCode = CharCode + Rest
        Else
            CharCode = Asc("0") + Rest
        End If
        Hex_Zahl = Chr(CharCode) + Hex_Zahl ' Hex_Zahl zusammenstellen
        Trim(Hex_Zahl)                   ' von hinten nach vorn
        Zahl = Temp                     ' Zahl für nächsten Durchlauf setzen
    End While
    If Format = "Hex16" Then Laenge = 4 ' Formatlänge evtl. korrigieren
    If Format = "Hex32" Then Laenge = 8
    While Len(Hex_Zahl) < Laenge
        Hex_Zahl = "0" + Hex_Zahl       ' Hex_Zahl mit 0 auffüllen
    End While
    Return (Hex_Zahl)
End Function
```

### *1.9.2.4 Funktion IntToBin*

```
Public Function IntToBin(ByVal Wert As Integer) As String
    Dim Bin_Zahl As String
    Dim Zahl As Integer
    Dim Temp As Integer
    Dim Rest As Integer
    Dim i As Integer
    Bin_Zahl = ""                    ' Bin_Zahl leeren
    Zahl = Wert                      ' Wert nach Zahl übertragen
    While Zahl > 0                    ' verbleibenden Zahlenwert prüfen
        Temp = Math.Truncate(Zahl / 2) ' Zwischenwert bilden
        Rest = Zahl - (Temp * 2)      ' Rest aus Division
        Bin_Zahl = Chr(Asc("0") + Rest) + Bin_Zahl ' Bin_Zahl zusammenstellen
        Trim(Bin_Zahl)                 ' von hinten nach vorn
        Zahl = Temp                    ' Zahl für nächsten Durchlauf setzen
    End While
    While Len(Bin_Zahl) < 8
        Bin_Zahl = "0" + Bin_Zahl      ' Bin_Zahl mit 0 auffüllen
    End While
    Return (Bin_Zahl)
End Function
```

### *1.9.2.5 Funktion Chk_Is_Connect*

```
Public Function Chk_Is_Connect() As Boolean
    Dim Is_Connect As Boolean
    Is_Connect = True
    Try                 ' Set up structured error handling.
        SerialPort1.Open()
    Catch ex As Exception
        MsgBox("Schnittstelle belegt", MsgBoxStyle.OkOnly, AcceptButton)
        Is_Connect = False
    Finally
    End Try
    Return Is_Connect
End Function
```

### 1.9.3 Unterprogramme

#### 1.9.3.1 Set_Default

```
Public Sub Set_Default()
    PB_Bilder.Parent = Pn_Doku
    PB_Bilder.Width = Pn_Doku.Width
    PB_Bilder.Height = Pn_Doku.Height - 26
    PB_Bilder.Top = 26
    PB_Bilder.Left = 0
    Pb_Grafik.Parent = Pn_Doku
    Pb_Grafik.Width = Pn_Doku.Width
    Pb_Grafik.Height = Pn_Doku.Height - 26
    Pb_Grafik.Top = 26
    Pb_Grafik.Left = 0
    Rtb_Doku.Parent = Pn_Doku
    Rtb_Doku.Width = Pn_Doku.Width
    Rtb_Doku.Height = Pn_Doku.Height - 26
    Rtb_Doku.Top = 26
    Rtb_Doku.Left = 0
    TB_Kennung.Text = ""
    Tb_Doku_Pfad.Text = ""
    Rb_Data_Rec.Checked = False
    TB_Empfang.Text = "0"
    Read_Pointer = 0
    Write_Pointer = 0
    TB_Takt.Text = CB_Takt.Items(1)
    TB_Baud.Text = "2400"
    TB_Daten.Text = "8"
    TB_Stop.Text = "1"
    TB_Parity.Text = CB_Parity.Items(0)
    TB_Read.Text = "200"
    TB_Write.Text = "10"
    TB_Kopf.Text = "VALUE"
    TB_Befehl.Text = "V0"
    TB_Datum.Text = DTP_Datum.Text
    Rt_Kommentar.Text = "-"
    Rb_TriggerOff.Checked = True
    Tr_Chk_Rs232.Enabled = False
    SerialPort1.ReadBufferSize = Val(TB_Read.Text)
    SerialPort1.WriteBufferSize = Val(TB_Write.Text)
    SerialPort1.BaudRate = Val(TB_Baud.Text)
    SerialPort1.DataBits = Val(TB_Daten.Text)
    SerialPort1.StopBits = Val(TB_Stop.Text)
    If TB_Parity.Text = "None" Then SerialPort1.Parity = IO.Ports.Parity.None
    If TB_Parity.Text = "Odd" Then SerialPort1.Parity = IO.Ports.Parity.Odd
```

```
    If TB_Parity.Text = "Even" Then SerialPort1.Parity = IO.Ports.Parity.Even
    If TB_Parity.Text = "Mark" Then SerialPort1.Parity = IO.Ports.Parity.Mark
    If TB_Parity.Text = "Space" Then SerialPort1.Parity = IO.Ports.Parity.Space
End Sub
```

### 1.9.3.2 Filter Assembler

```
Public Sub Filter_Assembler()
    Dim Work_Buf As String            ' Arbeitspuffer zum Abarbeiten
    Dim Zeile As String               ' wird für Zeilenweise Abarbeitung gebraucht
    Dim Var_Name As String            'wird für Bit-Beschreibung benötigt
    Dim Format_Str As String          ' Wird zur Auswertung der Bit-Beschreibung gebraucht
    Dim Bit_Info As String            ' Ergebnis aus Bit-Beschreibung
    Dim Bit_Liste(7) As String        ' behalten wir  als Zwischenspeicher
    Dim Bit_Nr As Integer             ' behalten wir für Index zum Zwischenspeicher
    Dim I As Integer                  ' benötigen wir zur Vorbesetzung der Bit_Liste
    Dim LF_Pos As Integer             ' Zur Zeilenbearbeitung erforderlich
    Dim Dp_Pos As Integer             ' benötigt zur Erkennung eines Variablennamens
    Dim Startflag As Boolean          ' Format Byte beim ersten Durchlauf nicht
berücksichtigen
    Work_Buf = Rt_Buffer.Text         ' Kopie der Assembler Variablendeklaration
    Zeile = ""                        ' mit Leerstring vorbesetzen
    Var_Name = ""                     ' mit Leerstring vorbesetzen
    Startflag = False                 ' Startflag löschen
    Format_Str = Tb_DefaultFormat.Text      ' Defaultwert Format setzen
    Lb_Variablen.Items.Clear()        ' Variablenliste leeren
    While Len(Work_Buf) > 0           ' Beginn der Auswerteschleife
        LF_Pos = InStr(Work_Buf, Chr(10))   ' Zeilenende  abfragen
        If LF_Pos > 0 Then
            Zeile = Mid(Work_Buf, 1, LF_Pos - 1)     ' Zeile herauskopieren und Work_Buf
kürzen
            Work_Buf = Mid(Work_Buf, LF_Pos + 1, Len(Work_Buf) - LF_Pos)
        Else
            Zeile = Work_Buf                  ' letzte Zeile und  Work_Buf leeren
            Work_Buf = ""
        End If
        If Zeile <> " " Then                  ' Zeile vorhanden, dann Auswertung beginnen
            Dp_Pos = InStr(Zeile, ":")        ' zuerst Zeile auf  Variablennamen prüfen
            If Dp_Pos > 0 Then                ' neuer Variablenname
                If (Format_Str = "Byte") And Startflag Then ' Vorgänger hat Bitliste
                    For I = 0 To 7
                        ' Bitbeschreibung Vorgänger in Variablen- und Formatlisten ergänzen
                        Lb_Variablen.Items.Add(Bit_Liste(I))
                        Lb_Formate.Items.Add("Bit")
                    Next I
                End If
                Startflag = True
                Zeile = Get_Var_Name(Zeile, Dp_Pos)     ' Zeile hat Name und Format
                If Zeile <> "" Then         ' Zeile gültig?
                    Var_Name = Mid(Zeile, 1, Dp_Pos - 1)   ' Name eintragen
```

```
                Dp_Pos = InStr(Zeile, ";")          ' Trennzeichen ist „;“
                Format_Str = Mid(Zeile, Dp_Pos + 1, Len(Zeile) - Dp_Pos)
                If Format_Str = "Byte" Then
                   For I = 0 To 7                   ' dann  Bitliste  mit Defaultwert besetzen
                      Bit_Liste(I) = Var_Name + ".Bit" + Str(I) + " :nicht verwendet"
                   Next I
                End If
             End If
          Else
             If Format_Str = "Byte" Then                          ' prüfen, ob Bit-Beschreibung
erforderlich
                Bit_Info = Get_Bit_Info(Zeile)          ' hier kommt der Aufruf der Auswertung
                If Bit_Info <> "" Then                  ' Bitinfo liegt vor
                   Bit_Nr = Val(Mid(Bit_Info, 6, 1))      ' Bitnummer holen
                   Bit_Liste(Bit_Nr) = Var_Name + Bit_Info   ' in Liste eintragen
                End If
             End If
          End If
       End If
    End While
    If (Format_Str = "Byte") And Startflag Then ' Vorgänger hat Bitliste
       For I = 0 To 7
          ' Bitbeschreibung Vorgänger in Variablen- und Formatlisten ergänzen
          Lb_Variablen.Items.Add(Bit_Liste(I))
          Lb_Formate.Items.Add("Bit")
       Next I
    End If
  End Sub
```

### *1.9.3.3 Filter Basic*

```
Public Sub Filter_Basic()

End Sub
'*****************************************************************
'*          Subroutine ist für einen Variablenfiler          *
'*          eines C-Controllerprogrammes vorgesehen          *
'*****************************************************************
Public Sub Filter_C()

End Sub
```

### *1.9.3.4 Filter Eintragen*

```
Public Sub Filter_Eintragen()
    Dim i As Integer            ' Schleifenvariable
    Dim Anzahl As Integer
    Dim lfd_Nr_Var As Integer   ' Zähler für Tabellenzeile Variablen
    Dim lfd_Nr_Bit As Integer   ' Zähler für Tabellenzeile Einzelbit
    Dim Trenn_Pos As Integer    ' Hilfsvariable für Textbearbeitung
    Dim Ref_Str As String       ' Hilfsvariable für Listeneintrag
    DG_Variablen.Rows.Clear()
    Dg_All_Bits.Rows.Clear()
    lfd_Nr_Var = -1
    lfd_Nr_Bit = 0
    Anzahl = Lb_Variablen.Items.Count
    For i = 0 To Lb_Variablen.Items.Count - 1
        Ref_Str = Lb_Variablen.Items(i)
        Trenn_Pos = InStr(Ref_Str, ".")
        If Trenn_Pos = 0 Then
            Store_Variable(Ref_Str, Lb_Formate.Items(i), "Ja", "Ja", 0)
            lfd_Nr_Var = lfd_Nr_Var + 1
        Else
            Dg_All_Bits.Rows.Add()
            Dg_All_Bits.Item("CL_Bit_Id", lfd_Nr_Bit).Value = lfd_Nr_Bit + 1
            Dg_All_Bits.Item("CL_Proj_Id", lfd_Nr_Bit).Value = Val(Tb_Projekt_Id.Text)
            Dg_All_Bits.Item("CL_Var_Id", lfd_Nr_Bit).Value = lfd_Nr_Var
            Dg_All_Bits.Item("CL_BitNr", lfd_Nr_Bit).Value = "Bit " + Mid(Ref_Str, Trenn_Pos +
5, 1)
            Trenn_Pos = InStr(Ref_Str, ":")
            Ref_Str = Mid(Ref_Str, Trenn_Pos + 1, Len(Ref_Str) - Trenn_Pos)
            Dg_All_Bits.Item("CL_BitFunktion", lfd_Nr_Bit).Value = Ref_Str
            lfd_Nr_Bit = lfd_Nr_Bit + 1
        End If
    Next i
    Load_Variable()
    ' hier Zwischenschritt zur Speicherung aller Einzelbitinfo
    Store_All_Einzelbit()
    Load_Projektbits(Val(Tb_Projekt_Id.Text))
    TC_Auswahl.SelectTab(2) ' hier erfolgt der Wechsel zur nächsten Seite
    TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
    Set_Edit_Variable(0)
    If Tb_Format.Text = "Byte" Then
        Show_Einzelbit(Val(Tb_Var_ID.Text))
    End If
```

```
End Sub
```

### *1.9.3.5 Show_Einzelbit*

```
Public Sub Show_Einzelbit(ByVal Var_Id As Integer)
    Dim i As Integer
    Dim Zeilen_Nr As Integer
    Dim Anzahl As Integer
    Dg_Einzelbit.Columns.Item(3).HeaderText = "Variable: " + TB_Variable.Text + Chr(13) +
"Funktion"
    Dg_Einzelbit.Rows.Clear()            ' Tabelle löschen
    For i = 0 To 7                       ' Tabelle vorbesetzen
        Dg_Einzelbit.Rows.Add()
        Dg_Einzelbit.Item("CL_BitId_Nr", i).Value = -1
        Dg_Einzelbit.Item("CL_Variable_Id", i).Value = Str(Var_Id)
        Dg_Einzelbit.Item("Cl_Bit", i).Value = "Bit" + Str(i)
        Dg_Einzelbit.Item("CL_Funktion", i).Value = "nicht verwendet"
    Next
    Anzahl = EinzelbitTableAdapter.Get_VariablenBit(Val(Tb_Var_ID.Text)).Count
    If Anzahl > 0 Then
        For i = 0 To Anzahl - 1          ' Ergebnis  in Tabelle übertragen
            Zeilen_Nr                                                          =
EinzelbitTableAdapter.Get_VariablenBit(Val(Tb_Var_ID.Text)).Item(i).Bit
            Dg_Einzelbit.Item("CL_BitId_Nr",                Zeilen_Nr).Value   =
Str(EinzelbitTableAdapter.Get_VariablenBit(Val(Tb_Var_ID.Text)).Item(i).Id_Nr)
            Dg_Einzelbit.Item("CL_Funktion",                Zeilen_Nr).Value   =
EinzelbitTableAdapter.Get_VariablenBit(Val(Tb_Var_ID.Text)).Item(i).Funktion
        Next
    End If
    Set_Edit_Bit(0)      ' Inhalt erster Zeile in den Editierbereich
End Sub
```

### *1.9.3.6 Set_Edit_Variable*

```
Public Sub Set_Edit_Variable(ByVal Row_Nr As Integer)
    Dim Trigger As Integer
    Dim Infosatz As String
    Tb_lfd_Nr.Text = Str(Row_Nr)
    If DG_Variablen.Rows.Count > 1 Then
        If DG_Variablen.Item("CL_Freigabe", Row_Nr).Value = "Ja" Then
            Tb_Var_ID.Text = Str(DG_Variablen.Item("CL_Id_Nr", Row_Nr).Value)
            TB_Variable.Text = DG_Variablen.Item("CL_Variable", Row_Nr).Value
            Cb_Format.Text = DG_Variablen.Item("CL_Format", Row_Nr).Value
            CB_Aktiv.Checked = DG_Variablen.Item("CL_Aktiv", Row_Nr).Value = "Ja"
            Trigger = DG_Variablen.Item("Cl_Trigger", Row_Nr).Value
            If Trigger < 0 Then Trigger = 0
            ' CB_Trigger.Text = CB_Trigger.Items(Trigger)
            Tb_Trigger.Text = CB_Trigger.Items(Trigger)
        Else
            Infosatz = "Formatänderung nicht möglich. Byte gehört zu einem anderen Format."
            Infosatz = Infosatz + Chr(13) + "Es muss erst das vorhergehende Format geändert
werden,"
            Infosatz = Infosatz + Chr(13) + "um Zugriff auf dieses Byte zu bekommen."
            MsgBox(Infosatz, MsgBoxStyle.Information, AcceptButton)
        End If
    End If
End Sub
```

### *1.9.3.7 Set_Edit_Bit*

```
Public Sub Set_Edit_Bit(ByVal Row_Nr As Integer)
    TB_Bit_Id.Text = Dg_Einzelbit.Item("CL_BitId_Nr", Row_Nr).Value
    TB_Bit.Text = "Bit" + Str(Row_Nr)
    TB_Funktion.Text = Dg_Einzelbit.Item("CL_Funktion", Row_Nr).Value
End Sub
```

### *1.9.3.8 Chk_Format8*

```
Public Sub Chk_Format8(ByVal Old_Format As String, ByVal New_Format As String,
ByVal Zeile_Nr As Integer)
    Dim Bereich As Integer
    Dim i As Integer
    Bereich = 1
    If (Old_Format = "Int16") Or (Old_Format = "Hex16") Then
        Bereich = 2
    End If
    If (Old_Format = "Hex32") Or (Old_Format = "Int32") Then
        Bereich = 4
    End If
    For i = 0 To Bereich - 1
        DG_Variablen.Item("CL_Freigabe", Zeile_Nr + i).Value = "Ja"
        If i = 0 Then
            DG_Variablen.Item("CL_Format", Zeile_Nr + i).Value = New_Format
        Else
            DG_Variablen.Item("CL_Format", Zeile_Nr + i).Value = Cb_DefaultFormat.Text
        End If
        Update_Variable(Zeile_Nr + i)
    Next
End Sub
```

### *1.9.3.9 Chk_Format16*

```
Public Sub Chk_Format16(ByVal Old_Format As String, ByVal New_Format As String,
ByVal Zeile_Nr As Integer)
    If (Old_Format = "Hex32") Or (Old_Format = "Int32") Then
        DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 2).Value = "Ja"     ' Zeile freigeben
        Update_Variable(Zeile_Nr + 2)                                   ' für jede Zeile ein Update
        DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 3).Value = "Ja"     ' Zeile freigeben
        Update_Variable(Zeile_Nr + 3)                                   ' für jede Zeile ein Update
    End If
    DG_Variablen.Item("CL_Format", Zeile_Nr).Value = New_Format
    Update_Variable(Zeile_Nr)                                           ' für jede Zeile ein Update
    DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 1).Value = "Nein"       ' Zeile sperren
    Update_Variable(Zeile_Nr + 1)                                       ' für jede Zeile ein Update
End Sub

Public Sub Chk_Format32(ByVal New_Format As String, ByVal Zeile_Nr As Integer)
    Dim i As Integer                        ' Schleifenzähler
    DG_Variablen.Item("CL_Format", Zeile_Nr).Value = New_Format
    Update_Variable(Zeile_Nr)                                           ' für jede Zeile ein Update
    For i = 1 To 3
        DG_Variablen.Item("CL_Freigabe", Zeile_Nr + i).Value = "Nein"   ' Zeile sperren
        Update_Variable(Zeile_Nr + i)                                   ' für jede Zeile ein Update
    Next
End Sub
```

### *1.9.3.10 Set_ProjektDaten*

```
Public Sub Set_ProjektDaten(ByVal Projekt_Id As Integer, ByVal Projekt As String)
    Dim Projektdaten As ProjektRecord
    Projektdaten.Id_Nr = Projekt_Id
    Projektdaten.Projekt = Projekt
    Projektdaten.Controller = TB_Controller.Text
    Projektdaten.Takt = TB_Takt.Text
    Projektdaten.Baud = CB_Baud.Items.IndexOf(TB_Baud.Text)
    Projektdaten.Daten = Val(TB_Daten.Text)
    Projektdaten.Stopp = Val(TB_Stop.Text)
    Projektdaten.Parity = CB_Parity.Items.IndexOf(TB_Parity.Text)
    Projektdaten.Kopf = TB_Kopf.Text
    Projektdaten.Auftrag = TB_Befehl.Text
    Projektdaten.RPuffer = Val(TB_Read.Text)
    Projektdaten.WPuffer = Val(TB_Write.Text)
    Projektdaten.Sprache = TB_Sprache.Text
    Projektdaten.Autor = TB_Autor.Text
    Projektdaten.Datum = TB_Datum.Text
    Projektdaten.Kommentar = Rt_Kommentar.Text
    If Projekt_Id < 0 Then
        Insert_Projekt(Projektdaten)
    Else
        Update_Projekt(Projektdaten)
    End If
End Sub
```

### *1.9.3.11 Insert_Projekt*

```
Public Sub Insert_Projekt(ByVal Projektdaten As ProjektRecord)

    ProjektTableAdapter.Insert_Projekt(Projektdaten.Projekt, Projektdaten.Controller,
Projektdaten.Takt, Projektdaten.Baud, Projektdaten.Daten, Projektdaten.Stopp,
Projektdaten.Parity, Projektdaten.Kopf, Projektdaten.Auftrag, Projektdaten.Sprache,
Projektdaten.WPuffer, Projektdaten.RPuffer, Projektdaten.Autor, Projektdaten.Datum,
Projektdaten.Kommentar)
    Load_Projekte(Tb_Projekte.Text)          ' Nach Einfügen Projektliste neu laden

End Sub
```

### 1.9.3.12 Delete_Projekt

```
Public Sub Delete_Projekt(ByVal Id_Nr As Integer)
    Dim Antwort As Integer
    Antwort = MsgBox("Soll der Datensatz wirklich gelöscht werden?", MsgBoxStyle.YesNo,
"Warnung")
    If Antwort = vbYes Then
        ' Nachtragen, Variablen zum aktuellen Projekt löschen
        VariablenTableAdapter.Delete_Variable(Val(Tb_Projekt_Id.Text))
        ' Nachtragen, Einzelbits zum aktuellen Projekt löschen
        EinzelbitTableAdapter.Delete_Projektbits(Val(Tb_Projekt_Id.Text))
        ' Nachtragen zum Löschen von Teilelisten
        TeilelisteTableAdapter.Delete_ListeProjekt(Val(Tb_Projekt_Id.Text))
        ' Nachtragen, Dokumente zum aktuellen Projekt löschen
        DokumentationTableAdapter.Delete_ProjektDoku(Val(Tb_Projekt_Id.Text))
        ' ****************************************************
        ProjektTableAdapter.Delete_Projekt(Id_Nr)
        Load_Projekte("")
        If ProjekteTableAdapter.Get_Projekte.Rows.Count > 0 Then
            Tb_Projekte.Text = ProjekteTableAdapter.Get_Projekte.Item(0).Projekt
            TB_Projekt_Alt.Text = Tb_Projekte.Text
            Tb_Projekt_Id.Text = Str(ProjekteTableAdapter.Get_Projekte.Item(0).Id_Nr)
            Load_Variable()    ' Nachtragen, Variablen zum Projekt laden
        Else
            Tb_Projekte.Text = ""
            TB_Projekt_Alt.Text = Tb_Projekte.Text
            Tb_Projekt_Id.Text = ""
        End If
    End If
End Sub
```

### 1.9.3.13 Update_Projekt

```
Public Sub Update_Projekt(ByVal Projektdaten As ProjektRecord)
    ProjektTableAdapter.Update_Projekt(Projektdaten.Projekt,        Projektdaten.Controller,
Projektdaten.Takt,        Projektdaten.Baud,        Projektdaten.Daten,        Projektdaten.Stopp,
Projektdaten.Parity,      Projektdaten.Kopf,      Projektdaten.Auftrag,      Projektdaten.Sprache,
Projektdaten.WPuffer,     Projektdaten.RPuffer,     Projektdaten.Autor,     Projektdaten.Datum,
Projektdaten.Kommentar, Projektdaten.Id_Nr)
End Sub
```

### *1.9.3.14 Load_Projekte*

```
Public Sub Load_Projekte(ByVal Projektname As String)
    Dim I As Integer
    Dim Index As String
    Dim Anzahl As Integer
    Dim Projekt As String
    Dim Id_Nr As Integer
    Anzahl = ProjekteTableAdapter.Get_Projekte.Rows.Count
    Cb_Projekte.Items.Clear()
    LB_Id_Nr.Items.Clear()
    If Anzahl > 0 Then
        For I = 0 To Anzahl - 1
            Projekt = ProjekteTableAdapter.Get_Projekte.Item(I).Projekt
            Id_Nr = ProjekteTableAdapter.Get_Projekte.Item(I).Id_Nr
            Cb_Projekte.Items.Add(Projekt)
            LB_Id_Nr.Items.Add(Str(Id_Nr))
        Next
        If Projektname = "" Then
            Tb_Projekte.Text = Cb_Projekte.Items(0)
            TB_Projekt_Alt.Text = Tb_Projekte.Text
            Tb_Projekt_Id.Text = LB_Id_Nr.Items(0)
        Else
            Index = Cb_Projekte.Items.IndexOf(Projektname)
            Tb_Projekt_Id.Text = LB_Id_Nr.Items(Index)
        End If
        Load_Controller()
    End If
    Bt_Del_Projekt.Enabled = Anzahl > 0
    Pn_Einstellung.Enabled = Anzahl > 0
End Sub
```

### *1.9.3.15 Load_Projekt*

```
Public Sub Load_Projekt(ByVal Id_Nr As Integer)
    Dim Anzahl As Integer
    Anzahl = ProjektTableAdapter.Get_Projekt(Id_Nr).Rows.Count
    If Anzahl > 0 Then
        Tb_Projekte.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Projekt
        TB_Projekt_Alt.Text = Tb_Projekte.Text
        TB_Controller.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Controller
        TB_Takt.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).CPU_Takt
        TB_Baud.Text                                                                    =
CB_Baud.Items(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Baud)
        SerialPort1.BaudRate = Val(TB_Baud.Text)
        TB_Daten.Text = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Datenbit)
        SerialPort1.DataBits = Val(TB_Daten.Text)
        TB_Stop.Text = Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Stoppbit)
        SerialPort1.StopBits = Val(TB_Stop.Text)
        TB_Parity.Text                                                                  =
CB_Parity.Items(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Parity)
        If TB_Parity.Text = "None" Then SerialPort1.Parity = IO.Ports.Parity.None
        If TB_Parity.Text = "Odd" Then SerialPort1.Parity = IO.Ports.Parity.Odd
        If TB_Parity.Text = "Even" Then SerialPort1.Parity = IO.Ports.Parity.Even
        If TB_Parity.Text = "Mark" Then SerialPort1.Parity = IO.Ports.Parity.Mark
        If TB_Parity.Text = "Space" Then SerialPort1.Parity = IO.Ports.Parity.Space
        TB_Kopf.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Kopf
        TB_Befehl.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Auftrag
        TB_Read.Text = Trim(Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).RPuffer))
        SerialPort1.ReadBufferSize = Val(TB_Read.Text)
        TB_Write.Text = Trim(Str(ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).WPuffer))
        SerialPort1.WriteBufferSize = Val(TB_Write.Text)
        TB_Sprache.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Sprache
        TB_Autor.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Autor
        TB_Datum.Text = ProjektTableAdapter.Get_Projekt(Id_Nr).Item(0).Datum
        Load_Doku(Id_Nr, 5)
    End If
End Sub
```

### *1.9.3.16 Store _Variable*

```
  Private Sub Store_Variable(ByVal Variable As String, ByVal Format As String, ByVal
Freigabe As String, ByVal aktiv As String, ByVal Trigger As Byte)
    Dim ProjektId As Integer
    Dim FormatIndex As Byte
    FormatIndex = Cb_Format.Items.IndexOf(Format)
    ProjektId = Val(Tb_Projekt_Id.Text)
    VariablenTableAdapter.Insert_Variable(ProjektId, Variable, FormatIndex, Freigabe, aktiv,
Trigger)
  End Sub
```

### *1.9.3.17 Load_Variable*

```
  Public Sub Load_Variable()
    Dim Projekt_ID As Integer
    Dim Anzahl As Integer
    Dim i As Integer
    Dim Id_Nr As Integer
    Dim FormatIndex As Integer
    DG_Variablen.Rows.Clear()
    Projekt_ID = Val(Tb_Projekt_Id.Text)
    Anzahl = VariablenTableAdapter.Get_Variable(Projekt_ID).Count
    If Anzahl > 0 Then
      For i = 0 To Anzahl - 1
        DG_Variablen.Rows.Add()
        DG_Variablen.Item("CL_Id_Nr",                    i).Value                    =
Str(VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Id_Nr)
        DG_Variablen.Item("CL_Lfd_Nr", i).Value = i + 1
        DG_Variablen.Item("CL_Projekt",                  i).Value                    =
Str(VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Projekt_Id)
        DG_Variablen.Item("CL_Variable",                 i).Value                    =
VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Variable
        FormatIndex = VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Format
        DG_Variablen.Item("CL_Format", i).Value = Cb_Format.Items(FormatIndex)
        DG_Variablen.Item("CL_Freigabe",                 i).Value                    =
VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Freigabe
        DG_Variablen.Item("CL_aktiv",                    i).Value                    =
VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Aktiv
        DG_Variablen.Item("CL_Trigger",                  i).Value                    =
Str(VariablenTableAdapter.Get_Variable(Projekt_ID).Item(i).Trigger)
      Next
    End If
```

```
End Sub
```

### 1.9.3.18 Update_Variable

```
Public Sub Update_Variable(ByVal Zeile As Integer)
    Dim Var_Id As Integer
    Dim ProjektId As Integer
    Dim Format As Byte
    Dim Freigabe As String
    Dim Aktiv As String
    Dim Trigger As Byte
    Dim Ref_Text As String
    If Tb_Projekt_Id.Text <> "-1" Then
        Var_Id = Val(DG_Variablen.Item("CL_Id_Nr", Zeile).Value)
        ProjektId = Val(Tb_Projekt_Id.Text)
        Ref_Text = DG_Variablen.Item("CL_Format", Zeile).Value
        Trim(Ref_Text)
        Format = Cb_Format.Items.IndexOf(Ref_Text)
        Freigabe = DG_Variablen.Item("CL_Freigabe", Zeile).Value
        Aktiv = DG_Variablen.Item("CL_Aktiv", Zeile).Value
        Trigger = Val(DG_Variablen.Item("CL_Trigger", Zeile).Value)
        VariablenTableAdapter.Update_Variable(Format, Freigabe, Aktiv, Trigger, Var_Id)
    End If
End Sub
```

### *1.9.3.19 Store_All_Einzelbit*

```
Public Sub Store_All_Einzelbit()
    Dim I As Integer          'Schleifenvariable für Variablentabelle
    Dim j As Integer          'Schleifenvariable für Einzelbiteinträge
    Dim Anz_Variable As Integer ' Anzahl Einträge in der Variablentabelle
    Dim Variable_Id As Integer  ' Variable zur Konvertierung der Variablen_Id
    Dim Projekt_Id As Integer   ' Variable zur Konvertierung der Projekt_Id
    Dim Bit_Nr As Byte          ' Bitnummer
    Dim Bitname As String       ' Text der Spalte Bit
    Dim Info As String
    Dim Varname As String
    Dim Ref_Wert As Integer
    Anz_Variable = DG_Variablen.Rows.Count
    If Anz_Variable > 0 Then
        Projekt_Id = Val(Tb_Projekt_Id.Text)
        For I = 0 To Anz_Variable - 1
            Varname = DG_Variablen.Item("CL_Variable", I).Value
            If DG_Variablen.Item("CL_Format", I).Value = "Byte" Then
                Variable_Id = Val(DG_Variablen.Item("CL_Id_Nr", I).Value)

                For j = 0 To Dg_All_Bits.Rows.Count - 1
                    Ref_Wert = Val(Dg_All_Bits.Item("Cl_Var_Id", j).Value)
                    If Ref_Wert = I Then
                        Bitname = Dg_All_Bits.Item("CL_BitNr", j).Value
                        Bit_Nr = Val(Mid(Bitname, 5, 1))
                        Info = Dg_All_Bits.Item("Cl_BitFunktion", j).Value
                        If Info = "" Then Info = "nicht verwendet"
                        Trim(Info)
                        If Info <> "nicht verwendet" Then
                            EinzelbitTableAdapter.Insert_Einzelbit(Projekt_Id,   Variable_Id,   Bit_Nr,
Info)
                        End If
                    End If
                Next j
            End If
        Next I
    End If

End Sub
```

### *1.9.3.20 Store_Einzelbit*

```
Public Sub Store_Einzelbit(ByVal Funktion As String)
    Dim Projekt_Id As Integer
    Dim Variable_Id As Integer
    Dim Bit As Byte
    Projekt_Id = Val(Tb_Projekt_Id.Text)
    Variable_Id = Val(Tb_Var_ID.Text)
    Bit = Val(Mid(TB_Bit.Text, 5))
    EinzelbitTableAdapter.Insert_Einzelbit(Projekt_Id, Variable_Id, Bit, Funktion)
End Sub
```

### *1.9.3.21 Load_Projektbits*

```
Public Sub Load_Projektbits(ByVal Projekt_Id As Integer)
    Dim Anzahl As Integer
    Dim i As Integer
    Anzahl = EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Count
    If Anzahl > 0 Then
        Dg_All_Bits.Rows.Clear()
        For i = 0 To Anzahl - 1
            Dg_All_Bits.Rows.Add()
            Dg_All_Bits.Item("Cl_Bit_Id", i).Value =
Str(EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Id_Nr)
            Dg_All_Bits.Item("Cl_Proj_Id", i).Value =
Str(EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Projekt_Id)
            Dg_All_Bits.Item("Cl_Var_Id", i).Value =
Str(EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Variable_Id)
            Dg_All_Bits.Item("Cl_BitNr", i).Value = "Bit " +
Str(EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Bit)
            Dg_All_Bits.Item("Cl_BitFunktion", i).Value =
EinzelbitTableAdapter.Get_ProjektBits(Projekt_Id).Item(i).Funktion
        Next
    End If
End Sub
```

### *1.9.3.22 Load_Variablenbits*

```
Public Sub Load_Variablenbits(ByVal Variablen_Id As Integer)
    Dim Anzahl As Integer
    Dim i As Integer
    Anzahl = EinzelbitTableAdapter.Get_VariablenBit(Variablen_Id).Count
    Dg_Einzelbit.Rows.Clear()
    For i = 0 To Anzahl - 1
        Dg_Einzelbit.Rows.Add()
        Dg_Einzelbit.Item("Cl_Bit_Id", i).Value =
Str(EinzelbitTableAdapter.Get_VariablenBit(Variablen_Id).Item(i).Id_Nr)
        Dg_Einzelbit.Item("Cl_Proj_Id", i).Value =
Str(EinzelbitTableAdapter.Get_VariablenBit(Variablen_Id).Item(i).Projekt_Id)
        Dg_Einzelbit.Item("Cl_Var_Id", i).Value =
Str(EinzelbitTableAdapter.Get_VariablenBit(Variablen_Id).Item(i).Variable_Id)
        Dg_Einzelbit.Item("Cl_BitNr", i).Value = "Bit " +
Str(EinzelbitTableAdapter.Get_VariablenBit(Variablen_Id).Item(i).Bit)
        Dg_Einzelbit.Item("Cl_BitFunktion", i).Value =
EinzelbitTableAdapter.Get_VariablenBit(Variablen_Id).Item(i).Funktion
    Next
End Sub
```

### *1.9.3.23 Delete_Projektbits*

```
Public Sub Delete_Projektbits(ByVal Projekt_Id As Integer)
    EinzelbitTableAdapter.Delete_Projektbits(Projekt_Id)
End Sub
```

### *1.9.3.24 Delete_Variablenbits*

```
Public Sub Delete_Variablenbit(ByVal Variable_Id As Integer)
    If MsgBox("Sollen die Einzelbitinformationen  entfernt werden?", MsgBoxStyle.YesNo) =
MsgBoxResult.Yes Then
        EinzelbitTableAdapter.Delete_VariablenBits(Variable_Id)
    End If
End Sub
```

### *1.9.3.25 Load_Controller*

```
Public Sub Load_Controller()
    Dim i As Integer
    Dim Anzahl As Integer
    CB_Controller.Items.Clear()                          ' Liste leeren
    Anzahl = ControllerlisteTableAdapter.Get_Controller.Rows.Count  ' Anzahl Datensätze
    If Anzahl > 0 Then                                  ' wenn ja
        For i = 0 To Anzahl - 1                         ' dann eintragen
CB_Controller.Items.Add(ControllerlisteTableAdapter.Get_Controller.Item(i).Controller)
        Next
    Else
        TB_Controller.Text = "Atmega 8"
    End If
End Sub
```

### *1.9.3.26 Load_Doku*

```
Public Sub Load_Doku(ByVal Projekt As Integer, ByVal DateiTyp As Integer)
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Id_Pos As Integer
    Cb_DokuPfad.Items.Clear()
    Pn_TeileKorr.Enabled = False
    Tb_Doku_Nr.Text = "-1"            ' keine Freigabe Teileliste
    If DateiTyp = 5 Then              ' alle Dateien
        Anzahl = DokumentationTableAdapter.Get_ProjektDoku(Projekt).Rows.Count
        If Anzahl > 0 Then
            For i = 0 To Anzahl - 1

Cb_DokuPfad.Items.Add(DokumentationTableAdapter.Get_ProjektDoku(Projekt).Item(i).Dat
eipfad)

LB_Doku_Id.Items.Add(DokumentationTableAdapter.Get_ProjektDoku(Projekt).Item(i).Id_Nr)
            Next
            ' Bei Neuer Datei bleibt der Eintrag im Textfeld erhalten
            If Tb_Doku_Pfad.Text = "" Then Tb_Doku_Pfad.Text = Cb_DokuPfad.Items(0)
        End If
    Else
        Anzahl = DokumentationTableAdapter.Get_ProjektDoku_Typ(Projekt,
DateiTyp).Rows.Count
        If Anzahl > 0 Then
            For i = 0 To Anzahl - 1

Cb_DokuPfad.Items.Add(DokumentationTableAdapter.Get_ProjektDoku_Typ(Projekt,
DateiTyp).Item(i).Dateipfad)

LB_Doku_Id.Items.Add(DokumentationTableAdapter.Get_ProjektDoku_Typ(Projekt,
DateiTyp).Item(i).Id_Nr)
            Next
            ' Bei Neuer Datei bleibt der Eintrag im Textfeld erhalten
            If Tb_Doku_Pfad.Text = "" Then Tb_Doku_Pfad.Text = Cb_DokuPfad.Items(0)
        End If
    End If
    If Anzahl > 0 Then
        Id_Pos = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
        If Id_Pos >= 0 Then
            Tb_Doku_Nr.Text = LB_Doku_Id.Items(Id_Pos) ' Dokument-Id ermitteln
            Load_Dokudaten(Val(Tb_Doku_Nr.Text), DateiTyp)
```

```
        End If
      Else
        Cb_DokuPfad.Enabled = False
        Rtb_Bemerkung.Enabled = False
        Cb_Sonstiges.Enabled = False
        Tb_Sonstiges.Enabled = False
      End If
    End Sub
```

### *1.9.3.27 Load_Dokudaten*

```
    Public Sub Load_Dokudaten(ByVal Doku_Nr As Integer, ByVal Doku_typ As Integer)
      Dim Anzahl As Integer
      Anzahl = DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Rows.Count
      If Anzahl > 0 Then
        Rtb_Bemerkung.Text                                                    =
DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Item(0).Bemerkung
        Tb_Sonstiges.Text                                                     =
Str(DokumentationTableAdapter.Get_Dokumentation(Doku_Nr).Item(0).sonstiges)
        If Doku_typ = 1 Then
          Load_Teileliste(Val(Tb_Doku_Nr.Text))        ' Teileliste laden, wenn vorhanden
          Pn_TeileKorr.Enabled = True                 ' Dateifunktionen freigeben
        End If
        Cb_DokuPfad.Enabled = True
        Rtb_Bemerkung.Enabled = True
        Cb_Sonstiges.Enabled = True
        Tb_Sonstiges.Enabled = True
      End If
    End Sub
```

### *1.9.3.28 Load_Kategorie*

```
Public Sub Load_Kategorie()
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Kategorie As String
    Anzahl = BauteilTableAdapter.Get_Bauteil.Rows.Count
    If Anzahl > 0 Then
        For i = 0 To Anzahl - 1
            Kategorie = KategorieTableAdapter.Get_Kategorie.Item(i).Bezeichnung
            If Cb_Kategorie.Items.IndexOf(Kategorie) < 0 Then
                Cb_Kategorie.Items.Add(Kategorie)
            End If
        Next
    End If
End Sub
```

### *1.9.3.29 Load_Bauteil*

```
Public Sub Load_Bauteil()
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Bauteil As String
    Anzahl = BauteilTableAdapter.Get_Bauteil.Rows.Count
    If Anzahl > 0 Then
        For i = 0 To Anzahl - 1
            Bauteil = BauteilTableAdapter.Get_Bauteil.Item(i).Bauteil
            If CB_Bauteil.Items.IndexOf(Bauteil) < 0 Then
                CB_Bauteil.Items.Add(Bauteil)
            End If
        Next
    End If
End Sub
```

### *1.9.3.30 Load_Teileliste*

```
Public Sub Load_Teileliste(ByVal Doku_Id As Integer)
    Dim i As Integer
    Dim Anzahl As Integer
    Dim Id_Nr As Integer
    Dim Projekt_Nr As Integer
    Dim Doku_Nr As Integer
    Dim Bauteil As String
    Dim Kategorie As String
    Dim Wert As Integer
    Dim Einheit As String
    Dim Anz As Integer
    Dim Preis As Single
    Dim Datenblatt As String
    Dim Sonst As Integer
    Dg_Bauteile.Rows.Clear()
    Tb_Eintrag_Nr.Text = "-1"
    Anzahl = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Rows.Count
    If Anzahl > 0 Then
        For i = 0 To Anzahl - 1
            Id_Nr = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Id_Nr
            Projekt_Nr =
TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Projekt_Id
            Doku_Nr = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Doku_Id
            Bauteil = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Bauteil
            Kategorie =
TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Bezeichnung
            Wert = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Wert
            Einheit = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Einheit
            Anz = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Anzahl
            Preis = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Preis
            Datenblatt =
TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).Datenblatt
            Sonst = TeilelisteTableAdapter.Get_TeileListeDoku(Doku_Id).Item(i).sonstiges
            Dg_Bauteile.Rows.Add()
            Dg_Bauteile.Item("Cl_Id_Bauteil", i).Value = Str(Id_Nr)
            Dg_Bauteile.Item("Cl_Projekt_Nr", i).Value = Str(Projekt_Nr)
            Dg_Bauteile.Item("Cl_Doku_Nr", i).Value = Str(Doku_Nr)
            Dg_Bauteile.Item("Cl_Bauteil", i).Value = Bauteil
            Dg_Bauteile.Item("Cl_Bezeichnung", i).Value = Kategorie
            Dg_Bauteile.Item("Cl_Wert", i).Value = Str(Wert)
            Dg_Bauteile.Item("Cl_Einheit", i).Value = Einheit
            Dg_Bauteile.Item("Cl_Anzahl", i).Value = Str(Anz)
            Dg_Bauteile.Item("Cl_Preis", i).Value = Str(Preis)
```

```
            Dg_Bauteile.Item("Cl_Datenblatt", i).Value = Datenblatt
            Dg_Bauteile.Item("Cl_sonstiges", i).Value = Str(Sonst)
            Tb_Eintrag_Nr.Text = Str(Id_Nr)
        Next
    End If
End Sub
```

### *1.9.3.31 Store_Bauteil*

```
Public Sub Store_Bauteil(ByVal Id_Nr As Integer)
  Dim Info As String
  Dim Projekt_Nr As Integer
  Dim Doku_Nr As Integer
  Dim Bauteil As String
  Dim Kategorie As String
  Dim Wert As Integer
  Dim Einheit As String
  Dim Anzahl As Integer
  Dim Preis As Single
  Dim Doku As String
  Projekt_Nr = Val(Tb_Projekt_Id.Text)
  Doku_Nr = Val(Tb_Doku_Nr.Text)
  If Doku_Nr < 0 Then
    Info = "Zeichnung noch nicht gespeichert. Teileliste kann nicht übernommen werden "
  Else
    If Id_Nr < 0 Then
      Info = "Bauteil zur Liste hinzugefügt"
    Else
      Info = "Eintrag wurde korrigiert"
    End If
    Bauteil = Tb_Bauteil.Text
    Kategorie = TB_Kategorie.Text
    Wert = 0
    Einheit = "-"
    If Pn_Kondensator.Visible Then
      Einheit = TB_Einheit_Farad.Text
      Wert = Val(TB_Wert_Farad.Text)
    End If
    If Pn_Widerstand.Visible Then
      Einheit = Tb_Einheit_Ohm.Text
      Wert = Val(Tb_Wert_Ohm.Text)
    End If
    If Pn_Spule.Visible Then
      Einheit = Tb_Einheit_Spule.Text
      Wert = Tb_Wert_Spule.Text
    End If
    If Cb_Faktor10.Checked Then
      Wert = Wert * 10
    End If
    Anzahl = Val(Tb_Anzahl.Text)
    Preis = Val(Tb_Preis.Text)
    Doku = TB_Bauteildoku.Text
```

```
        If Id_Nr < 0 Then
            TeilelisteTableAdapter.Insert_Teileliste(Projekt_Nr, Doku_Nr, Bauteil, Kategorie,
Einheit, Wert, Anzahl, Preis, Doku, 0)
        Else
            TeilelisteTableAdapter.Update_TeileListe(Bauteil, Kategorie, Einheit, Wert, Anzahl,
Preis, Doku, 0, Id_Nr)
        End If
        Load_Teileliste(Doku_Nr)
    End If
    MsgBox(Info, MsgBoxStyle.OkOnly, AcceptButton)
End Sub
```

### *1.9.4 Set_Bitliste*

```
Public Sub Set_Bitliste(ByVal Zeile_Nr As Integer)
    Dim I As Integer
    Dim Id_Nr As Integer
    Dim Variable As String
    Dim Anzahl As Integer
    Dim Bit_Nr As Integer
    Dim BitInfo As String
    Dim Listeintrag As String
    Id_Nr = Val(DG_Variablen.Item("CL_Id_Nr", Zeile_Nr).Value)
    Variable = DG_Variablen.Item("CL_Variable", Zeile_Nr).Value
    Anzahl = EinzelbitTableAdapter.Get_VariablenBit(Id_Nr).Rows.Count
    If Anzahl > 0 Then
        For I = 0 To Anzahl - 1
            Bit_Nr = EinzelbitTableAdapter.Get_VariablenBit(Id_Nr).Item(I).Bit
            BitInfo = EinzelbitTableAdapter.Get_VariablenBit(Id_Nr).Item(I).Funktion
            Listeintrag = Variable + "(Bit " + Str(Bit_Nr) + ")." + BitInfo
            Clb_Bit_Info.Items.Add(Listeintrag, False)
            Lb_Bit_Ref.Items.Add(Str(Zeile_Nr) + "." + Str(Bit_Nr))
        Next
    End If
End Sub
```

### *1.9.3.32 Set_Checkbit*

```
Public Sub Set_CheckBit(ByVal Zeile_Nr As Integer, ByVal ByteWert As String)
    Dim i As Integer
    Dim Ref_Str As String          ' Referentext für Index der Liste
    Dim Ref_Index As Integer       ' Index Listeneintrag
    Dim Set_Chk As Boolean         ' Hilfsvariable für Checkbox
    For i = 0 To 7
        Ref_Str = Str(Zeile_Nr) + "." + Str(i)         ' Referenztext bilden
        Ref_Index = Lb_Bit_Ref.Items.IndexOf(Ref_Str)  ' index holen
        If Ref_Index >= 0 Then                         ' wenn Index gültig
            Set_Chk = Mid(ByteWert, 8 - i, 1) = "1"    ' Text von rechs nach Links prüfen
            Clb_Bit_Info.SetItemChecked(Ref_Index, Set_Chk) ' Status Checkbox zuweisen
        End If
    Next
End Sub
```

### *1.9.3.33 Show_Dokument*

```
'*****************************************************************************
'     Anzeigen von Bildern und Aufrufen von PDF-Files        *
'*****************************************************************************
Public Sub Show_Dokument()
    Dim Typ As Integer
    Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
    ' Freischaltung Button zum Speichern der Dokumentation
    Bt_Store_Doku.Enabled = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text) < 0
    Cb_Picture_Scale.Enabled = True
    Select Case Typ
        Case 0                          ' Bilder
            Cb_Picture_Scale.Checked = False
            PB_Bilder.Visible = Not Cb_Picture_Scale.Checked
            Pb_Grafik.Visible = Cb_Picture_Scale.Checked
            Rtb_Doku.Visible = False
            PB_Bilder.ImageLocation = Tb_Doku_Pfad.Text
            Pb_Grafik.ImageLocation = Tb_Doku_Pfad.Text
        Case 1                          ' Skizzen
            Cb_Picture_Scale.Checked = True
            Rtb_Doku.Visible = False
            PB_Bilder.Visible = Not Cb_Picture_Scale.Checked
            Pb_Grafik.Visible = Cb_Picture_Scale.Checked
            PB_Bilder.ImageLocation = Tb_Doku_Pfad.Text
            Pb_Grafik.ImageLocation = Tb_Doku_Pfad.Text
        Case 2                          ' RichText (WordPad)
            Cb_Picture_Scale.Enabled = False
            PB_Bilder.Visible = False
            Pb_Grafik.Visible = False
            Rtb_Doku.Visible = True
            Rtb_Doku.Clear()
            Rtb_Doku.LoadFile(Tb_Doku_Pfad.Text, RichTextBoxStreamType.RichText)
        Case 3                          ' Text ( Editor )
            Cb_Picture_Scale.Enabled = False
            PB_Bilder.Visible = False
            Pb_Grafik.Visible = False
            Rtb_Doku.Visible = True
            Rtb_Doku.Clear()
            Rtb_Doku.LoadFile(Tb_Doku_Pfad.Text, RichTextBoxStreamType.PlainText)
        Case 4                          ' Pdf-Files
            PB_Bilder.Visible = False
            Pb_Grafik.Visible = False
```

```
            Rtb_Doku.Visible = True
            Rtb_Doku.Clear()
            Pb_Fortschritt.Visible = True
            Tr_Fortschritt.Enabled = True
            Ext_Programm.Start(Tb_Doku_Pfad.Text)
            'Ext_Programm.StartInfo.FileName = Tb_Doku_Pfad.Text
            'Ext_Programm.Start()
        End Select
    End Sub
```

### 1.9.3.34 Init_Ports

```
    '*********************************************************************
    '*          Erreichbare Ports des PC ermitteln                 *
    '*********************************************************************
    Public Sub Init_Ports()
        CB_Port.Items.Clear()
        For Each sp As String In My.Computer.Ports.SerialPortNames
            CB_Port.Items.Add(sp)
        Next
        TB_Port.Text = "-"
        If CB_Port.Items.Count > 0 Then
            CB_Port.Text = CB_Port.Items.Item(0)
            TB_Port.Text = CB_Port.Text
            SerialPort1.PortName = TB_Port.Text
        End If
        CB_Port.Enabled = CB_Port.Items.Count > 0
        Bt_Connect.Enabled = CB_Port.Items.Count > 0
    End Sub
```

### 1.9.3.35 Send_Befehl

```
    Public Sub Send_Befehl(ByVal Order As String)
        Dim Befehl As String
        Dim Anz As Integer
        Befehl = Trim(Order)
        Anz = Len(Befehl)
        If Anz > 0 Then
            SerialPort1.Write(Befehl)
        End If
    End Sub
```

### *1.9.3.36 Chk_Connect_Status*

```
Public Sub Chk_Connect_Status()
    Dim is_Connect As Boolean
    is_Connect = SerialPort1.IsOpen        ' Port ist verbunden
    CB_Baud.Enabled = Not is_Connect       ' Zugriff nur Offline
    CB_Controller.Enabled = Not is_Connect
    CB_Daten.Enabled = Not is_Connect
    CB_Port.Enabled = Not is_Connect
    CB_Parity.Enabled = Not is_Connect
    CB_Stop.Enabled = Not is_Connect
    CB_Takt.Enabled = Not is_Connect
    TB_Controller.Enabled = Not is_Connect
    TB_Read.Enabled = Not is_Connect
    TB_Takt.Enabled = Not is_Connect
    TB_Write.Enabled = Not is_Connect
    Bt_Port_New.Enabled = Not is_Connect
    Bt_Test.Enabled = is_Connect           ' Zugriff nur Online
    Bt_Single.Enabled = is_Connect
    Bt_Zyklus.Enabled = is_Connect
    Tr_Chk_Rs232.Enabled = is_Connect
    '*  Trigger nur bei verbundener Schnittstelle
    Pn_Trigger.Enabled = is_Connect
    If Not is_Connect Then
        Rb_TriggerOff.Checked = True
        TB_Watchdog.BackColor = Color.White
        Tr_Watchdog.Enabled = False
        Bt_Set_Trigger.Enabled = False
        Bt_Zyklus.Text = "zyklisch lesen"
        Tr_Zyklus.Enabled = False
    End If
End Sub
```

### 1.9.3.37 Clear_Anzeige

```
'*********************************************************************
'*      Anzeigefeld von  Ausgabeobjekten leeren          *
'*********************************************************************
  Public Sub Clear_Anzeige()
    Dim i As Integer
    Clb_Bit_Info.Items.Clear()
    If Pn_Variablen.Controls.Count > 0 Then
      For i = 0 To Pn_Variablen.Controls.Count - 1
        Pn_Variablen.Controls.Item(0).Dispose()
      Next i
    End If
  End Sub
```

### 1.9.3.38 Generate_ValueBox

```
  Public Sub Generate_ValueBox(ByVal Var_Daten As Var_Rec)
    Dim My_Textfeld As ValueBox            ' Die Variable My_Textfeld wird als Typ Valuebox
    My_Textfeld = New ValueBox             ' lokal deklariert. Mit “New Valuebox” wird das
Objekt erzeugt
    My_Textfeld.Parent = Pn_Variablen      ' und Panel “Pn_Anzeige” zugeordnet
    My_Textfeld.Variable_Id = Var_Daten.Var_Id
    My_Textfeld.Name = "VB_" + Var_Daten.Name        ' Der Name des Objektes wird
vergeben, bezogen auf die Variable
    My_Textfeld.Lb_Variable.Text = Var_Daten.Name    ' Der Labeltext bekommt den
Variablennamen zugewiesen
    My_Textfeld.Format = Var_Daten.Format        ' Die Eigenschaft “Format” wird besetzt
    My_Textfeld.Triggerwert = "0"              ' Der Triggerwert wird mit “0” vorbesetzt.
    If Var_Daten.Trigger>0 then                ' Triggerlevel aus Tabelle
      My_Textfeld.Triggerlevel = 2^(Var_Daten.Trigger -1)        ' anpassen
    Else
      My_Textfeld.Triggerlevel =0
    End If

    My_Textfeld.Visible = Var_Daten.Aktiv          ' Parameter definiert die Sichtbarkeit des
Objektes
    My_Textfeld.TB_In.Text = "0"               ' Die Anzeige wird mit “0” vorbesetzt
    My_Textfeld.Top = Var_Daten.Top            ' nun noch die Position festlegen
    My_Textfeld.Left = Var_Daten.Left
    My_Textfeld.Width = Var_Daten.Width        ' und die Größe
    My_Textfeld.Height = Var_Daten.Height
  End Sub
```

### *1.9.3.39 Reset_Trigger*

```
Public Sub Reset_Trigger(ByVal Trigger As Integer)
   If (Trigger = 1) And (Rb_Bit_0.Checked) Then Rb_TriggerOff.Checked = True
   If (Trigger = 2) And (Rb_Bit_1.Checked) Then Rb_TriggerOff.Checked = True
   If (Trigger = 4) And (Rb_Bit_2.Checked) Then Rb_TriggerOff.Checked = True
   If (Trigger = 8) And (Rb_Bit_3.Checked) Then Rb_TriggerOff.Checked = True
   If (Trigger = 16) And (Rb_Bit_4.Checked) Then Rb_TriggerOff.Checked = True
   If (Trigger = 32) And (Rb_Bit_5.Checked) Then Rb_TriggerOff.Checked = True
   If (Trigger = 64) And (Rb_Bit_6.Checked) Then Rb_TriggerOff.Checked = True
   If (Trigger = 128) And (Rb_Bit_7.Checked) Then Rb_TriggerOff.Checked = True
   If Rb_TriggerOff.Checked Then
      TB_Watchdog.BackColor = Color.LawnGreen   ' Textbox grün Empfang Trigger
      Pn_Trigger.Enabled = True                 ' Bereich Trigger freigeben
      Tr_Watchdog.Enabled = False               ' Wachhund stoppen
      Bt_Zyklus.Enabled = True                  ' zyklische bearbeitung freigeben
   End If
End Sub
```

### 1.9.4 Ereignisbearbeitung

```
'*****************************************************************************************
'*                          Begin der Ereignisbearbeitung                                *
'*****************************************************************************************
```

#### 1.9.4.1 Button Filter starten ( Seite 2 )

```
Private Sub Bt_Filter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Filter.Click
    If TB_Sprache.Text = "Assembler" Then
        Filter_Assembler()
    End If
    If TB_Sprache.Text = "Bascom" Then
        Filter_Basic()
    End If
    If TB_Sprache.Text = "C" Then
        Filter_C()
    End If
    Tb_Projekte.Text = "Neu"
    Tb_Projekt_Id.Text = "-1"
    TB_Projekt_Alt.Text = Tb_Projekte.Text
End Sub
```

#### 1.9.4.2 Button Schließen ( Anwendung )

```
Private Sub Bt_Close_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Close()
End Sub
```

#### 1.9.4.3 Listbox Variablen Indexwechsel ( Seite 2)

```
Private Sub Lb_Variablen_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Lb_Variablen.SelectedIndexChanged
    Dim List_Nr As Integer
    List_Nr = Lb_Variablen.Items.IndexOf(Lb_Variablen.SelectedItem)
    Lb_Formate.SetSelected(List_Nr, True)
End Sub
```

### *1.9.4.4 Sprachfilter ( Seite 2 )*

```
Private Sub CB_Sprache_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Sprache.TextChanged
    If CB_Sprache.Text <> "Assembler" Then
        MsgBox("Es ist nur der Filter für Assembler Variablen verfügbar. Auswahl wird daher abgewiesen.", MsgBoxStyle.Information, "Hinweis")
    Else
        TB_Sprache.Text = CB_Sprache.Text
    End If
End Sub
```

### *1.9.4.5 Combobox Formatvorgabe ( Seite 2 )*

```
Private Sub Cb_DefaultFormat_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Cb_DefaultFormat.TextChanged
    Tb_DefaultFormat.Text = Cb_DefaultFormat.Text
End Sub
```

### *1.9.4.6 Textbox Funktion ( Seite 3 )*

```
Private Sub TB_Funktion_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Funktion.Leave
    Dim Id_Nr As Integer
    Id_Nr = Val(TB_Bit_Id.Text)
    If Id_Nr < 0 Then
        Store_Einzelbit(TB_Funktion.Text)
    Else
        EinzelbitTableAdapter.Update_Einzelbit(TB_Funktion.Text, Id_Nr)
    End If
    Load_Projektbits(Val(Tb_Projekt_Id.Text))
    Show_Einzelbit(Val(Tb_Var_ID.Text))
End Sub
```

### *1.9.4.7 Checkbox Aktiv ( Seite 3 )*

```
Private Sub CB_Aktiv_CheckedChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Aktiv.CheckedChanged
    Dim Zeile_Nr As Integer
    Zeile_Nr = Val(Tb_lfd_Nr.Text)
    If CB_Aktiv.Checked Then
        DG_Variablen.Item("CL_Aktiv", Zeile_Nr).Value = "Ja"
    Else
        DG_Variablen.Item("CL_Aktiv", Zeile_Nr).Value = "Nein"
    End If
End Sub
```

### *1.9.4.8 Button Übernahme ( Seite 2 )*

```
Private Sub Bt_Uebernahme_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Uebernahme.Click
    Dim Ref_Text As String
    Dim Name_Neu As Boolean
    Dim Name_Ok As Boolean
    Dim Filter_Ok As Boolean
    Dim Information As String
    Information = ""
    Ref_Text = Tb_Projekte.Text
    Ref_Text = Trim(Ref_Text)
    Name_Neu = Cb_Projekte.Items.IndexOf(Ref_Text) < 0
    Name_Ok = Tb_Projekte.Text <> "Neu"
    Filter_Ok = Lb_Variablen.Items.Count > 0
    If Name_Neu And Name_Ok And Filter_Ok Then
        Set_ProjektDaten(-1, Ref_Text)
        Filter_Eintragen()
    Else
        If Not Name_Neu Then Information = Information + "Bitte neuen Namen wählen. Projekt schon vorhanden" + Chr(13)
        If Not Name_Ok Then Information = Information + "Neu ist kein gültiger Name" + Chr(13)
        If Not Filter_Ok Then Information = Information + "Es liegen keine Daten zur Übernahme vor. Bitte erst den Filter benutzen" + Chr(13)
        MsgBox(Information, MsgBoxStyle.OkOnly, "Information")
        TB_Projekt_Alt.Text = Tb_Projekte.Text
    End If
End Sub
```

### *1.9.4.9 Tabelle Variablen ( Seite 3 )*

```
Private Sub DG_Variablen_CellClick(ByVal sender As System.Object, ByVal e As System.Windows.Forms.DataGridViewCellEventArgs) Handles DG_Variablen.CellClick
    Set_Edit_Variable(DG_Variablen.CurrentCell.RowIndex)
    If Tb_Format.Text = "Byte" Then
        Pn_Bitfeld.Visible = True  ' Bitinformation anzeigen
        Show_Einzelbit(Val(Tb_Var_ID.Text))
    Else
        Pn_Bitfeld.Visible = False  ' keine Bitinformation
    End If
```

```
End Sub
```

### 1.9.4.10 Tabelle Einzelbit ( Seite 3 )

```
Private Sub Dg_Einzelbit_CellClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DataGridViewCellEventArgs) Handles Dg_Einzelbit.CellClick
    Set_Edit_Bit(Dg_Einzelbit.CurrentCell.RowIndex)
End Sub
```

### 1.9.4.11 Start Anwendung ( Anwendungsfenster )

```
Private Sub Frm_Open_Eye_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim Projekt_Nr As Integer
    Load_Controller()
    Rtb_Doku.Parent = Pn_Doku
    Rtb_Doku.Width = 542
    Rtb_Doku.Height = 342
    PB_Bilder.Parent = Pn_Doku
    PB_Bilder.Width = 542
    PB_Bilder.Height = 342
    Pb_Grafik.Parent = Pn_Doku
    Pb_Grafik.Width = 542
    Pb_Grafik.Height = 342
    Set_Default()
    Init_Ports()
    Load_Projekte("")
    Projekt_Nr = Val(Tb_Projekt_Id.Text)
    Load_Projekt(Projekt_Nr)
    Load_Variable()
    Load_Projektbits(Val(Tb_Projekt_Id.Text))
    TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
    Set_Edit_Variable(0)
    If Tb_Format.Text = "Byte" Then
        Show_Einzelbit(Val(Tb_Var_ID.Text))
    End If
    Chk_Connect_Status()
    EntwicklungDataSet.BeginInit()
    EntwicklungDataSet.GetChanges()
    EntwicklungDataSet.EndInit()
    Tb_Dateityp.Text = Cb_DateiTyp.Items(1) ' Beginnen mit Skizzen
    If Projekt_Nr >= 0 Then
        Load_Doku(Projekt_Nr, 1)
        Load_Kategorie()
```

```
        Load_Bauteil()
    End If
End Sub
```

### *1.9.4.12 Combobox Projekte ( Anwendung )*

```
Private Sub Cb_Projekte_SelectedIndexChanged(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles Cb_Projekte.SelectedIndexChanged
    Dim Projekt_Pos As Integer                          ' für Position in der Liste
    Tb_Projekte.Text = Cb_Projekte.Text
    TB_Projekt_Alt.Text = Tb_Projekte.Text
    Projekt_Pos = Cb_Projekte.Items.IndexOf(Tb_Projekte.Text)   ' Position ermitteln
    Tb_Projekt_Id.Text  =  LB_Id_Nr.Items(Projekt_Pos)                    '  und  zugehörige
Projektnummer eintragen
    Load_Projekt(Val(Tb_Projekt_Id.Text))                   ' Einstellparameter Projekt
    Load_Variable()
    Load_Projektbits(Val(Tb_Projekt_Id.Text))
    TB_Anzahl_Var.Text = Str(DG_Variablen.RowCount - 1)
    Set_Edit_Variable(0)
    If Tb_Format.Text = "Byte" Then
        Show_Einzelbit(Val(Tb_Var_ID.Text))
    End If
End Sub
```

### *1.9.4.13 Combobox Format ändern ( Seite 3 )*

```
  Private Sub Cb_Format_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Cb_Format.SelectedIndexChanged
    Dim Zeile_Nr As Integer
    Dim Bereich As Integer        ' Wert für 1-, 2- oder 4- Byte Zahl
    Dim Frei16 As Boolean         ' Hilfsvariable für Bereichsprüfung Zweibyte Format
    Dim Frei32 As Boolean         ' Hilfsvariable für Bereichsprüfung Vierbyte Format
    Dim old_Format As String          ' zur Prüfung, ob manueller Formatwechsel oder von
Tabelle
    Zeile_Nr = Val(Tb_lfd_Nr.Text)  ' aktuelle Zeilennummer
    Bereich = 1                  ' Bereich vorbesetzen für Einbyte-Format
    old_Format = DG_Variablen.Item("CL_Format", Zeile_Nr).Value
    If old_Format <> Cb_Format.Text Then
      If (Cb_Format.Text = "Int16") Or (Cb_Format.Text = "Hex16") Then
        Bereich = 2              '  Zweibytewert
        Frei16 = Zeile_Nr + 1 < DG_Variablen.Rows.Count - 1
        If Frei16 Then
          Frei16 = DG_Variablen.Item("CL_Format", Zeile_Nr + 1).Value = "Byte"
          Frei16  =  Frei16  Or  (DG_Variablen.Item("CL_Format",  Zeile_Nr  +  1).Value  =
"ASCII")
          Frei16  =  Frei16  Or  (DG_Variablen.Item("CL_Format",  Zeile_Nr  +  1).Value  =
"Int8")
          Frei16  =  Frei16  Or  (DG_Variablen.Item("CL_Format",  Zeile_Nr  +  1).Value  =
"Hex8")
        End If
        If Frei16 Then
          Chk_Format16(Tb_Format.Text, Cb_Format.Text, Zeile_Nr)
        Else
          MsgBox(" Bereich zu klein", MsgBoxStyle.Information, AcceptButton)
        End If
      End If
      If (Cb_Format.Text = "Hex32") Or (Cb_Format.Text = "Int32") Then
        Bereich = 4              '  Vierbytewert
        Frei32 = Zeile_Nr + 3 < DG_Variablen.Rows.Count - 1
        If Frei32 Then
          Frei32 = (DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 2).Value = "Ja")
          Frei32 = Frei32 And (DG_Variablen.Item("CL_Freigabe", Zeile_Nr + 3).Value =
"Ja")
          Frei32 = Frei32 And Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value
= "Int16")
          Frei32 = Frei32 And Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value
= "Hex16")
          Frei32 = Frei32 And Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value
= "Hex32")
```

```
                Frei32 = Frei32 And Not (DG_Variablen.Item("CL_Format", Zeile_Nr + 3).Value
= "Int32")
            End If
            If Frei32 Then
                Chk_Format32(Cb_Format.Text, Zeile_Nr)
            Else
                MsgBox(" Bereich zu klein", MsgBoxStyle.Information, AcceptButton)
            End If
        End If
        If Bereich = 1 Then
            Chk_Format8(Tb_Format.Text, Cb_Format.Text, Zeile_Nr)
        End If
    End If
    Tb_Format.Text = Cb_Format.Text
    If Tb_Format.Text = "Byte" Then
        Show_Einzelbit(Val(Tb_Var_ID.Text))
    End If
End Sub
```

### *1.9.4.14 Textbox Format ändern ( Seite 3 )*

```
Private Sub Tb_Format_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Tb_Format.TextChanged
    If Tb_Format.Text = "Byte" Then
        Pn_Bitfeld.Visible = True
        Show_Einzelbit(Val(Tb_Var_ID.Text))
    Else
        Pn_Bitfeld.Visible = False
    End If
End Sub
```

### *1.9.4.15 Combobox Port ( Seite 4 )*

```
Private Sub CB_Port_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Port.SelectedIndexChanged
    TB_Port.Text = CB_Port.Text
    SerialPort1.PortName = TB_Port.Text
End Sub
```

### *1.9.4.16 Button Port neu laden ( Seite 4 )*

```
Private Sub Bt_Port_New_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Port_New.Click
    Init_Ports()
End Sub
```

### *1.9.4.17 Combobox Baud ( Seite 4 )*

```
Private Sub CB_Baud_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Baud.SelectedIndexChanged
    TB_Baud.Text = CB_Baud.Text
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    SerialPort1.BaudRate = Val(TB_Baud.Text)
End Sub
```

### *1.9.4.18 Combobox Datenbit ( Seite 4 )*

```
Private Sub CB_Daten_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Daten.SelectedIndexChanged
    TB_Daten.Text = CB_Daten.Text
    SerialPort1.DataBits = Val(TB_Daten.Text)
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
End Sub
```

### *1.9.4.19 Combobox Stopbit ( Seite 4 )*

```
Private Sub CB_Stop_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles CB_Stop.SelectedIndexChanged
    TB_Stop.Text = CB_Stop.Text
    SerialPort1.StopBits = Val(TB_Stop.Text)
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text),
Tb_Projekte.Text)
End Sub
```

### *1.9.4.20 Combobox Parity ( Seite 4 )*

```
Private Sub CB_Parity_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles CB_Parity.SelectedIndexChanged
    TB_Parity.Text = CB_Parity.Text
    If TB_Parity.Text = "None" Then SerialPort1.Parity = IO.Ports.Parity.None
    If TB_Parity.Text = "Odd" Then SerialPort1.Parity = IO.Ports.Parity.Odd
    If TB_Parity.Text = "Even" Then SerialPort1.Parity = IO.Ports.Parity.Even
    If TB_Parity.Text = "Mark" Then SerialPort1.Parity = IO.Ports.Parity.Mark
    If TB_Parity.Text = "Space" Then SerialPort1.Parity = IO.Ports.Parity.Space
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text),
Tb_Projekte.Text)
End Sub
```

### *1.9.4.21 Textbox Controller ( Seite 4 )*

```
Private Sub TB_Controller_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TB_Controller.Leave
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text),
Tb_Projekte.Text)
End Sub
```

### *1.9.4.22 Combobox Controller ( Seite 4 )*

```
Private Sub CB_Controller_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Controller.SelectedIndexChanged
    If TB_Controller.Text <> CB_Controller.Text Then
        TB_Controller.Text = CB_Controller.Text
        If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
    End If
End Sub
```

### *1.9.4.23 Textbox Takt ( Seite 4 )*

```
Private Sub TB_Takt_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Takt.Leave
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
End Sub
```

### *1.9.4.24 Combobox Takt ( Seite 4 )*

```
Private Sub CB_Takt_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Takt.SelectedIndexChanged
    TB_Takt.Text = CB_Takt.Text
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
End Sub
```

### *1.9.4.25 Textbox Puffer lesen ( Seite 4 )*

```
Private Sub TB_Read_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Read.Leave
    Dim Korrektur As Integer
    Korrektur = Math.Round(Val(TB_Read.Text) / 2) * 2
    TB_Read.Text = Str(Korrektur)
    SerialPort1.ReadBufferSize = Korrektur
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
End Sub
```

### *1.9.4.26 Textbox Puffer senden ( Seite 4 )*

```
Private Sub TB_Write_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Write.Leave
    Dim Korrektur As Integer
    Korrektur = Math.Round(Val(Trim(TB_Write.Text)) / 2) * 2
    TB_Write.Text = Str(Korrektur)
    SerialPort1.WriteBufferSize = Korrektur
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
End Sub
```

### *1.9.4.27 Textbox Empfängerkennung ( Seite 4 )*

```
Private Sub TB_Kopf_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Kopf.Leave
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
End Sub
```

### *1.9.4.28 Textbox Zeichen senden ( Seite 4 )*

```
Private Sub TB_Befehl_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Befehl.Leave
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
End Sub
```

### *1.9.4.29 Button Projekt löschen ( Anwendung )*

```
Private Sub Bt_Del_Projekt_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Del_Projekt.Click
    Delete_Projekt(Val(Tb_Projekt_Id.Text))
End Sub
```

### *1.9.4.30 Richtextbox Buffer ( Seite 2 )*

```
Private Sub Rt_Buffer_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Rt_Buffer.TextChanged
    Tb_Projekte.Text = "Neu"
    Tb_Projekt_Id.Text = "-1"
    TB_Projekt_Alt.Text = Tb_Projekte.Text
End Sub
```

### *1.9.4.31 Textbox Projekte ( Anwendung )*

```
Private Sub Tb_Projekte_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Tb_Projekte.Leave
    Dim Ref_Text As String
    Dim Name_Neu As Boolean
    Dim Name_Ok As Boolean
    Dim Information As String
    Dim Projektdaten As ProjektRecord
    If TB_Projekt_Alt.Text <> "Neu" Then
        Information = ""                                        ' Information ist leer
        Ref_Text = Tb_Projekte.Text                             ' Kopie aus der Textbox
        Ref_Text = Trim(Ref_Text)                               ' Leerzeichen entfernen
        Name_Neu = Cb_Projekte.Items.IndexOf(Ref_Text) < 0      ' 1. Bedingung
        Name_Ok = Ref_Text <> "Neu"                             ' 2. Bedingung
        If Name_Neu And Name_Ok Then
            If Ref_Text <> TB_Projekt_Alt.Text Then
                Projektdaten.Projekt = Tb_Projekte.Text
                Set_ProjektDaten(Val(Tb_Projekt_Id.Text), Tb_Projekte.Text)
                TB_Projekt_Alt.Text = Tb_Projekte.Text
            End If
            Load_Projekte(Tb_Projekte.Text)
        Else                ' Information, welche Bedingung nicht erfüllt ist
            If Not Name_Neu Then Information = Information + "Bitte neuen Namen wählen. Projekt schon vorhanden" + Chr(13)
            If Not Name_Ok Then Information = Information + "Neu ist kein gültiger Name" + Chr(13)
            MsgBox(Information, MsgBoxStyle.OkOnly, "Information")
            Tb_Projekte.Text = TB_Projekt_Alt.Text      ' alten Projektnamen eintragen
        End If
    End If
End Sub
```

### *1.9.4.32 Combobox Trigger ( Seite 3 )*

```
Private Sub CB_Trigger_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles CB_Trigger.SelectedIndexChanged
    Dim Zeile_Nr As Integer
    Dim Trigger As Integer
    Zeile_Nr = Val(Tb_lfd_Nr.Text)
    Tb_Trigger.Text = CB_Trigger.Text
    Trigger = CB_Trigger.Items.IndexOf(CB_Trigger.Text)
    DG_Variablen.Item("CL_Trigger", Zeile_Nr).Value = Str(Trigger)
    Update_Variable(Zeile_Nr)
End Sub
```

### *1.9.4.33 Datum ( Anwendung )*

```
Private Sub DTP_Datum_CloseUp(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles DTP_Datum.CloseUp
    TB_Datum.Text = DTP_Datum.Text
End Sub
```

### *1.9.4.34 RichTextbox Kommentar ( Seite 4 )*

```
Private Sub Rt_Kommentar_Leave(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Rt_Kommentar.Leave
    If Tb_Projekt_Id.Text <> "-1" Then Set_ProjektDaten(Val(Tb_Projekt_Id.Text),
Tb_Projekte.Text)
End Sub
```

### *1.9.4.35 Textbox Puffer Empfangen ( Seite 4 )*

```
Private Sub TB_Read_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TB_Read.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(TB_Read.Text) = 3 Then
        e.KeyChar = ""
    End If
End Sub
```

### *1.9.4.36 Textbox Puffer senden ( Seite 4 )*

```
Private Sub TB_Write_KeyPress(ByVal sender As System.Object, ByVal e As System.Windows.Forms.KeyPressEventArgs) Handles TB_Write.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(TB_Write.Text) = 2 Then
        e.KeyChar = ""
    End If
End Sub
```

### *1.9.4.37 Textbox Test ( Seite 4 )*

```
Private Sub Bt_Test_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Test.Click
    Send_Befehl(TB_Befehl.Text)
End Sub
```

### *1.9.4.38 Button verbinden ( Seite 4 )*

```
Private Sub Bt_Connect_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Connect.Click
    If Bt_Connect.Text = "Verbinden" Then
        If Chk_Is_Connect() Then
            Bt_Connect.Text = "Trennen"
        End If
    Else
        Bt_Connect.Text = "Verbinden"
        SerialPort1.Close()
    End If
    Chk_Connect_Status()
End Sub
```

### *1.9.4.39 Serialport ( Anwendung )*

```
Private Sub SerialPort1_DataReceived(ByVal sender As System.Object, ByVal e As System.IO.Ports.SerialDataReceivedEventArgs) Handles SerialPort1.DataReceived
    Dim I As Integer            ' Schleifenvariable zur Übernahme empfangener Daten
    Dim Read_Count As Integer      ' Anzahl eingetroffener Daten
    Dim Rec_Data As Byte          ' Einzelner Wert aus Portpuffer
    Read_Count = SerialPort1.BytesToRead
    For I = 0 To Read_Count - 1
        Rec_Data = SerialPort1.ReadByte
        Rec_Feld(Write_Pointer) = Rec_Data
        Write_Pointer = Write_Pointer + 1
        If Write_Pointer > 999 Then Write_Pointer = 0 ' Ende Ringpuffer, beginn von vorn
    Next
End Sub
```

### *1.9.4.40 Radiobutton Empfang ( Seite 4 )*

```
Private Sub Rb_Received_CheckedChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Rb_Received.CheckedChanged
    RT_Integer.Clear()
    RT_Hex.Clear()
End Sub
```

### *1.9.4.41 Anwendung wird beendet ( Anwendung )*

```
Private Sub Frm_Open_Eye_FormClosing(ByVal sender As System.Object, ByVal e As System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    EntwicklungDataSet.BeginInit()
    EntwicklungDataSet.AcceptChanges()
    EntwicklungDataSet.Einzelbit.AcceptChanges()
    EntwicklungDataSet.Variablen.AcceptChanges()
    EntwicklungDataSet.Projekte.AcceptChanges()
    EntwicklungDataSet.Projekt.AcceptChanges()
    EntwicklungDataSet.Teileliste.AcceptChanges()
    EntwicklungDataSet.Dokumentation.AcceptChanges()
    EntwicklungDataSet.EndInit()
End Sub
```

### *1.9.4.42 Timer Datenempfang ( Anwendung )*

```
Private Sub Tr_Chk_RS232_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Tr_Chk_Rs232.Tick
    Dim AnzByte As Integer
    Dim Cnt_Data As Integer
    If Not Rb_Data_Rec.Checked Then            ' nur bearbeiten, wenn keine Werte eingetragen werden
        Tr_Chk_Rs232.Enabled = False
        AnzByte = Write_Pointer - Read_Pointer
        While Write_Pointer <> Read_Pointer
            If Rb_Received.Checked Then
                Werte_Feld(Cnt_Data) = Rec_Feld(Read_Pointer)
                Cnt_Data = Cnt_Data + 1
                If Cnt_Data = Val(TB_Anzahl_Var.Text) Then
                    Cnt_Data = 0
                    Rb_Received.Checked = False
                    Rb_Data_Rec.Checked = True
                    Telekopf = ""
                End If
            Else
                Telekopf = Telekopf + Chr(Rec_Feld(Read_Pointer))  'ist globale Variable
                If InStr(TB_Kopf.Text, Telekopf) = 0 Then
                    Telekopf = ""
                Else
                    If Telekopf = TB_Kopf.Text Then
                        Rb_Received.Checked = True
                        TB_Kennung.Text = Telekopf
                    End If
                End If
            End If
            RT_Integer.Text = RT_Integer.Text + Str(Rec_Feld(Read_Pointer)) + ";"
            RT_Hex.Text = RT_Hex.Text + IntToHex(Rec_Feld(Read_Pointer), "Byte") + ";"
            Read_Pointer = Read_Pointer + 1
            If Read_Pointer > 999 Then Read_Pointer = 0
        End While
        TB_Empfang.Text = Str(Cnt_Data)
        Tr_Chk_Rs232.Enabled = True
    End If
End Sub
```

### *1.9.4.43 Button Anzeige generieren ( Seite 3 )*

```
Private Sub Bt_Gen_Anzeige_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Gen_Anzeige.Click
    Dim Par_Satz As Var_Rec
    Dim i As Integer
    Pn_Variablen.Controls.Clear()    ' Panel leeren
    Clb_Bit_Info.Items.Clear()       ' Checklistbox leeren
    Lb_Bit_Ref.Items.Clear()         ' Referenzliste leeren
    Par_Satz.Left = 4                ' Position
    Par_Satz.Top = 10
    Par_Satz.Width = 88
    Par_Satz.Height = 46
    For i = 0 To DG_Variablen.Rows.Count - 2
      If DG_Variablen.Item("CL_Freigabe", i).Value = "Ja" Then
        Par_Satz.Var_Id = DG_Variablen.Item("Cl_Id_Nr", i).Value ' Übergabe der ID-Nr der Variablen
        Par_Satz.Aktiv = DG_Variablen.Item("CL_Aktiv", i).Value = "Ja"
        ' bildet einen boolschen Ausdruck aus “ja”.
        Par_Satz.Format = DG_Variablen.Item("CL_Format", i).Value
        ' kopiert das Format aus der Tabellenzeile.
        Par_Satz.Trigger = Val(DG_Variablen.Item("CL_Trigger", i).Value)
        If Par_Satz.Trigger > 0 Then  ' Bitnummer in Bit vom Byte darstellen
          Par_Satz.Trigger = 2 ^ (Par_Satz.Trigger - 1)  ' Bit setzen
        End If
        ' kopiert den Triggerwert aus der Tabellenzeile
        Par_Satz.Name = DG_Variablen.Item("CL_Variable", i).Value
        ' kopiert den Variablennamen aus der Tabellenzeile
        Generate_ValueBox(Par_Satz)
        If Par_Satz.Format = "Byte" Then
          Set_Bitliste(i)
        End If
        If Par_Satz.Aktiv Then
          Par_Satz.Left = Par_Satz.Left + 94     ' nächste Position
          If Par_Satz.Left + 94 > Pn_Variablen.Width Then
            Par_Satz.Left = 4
            Par_Satz.Top = Par_Satz.Top + 50
          End If
        End If
      End If
    Next
    TC_Auswahl.SelectTab(3)              ' Weiterschalten zur Ansicht
  End Sub
```

### *1.9.4.44 Radiobutton Triggerbits ( Seite 5 )*

```
Private Sub Rb_Bit_0_CheckedChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Rb_Bit_0.CheckedChanged, Rb_Bit_7.TextChanged, Rb_Bit_6.CheckedChanged, Rb_Bit_5.CheckedChanged, Rb_Bit_4.CheckedChanged, Rb_Bit_3.CheckedChanged, Rb_Bit_2.CheckedChanged, Rb_Bit_1.CheckedChanged
    TB_Wert.Text = "0"
    If Rb_Bit_0.Checked Then TB_Wert.Text = Str(2 ^ 0)
    If Rb_Bit_1.Checked Then TB_Wert.Text = Str(2 ^ 1)
    If Rb_Bit_2.Checked Then TB_Wert.Text = Str(2 ^ 2)
    If Rb_Bit_3.Checked Then TB_Wert.Text = Str(2 ^ 3)
    If Rb_Bit_4.Checked Then TB_Wert.Text = Str(2 ^ 4)
    If Rb_Bit_5.Checked Then TB_Wert.Text = Str(2 ^ 5)
    If Rb_Bit_6.Checked Then TB_Wert.Text = Str(2 ^ 6)
    If Rb_Bit_7.Checked Then TB_Wert.Text = Str(2 ^ 7)
End Sub
```

### *1.9.4.45 Radiobutton Dateneingang ( Seite 4 )*

```
  Private Sub Rb_Data_Rec_CheckedChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Rb_Data_Rec.CheckedChanged
    Dim Data_Cnt As Integer          ' Zähler für Wertearray
    Dim Feld_Cnt As Integer          ' Zähler für Anzeigeobjekte
    Dim Anzahl As String             ' Anzahl für Vergleichswert
    Dim Format As String             ' Format aus der Tabelle
    Dim Wertsatz As String           ' Zusammenstellung Werte für das Anzeigeobjekt
    Dim Trigger As Byte              ' Trigger_Out - zweiter empfangener Wert
    Dim Variable As String           ' Variablennamen aus der Tabelle
    Dim Feld As ValueBox             ' Referenzvariable für das akt. Anzeigeobjekt
    If Rb_Data_Rec.Checked Then
      Data_Cnt = 0                   ' Zähler beginnen bei 0
      Feld_Cnt = 0
      Anzahl = Val(TB_Anzahl_Var.Text)   ' Genzwert für Schleife
      ' keine for next, da Data_Cnt unterschiedliche Schrittweite
      Trigger = Werte_Feld(1) ' bereits getestet, jetzt Reset Trigger
      Reset_Trigger(Trigger)
      While Data_Cnt < Anzahl
        Wertsatz = ""                    ' beginnen mit leerem Wertesatz
        Format = DG_Variablen.Item("CL_Format", Data_Cnt).Value
        Variable = DG_Variablen.Item("CL_Variable", Data_Cnt).Value
        If Format = "Byte" Or Format = "Int8" Or Format = "Hex8" Or Format = "ASCII"
Then
          Wertsatz = Str(Werte_Feld(Data_Cnt))   ' Wertesatz nur ein Byte
          Feld = Pn_Variablen.Controls(Feld_Cnt)  ' Variable ist aktuelles Anzeigeobjekt
          Feld.Triggerwert = Trigger
          Feld.TB_In.Text = ""                   ' Änderung erzwingen
          Feld.TB_In.Text = Wertsatz
          If Format = "Byte" Then
            Set_CheckBit(Data_Cnt, Feld.TB_Out.Text)  ' Status Einzelbits setzen
          End If
          Data_Cnt = Data_Cnt + 1                  ' Datenzähler hochsetzen
        End If
        If Format = "Int16" Or Format = "Hex16" Then
          Wertsatz = Str(Werte_Feld(Data_Cnt))
          Wertsatz = Wertsatz + ";" + Str(Werte_Feld(Data_Cnt + 1)) ' Wertesatz 2 Byte
          Feld = Pn_Variablen.Controls(Feld_Cnt)
          Feld.Triggerwert = Trigger
          Feld.TB_In.Text = ""                   ' Änderung erzwingen
          Feld.TB_In.Text = Wertsatz
```

```
            Data_Cnt = Data_Cnt + 2              ' Datenzähler zwei hochsetzen
        End If
        If Format = "Int32" Or Format = "Hex32" Then
            Wertsatz = Str(Werte_Feld(Data_Cnt))
            Wertsatz = Wertsatz + ";" + Str(Werte_Feld(Data_Cnt + 1))
            Wertsatz = Wertsatz + ";" + Str(Werte_Feld(Data_Cnt + 2))
            Wertsatz = Wertsatz + ";" + Str(Werte_Feld(Data_Cnt + 3))  ' Wertesatz 4 Byte
            Feld = Pn_Variablen.Controls(Feld_Cnt)
            Feld.Triggerwert = Trigger
            Feld.TB_In.Text = ""                 ' Änderung erzwingen
            Feld.TB_In.Text = Wertsatz
            Data_Cnt = Data_Cnt + 4              ' Datenzähler vier hochsetzen
        End If
        Feld_Cnt = Feld_Cnt + 1                  ' Anzeigezähler nachführen
    End While
    Rb_Data_Rec.Checked = False
    Rb_Received.Checked = False
  End If
End Sub
```

### *1.9.4.46 Button einmalig lesen ( Seite 5 )*

```
Private Sub Bt_Single_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Single.Click
  Send_Befehl(TB_Befehl.Text)
End Sub
```

### *1.9.4.47 Button zyklisch lesen ( Seite 5 )*

```
Private Sub Bt_Zyklus_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_Zyklus.Click
  If Bt_Zyklus.Text = "zyklisch lesen" Then
    Bt_Zyklus.Text = "Zyklus Stoppen"
    Tr_Zyklus.Enabled = True
  Else
    Bt_Zyklus.Text = "zyklisch lesen"
    Tr_Zyklus.Enabled = False
  End If
End Sub
```

### *1.9.4.48 Timer Zyklus ( Anwendung )*

```
Private Sub Tr_Zyklus_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Tr_Zyklus.Tick
    Send_Befehl(TB_Befehl.Text)
End Sub
```

### *1.9.4.49 Timer Watchdog ( Anwendung )*

```
Private Sub Tr_Watchdog_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Tr_Watchdog.Tick
    Dim Trigger_on(2) As Byte
    TB_Watchdog.BackColor = Color.Red
    Pn_Trigger.Enabled = True
    Rb_TriggerOff.Checked = True
    Tr_Watchdog.Enabled = False
    Trigger_on(0) = Asc("v")
    Trigger_on(1) = Val(TB_Wert.Text)
    SerialPort1.Write(Trigger_on, 0, 2)
    Bt_Zyklus.Enabled = True
End Sub
```

### *1.9.4.50 Textbox Überwachungszeit ( Seite 5 )*

```
Private Sub TB_Watchdog_KeyPress(ByVal sender As System.Object, ByVal e As System.Windows.Forms.KeyPressEventArgs) Handles TB_Watchdog.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(TB_Watchdog.Text) = 9 Then
        e.KeyChar = ""
    End If
End Sub
```

### *1.9.4.51 Button Trigger senden ( Seite 5 )*

```
Private Sub Bt_Set_Trigger_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Set_Trigger.Click
    Dim Trigger_on(2) As Byte
    Bt_Zyklus.Enabled = False
    Bt_Zyklus.Text = "zyklisch lesen"
    Tr_Zyklus.Enabled = False
    Trigger_on(0) = Asc("V")
```

```
        Trigger_on(1) = Val(TB_Wert.Text)
        Tr_Watchdog.Interval = Val(TB_Watchdog.Text)
        Tr_Watchdog.Enabled = True
        TB_Watchdog.BackColor = Color.Yellow
        SerialPort1.Write(Trigger_on, 0, 2)
        Pn_Trigger.Enabled = False
    End Sub
```

### *1.9.4.52 Textbox Triggerwert ( Seite 5 )*

```
    Private Sub TB_Wert_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TB_Wert.TextChanged
        If TB_Wert.Text <> "0" Then
            Bt_Set_Trigger.Enabled = True
            TB_Watchdog.BackColor = Color.White
        Else
            Bt_Set_Trigger.Enabled = False
        End If
    End Sub
```

### *1.9.4.53 Combobox Dokumentation ( Seite 6 )*

```
    Private Sub Cb_DokuPfad_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Cb_DokuPfad.SelectedIndexChanged
        Dim Id_Nr As Integer
        Dim Pos_Nr As Integer
        Dim Anzahl As Integer
        Tb_Doku_Pfad.Text = Cb_DokuPfad.Text
        Pos_Nr = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
        If Pos_Nr >= 0 Then
            Id_Nr = Val(LB_Doku_Id.Items(Pos_Nr))
            Tb_Doku_Nr.Text = LB_Doku_Id.Items(Pos_Nr)
        End If
        Pos_Nr = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
        Show_Dokument()
        Load_Dokudaten(Id_Nr, Pos_Nr)
    End Sub
```

### *1.9.4.54 Combobox Auswahl Dateityp ( Seite 6 )*

```
Private Sub Cb_DateiTyp_SelectedIndexChanged(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles Cb_DateiTyp.SelectedIndexChanged
    Dim Typ As Integer
    Tb_Dateityp.Text = Cb_DateiTyp.Text
    Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
    DateiDialog.Filter = "alle Dateien (*.*)|*.*"
    If Tb_Dateityp.Text = "Bilder" Then
        DateiDialog.Filter = "Bilder (*.jpg;*.jpeg;*.jpe)|*.jpg;*.jpeg;*.jpe|Bilder (*.Gif)|*.gif|Bilder
(*.Tif;*.Tiff)|*.tif;tiff"
    End If
    If Tb_Dateityp.Text = "Skizzen" Then
        DateiDialog.Filter = "Grafikdateien (*.png)|*.png|Paint Dateien (*.bmp)|*.bmp"
    End If
    If Tb_Dateityp.Text = "PDF-Files" Then DateiDialog.Filter = "Pdf Dateien (*.pdf)|*.pdf"
    If Tb_Dateityp.Text = "RTF-Text" Then DateiDialog.Filter = "RichTextdatei (*.rtf)|*.rtf"
    If Tb_Dateityp.Text = "Text" Then DateiDialog.Filter = "Textdatei (*.txt)|*.txt"
    Rtb_Bemerkung.Text = "-"
    Tb_Doku_Pfad.Text = ""
    Load_Doku(Val(Tb_Projekt_Id.Text), Typ)
End Sub
```

### *1.9.4.55 Checkbx Bild anpassen*

```
Private Sub Cb_Picture_Scale_CheckedChanged(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles Cb_Picture_Scale.CheckedChanged
    PB_Bilder.Visible = Not Cb_Picture_Scale.Checked
    Pb_Grafik.Visible = Cb_Picture_Scale.Checked
End Sub
```

### *1.9.4.56 Button Dokumentation laden*

```
Private Sub Bt_LoadDoku_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Bt_LoadDoku.Click
    DateiDialog.InitialDirectory = "c:\"
    DateiDialog.ShowDialog()
    Tb_Doku_Pfad.Text = DateiDialog.FileName
    Show_Dokument()
End Sub
```

### *1.9.4.57 Anwendung aktivieren, Rückkehr von anderem Programm ( Anwendung )*

```
Private Sub Frm_Open_Eye_Activated(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Activated
    Pb_Fortschritt.Value = 0
    Pb_Fortschritt.Visible = False
    Tr_Fortschritt.Enabled = False
End Sub
```

### *1.9.4.58 Fortschrittsanzeige ( Seite 6 )*

```
Private Sub Tr_Fortschritt_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Tr_Fortschritt.Tick
    Pb_Fortschritt.Value = Pb_Fortschritt.Value + 1
    If Pb_Fortschritt.Value > Pb_Fortschritt.Maximum - 2 Then
        Pb_Fortschritt.Value = 0
    End If
End Sub
```

### *1.9.4.59 Button Dokumentation speichern ( Seite 6 )*

```
Private Sub Bt_Store_Doku_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Store_Doku.Click
    Dim Proj_Nr As Integer
    Dim Datei As String
    Dim Typ As Integer
    Dim Bem As String
    Dim sonst As Integer
    Dim Info As String
    Proj_Nr = Val(Tb_Projekt_Id.Text)
    If Proj_Nr > 0 Then
        Datei = Tb_Doku_Pfad.Text
        Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
        Bem = Rtb_Bemerkung.Text
        sonst = Val(Tb_Sonstiges.Text)
        DokumentationTableAdapter.Insert_Doku(Proj_Nr, Datei, Typ, Bem, sonst)
        Load_Doku(Proj_Nr, Typ)
        Info = "Der Dateipfad ist auf der Datenbank gespeichert" + Chr(13)
    Else
        Info = "Es ist noch kein Projekt angelegt" + Chr(13)
        Info = Info + "Dateipfad konnte nicht gespeichert werden"
    End If
    MsgBox(Info, MsgBoxStyle.OkOnly, AcceptButton)
End Sub
```

### *1.9.4.60 Button Dokumentation löschen ( Seite 6 )*

```
Private Sub Bt_Delete_Doku_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Delete_Doku.Click
    Dim Id_Nr As Integer
    Dim Typ As Integer
    Dim Eintrag_Nr As Integer
    Dim Antwort As Integer
    Dim Info As String
    Antwort = MsgBox("Soll die Dokumentation wirklich entfernt werden?", MsgBoxStyle.YesNo, "Warnung")
    If Antwort = vbYes Then
      Eintrag_Nr = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
      If Eintrag_Nr < 0 Then
        Info = "Eintrag nicht gefunden. Löschen fehlgeschlagen"
        MsgBox(Info, MsgBoxStyle.OkOnly, AcceptButton)
      Else
        Id_Nr = Val(LB_Doku_Id.Items(Eintrag_Nr))
        DokumentationTableAdapter.Delete_Doku(Id_Nr)
        TeilelisteTableAdapter.Delete_ListeDoku(Id_Nr)
        Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
        Tb_Doku_Pfad.Text = ""
        Load_Doku(Val(Tb_Projekt_Id.Text), Typ)
        Info = "Eintrag ist entfernt"
      End If
      MsgBox(Info, MsgBoxStyle.OkOnly, AcceptButton)
    End If
  End Sub
```

### *1.9.4.61 Richtextbox Bemerkung ( Seite 6 )*

```
Private Sub Rtb_Bemerkung_Leave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Rtb_Bemerkung.Leave
    Dim Id_Nr As Integer
    Dim Pos_Nr As Integer
    Dim Typ As Integer
    If Bt_Store_Doku.Enabled = False Then
      Typ = Cb_DateiTyp.Items.IndexOf(Tb_Dateityp.Text)
      Pos_Nr = Cb_DokuPfad.Items.IndexOf(Tb_Doku_Pfad.Text)
      Id_Nr = LB_Doku_Id.Items(Pos_Nr)
      DokumentationTableAdapter.Update_Doku(Tb_Doku_Pfad.Text, Typ, Rtb_Bemerkung.Text, Val(Tb_Sonstiges.Text), Id_Nr)
    End If
  End Sub
```

### *1.9.4.62 Button Bauteil speichern ( Seite 7 )*

```
Private Sub Bt_Store_Bauteil_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Store_Bauteil.Click
    Tb_Eintrag_Nr.Text = "-1"
    Store_Bauteil(-1)
    Load_Bauteil()
    Load_Kategorie()
End Sub
```

### *1.9.4.63 Button Bauteil löschen ( Seite 7 )*

```
Private Sub Bt_del_Bauteil_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_del_Bautteil.Click
    Dim Id_Nr As Integer
    Dim Antwort As Integer
    Id_Nr = Val(Tb_Eintrag_Nr.Text)
    If Id_Nr >= 0 Then
      Antwort = MsgBox("Soll die Dokumentation wirklich entfernt werden?", MsgBoxStyle.YesNo, "Warnung")
      If Antwort = vbYes Then
        TeilelisteTableAdapter.Delete_Bauteil(Id_Nr)
        Load_Teileliste(Val(Tb_Doku_Nr.Text))
        Load_Bauteil()
        Load_Kategorie()
      End If
    End If
End Sub
```

### *1.9.4.64 Button Korrektur Bauteil ( Seite 7 )*

```
Private Sub Bt_Korr_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Korr.Click
    Dim Id_Nr As Integer
    Id_Nr = Val(Tb_Eintrag_Nr.Text)
    Store_Bauteil(Id_Nr)
End Sub
```

### *1.9.4.65 Button Datenblatt holen ( Seite 7 )*

```
Private Sub Bt_Datenblatt_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Datenblatt.Click
    Dim Old_Filter As String
    Old_Filter = DateiDialog.Filter
    DateiDialog.Filter = "Pdf Dateien (*.pdf)|*.pdf"
    DateiDialog.InitialDirectory = "c:\"
    DateiDialog.ShowDialog()
    TB_Bauteildoku.Text = DateiDialog.FileName
    TB_Bauteildoku.Enabled = True
    Bt_Lesen.Enabled = True
    DateiDialog.Filter = Old_Filter
End Sub
```

### *1.9.4.66 Button Datenblatt öffnen ( Seite 7 )*

```
Private Sub Bt_Lesen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Bt_Lesen.Click
    Ext_Programm.Start(TB_Bauteildoku.Text)
End Sub
```

### *1.9.4.67 Textbox Bauteil ( Seite 7 )*

```
Private Sub TB_Bauteildoku_Enter(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_Bauteildoku.Enter
    TB_Bauteildoku.Text = "-"
    TB_Bauteildoku.Enabled = False
    Bt_Lesen.Enabled = False
End Sub
```

### *1.9.4.68 Combobox Einheit Farad ( Seite 7 )*

```
Private Sub CB_Einheit_Farad_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles CB_Einheit_Farad.SelectedIndexChanged
    TB_Einheit_Farad.Text = CB_Einheit_Farad.Text
End Sub
```

### *1.9.4.69 Combobox Einheit Ohm ( Seite 7 )*

```
Private Sub Cb_Einheit_Ohm_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Cb_Einheit_Ohm.SelectedIndexChanged
    Tb_Einheit_Ohm.Text = Cb_Einheit_Ohm.Text
End Sub
```

### *1.9.4.70 Combobox Einheit Henry ( Seite 7 )*

```
Private Sub Cb_Einheit_Spule_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Cb_Einheit_Spule.SelectedIndexChanged
    Tb_Einheit_Spule.Text = Cb_Einheit_Spule.Text
End Sub
```

### *1.9.4.71 Combobox Kategorie ( Seite 7 )*

```
Private Sub Cb_Kategorie_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Cb_Kategorie.SelectedIndexChanged
    Dim Ref_Text As String
    Pn_Spule.Visible = False
    Pn_Widerstand.Visible = False
    Pn_Kondensator.Visible = False
    TB_Kategorie.Text = Cb_Kategorie.Text
    Ref_Text = TB_Kategorie.Text
    If (Ref_Text = "Widerstand") Or (Ref_Text = "Kondensator") Or (Ref_Text = "Spule")
Then
        Pn_Einheit.Visible = True
        If TB_Kategorie.Text = "Widerstand" Then
            Pn_Widerstand.Visible = True
        End If
        If TB_Kategorie.Text = "Kondensator" Then
            Pn_Kondensator.Visible = True
        End If
        If TB_Kategorie.Text = "Spule" Then
            Pn_Spule.Visible = True
        End If
    Else
        Pn_Einheit.Visible = False
    End If
End Sub
```

### *1.9.4.72 Combobox Wertevorgabe Farad ( Seite 7 )*

```
Private Sub CB_Wert_Farad_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles CB_Wert_Farad.SelectedIndexChanged
    TB_Wert_Farad.Text = CB_Wert_Farad.Text
End Sub
```

### *1.9.4.73 Combobox Wertevorgabe Ohm ( Seite 7 )*

```
Private Sub Cb_Wert_Ohm_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Cb_Wert_Ohm.SelectedIndexChanged
    Tb_Wert_Ohm.Text = Cb_Wert_Ohm.Text
End Sub
```

### *1.9.4.74 Combobox Wertevorgabe Henry ( Seite 7 )*

```
Private Sub Cb_Wert_Spule_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Cb_Wert_Spule.SelectedIndexChanged
    Tb_Wert_Spule.Text = Cb_Wert_Spule.Text
End Sub
```

### *1.9.4.75 Tabelle Bauteile ( Seite 7 )*

```
Private Sub Dg_Bauteile_CellClick(ByVal sender As System.Object, ByVal e As System.Windows.Forms.DataGridViewCellEventArgs) Handles Dg_Bauteile.CellClick
    Dim Wert As Integer
    Dim Row_Nr As Integer
    Dim Ref_Text As String
    Row_Nr = Dg_Bauteile.CurrentCell.RowIndex
    If Dg_Bauteile.Item("CL_Id_Bauteil", Row_Nr).Value <> Nothing Then          ' Zelle muss Wert enthalten
        Tb_Eintrag_Nr.Text = Str(Dg_Bauteile.Item("CL_Id_Bauteil", Row_Nr).Value)
        If Val(Tb_Eintrag_Nr.Text) > 0 Then
            Tb_Bauteil.Text = Dg_Bauteile.Item("CL_Bauteil", Row_Nr).Value
            Ref_Text = Trim(Tb_Bauteil.Text) ' Leerzeichen entfernen
            TB_Kategorie.Text = Trim(Dg_Bauteile.Item("CL_Bezeichnung", Row_Nr).Value)
            Pn_Kondensator.Visible = False
            Pn_Widerstand.Visible = False
            Pn_Spule.Visible = False
            Pn_Einheit.Visible = False
            Wert = Val(Dg_Bauteile.Item("CL_Wert", Row_Nr).Value)
            If Wert > 99 Then
                Wert = Wert / 10
                Cb_Faktor10.Checked = True
            Else
                Cb_Faktor10.Checked = False
            End If
            If (Ref_Text = "Widerstand") Or (Ref_Text = "Kondensator") Or (Ref_Text = "Spule") Then
                Pn_Einheit.Visible = True
                If Ref_Text = "Kondensator" Then
                    Pn_Kondensator.Visible = True
                    TB_Wert_Farad.Text = Str(Wert)
                    TB_Einheit_Farad.Text = Dg_Bauteile.Item("CL_Einheit", Row_Nr).Value
                End If
                If Ref_Text = "Spule" Then
                    Pn_Spule.Visible = True
                    Tb_Wert_Spule.Text = Str(Wert)
                    Tb_Einheit_Spule.Text = Dg_Bauteile.Item("CL_Einheit", Row_Nr).Value
                End If
                If Ref_Text = "Widerstand" Then
                    Pn_Widerstand.Visible = True
                    Tb_Wert_Ohm.Text = Str(Wert)
```

```
                Tb_Einheit_Ohm.Text = Dg_Bauteile.Item("CL_Einheit", Row_Nr).Value
            End If
        Else
            Pn_Einheit.Visible = False
        End If
        Pn_Einheit.Visible = Pn_Widerstand.Visible Or Pn_Spule.Visible Or
Pn_Kondensator.Visible
        Tb_Anzahl.Text = Dg_Bauteile.Item("CL_Anzahl", Row_Nr).Value
        Tb_Preis.Text = Dg_Bauteile.Item("Cl_Preis", Row_Nr).Value
        TB_Bauteildoku.Text = Dg_Bauteile.Item("Cl_Datenblatt", Row_Nr).Value
        Bt_Lesen.Enabled = TB_Bauteildoku.Text <> "-" ' Sperrt Button, um Datei zu
öffnen oder gibt es frei
    End If
  Else
    'wenn leere Tabellenspalte dann neuer Datensatz
    Tb_Eintrag_Nr.Text = "-1"
    Tb_Bauteil.Text = "-"
    TB_Kategorie.Text = "-"
    Cb_Faktor10.Checked = False
    TB_Wert_Farad.Text = "0"
    TB_Einheit_Farad.Text = "µF"
    Tb_Wert_Spule.Text = "0"
    Tb_Einheit_Spule.Text = "mH"
    Tb_Wert_Ohm.Text = "0"
    Tb_Einheit_Ohm.Text = "Ohm"
    Tb_Anzahl.Text = "0"
    Tb_Preis.Text = "0"
    TB_Bauteildoku.Text = "-"
    Tb_Bauteil.Select()
  End If
End Sub
```

### *1.9.4.76 Textbox Preis ( Seite 7 )*

```
Private Sub Tb_Preis_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles Tb_Preis.KeyPress
    If (e.KeyChar = ",") Then e.KeyChar = "."
    If (InStr(Tb_Preis.Text, ".") > 0) And e.KeyChar = "." Then
        e.KeyChar = ""
    End If
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 And e.KeyChar <>
"." Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(Tb_Preis.Text) = 8 Then
        e.KeyChar = ""
    End If
End Sub
```

### *1.9.4.77 Texbox Anzahl ( Seite 7 )*

```
Private Sub Tb_Anzahl_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles Tb_Anzahl.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(Tb_Anzahl.Text) = 4 Then
        e.KeyChar = ""
    End If
End Sub
```

### *1.9.4.78 Textbox Wert Farad ( Seite 7 )*

```
Private Sub TB_Wert_Farad_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TB_Wert_Farad.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(TB_Wert_Farad.Text) = 3 Then
        e.KeyChar = ""
    End If
End Sub
```

### *1.9.4.79 Textbox Wert Ohm ( Seite 7 )*

```
Private Sub Tb_Wert_Ohm_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles Tb_Wert_Ohm.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(Tb_Wert_Ohm.Text) = 3 Then
        e.KeyChar = ""
    End If
End Sub
```

### *1.9.4.80 Textbox Wert Henry ( Seite 7 )*

```
Private Sub Tb_Wert_Spule_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles Tb_Wert_Spule.KeyPress
    If ((e.KeyChar < "0") Or (e.KeyChar > "9")) And Asc(e.KeyChar) <> 8 Then
        e.KeyChar = ""
    End If
    If Asc(e.KeyChar) <> 8 And Len(Tb_Wert_Spule.Text) = 3 Then
        e.KeyChar = ""
    End If
End Sub
```

### *1.9.5 Startseite aufbauen*

```
Private Sub Tp_Start_Paint(ByVal sender As System.Object, ByVal e As System.Windows.Forms.PaintEventArgs) Handles Tp_Start.Paint
    ' Schwarze Linie ist der USB-ISP Weg
    e.Graphics.DrawLine(Pens.Black, 100, 15, 100, 210)
    e.Graphics.DrawLine(Pens.Black, 100, 15, 650, 15)
    e.Graphics.DrawLine(Pens.Black, 650, 15, 650, 80)

    ' rote Linie ist Programmieren über RS 232 zum Evaluationsboard  mit PonyProg
    e.Graphics.DrawLine(Pens.Red, 140, 80, 140, 210)
    e.Graphics.DrawLine(Pens.Red, 140, 80, 630, 80)

    ' blaue Linie ist die USB-RS232 Verbindung zum Evaluationsboard Datenleitung
    e.Graphics.DrawLine(Pens.Blue, 180, 110, 180, 220)
    e.Graphics.DrawLine(Pens.Blue, 180, 110, 630, 110)

    'Grüne Linien bauen den Pfeil auf
    e.Graphics.DrawLine(Pens.Green, 680, 150, 680, 200)
    e.Graphics.DrawLine(Pens.Green, 700, 150, 700, 200)
    e.Graphics.DrawLine(Pens.Green, 690, 220, 670, 200)
    e.Graphics.DrawLine(Pens.Green, 690, 220, 710, 200)
    e.Graphics.DrawLine(Pens.Green, 680, 150, 700, 150)
    e.Graphics.DrawLine(Pens.Green, 670, 200, 680, 200)
    e.Graphics.DrawLine(Pens.Green, 700, 200, 710, 200)
End Sub
```

```
End Class
```

### 1.9.6 Programm Ausgabeobjekt

#### 1.9.6.1 Interne Variablen

```
Public Class ValueBox
  Public Format As String
  Public Variable_Id As Integer   ' neu hinzugekommen
  Public Triggerwert As Byte
  Public Triggerlevel As Byte
```

#### 1.9.6.2 Funktion Get_Wert

```
  Public Function Get_Wert(ByVal Eingang As String) As Integer
    Dim MarkCnt As Integer
    Dim MarkPos As Integer
    Dim CalcWert As Integer
    Dim RefText As String
    Dim WorkText As String
    WorkText = Eingang                    ' Zuerst Arbeitsstring besetzen
    CalcWert = 0                          ' und Defaultwerte setzen
    MarkCnt = 0
    MarkPos = InStr(WorkText, ";")        ' Semikolon im String?
    If MarkPos = 0 Then
      CalcWert = Val(WorkText)            ' ein Byte gleich Zahl
    Else
      RefText = WorkText                  ' ein Byte gleich Zahl
      While MarkPos > 0
        RefText = Mid(WorkText, 1, MarkPos - 1)     ' Byte herauslösen
        WorkText = Mid(WorkText, MarkPos + 1, Len(WorkText) - MarkPos)
        ' Arbeitstext kürzen
        CalcWert = CalcWert + (Val(RefText) * 256 ^ MarkCnt) ' Zahlenwert aufaddieren
        MarkCnt = MarkCnt + 1                 ' Exponenten zähler erhöhen
        MarkPos = InStr(WorkText, ";")
        If MarkPos = 0 Then RefText = WorkText ' Letztes Byte
      End While
      CalcWert = CalcWert + (Val(RefText) * 256 ^ MarkCnt)    ' Letzes Byte aufaddieren
    End If
    Return (CalcWert)
  End Function
```

### *1.9.6.3 Textbox Dateneingang ( Tb_In )*

```
Private Sub TB_In_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TB_In.TextChanged
  Dim Zahlenwert As Int64
  If TB_In.Text <>“” then
    Zahlenwert = Get_Wert(TB_In.Text)
    If (Format = "Hex8") Or (Format = "Hex16") Or (Format = "Hex32") Then
      TB_Out.Text = Frm_Open_Eye.IntToHex(Zahlenwert, Format)
    End If
    If (Format = "Byte") Then
      TB_Out.Text = Frm_Open_Eye.IntToBin(Zahlenwert)
    End If
    If (Format = "Int8") Or (Format = "Int16") Or (Format = "Int32") Then
      TB_Out.Text = Str(Zahlenwert)
    End If
    If (Format = "ASCII") Then
      TB_Out.Text = Chr(Zahlenwert)
    End If
    TB_Out.BackColor = TB_In.BackColor
    If (Triggerlevel = Triggerwert) And (Triggerlevel <> 0) Then
      TB_Out.BackColor = Color.Pink
    End If
  End If
End Sub
```

### *1.9.6.4 Erfassung Maus mit Infofeld*

```
Private Sub ValueBox_MouseHover(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.MouseHover
    Dim Anzahl As Integer
    Dim i As Integer
    Dim InfoNr As Integer
    Dim InfoText(7) As String
    If Format = "Byte" Then        ' Format ist Objektvariable, nicht lokal
      Anzahl = Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Rows.Count
      For i = 0 To 7
        InfoText(i) = "nicht verwendet"
      Next
      For i = 0 To Anzahl - 1
        InfoNr = Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Item(i).Bit
        InfoText(InfoNr) = Frm_Open_Eye.EinzelbitTableAdapter.Get_VariablenBit(Variable_Id).Item(i).Funktion
      Next
      Frm_Open_Eye.LB_Name.Text = Lb_Variable.Text
      Frm_Open_Eye.Cb_Bit0.Checked = Mid(TB_Out.Text, 8, 1) = "1"
      Frm_Open_Eye.Cb_Bit0.Text = InfoText(0)

      Frm_Open_Eye.Cb_Bit1.Checked = Mid(TB_Out.Text, 7, 1) = "1"
      Frm_Open_Eye.Cb_Bit1.Text = InfoText(1)
      Frm_Open_Eye.Cb_Bit2.Checked = Mid(TB_Out.Text, 6, 1) = "1"
      Frm_Open_Eye.Cb_Bit2.Text = InfoText(2)
      Frm_Open_Eye.Cb_Bit3.Checked = Mid(TB_Out.Text, 5, 1) = "1"
      Frm_Open_Eye.Cb_Bit3.Text = InfoText(3)
      Frm_Open_Eye.Cb_Bit4.Checked = Mid(TB_Out.Text, 4, 1) = "1"
      Frm_Open_Eye.Cb_Bit4.Text = InfoText(4)
      Frm_Open_Eye.Cb_Bit5.Checked = Mid(TB_Out.Text, 3, 1) = "1"
      Frm_Open_Eye.Cb_Bit5.Text = InfoText(5)
      Frm_Open_Eye.Cb_Bit6.Checked = Mid(TB_Out.Text, 2, 1) = "1"
      Frm_Open_Eye.Cb_Bit6.Text = InfoText(6)
      Frm_Open_Eye.Cb_Bit7.Checked = Mid(TB_Out.Text, 1, 1) = "1"
      Frm_Open_Eye.Cb_Bit7.Text = InfoText(7)
      Frm_Open_Eye.Tb_WertHex.Text = Frm_Open_Eye.IntToHex(Val(TB_In.Text), Format)
      Frm_Open_Eye.Tb_WertInt.Text = TB_In.Text
      Frm_Open_Eye.Tb_WertASCII.Text = Chr(Val(TB_In.Text))
      Frm_Open_Eye.PN_BitAnzeige.Parent = Frm_Open_Eye.Tp_Visu
      Frm_Open_Eye.PN_BitAnzeige.Left = 590
      Frm_Open_Eye.PN_BitAnzeige.Top = 34
```

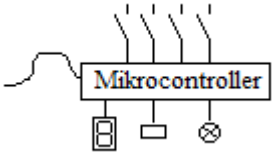
```
        Frm_Open_Eye.PN_BitAnzeige.BringToFront()
        Frm_Open_Eye.Pn_Trigger.Visible = False
        Frm_Open_Eye.PN_BitAnzeige.Visible = True
    End If
End Sub
```

#### *1.9.6.5 Erfassung Maus Bereich verlassen*

```
Private Sub ValueBox_MouseLeave(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.MouseLeave
    Frm_Open_Eye.PN_BitAnzeige.Visible = False
    Frm_Open_Eye.Pn_Trigger.Visible = True
End Sub
```

```
End Class
```

# 2 Controllerprogramme in Assembler




*Der Arbeiter*

## 2.1 Der Controller im Überblick

Ein kleiner Baustein mit 28 Anschlusspins, so präsentiert sich der Atmega8 dem neugierigen Bastler. Der Name „Mikrocontroller“ deutet etwas Kompliziertes an. Das ist sicherlich nichts für den Hobbybastler mit bescheidenen Kenntnissen in Elektrotechnik, geschweige Elektronik. Falsch gedacht, denn so schwierig ist es nicht, einen solchen Baustein für sein eigenes Projekt einzusetzen. Der Modelleisenbahner baut damit Ampeln, die richtig funktionieren. Vielleicht steuert er sogar die gesamte Bahnanlage mit einem PC und nutzt die Mikrocontroller als Schnittstelle für Signalausgabe und Signalerfassung. Der Slotcarfreund findet damit den Weg, die Runden der Rennwagen zu zählen und Zeiten zu nehmen. Der Modellbauer setzt sie ein, um seinen Modellen interessante Effekte mitzugeben. Ein anderer setzt den µC für eine Alarmanlage ein oder steuert Haus- und Hofbeleuchtung. Und dies alles erledigt der gleiche Baustein.

Klar, ein Laie findet nicht nur mal so zum Controller. Es ist schon etwas Arbeit damit verbunden und es gilt, einige Aufgaben zu lösen. Auch muß man sich mit der Programmierung dieser hoch integrierten Bausteine befassen, denn das ist der Preis, den der universelle Einsatz fordert.

Diese Lektüre soll ohne großen Ballast den Einstieg zur Controllerwelt ermöglichen. Gut, die Grundlagen eines Stromkreises sollten schon bekannt sein, denn schließlich arbeiten wir mit elektrischen Bauteilen, oder genauer gesagt elektronischen Bauteilen. Und damit sind wir auch schon mittendrin, im Thema. Ein Controller allein nützt nicht viel. Es muß schon ein wenig **drangebastelt** werden.

### 2.1.1 Zubehör

Damit wir auch sehen, wie so eine Schaltung aufgebaut ist, fertigen wir einen Schaltplan an. Doch nicht jeder hat CAD-Programme für elektrische Schaltungen. Ein kostenloser Ersatz ist **Paint** von Microsoft. Es ist bei jedem Windows im Zubehör dabei und lässt sich leicht für Skizzen verwenden. Schaltpläne sind aber die Basis für jedwede elektronische Schaltung. Ohne Schaltplan geht nichts. Klar, man kann mal eben drauf los löten oder im Steckbrett aufbauen, aber es wird nicht lange dauern bis man die Schaltung mangels Durchblick wieder zerpflückt.

Der Schaltplan zeigt auch, welche Bauteile noch zum Controller erforderlich sind. Daher ist auch bereits im Visual Basic Programm Open_Eye die Ablage von Skizzen und Schaltungen eingepflegt. Hier werden wir sie brauchen, denn bei allen Experimenten ist die Grundlage der Schaltplan.

Wenn wir eine Schaltung zeichnen wollen und uns den Controller ansehen, werden vermutlich wieder Zweifel aufkommen, ob man dem Thema gewachsen ist. Na ja, eine Pinbelegung ist schon erforderlich, denn aus dem Kopf die Anschlüsse definieren, das wird nix. Also kurz mal eine Internet-Suchmaschine bemühen und nach **Atmega8** (**Atmega16**) suchen. Das Datenblatt gibt es dann direkt von Atmel im Download als **PDF File**. Auch diese sind über Open_Eye erreichbar und die Pfade zur Dateiablage lassen sich im Progamm zum Projekt ablegen.

Auch das Programm **AVR Studio** ist kostenlos erhältlich. Es ist für die Programmierung der Controller erforderlich.

Die reine Theorie ist bei weitem nicht so interessant, wie die Praxis. Leider ist dieser Teil mit Kosten verbunden. Sicher, die Grundausstattung ist immer wieder verwendbar, aber dennoch sind 100 € schnell zusammen und oft auch überschritten.

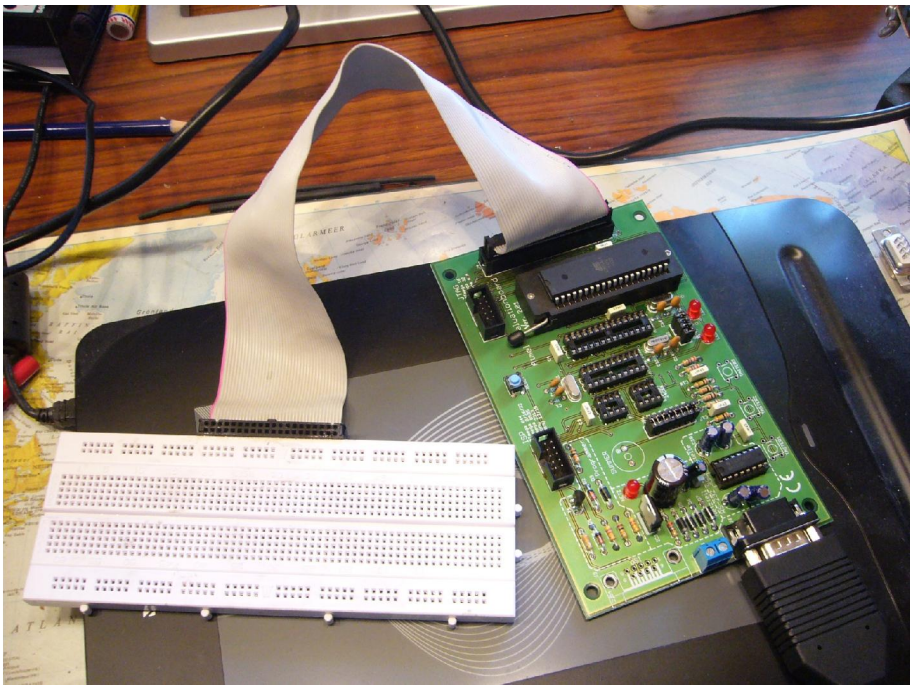
Die folgende Aufstellung ist lediglich ein Vorschlag. Es ist durchaus denkbar, das ein oder andere Teil mit Alternativen zu ersetzen. Für diese Anleitung spielt es auch keine Rolle, welche Platinen zur Programmierung eingesetzt werden. Ich habe mit dem Atmel-Evaluationsboard gearbeitet und werde kurz die Vorteile erklären:

Das Board ist für verschiedene Controller von Atmel geeignet. Die Stromversorgung ist bereits mit einem Festspannungsregler eingebaut, es

fehlt lediglich eine externe Versorgung. Dafür eignet sich ein preiswertes 12V Steckernetzteil. Auf der Karte ist eine serielle Schnittstelle. Nicht zu verwechseln mit der ISP-Programmierschnittstelle, welche ebenfalls mit einer 9 pol. Sub-D-Buchsenleiste ausgeführt ist. Verfügt der PC über eine echte serielle Schnittstelle, kann mit dem Programm PonyProg die Programmierung vorgenommen werden. Eine programmierung direkt aus AVR-Studio geht nur mit einem USB-ISP Programmieradapter. Hier ist auf die Bezwichnung ISP zu achten. Es gibt im Internet genügend preiswerte Programmieradapter, die unter 10 € angeboten werden. Das sind allerdings keine ISP sondern ISP USBASP Programmiergeräte, die AVR Studio auch nicht unterstützt.

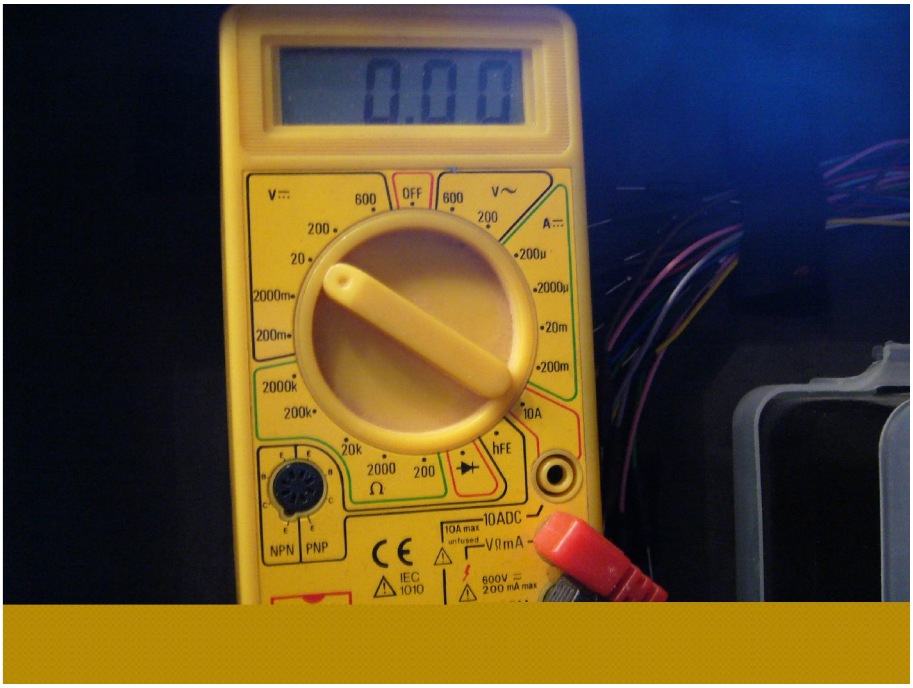
Auf dem Board ist eine zusätzliche ISP-Schnittstelle als Stiftleiste ausgeführt , die den Einsatz eines USB-ISP-Sticks, z.B. eines AVR-ISP-MKII ermöglicht. Damit ist die Programmierung direkt aus AVR-Studio gegeben und der Brennvorgang erfolgt sehr schnell. Wer sich den Bausatz für knapp 15 € bestellt, sollte statt der Fassungen einen Nullkraft-Sockel einsetzen. Dafür können die Taster sowie deren Beschaltung durchaus entfallen.(Bausatz)

Für die Verbindung zum Steckbrett habe ich ein 40 pol. IDE Flachkabel benutzt und am Steckbrett mit einem Zweikomponentenkleber angeklebt.



*.Atmel Evaluationsboard mit Steckbrett*

Ein Multimeter sollte auch nicht fehlen. Es gibt doch hin und wieder Situationen, wo es unentbehrlich ist. Es dürfen auch ruhig zwei sein, aber eines sollte mindestens von guter Qualität sein. Das abgebildete Multimeter kostet keine 5 €. Ich benutze es nur in den Schaltungen oder um mal schnell einen Widerstanf zu bestimmen. Da die Widerstände ggenormt sind, ist die Genauigkeit der Anzeige völlig ausreichend.




*Kleine Helfer Multimeter*

Es ist auch sinnvoll, einen Lieferanten für eine erste Lieferung auszuwählen, wenn ein Elektronikhandel nicht gleich um die Ecke ist.

### *2.1.2 Regeln für Arbeiten mit elektronischen Bauteilen*

Hitze:

Elektronische Bauteile vertragen keine Hitze. Möchte man Bauteile löten, sollte dieser Lötvorgang kurz sein. Wer seine Transistoren und Dioden minutenlang erhitzt, braucht sich nicht wundern, wenn die Lebensgeister entschwunden sind. Integrierte Schaltungen sollten prinzipiell auf Fassungen gesteckt werden. Aber auch bei Fassungen ist Dauererhitzung schädlich. Hier schmilzt der Kunststoff und verklebt die Kontaktfedern. Da dieses Buch die Arbeit mit einem Steckbrett beschreibt, besteht lediglich die Gefahr, das der Spannungsregler auf dem Evaluationsboard durch zu hohe Ströme zu heiß wird.

Statische Spannungen:

Integrierte Schaltungen sterben bei statischen Überladungen. Bauteile wie der µC Atmega8 sind hoch integrierte Schaltkreise mit mehreren Millionen Transistoren. Die Leiterbahnen sind mikroskopisch und entsprechend empfindlich. Es ist deshalb Sitte, solche Bauteile in antistatischer Verpackung zu halten und nur zum Einsatz heraus zu nehmen. Die Anschlüsse sollten nicht unbedingt angefasst werden, da auch ein menschlicher Körper durchaus statisch aufgeladen sein kann. Wie oft hat man dies an der Autotür oder in Räumen mit Teppichboden dann an einem Treppengeländer zu spüren bekommen. Sicherlich haben die Hersteller mittlerweile reagiert und Schutzdioden eingebaut, trotzdem ist Vorsicht geboten.

Auch verschiedene Transistoren mögen statische Spannungen überhaupt nicht. Bei den hier verwendeten BC-Typen allerdings besteht keine Gefahr.

Überspannung/ Überstrom:

Was ein elektronischer Baustein ebenfalls mit Abdanken und fristloser Kündigung quittiert, ist das Betreiben mit zu hoher Betriebsspannung. Daher ist es für viele Bauteile unumgänglich, die Datenblätter dazu aus dem Internet herunter zu laden. Selbst bei einfachsten Bauteilen, einer LED ist es wichtig, den maximal zulässigen Strom zu erfahren.

Als Bastler muss man nicht unbedingt die Details von LED's wissen. Mache Lieferanten bieten auch Datenblätter zum Download. So kann man die richtigen Betriebswerte erfahren.

Eine LED mit 20 mA kann demnach nach dem ohmschen Gesetz mit 250 Ohm an 5 V betrieben werden. Die Helligkeit wird sich aber kaum verändern, wenn ein Widerstand mit 300 Ohm gewählt wird. Der geringere Strom aber schont den Ausgang eines Controllers. Anders ist es bei Anzeigen im Multiplexbetrieb. Da sollte man schon nahe an die 20 mA kommen, ja selbst ein Überschreiten dieses Wertes ist nicht schädlich, da die LED's nur ganz kurz für den Bruchteil einer Sekunde zugeschaltet werden.

Bei anderen Bausteinen gilt es, die Betriebsparameter exakt zu beachten. Bei unseren Experimenten bewegen wir uns zu keiner Zeit in einem kritischen Bereich. Es sei denn, bei der Beschaltung der Bausteine werden grobe Fehler gemacht. Die Sorgfalt beim Verdrahten eines Experimentes ist daher unumgänglich.

## Wichtige Regeln:

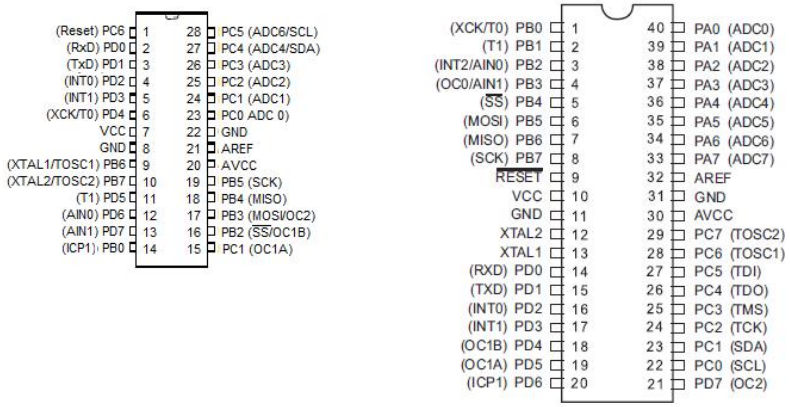
Keine Ausgänge kurzschließen nach **GND** (Ground) oder **VCC** (Versorgungsspannung)

Eingänge nie mit höherer Spannung beschalten als **VCC** und nicht kleiner als **GND** (negative Spannung)

Einen Mikrocontroller nicht mit 12 V beschalten (wenn 12V für Experimente genutzt werden)

### 2.1.3 *Auswahl Controller*

Um den geeigneten Controller auszuwählen, sollten wir uns erst einmal die Schaltbilder ansehen und prüfen, ob auch ausreichend IO-Pins zur Verfügung sind. Um eine 7-Segmentanzeige im Multiplex zu betreiben, erfordert es schon 7 Ausgänge für die Segmente plus 1 Ausgang je Ziffer. Das sind elf IO-Pins bei einer vierstelligen Anzeige. Zusätzlich wollen wir auch ein Relais schalten, vier Taster anschließen und eine Gabellichtschranke. Das sind weitere sechs IO-Pins, also insgesamt siebzehn. Da muss man schon prüfen, ob ein **Atmega8** ausreicht. Werfen wir daher einen Blick in das Datenblatt auf den Bereich **Pin Configuration**




*Pin Konfiguration Atmega8 Atmega16*

An den Anschlusspins stehen neben der Portbezeichnung noch weitere Begriffe. So ist an PC 6 beim Atmega8 auch der Reset des Controllers. Dieser Reset wird u.a. zur Programmierung verwendet und ist oft für eigene Anwendungen nicht geeignet. Der Atmega16 hingegen hat dafür einen eigenen Anschluss. Die Beispiele und Experimente sind mit beiden Controllern durchführbar. Lediglich die Ein- und Ausgabe an die Portpins muss angepasst werden.

PD 0 und PD 1 entfallen auch durch die Belegung mit den Signalen TxD und RxD, die für die serielle Kommunikation erforderlich sind. Wird der USART des Controllers parametriert, sind diese Anschlüsse automatisch verbunden.

PC 6 und PC 7 sind beim Atmega 8 zum Anschluss eines externen Quarzes vorgesehen und sind bei Verwendung der seriellen Schnittstelle

ebenfalls nicht nutzbar. Ein interner Takt ist zu ungenau für eine saubere und schnelle Datenübertragung oder der Generierung einer Uhrzeit.

Die Portpins PC 0 bis PC 5 sind auch für analoge Signale geeignet. (Atmega 8) / (PA 0 bis PA 7 beim Atmega 16)

Für weitere Controller sind die entsprechenden Datenblätter zu benutzen.

### *2.1.4 Portpin nutzen und festlegen*

Welche Pins darf ich verwenden? Eigentlich alle, aber trotzdem gibt es beim Atmega8 eine Einschränkung. Der Reset der Portpin PC 6. Dieser sollte nicht in die Planung einfließen, denn die Beschaltung des Reset wird vom Programmer benutzt, der ihn während des Schreibvorganges auf 0 zieht. Wenn der Controller seinen Dienst leistet, ist der Reset natürlich der Reset und sollte 1 Signal führen. Und das nützt uns nicht viel. Mit viel Vorsicht ist er aber brauchbar. Für uns ist er Tabu.

Auch sind die Pins PB6 und PB7 für die Beschaltung mit einem externen Quarz vorgesehen. Dies ist erforderlich bei Anwendungen, wo es eine genaue Taktzeit erfordert. Z. B. in Uhren oder auch bei der Datenübertragung. Wir setzen einen 16 MHz Quarz ein. Er ist auf dem Evaluationsbord bestückt. Allerdings sind die Controller im Auslieferzustand auf den internen Taktgeber eingerichtet und müssen erst auf den externen Quarz umgeschaltet werden. Doch dies zu einem späteren Zeitpunkt.

Die Festlegung der Pins erfolgt in einem Initialisierungsteil des Programmes. Je nach Einsatz und Aufgabe werden die Portpins unterschiedlich definiert und der Anwendung angepasst.

Bei unseren Experimenten werden in der Regel nur die einfachen Beschaltungen angewendet, das heißt Eingang oder Ausgang.

Für die serielle Verbindung werden RxD und TxD benutzt, also sind PD0 und PD1 auch nicht mehr verfügbar..

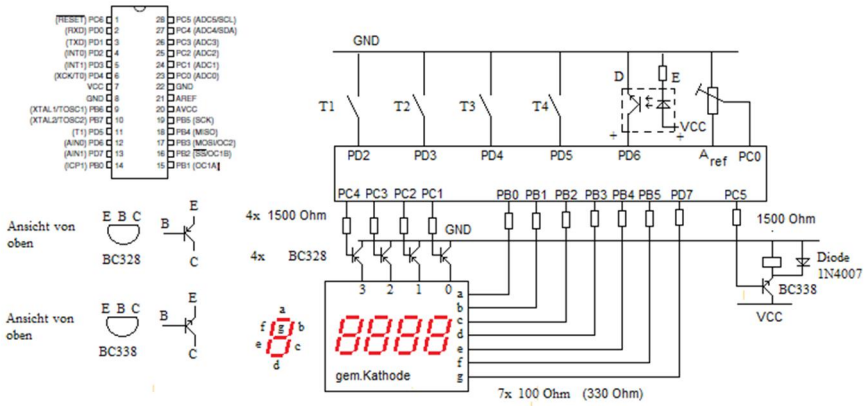
In einem Experiment wird PC 0 als Analogeingang verwendet.

So verbleiben, wenn ich richtig gezählt habe 18 freie IO-Pins für den Atmega8. Nicht gerade viel und wenn ich auf eine 6 stellige Anzeige erweitern möchte, muß ich mir schon etwas einfallen lassen. Da wir aber hier keine IC-Palette zur Pinerweiterung besprechen wollen, müssen die IO-Pin ausreichend sein. Wer mehr braucht, sollte auf den Atmega16 ausweichen. Hier gehen von den 32 IO-Pins nur die beiden für die serielle Verbindung verloren. Reset und auch Quarz haben eigene Anschlüsse.

Unabhängig davon werde ich mit dem Atmega8 arbeiten.

### *2.1.5 Schaltplan Ziel der Experimente*

An dieser Stelle zeige ich den Schaltplan, den wir Schritt für Schritt erarbeiten und die Funktionalität der einzelnen Komponenten in einem Assemblerprogramm einbinden. Schließlich sollt ihr ja wissen, was euch erwartet und welche Bauteile erforderlich sind.




*Schaltplan des Zielaufbaus*

Die Aufgaben umfassen das Betreiben von LED, Ansteuern von Relais und 7-Segmentanzeige, Funktion eines Transistors als Signalverstärker und natürlich auch die Signalerfassung über Schalter, Potentiometer oder Optokoppler.

Wir werden uns die einzelnen Aufgaben vornehmen und mit kleinen Experimenten das erforderliche Wissen um elektronische Schaltungen erwerben.

Lasst euch also nicht vom Umfang dieser Schaltung beeindrucken.

### *2.1.6 Portpins Einrichten*

Wie im Schaltbild zu erkennen ist, sind die Portpins mit unterschiedlichen Aufgaben belegt. Der Controller lässt da sehr viel Freiheit, so das jeder Pin speziel an eine Aufgabe angepasst werden kann. Ausnahmen sind Sonderfunktionen wie TxD und RxD oder Analoge Eingänge. Aber die normale IO Ebene ist frei programmierbar. Es ist auch nicht erforderlich, das ein Port nur Ausgang oder nur Eingang ist. Hier kann durchaus alles bunt gemischt sein. Das hört sich im ersten Moment doch ziemlich kompliziert an. Mag sein, aber dafür schreibe ich diese Anleitung. Ihr werdet sehen, das es so kompliziert auch nicht ist.

Nach unserer Schaltung ist

Ausgang: PB0 bis PB5, PC 1 bis PC 5 und PD7

Eingang digital: PD 2 bis PD6

Eingang analog: PC0 (Erst einmal aber noch digitaler Eingang)

Die Auflistung zeigt, das die IO-Pins nicht unbedingt zusammen hängend sein müssen, denn das ist hier offensichtlich. Die 7 Segmente bekomme ich in keinem Port unter.

### 2.1.7 *Experiment LED*

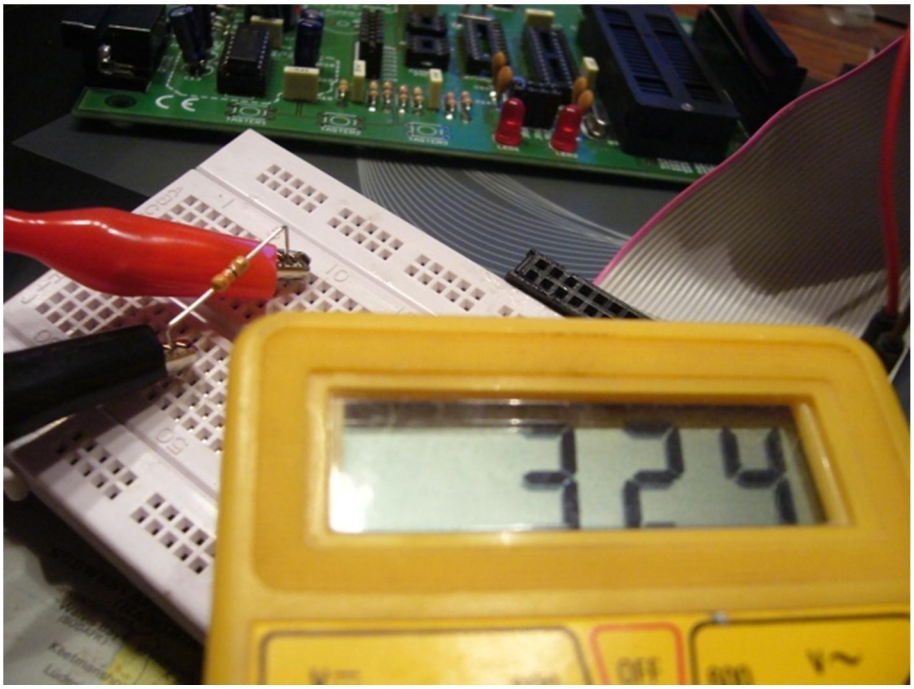
Damit wir ein wenig Übung mit den Bauteilen bekommen, nehmen wir zuerst einmal eine LED. Das ist eine Diode, die einen Strom nur in einer Richtung fließen lässt. Dabei emittiert sie Photonen, also Licht. Es ist eine „light emitting diode“, daher auch der Name. Allerdings ist darauf zu achten, dass der Strom nicht größer als zulässig ist. Genaueres steht wieder im Datenblatt. In der Regel zeigt eine LED aber schon zwischen 5 bis 10 mA ein deutliches Leuchten. Auch wenn ich versprochen habe, ohne Ballast die Thematik zu beschreiben, muss an dieser Stelle etwas Mathematik angewendet werden. Vielleicht ist ja das Ohmsche Gesetz bekannt, vielleicht aber auch nicht. Der Herr Ohm hat erkannt, das bei gleich bleibender Spannung der Strom sinkt, wenn der Widerstand größer wird. Steigt hingegen die Spannung bei gleichem Widerstand, wird auch der Strom größer. Aus dieser Erkenntnis hat er dann eine Formel aufgestellt. Für den Widerstand wird der Formelbuchstabe R, für die Spannung U und für den Strom I eingesetzt. Daraus folgen die drei Gleichungen:

$$I = \frac{U}{R} \qquad U = I * R \qquad R = \frac{U}{I} \Rightarrow R = \frac{5\,V}{0{,}010\,A}$$

*das Ohmsche Gesetz*

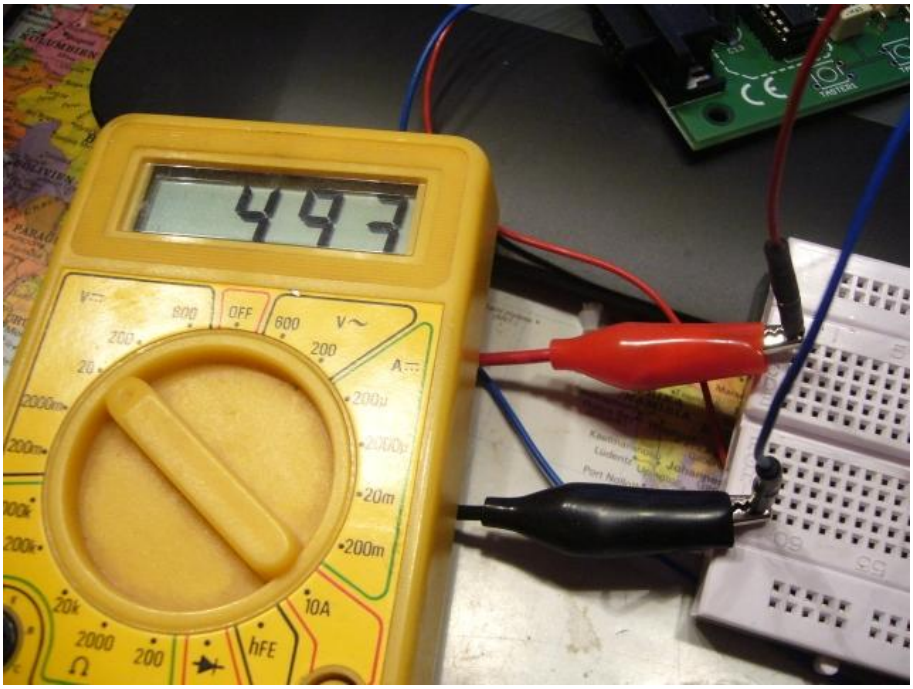
Wichtig bei der Anwendung von Formeln, das die Werte immer in der Grundeinheit angegeben werden. Möchte ich den Strom bei 5 V auf 10 mA begrenzen, muss ich 5 V durch 0,01 A teilen. Das Ergebnis ist 500 Ohm.

Nun, so genau brauchen wir es nicht. Bei dieser Faustformel ist die Diodenspannung noch gar nicht berücksichtigt und die liegt in etwa bei 2 V. Das ist von Typ zu Typ unterschiedlich. Ziehen wir die 2 V von den 5 V ab, liegen wir bei 300 Ohm. Zwischen 270 und 330 Ohm gibt es da keine Probleme. Dazu ein paar Messungen. Zuerst überprüfen wir den Widerstand. Dazu das Multimeter in einen geeigneten Messbereich stellen und den Widerstand anschließen. Er sollte bei einer Einstellung 2000 Ohm Messbereich ca. 330 Ohm anzeigen.



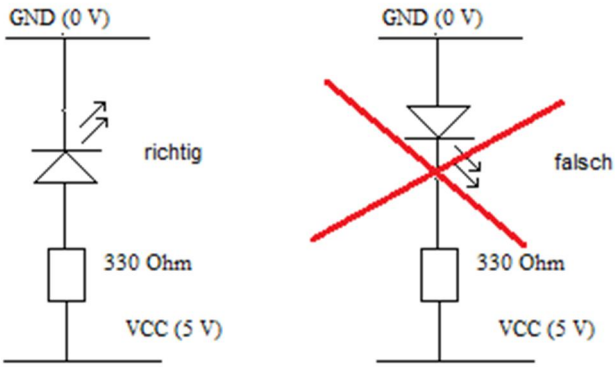

*Messung Widerstand*

Im nächsten Schritt messen wir die Spannung. Dazu stecken wir Steckbrücken nicht ganz so tief in die Kontakte, so dass wir mit Messspitzen oder wie hier verwendet, Krokodilklemmen an die blanken Stifte kommen.




*Messung Spannung*

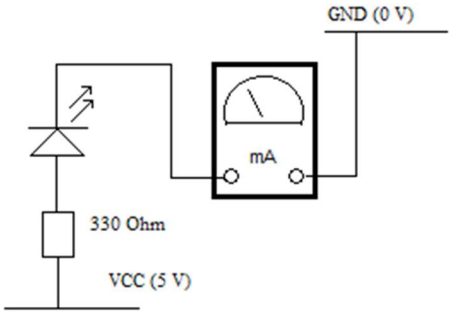
Ist die Spannung 5 V können wir die Schaltung aufbauen und anschließen. Auf einem Steckbrett ist das schnell erledigt. Prüfen wir doch gleich einmal, ob das stimmt und die LED sichtbar leuchtet:



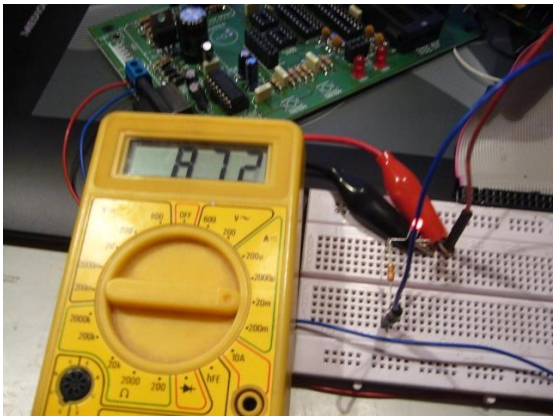

*LED Schaltung*

Sind die Anschlüsse richtig gepolt, sollte die LED leichten. Leuchtet sie nicht, ist die LED vermutlich falch herum angeschlossen. Das macht aber einer LED nichts aus, denn es ist ja eine Diode. Es fließt einfach kein Strom.

Wenn die LED leuchtet, können wir auch den Strom messen. Dazu stellen wir das Multimeter auf den Messbereich 200 mA. Achtung, keine Spannung messen! Das Multimeter muss wie in der Skizze gezeichnet in Reihe mit der LED und dem Widerstand liegen.




*LED- Strommessung*

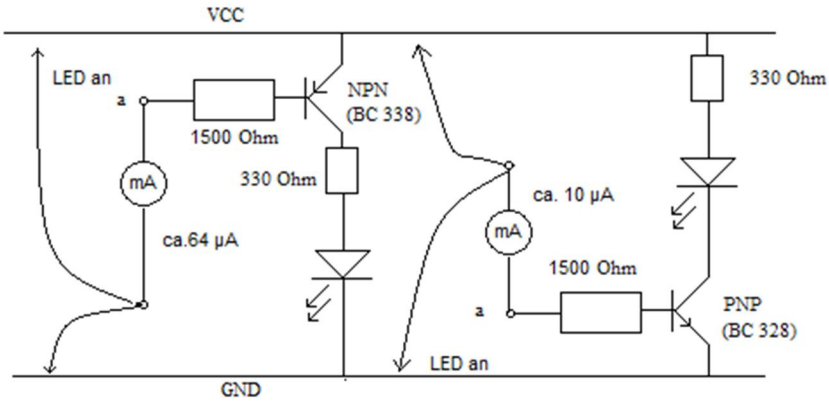


*Diodenstrom messen*

Nach einer Strommessung sollte das Multimeter sofort wieder auf Spannung schalten, um nicht mit einer anschließenden Spannungsmessung eine Kurzschluß auszulösen und das Gerät möglicherweise zu zerstören Ablageort  Datenbank

### *2.1.8 Experimente mit Transistor*

Damit uns auch die Arbeitsweise eines Transistors vertraut wird, möchte ich hier gleich zwei einfache Transistorschaltungen für einfache Experimente einfügen.

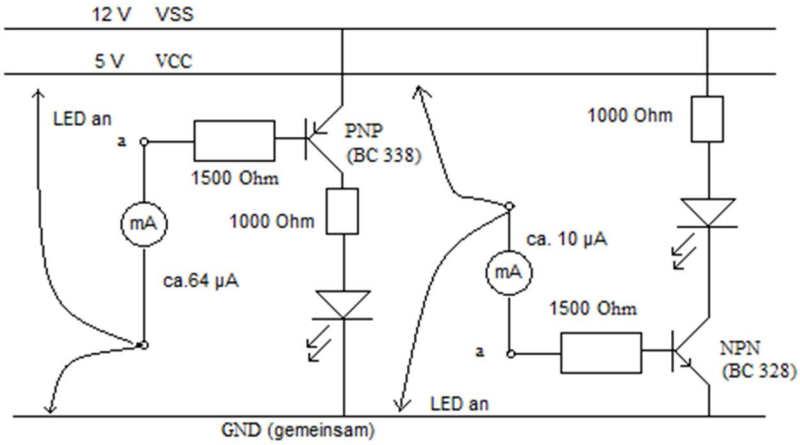



*Transistorschaltung mit LED*

Die Versuche zeigen es deutlich. Die Transistoren steuern nur durch, wenn über den Widerstand das passende Potenzial an die Basis gelegt wird. Ist die Beschaltung offen oder das falsche Potenzial idt die LED aus. Messen wir doch einmal den Strom, der benötigt wird, um die LED zum leuchten zu bringen. Dazu wird das Messgerat einfach in Reihe zum Basiswiderstand geschaltet. Der Strom ist deutlich geringer, als der Strom, der über den Transistor zur LED fließt. Aber wozu das Ganze?

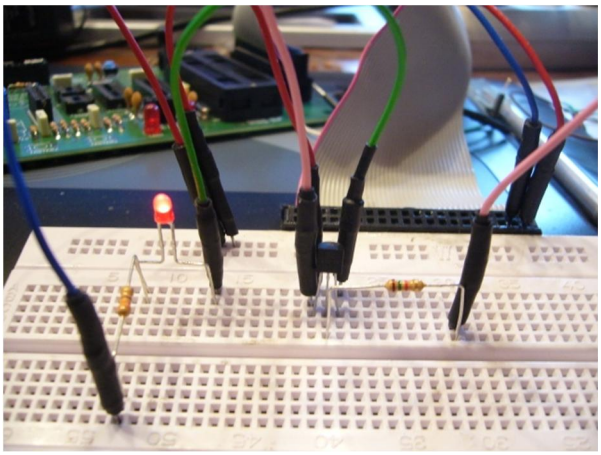
Nun, der Ausgang eines Mikrocontrollers kann nur begrenzt Strom liefern. Ist auch klar. Stellen wir uns einmal vor, alle Portpins eines Controllers sind auf Ausgang geschaltet und jeder Ausgang wird mit 20 mA belastet. Das kann er leisten, aber bei ca. 30 Ausgängen sind wir da schon bei 600 mA. Und ich habe es am Anfang ja angedeutet. Ein Mikrocontroller ist ein hochintegrierter Schaltkreis mit ganz feinen Verbindungen. Ein Haar ist ein dickes Seil dagegen. Die Belastung wäre auch für den Controller zu hoch. Wie weit man mit der Belastung eines Portpins gehen kann und wie weit insgesamt mit dem Controller steht im Datenblatt. Aber es gibt auch andere Bauteile, die ein Controller schalten soll. Ein Relais zum Beispiel und da sind wir nicht mehr bei 20 mA. Außerdem sind 5V Relais teuer und die Kontakte sind auch nicht hoch belastbar. 12 V oder 24 V Relais dagegen können mit den 5 V, die der Controller liefert, nix anfangen.

Und hier ist der Transistor zu Hause. Er muss nämlich nicht mit seinem Collektor an den 5 V hängen. Da dürfen auch 12 oder 24 V angeschlossen sein. Dennoch reicht ein 5V Signal über den Widerstand aus, um ihn durchzusteuern. Für unser Experiment ändern wir einmal den Vorwiderstand der LED auf 1kOhm und verbinden den Collektor mit +12V.




*Transistorschaltung LED an 12 V*

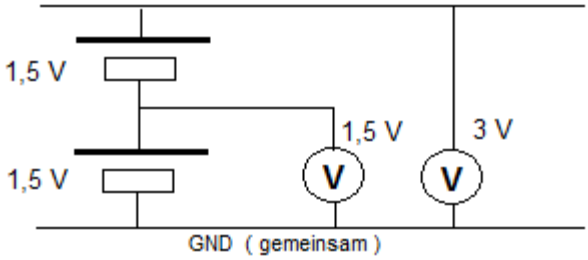
Auch diesmal leuchten die LED hell, obwohl der Strom zur Basis des transistors sich kaum verändert hat. Wichtig ist, das der Minus, also GND von beiden Spannungsquellen verbunden ist.



*Transistor PNP auf Steckbrett*

### *2.1.9 Arbeiten mit verschiedenen Spannungen*

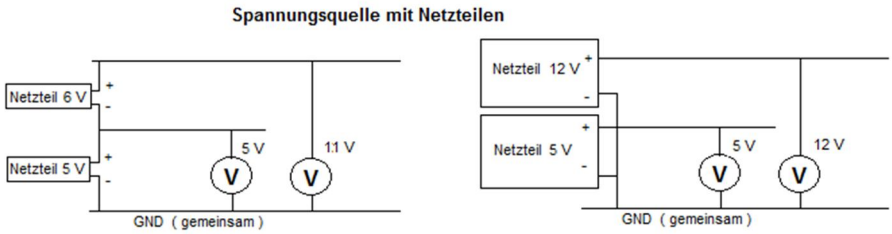
Der Ansatz ist bereits angesprochen und ich denke, das dieses Thema ein wenig mehr Information benötigt. Angenommen, wir haben zwei Batterien, die hintereinander geschaltet sind. Eine Baterie liefert eine Spannung von 1,5 V. Also haben wir an beiden Batterien 3 V.




*Spannungsquelle Batterie*

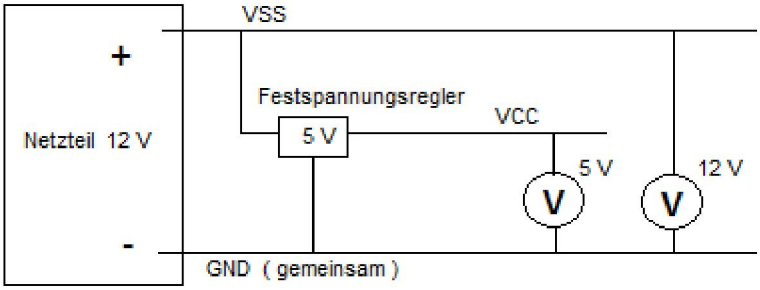
Im Schaltbild ist es einfach und nachvollziehbar zu erkennen. Beide Spannungsmessungen beziehen sich auf den Minus der ersten Batterie. Die gemessenen Werte sind einleuchtend.

Ersetze ich die Batterien durch Netzteile, ist das Ergebnis ähnlich.




*Spannungsquelle mit Netzteilen*

So werden aber in den seltesten Fällen Spannungen für elektronische Schaltungen erzeugt. In der Regel gibt es ein Netzteil mit 9 oder 12V. Auf der elektronischen Schaltung ist ein Festspannungsregler, der aus der Versorgungsspannung die benötigte Festspannung liefert. Das Grundprinzip wird in der nächsten Skizze deutlich.



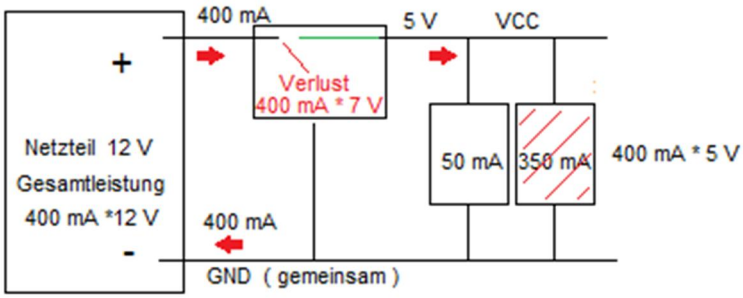

*Spannungsquelle Netzteil*

Alle Schaltungen haben eines gemeinsam. Der Minus oder wie er in Schaltungen auch immer wieder genannt wird GND, ist bei der Spannungsmessungen mit allen Spannungsquellen in irgendeiner Art verbunden. nur so ist ein Stromfluss möglich.


*Festspannungsregler*

### *2.1.10 Verlustleistung und Wärme*

Und nun noch ein klein wenig Theorie. Im Physikunterricht sollten wir es gelernt haben, das der Strom im Stromkreis überall gleich ist. Betrachten wir unter diesen Bedingungen einmal die Skizze mit dem Festspannungsregler und die Leistungsverteilung.




*Verlustleistung Festspannungsregler*

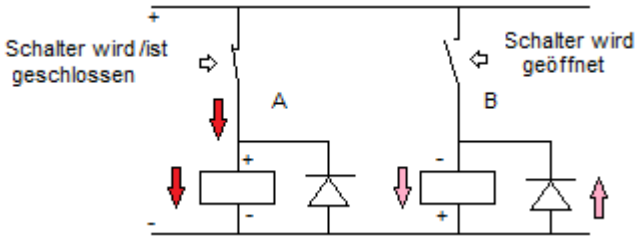
Nehmen wir einmal an, ein Netzteil liefert 12 V, die elektronische Schaltung nimmt 50 mA und weitere Bauteile wie Relais, LED und was auch immer 350 mA. Somit fließen aus dem Netzteil 400 mA Gesamtstrom. Die Leistungsformel sollte auch noch bekannt sein. Die Leistung ist das Produkt aus Strom und Spannung (U * I). Somit liefert das Netztei eine Leistung von 12 V * 400 mA = 4,8 VA (Watt hats erfunden)

Die Schaltung selbst setzt davon 5 V * 50 mA um, also 0,25 W. Das Zubehör hat eine Leistungsaufnahme von 5 V * 350 mA, entsprechend 1,75 W. Dann fehlt in der Leistungsbilanz ganze 2,8 W. Wo werden die verbraten? Nun ja, verbraten ist das richtige Wort, denn auch wenn so ein Festspannungsregler oftmals auch mehr als 1 A liefern kann, die Leistung muss er irgendwie loswerden. Und die ist nun mal Spannungsdifferenz zum Gesamtstrom. / V *400 mA sind die fehlenden 2,8 W, die der Regler in Wärme umsetzt. Deshalb besitzt er auch eine große Metallplatte, auf die dann ein noch größerer Kühlkörper geschraubt wird, wenn das Leistungsverhältnis entsprechend schlecht ist. Wäre nur die Elektronik mit ihren 50 mA dran, müssten zwar immer noch 7 * 50 mA = 350 mW in Wärme abgeführt werden, aber dadurch wird der Regler nur handwarm. Im ersten Fall könnte man sich schon mal die Finger verbrennen. Aus diesem Grund ist beim Einsatz eines Festspannungsregler die

Versorgungsspannung ein wichtiges Kriterium. Nur weil grad ein 24 V Netzteil verfügbar ist, muss die Schaltung nicht unbedingt funktionieren, denn der Regler müsste bei dieser Betrachtung die zusätzlichen 4,8 W. also insgesamt 7,6 W loswerden. Der clevere Elektroniker packt auch deshalb alle irgendwie stromziehende Verbraucher in den ungeregelten 12 V Kreis und versorgt nur Elektronik mit den geregelten 5 V, die eine solche stabilisierte Spannung benötigen. Selbst LED und Anzeigen lassen sich, wie unser Experiment mit den 12 V gezeigt hat, problemlos mit einem 12 V Steckernetzteil aus den 5 V raushalten. Die gelieferte Spannung ist ausreichend stabil, so das ein Übersteuern einer LED nicht zu erwarten ist. Relais sind sowieso unempfindlich, was Spannungsschwankung betrifft und gehören schon standartmäßig in den 12 V Kreis.

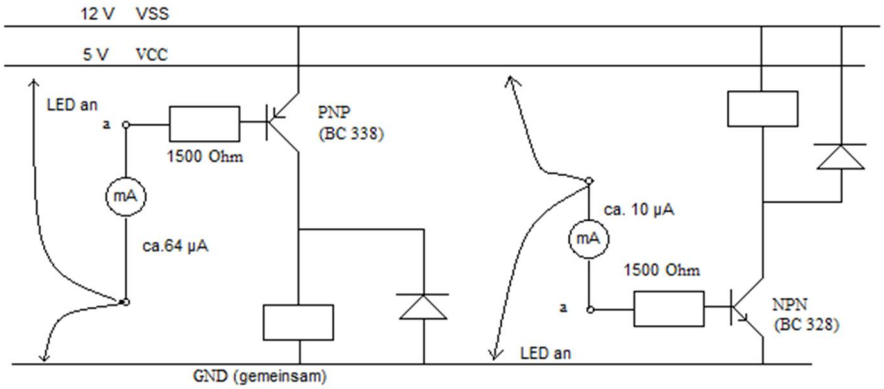
### *2.1.11 Ein Relais mit einem Transistor schalten.*

Das Schalten eines Relais oder Elektromagneten allgemein in einer elektronischen Umgebung erfordert eine Vorsichtsmaßnahme. Auch hier ein kleiner Rückblick in die Physik Magnetismus und Elektrizität. Die Eigenschaft, die es ermöglicht, das bei uns allen das Licht angeht ist dem Umstand zu verdanken, das ein Leiter, der durch ein Magnetfeld bewegt wird, eine Spannung erzeugt, die in der Höhe abhängig von der Anzahl der Windungen und von der Geschwindigkeit der Bewegung ist. Darüber sind wir froh, denn es ermöglicht uns vielfältige Bequemlichkeiten. Elektrizität ist aus unserem Leben nicht wegzudenken. Aber wieso müssen wir hier vorsichtig sein und Maßnahmen ergreifen? Wogegen? Nun, was uns einerseits nützlich ist, hat auch Schattenseiten. So ist eine Relaisspule ein Leiter mit vielen Windungen. Wird sie angesteuert, baut sich ein Magnetfeld auf und der Anker zieht an. Das Relais schaltet. Das ist gut so. Da gibt es nichts dran auszusetzen. Aber.. wir schalten ein Relais auch ab. Klar, das Licht soll ja wieder ausgehen. Nur was passiert mit der Spule im Relais. Das Magnetfeld wird durch den Abschaltvorgang schlagartig abgebaut, was einer Bewegung der Leiter im Magnetfeld gleichkommt und da dieser Vorgang sehr schnell ist, ist auch die erzeugte Spannung in der Relaisspule enorm hoch. So hoch, das Halbleiterschichten zerstört werden. Man kann es sich wie einen Blitzeinschlag vorstellen. Was kann dagegen getan werden, damit diese Eigenschaft von Induktivitäten die Elektronik massenweise in die Tonne kloppt. Vielleicht kann sich der ein oder andere noch an seine Schulzeit und den Physikunterricht erinnern. Die Richtung, in welcher der Leiter bewegt wurde eizeugte auch eine Spannung in die eine oder andere Richtung. Also von 0 nach plus und auch entgegengesetzt. Das bedeutet, ein abschaltendes Relais erzeugt eine der Versorgungsspannung entgegengesetze Spannung. Und nun kommt eine Diode ins Spiel. Dioden, das haben wir schon bei der LED gelernt, lassen eine Spannung nur in eine Richtung durch. Schalten wir eine Diode in Sperrrichtung an das Relais, stört das beim Zuschalten nicht. Durch die Diode fließt kein Strom. Wird das Relais abgeschaltet, wird die Induktionsspannung der Spule, da sie der Versorgungsspannung entgegengesetzt ist, kurzgeschlossen und kann sich nicht zu einer großen Spannung aufbauen. Genau gesagt, die Spannung wird -0,7 V kaum überschreiten. Eine Skizze hilft den Vorgang zu verstehen.



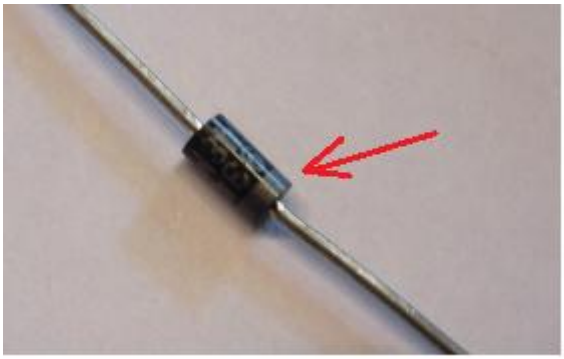

*Relais mit Freilaufdiode*

Setzen wir nun statt des Schalters einen Transistor ein. Wie bei den Experimenten mit Transistor und LED wird hier lediglich die LED gegen ein Relais getauscht. Aber nicht die Diode parallel zum Relais vergessen und auch richtig herum einbauen.




*Transistorschaltung mit 12V Relais*

Bei einer 1 N4007, einer Universaldiode ist der weiße Balken mit dem Strich über dem Dreieck in der Skizze vergleichbar. Man spricht auch von der Kathode, das ist der Anschluss, zu dem der Strom, kommend von der Anode fließt. Anode = +, Kathode = - Als Faustregel kann man sich auch merken der Ring ist ein langer Strich und ein Strich ist -



*Diode mit Ring*

Nun sind die ersten Experimente durchgeführt und ein wenig Grundwissen vermittelt. Die Funktion einer LED ist deutlich geworden und warum sie einen Vorwiderstand braucht. Der Transistor ist kein geheimnisvolles Bauteil mehr und wir wissen nun, wie er eingesetzt wird. Auch was Stromversorgung betrifft sind wir um einiges klüger. Es ist klar, warum ein Festspannungsregler heiß wird, obwohl er weit unter der genannten Stromgrenze betrieben wird. Das sollte auch erst einmal genug der Theorie sein und so wenden wir uns dem Controller und seiner Programmierung zu.

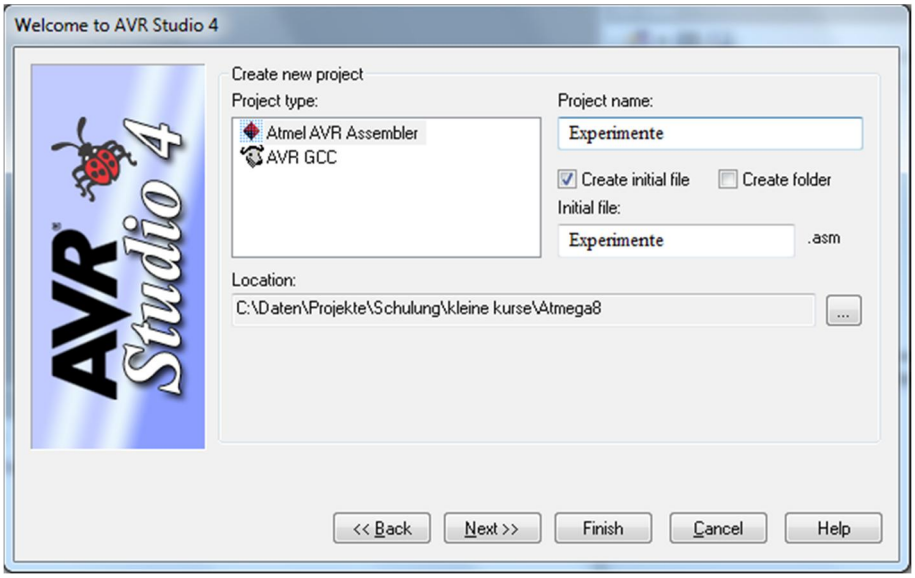
## 2.2 Programmieren von Atmega 8 (16)

Der µC, in unseren Experimenten ein Atmega8, wird in Assembler mit dem AVR-Studio programmiert. Diese Software arbeitet nicht direkt mit dem Evaluationsboard von Pollin. Daher ist es erforderlich, entweder PonyProg zum Flashen der Controller zu installieren, oder optional einen USB-ISP Stick anzuschaffen. Damit ist der Brennvorgang auch direkt aus AVR Studio heraus möglich und es geht auch um Welten schneller. Die Mehrkosten von ca. 20 -40 € sind schnell verschmerzt.

Wer auf ein Evaluationsboard verzichten möchte, kann natürlich auch relativ einfach auf dem Steckbrett einen Programmierplatz einrichten und den USB-ISP Stick über das Steckbrett einsetzen. Allerdings ist dann auch eine 5V Spannungserzeugung erforderlich. Die serielle Schnittstelle muss auch selbst aufgebaut werden. Auf diese Eigenbauten werde ich nicht näher eingehen. Das Internet ist voll mit entsprechenden Anleitungen. Die unterschiedlichen Handhabungen können aber unmöglich in diesem Buch berücksichtigt werden. Alle Beschreibungen bezüglich der Programmübertragung an den µC legen das Pollin AVR-Evaluations Board und einen USB-ISP Programmer zugrunde. Andere mit AVR-Studio kompatible Programmer sind natürlich auch anwendbar.

### *2.2.1 AVR Studio*

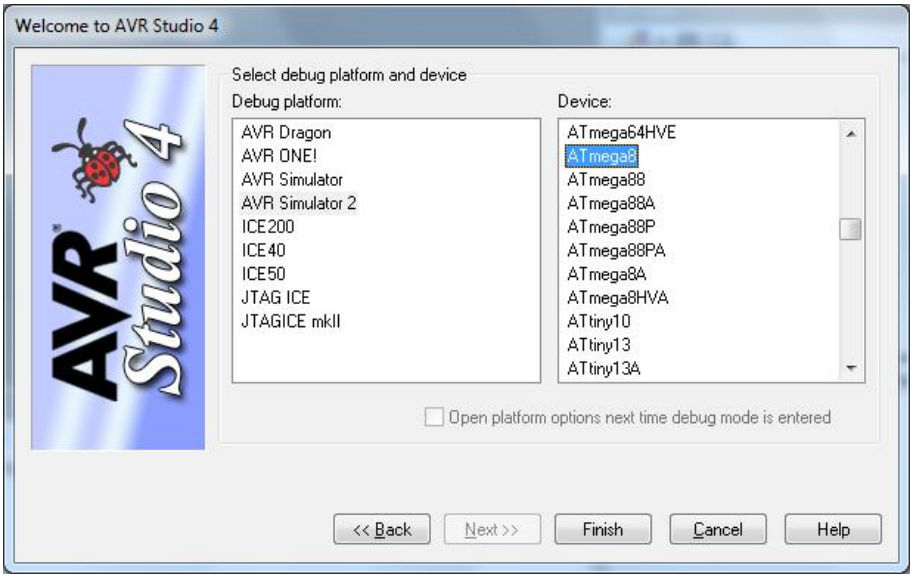
Beginnen wir mit AVR Studio. Dieses Programm kann kostenlos im Netz herunter geladen werden. Mit AVR Studio schreiben wir das Programm für den Mikrocontroller.In diesemm Buch arbeite ich mit Version 4.xx Wer bereits eine höhere Version hat, sollte auch zurechtkommen.




*AVR Studio Welcome*

In der Welcome Seite wird die Auswahl zwischen Assembler und AVR GCC, einem C Compiler angeboten. Dieser Buchabschnitt befasst sich ausschließlich mit dem Assembler.

Hier vergeben wir den Namen des Projektes **Experimente** und des Assembler- File, welches auch mit **Experimente** benannt wird. Mit next wechseln wir zur nächsten Seite.

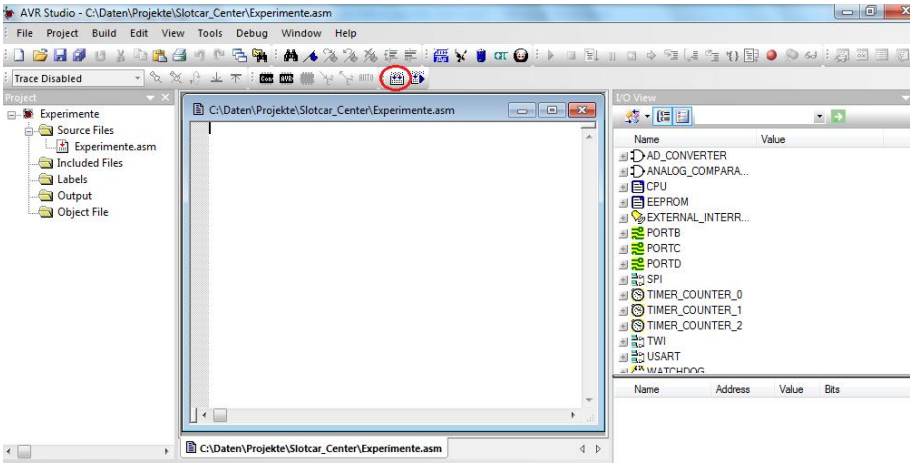



*AVRStudio Controllerauswahl*

In der Plattform **AVR Simulator 2** finden wir den **ATmega8** in der Auflistung der Controller. Mit **Finish** bestätigen wir die Auswahl und wenden uns dem Programmeditor zu.

### *2.2.2 Ein neues Projekt*

AVR Studio präsentiert sich mit einem Editor, der eine völlig andere Programmiergrundlage liefert, wie wir sie von Visual Basich kennen. Hier ist alles noch Handarbeit. Jede Zeile will geschrieben werden. Objekte einfügen? Nicht in Assembler. Aber keine Angst, wenn die Programmiertechnik richtig angewendet wird, sind die geschriebenen Routinen für weitere Programme verwendbar.

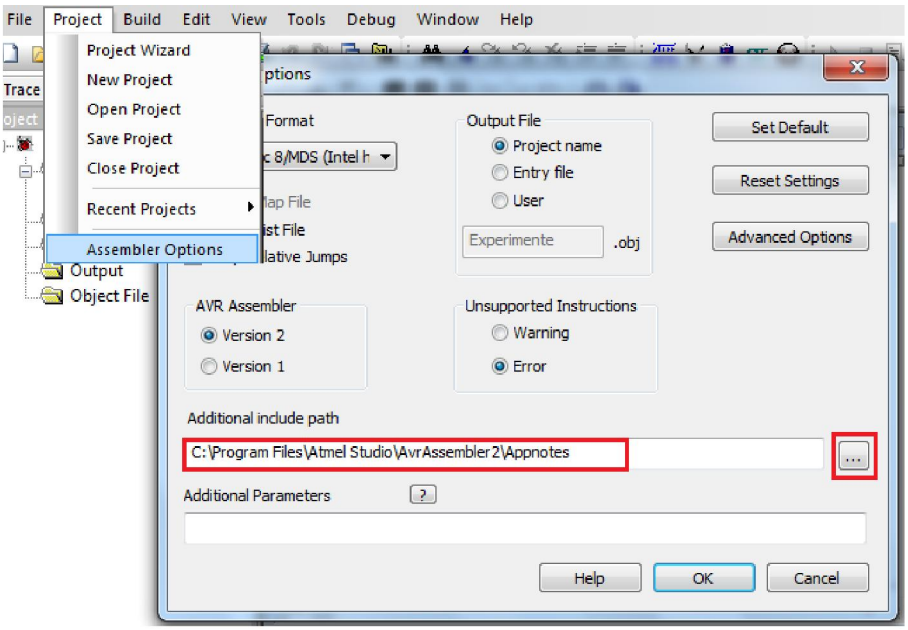



*AVRStudio Programmeditor*

Vielleicht sollten wir uns erst einmal mit den wichtigsten Funktionen vertraut machen, bevor wir mit der Programmierung beginnen.

Das File-Handling werde ich nicht besonders beschreiben. Die Funktionen zum Speichern und Laden sollten hinlänglich bekannt sein. Wichtig sind die Hilsmittel wie Suchfunktionen aber auch, wo die controllerspezifischen Dateien abgelegt sind. Mit diesen beginne ich die Beschreibung der wichtigsten Funktionen.

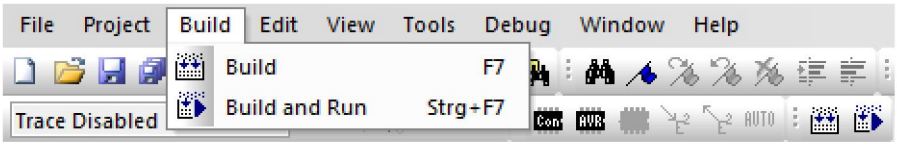
### 2.2.3 Beschreibung Menüleiste.




*Projekt Controller Fileablage*

Es ist manchmal notwendig, nachzusehen, wieso ein Registerbit plötzlich einen Namen hat. Zu gegebener Zeit komme ich darauf zurück.

Der Menüpunkt **Build** Compiliert das Programm in ein übertragungsfähiges Hex-File oder startet den Simulator.

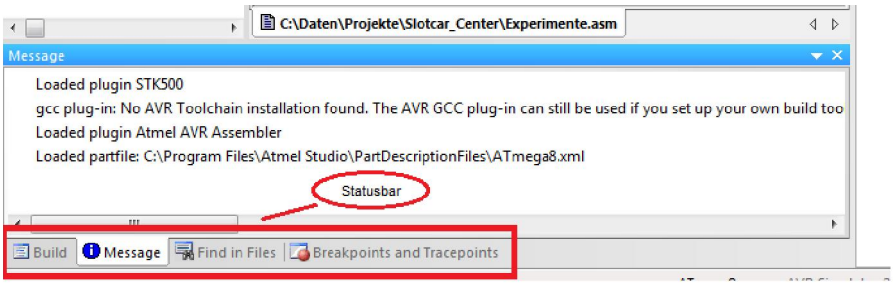



*Menüleiste Build*

**Build** generiert nur das Hexfile. **Build and Run** startet zusätzlich den Simulator. Das hat mit dem Controller noch nichts gemeinsam. Die beiden Icons tauchen auch in der Toolbar auf.

Unter **Edit** finden wir unter anderem auch die Suchfunktionen. Auch liegt hier die Verwaltung von Haltepunkten. Unter dem Menüpunkt **View**

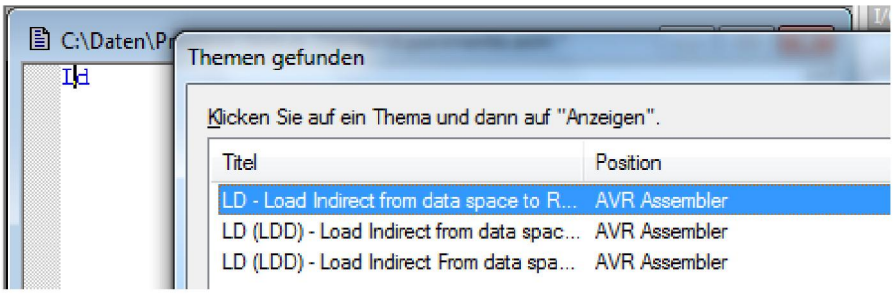
könnnen die Ansichten von **Toolbar** und **Statusbar** angepasst werden. Die **Statusbar** ist unter dem Editor angelegt.




*Statusbar*

In den **Tools** sind einige interessante Einstellungen und auch die Verbindungseinstellung zum Controller.

Das Menü Debug ist für den Simuletor interessant und Window ist auch nicht so wichtig für den Anfänger. Help allerdings beinhaltet die Hilfe zu Assembler und das ist, obwohl in Englisch, ein wichtiges Hilfsmittel. So reicht es, einen Assemblerbefehl zu schreiben, den Cursor in den Befehl zu setzen und F1 zu betätigen.



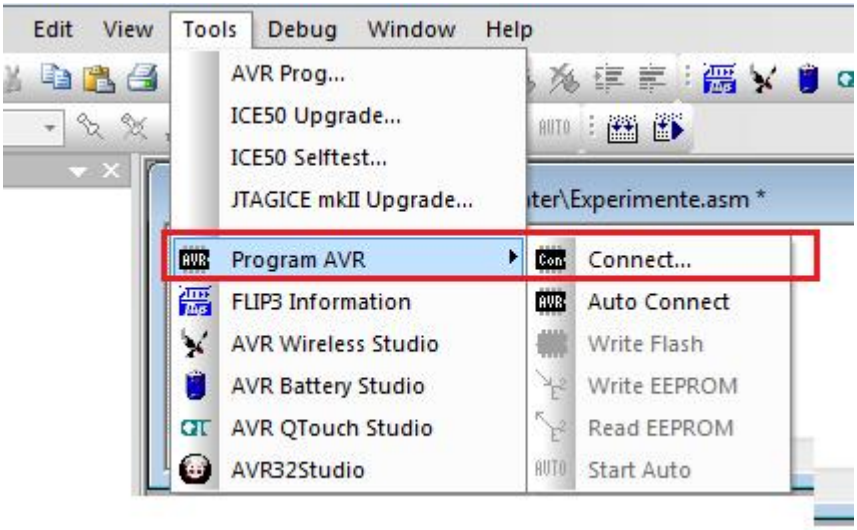

*Assembler Onlinehilfe*

Man findet in den Rubriken nicht nur Erklärung sondern auch Beispiele.

Kommen wir aber noch einmal zurück zu den Tools

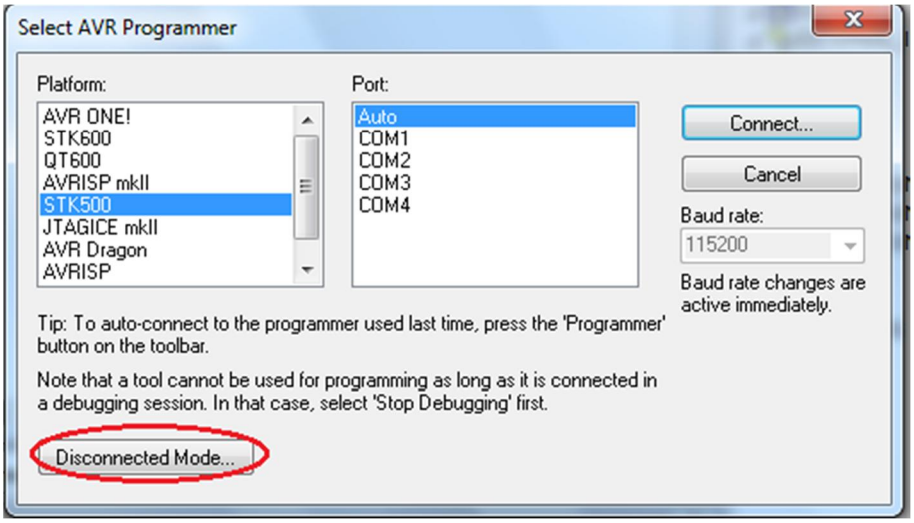
### 2.2.4 Wichtige Einstellungen

Von allen Einträgen interessiert eigentlich nur der rot eingerahmte Menüpunkt.



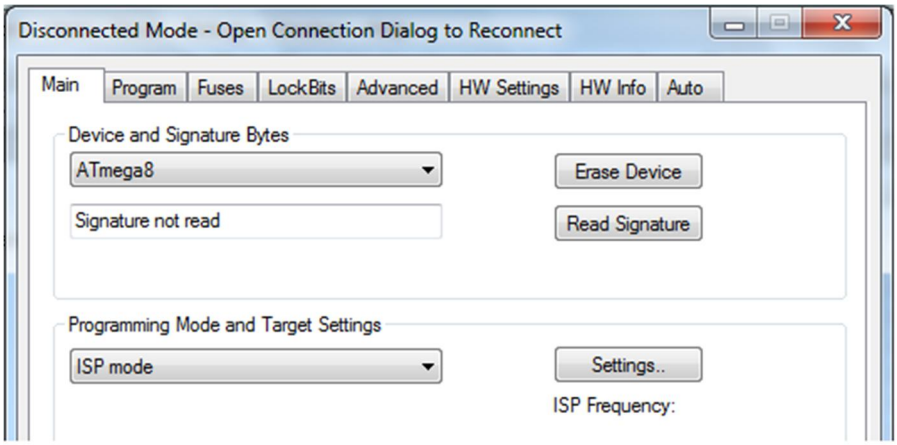

*Verbindungseinstellungen*

Hier wird die Schnittstelle vom Studio mit dem Programmiergerät festgelegt. Aber auch wichtige Einstellungen sind hier vorzunehmen.



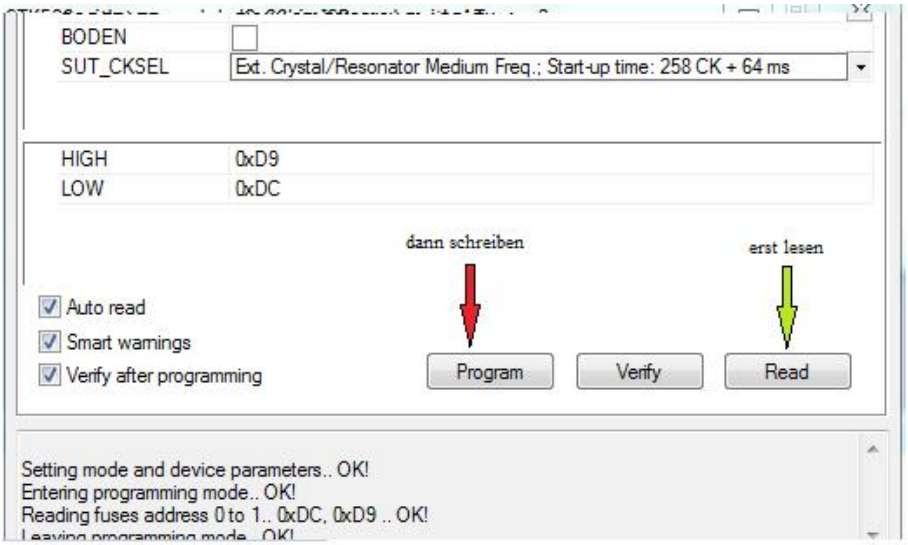

***Verbindung herstellen***

Einen besonderen Blick sollten wir auf den Button **Disconnect Mode** werfen. Dort verbirgt sich eine ganz wichtige Eigenschaft. Die ISP Frequenz. Ein Controller darf macimal mit ¼ seiner Taktfrequenz beschrieben werden. Diese Einstellung finden wir im Tabulator **Main** in der Combobox **Programming Mode and Target Setting** Über den Button **Settings** kann die Frequenz eingestellt werden. Manchmal vergisst man diese Vorgabe und wenn schon Controller mit einem Quarz unter 16 MHz laufen, nimmt man die **ISP-Frequenz** gern schon mal ein bisschen hoch. Wenn dann mal wieder ein Controller mit werksseitig eingestellen internen 1 MHz beschrieben werden soll, geht es in die Hose. Nicht selten liest man in den Foren **Controller lässt sich nicht programmieren**.




*ISP Frequenz einstellen*

Ebenfalls wichtig, aber mit Vorsicht und Bedacht zu verwendende Seite ist Fuses. Hier kann man ohne sich anzustrengen, den Kontakt zu einem Controller für immer verlieren. Sehen wir uns zuerst diese Seite einmal an.




*Bei FuseBits Beachten*

Bevor an den Fusebits irgendetwas verstellt wird ist es erforderlich diese zu **lesen**. Das ist ganz wichtig, um nicht ein Bit aus Versehen zu setzen. Wir werden nur die Fusebits **Sut_CKSEL** verändern, um einen externen Quarz als Taktgeber einzuschalten. Für eine zuverlässig funktionierende Kommunikation über die serielle Schnittstelle unumgänglich. Den Bereich habe ich grün eingerahmt.

Die Seite Programm hält auch noch eine kleine Falltür bereit. Wenn wir im Datenblatt die Features lesen, finden wir den Eintrag 512 Byte EEPROM. Um auf diesen Daten zu schreiben, ist ein separater Schreibvorgang erforderlich. Und dafür erzeugt der Compiler, wenn er Daten im ESEG vorfindet ein eigenes File

ESEG - Segment EEProm oder Speicherbereich EEProm. Dies muss hier ausdrücklich auch angegeben sein. Zu den einzelnen Speicherbereichen im Controller kommen wir aber noch. Erst einmal genug der Worte.

## 2.2.5 Speicherbereiche eines Mikrocontrollers

In einem Mikrocontroller sind im Datenblatt verschiedene Arten von Speicherbereichen angegeben. Diese Information ist wichtig für den Einsatz eines Controllers. Hier ein Auszug aus dem Datenblatt eines Atmega8.

High Endurance Non-volatile Memory segments
– 8Kbytes of In-System Self-programmable Flash program memory
– 512Bytes EEPROM
– 1Kbyte Internal SRAM
– Write/Erase Cycles: 10,000 Flash/100,000 EEPROM.

Interessant ist die letzte Zeile, denn sie beschreibt die Lebenszeit der verschiedenen Speicherbereiche.

Der FlashSpeicher (CSEG=Code-Segment) kann ca. 10000 Schreibzyklen verkraften. Danach muss mit Ausfällen gerechnet werden Klar, 10000 Schreibzyklen auf einen Programmspeicher, das ist völlig ausreichend. Dafür bleibt der Inhalt bei einem Stromausfall auch erhalten. Zeitlich nicht begrenzt. So kann er monatelang in der Schublade liegen, bevor er zum Einsatz kommt und das Programm ist immer noch vorhanden.

Ein ähnliches Konzept hat der EEPROM (ESEG). Auch ein EEPROM verliert seinen gespeicherten Inhalt nicht. Allerdings mit 512 Byte taugt er nicht für ein Programm. Dafür ist er auch nicht gedacht, aber zur Ablage von Parametersätzen oder wichtigen Daten bei einem detektierten Stromausfall ist er ausreichend. Dennoch, so frei nutzbar ist er auch nicht und unterliegt gewissen Auflagen. Das ergibt sich aus seiner Lebensdauer von 100000 Schreibzyklen. Sicherlich das ist das 10fache vom Flashspeicher, aber im Programm kann das innerhalb weniger Sekunden das Aus für den Speicher bedeuten, wenn man ihn wie einen normalen Speicher benutzen will und der Meinung ist, permanent eine Uhrzeit mitzuschreiben, um den Stromausfall genau festzuhalten. Das wäre ja dann der letzte Eintrag. Nun, ein Controller erledigt einen kompletten Programmdurchlauf einschließlich der Schreibzugriffe in wenigen Millisekunden. Nehmen wir ein furchtbar langsames Programm mit 100 mSek. Zykluszeit. Nach 3 bis vier Minuten ist dann der EEPROM zerstört.

Das Datensegment (DSEG): ein unendlich beschreibbarer Speicher, allerdings erkauft mit Datenverlust bei Spannungsausfall. Es ist in der

Regel aber auch nicht notwendig, diesen Inhalt zu behalten. Wenn es unbedingt erforderlich ist, gibt es Maßnahmen, die auf solche Ereignisse reagieren und eine Sicherung wichtiger Daten auf den EEPROM legen.

Das Datensegment ist ein Speicherbereich, der ständig beschrieben werden muss. Da sind Programmereignisse, die eine Adressenablage erfordern ( STACK) oder Werte ablegen, um sie im Programm bearbeiten zu können. (Variablen). Register sind ebenfalls solche Speicherzellen. Ständig werden Werte hinein geschrieben, Selbst wenn nur ein Schreibzugriff auf einen solchen Speicher in einem Zyklus stattfinden würde, hätten wir in einer Stunde über 2 Millionen. Und so ein Controller läuft jahrelang… und selten mit so einem langsamen Zyklus. Oftmals sind Programmzyklen im Bereich von Mikrosekunden. Wenn man bedenkt das ein Takt bei 8 MHz 1/8000000Sekunden ist, also 0,000000125 oder 125 Nanosekunden. 2 bis 3 Takte benötigt ein Befehl, also 325 nSek. Das sind 1000 Befehle in 325 µSek. Solche Laufzeitberechnungen werden wir uns gegebenenfalls im den Programmabschnitten noch ansehen. Hier möchte ich nur auf die Geschwindigkeit hinweisen und deutlich machen, warum der EEPROM nicht im Programm unbedacht beschrieben werden darf.

## 2.2.6 Assembler Programmstruktur

Ein genereller Unterschied ist bei einem Assemblerprogramm, das alle Variablen global sind. Das bedeutet, am Programmanfang werden die Adressmarken mit Namen dem Compiler kenntlich gemacht. Ihr lest richtig, ein Assembler ist im Grunde ein Compiler, der die Befehle für den Controller in die Welt der Maschinen übersetzt. Ein Controller kann mit **LDI R 16, 127** gar nichts anfangen. Was er braucht, das sieht im Prinzip so aus: **11100111 00001111** Erst mit diesen symbolisierten elektrischen Werten können die Arbeitsschritte im Controller durchgeführt werden. Wer diesbezüglich mehr Information haben möchte, kann das **AVR Instruction Set Manual** durcharbeiten. Hier führt es auf jeden Fall zu weit, denn das ist doch schon eher etwas auf Expertenebene.

Dennoch hielt ich diesen kleinen Exkurs für erforderlich, damit **Assembler** verstanden wird. Es ist nur eine Programmiersprache. Allerdings auf unterster Ebene. Nichts bewegt sich näher am Controller. Manche empfinden es als Nachteil, denn so sind die Programme nicht einfach portierbar. Ein in **C** erstelltes Programm braucht oftmals für einen anderen Controller einer anderen Familie lediglich eine Anpassung der Hardware. Den Rest erledigt der **C Compiler.**

Obwohl dieser Nachteil besteht, ist **Assembler** kein Abstellgleis, wie oftmals behauptet**.** Viele Vorgänge werden nirgends so deutlich**,** wie in dieser Sprache. Und wenn ein Assemblerprogramm gut strukturiert ist, verliert man nicht den Überblick.

## 2.2.7 *Assemblerbefehle*

So ein paar Befehle sollten wie auch schon im Basic Teil im Vorfeld kurz angesprochen werden. Dies ist keine vollständige Auflistung, denn diese ist auch in AVR Studio unter Assemblerhilfe zu finden.

Ein Controller kann nur Befehle in Registern ausführen. Dazu müssen diese mit Werten geladen werden. Deshalb beginnen wir mit den Ladebefehlen:

### *2.2.7.1 Ladebefehle:*

| | |
|---|---|
| MOV Register A, Register B | Kopiert den Wert von Register B nach Register A |
| MOVW r1: r2, r3:r4 | Kopiert Wort z.B.ZH:ZL, XH:XL Ergebnis in ZH:ZL |

Bewege zwischen Register

| | |
|---|---|
| LDI Register, <Konstante> | Lädt direkt einen festen Wert in Register |

(Load immediate) Die Konstante kann sein

Dezimalwert, Ganzzahl von 0 bis 255

Binär Bitwert von 0b00000000 bis 0b11111111

Hexadezimal Hexwert von 0x00 bis 0xff

ASCII Zeichen ‘**A‘. ‘B‘, ‘C‘** usw.

| | |
|---|---|
| LDS Register, <Variable> | Kopiert den Inhalt einer Speicherzelle in Register |

(Load direct from SRAM) Die Variable ist die mit einem Namen versehene Adresse der Speicherzelle, deren Inhalt gelesen wird

| | |
|---|---|
| LD Register, X (Y,Z) | Kopiert den Inhalt einer Speicherzelle in Register |

(Load indirect) Die Adresse auf die Speicherzelle steht im den Doppelregistern X, Y oder Z. Das Register setzt sich aus

XH ( High-Adressteil ) und XL ( Low-Adressteil ) zusammen.

X+ erhöht automatisch die Adresse nach dem Ladevorgang

-X reduziert automatisch die Adresse vor dem Ladevorgang

( auch Y+, -Y, Z+,- Z)

| | |
|---|---|
| IN Register, PINx | Register wird mit Portwert geladen |

Ein Port ist auch ein Register oder eine Speicherzelle, allerdings mit direkter verbindung zur Außenwelt. In der Bezeichnung PInA, PinB, PInC, oder PInD steckt auch wieder die Adresse des Portregisters.

| | |
|---|---|
| POP Register | Registerwert wird vom Stack geladen |

Die Anweisung POP ist ein spezieller Ladevorgang eines Registers. Dabei wird der Inhalt mit Hilfe eines weiteren Adressregisters, dem Stackpointer (Adresszeiger) aus dem DSeg (RAM) abgeholt. und der Adresszeigerr wieder erhöht, also gibt diese Speicherzelle sozusagen wieder frei. Man spricht vom Stack (Stapel) weil die Ablage so funktioniert. Stellen wir uns ein Brett mit einem Nagel vor. Ich schreibe mir Notizen und spieße diese auf den Nagel (Stapel) Zur Abarbeitung nehme ich nun einen Zettel nach dem anderen wieder vom Stapel. Somit wird der immer der nächste Zettell wieder zugänglich. Der Stack im Controller funktioniert genauso, nur das er bei der Ablage von oben nach unten adressiert und bei der Abnahme mit dem Zeiger wieder die höheren Speicherzellen adressiert.

### *2.2.7.2 Speicherbefehle:*

| STS Variable, Register | Kopiert den Inhalt eines Registers in eine Speicherzelle |
|---|---|

(Store Direct to SRAM) Die Variable ist die mit einem Namen versehene Adresse der Speicherzelle, die beschrieben wird.

| ST X, Register | Kopiert den Inhalt eines Registers in einer Speicherzelle |
|---|---|

(Store indirect) Die Adresse auf die Speicherzelle steht im

den Doppelregistern X, Y oder Z

Beschreibung sonst wie LD-Befehl

| PUSH Register | Registerwert wird auf Stack geschrieben |
|---|---|

PUSH ist das gegenteil von POP. Mit diesem Befehl wird der Inhalt eines Registers auf die mit dem Stackpointer adressierte Speicherzelle geschrieben, und der Stackpointer auf die nächstniedrigere freie Speicherzelle gesetzt. Also, das Ablegen von Notizen auf einem Nagelbrett wie unter der Beschreibung von dem Befehl POP.

| OUT PortX, Register | Registerwert wird in Port geschrieben |
|---|---|

Ein Port ist auch ein Register oder eine Speicherzelle, allerdings mit direkter verbindung zur Außenwelt. In der Bezeichnung PortA, PortB, PortC, oder PortD steckt auch wieder die Adresse des Portregisters.

### *2.2.7.3 Arithmetik und Logik:*

| ADD Register A, Register B | Addition zweier Registerinhalten, Ergebnis in Register A |
|---|---|

Addition der Werte in zwei Registern. Das Ergebnis ist im erstgenannten register abgelegt, der Inhalt von dem zweiten register ändert sich nicht.

| ADC Register A, Register B | wie ADD, aber mit Übertrag. Ergebnis in Register A |
|---|---|

(Add with Carry) Ist in einer vorherigen Addition ein Übertrag entstanden,

(z.B. 143+189 ist > 255) dann wird der Übertrag bei der Addition der beiden Registerwerte hinzugefügt.

Beispiel : Add Reg_A, Reg_B 143 + 189 = 332

Das 7.Bit hat in einem Byte den Wert 128 und ist das höchste. Wenn alle Bits in einem Byte gesetzt sind beträgt der Wert 255, also reichten 8 Bits nicht aus, um den Wert von 332 aufzunehmen. Ein weiteres Bit, das Carrybit hätte aber den Wert 256 und steckt die 77 locker ein. Also müssen wir die Addition Binär betrachten und dem achten Bit ein neuntes hinzufügen. Dann ist die Zahl 332 auch mit 101001100 abzubilden. Dieses neunte Bit ist das Carrybit und hat in diesem Fall den Wert 256 und in diesem Fall bliebe für die acht unteren Bits nur der Rest von 76 abzubilden.

Bei Bit-addition gilt: 1+0 = 1 Übertrag 0
1+1 = 0 Übertrag 1

| Register | Wert |
|---|---|
| Req_A | 1 0 0 0 1 1 1 1 |
| + | |
| Reg_B | 1 0 1 1 1 1 0 1 |
| + | |
| Übertrag | 1 0 1 1 1 1 1 1 |
| Ergebnis | 1 0 1 0 0 1 1 0 0 |

*Binäre Addition*

| ADIW rH: rL, 3 | Addiert 3 zu 16 Bit Wert ( Wortaddition mit Konstanten) |
|---|---|

(Add immediate to Word)

Wie der Befehl ADD, allerdings mit Doppelregister. Nicht für alle Controller gültig.

| SUB Register A, Register B | Subtraktion zweier Register, Ergebnis ist in Register A |
|---|---|

(Subtract Register)

Binäre Subtraktion ist eigentlich die Addition einer negativen Zahl. Daher wird aus dem Subtrahenten das 2er Komplement gebildet. Die Erklärung dazu ist allerdings etwas aufwändiger. Daher belasse ich es mit dieser Erklärung.

| SBC Register A, Register B | wie SUB, aber mit Übertrag. Ergebnis in Register A |
|---|---|

(Subtract Register with Carry) Ist in einer vorherigen Addition ein Übertrag entstanden,

(z.B. 67-125 ist < 0) dann wird der Übertrag

dieser Subtraktion abgezogen. So werden auch negative Ergebnisse gekennzeichnet.

| SBIW rH:rL, 5 | Subtrahiert 5 von 16 Bit-Wert (Wortsubtraktion mit Konstanten) |
|---|---|

(Subtract Immediate from Word)

Nicht in allen Controllern verfügbar

| AND Register A, Register B | Und-Verknüpfung zweier Register byteweise, |
|---|---|

(Logikal AND Registers) Ergebnis in Register A

Bit in Register A ist nur gesetzt, wenn das Bit in beiden Registern eine 1 hat. Ist im Ergebnis in Register A kein Bit gesetzt, wird das Zero-Bit gesetzt und ermöglicht einen bedingten Sprung (BREQ, BRNE)

| ANDI Register, 0b01010101 | Und-Verknüpfung mit Konstante byteweise |
|---|---|

(Logikal AND Register and Konstant)

Bit in Register A ist nur gesetzt, wenn das Bit in Register A und der Konstanten eine 1 hat. Ist im Ergebnis in Register A kein Bit gesetzt, wird das Zero-Bit gesetzt und ermöglicht einen bedingten Sprung (BREQ, BRNE)

| OR Register A, Register B | Oder-Verknüpfung zweier Register byteweise, |
|---|---|

(Logikal OR Registers) Ergebnis in Register A

Bit in Register A ist gesetzt, wenn das Bit in Register A oder Register B eine 1 hat. Ist im Ergebnis in Register A kein Bit gesetzt, wird das Zero-Bit gesetzt und ermöglicht einen bedingten Sprung (BREQ, BRNE)

| ORI Register, 0b01010101 | Oder-Verknüpfung mit Konstante byteweise |
|---|---|

(Logikal OR Register and Konstant)

Bit in Register A ist gesetzt, wenn das Bit in Register A oder in der Konstanten eine 1 hat. Ist im Ergebnis in Register A kein Bit gesetzt, wird das Zero-Bit gesetzt und ermöglicht einen bedingten Sprung (BREQ, BRNE)

| | |
|---|---|
| EOR Register A, Register B | Exclusiv Oder Verknüpfung zweier Register Byteweise |

(Exclusive OR Registers) Ergebnis ist in Register A

Bit in Register A ist gesetzt, wenn das Bit in Register A zu dem Bit in Register B einen unterschiedlichen Status hat. Ist im Ergebnis in Register A kein Bit gesetzt, wird das Zero-Bit gesetzt und ermöglicht einen bedingten Sprung (BREQ, BRNE)

| | |
|---|---|
| COM Register | Invertiert alle Bits, aus 1 wird 0 und aus 0 wird 1 |

(one's Complement) Bits im Register werden im Status geändert. Aus 1 wird 0 und aus 0 wird 1

| | |
|---|---|
| INC Register | Erhöht registerwert um 1 |
| DEC Register | Erniedrigt Registerwert um 1 |
| CLR Register | Setzt Register auf 0 |
| SER Register | Setzt Register auf 11111111 ( 255) |

### *2.2.7.4 Bitbefehle:*

| | |
|---|---|
| SWAP Register | Tauscht unteres und oberes Nibble |

(Swap Nibbles) Ein Nibble sind 4 Bit. Aus 00001111 wird 11110000

| | |
|---|---|
| ROL Register | Rotiert Bits nach links. Aus 00000001 wird 00000010 |

(Rotate left through Carry)

| | |
|---|---|
| ROR Register | Rotiert Bits nach rechts. Aus 10000000 wird 01000000 |

(Rotate right through Carry)

| | |
|---|---|
| LSL Register | Rotiert Bits nach links. Aus 00000001 wird 00000010 |

(logical Shift left)

| | |
|---|---|
| LSR Register | Rotiert Bits nach rechts. Aus 10000000 wird 01000000 |

(logical Shift right )

| | |
|---|---|
| SBI Port, Bit | Setzt Bit auf angegebenem Port |

(Set Bit in IO Register)

| | |
|---|---|
| CBI Port, Bit | Löscht Bit auf angegebenem Port |

(Clear Bit in IO Register)

| | |
|---|---|
| SEI | gibt den globalen Interrupt frei. |

(global Interrupt enabled)

| | |
|---|---|
| CLI | sperrt den globalen Interrupt |

(global Interrupt Disabled)

### *2.2.7.5 Sprungbefehle:*

| | |
|---|---|
| RJMP <Adressmarke> | Sprung zur Adressmarke, da geht es weiter |

(relative Jump)

| | |
|---|---|
| RCALL <Adressmarke > | Sprung zu Unterprogramm. Nach Bearbeitung weiter |

(relative Subroutine Call) mit Befehl hinter RCALL

| | |
|---|---|
| RET | Rücksprung aus Subroutine |

( Subroutine Return)

| | |
|---|---|
| RETI | Rücksprung aus Interrupt Service Routine |

(Interrupt Return)

### *2.2.7.6 Vergleiche und bedingte Sprünge:*

| CPSE Register_A, Register B | Überspringe einen Befehl wenn Gleich |
|---|---|

(Compare, Skip if Equal) Skip überspringt einen Befehl.

| CP Register_A, Register B | Vergleiche Register A mit Register B |
|---|---|

(Compare) Vergleiche zwei Register. Dabei wird eine Subtraktion durchgeführt, ohne die registerinhalte zu verändern. Lediglich die Statusbits im Statusregister, Ergebnis Positiv, Ergebnis negativ, Ergebnis 0 werden beeinflußt. Die Beschreibung der Statusbits findet man in der Hilfe bei den Befehlen.

| CPI Register, <konstante> | Vergleich Register mit Festwert |
|---|---|

(Compare Register with immediate) Vergleiche register mit einem konstanten Wert.

| BREQ <Sprungmarke> | Sprung, wenn Ergebnis aus Vergleich oder Arithmetik 0 ist |
|---|---|

(Branch if equal) Bei einem Vergleich wird A-B gerechnet. Ist Ergebnis 0

Dann ist **A= B**. **Zeroflag** wird gesetzt.

Wenn **Dec Register** 0 ist wird auch **Zeroflag** gesetzt

**Andi Register, 0b00000000** , auch **Zeroflag** gesetzt

Gleiches gilt bei Addition und Subtraktion.

| BRNE <Sprungmarke> | Sprung, wenn Ergebnis von Vergleich etc. nicht 0 ist |
|---|---|

(Branch if not equal)

| | |
|---|---|
| BRLO <Sprungmarke> | Sprung, wenn Ergebnis aus Vergleich negativ ist |

( Branch if lower)

| | |
|---|---|
| BRGE <Sprungmarke> | Sprung, wenn Ergebnis aus Vergleich gleich oder größer |

(Branch if equal or greater)

| | |
|---|---|
| SBRS Register, Bit | Überspringe einen Befehl , wenn Bit gesetzt ist |

(Skip if bit in register is set)

| | |
|---|---|
| SBRC Register, Bit | Überspringe einen Befehl , wenn Bit nichtgesetzt ist |

(Skip if bit in register is cleared)

| | |
|---|---|
| SBIS Port, Bit | Überspringe einen Befehl , wenn IO-Bit gesetzt ist |

(Skip if bit in io register is set)

| | |
|---|---|
| SBIC Port, Bit | Überspringe einen Befehl , wenn IO-Bit gelöscht ist |

(Skip if bit in io register is cleared)

Dies sind nicht alle Befehle, aber mit diesen werden wir arbeiten. Natürlich stehen weitere Informationen in der Hilfe von **AVR Studio** und im Datenblatt unter der Rubrik **Instruction Set Summary.**

Nicht alle Befehle sind noch einmal erläutert, aber die weitere Information bekommt mann im datenblatt oder in der Assembler-Hilfe im AVR-Studio.

### *2.2.8 Compilerdirektiven*

Nicht alle Befehle in einem Assemblerlisting landen auf dem Controller. Anweisungen, die mit einem Punkt beginnen sind für den Compiler bestimmt. Mit **.Def** legt er sich eine Referenztabelle zu einer Adresse an. Nein, ich habe mich nicht geirrt, auch wenn die Direktive **.Def** eine umbenennung eines Registers ist. Dazu ein kleiner Seitensprung.

Register sind Speicherzellen mit besonderen Fähigkeiten. Ob nun r16 oder Reg_A ist dem Controller völlig egal. Für ihn ist es wichtig, im Befehl bezüglich eines Registers die Adresse der Speicherzelle zu bekommen, damit die Arbeit dort erledigt wird. So ist auch die Deklaration einer Variablen nichts anderes als eine Compilerdirektive, die Speicherzelle mit einem Alias zu verwalten. Bei Aufruf in der Programmierung wird dann aus diesem Alias wieder eine Adresse für den Controllerbefehl.

Variablenmarken und Sprungadressen sind mit einem Doppelpunkt markiert. Auch hier werden Adressen über einen Alias referenziert. Mit der Direktive **.Byte n** wird der von einer Variablen belegte Speicherplatz reserviert.

Compilerdirektiven zeigen auch auf Speicherbereiche. So ist **.DSEG** eine wichtige Information, das die folgenden Adressmarken auf Adressen im dynamischen Speicherbereich sind. Der Eintrag **.CSEG** ist dem Codesegment, also dem Flashspeicher zugeordnet und **.ESEG** zeigt auf den Adressbereich EEPROM.

Die Anweisung .EQU definiert Konstanten zu einem Alias. Diese sind wie Zahlen zu betrachten. So ist zum Beispiel zwischen Variablenliste und Programm mit .EQU F_CPU =16000000 eine Konstante F_CPU mit dem Wert 16000000 definiert. Dahinter die Konstante .EQU Baud = 9600. Darauf folgen weitere Konstanten, die aber aus einer Formel befüllt werden. Das zeigt, es ist dem Compiler auch möglich, Berechnungen durchzuführen und die Ergebnisse in Form von Konstanten mit einem Alias bereit zu halten. Nun, eine Zahl wie 16000000 läßt sich im Programm nicht verwenden, zumindest nicht so einfach, weil sie nicht im Bereich von der Datenbreite von 8 Bit eines Registers und auch nicht im Bereich eines 16 Bit breiten Doppelregisters liegt. Aber der Compiler kann mit so großen Zahlen rechnen. Die Konstante Baud allerdings ließe sich mit dem Low- und Highanteil schon in ein Doppelregister packen.

Eine weitere erwähnenswerte Compilerdirektive ist **.Org**. Hier wird eine Adresse festgelegt. .Org 0000 bedeutet den Zeiger auf die Adresse 0000

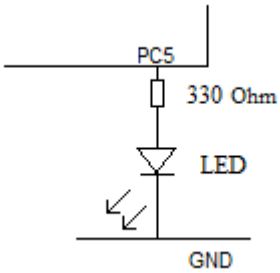
zu setzen und der Adressmarke Neustart diese Adresse zuzuweisen. Auch hier sind wie in der Folge der .Org Direkiven deutlich wird, durchaus auch wieder Aliase verwendbar. Die gesamte Interrupt Vektor Tabelle besteht aus Org-Direktiven, die mit Adressmarken gesetzt werden. Dazu ist wieder einmal die Include-Datei die Quelle. Dort sind diesen Aliasen die Adressen zugewiesen. Uns hilft es gewaltig. Klar, die Adressen in der IVT sind fest mit den Interrupts verbunden und wenn einer auslöst, adressiert der Controller die entsprechende Stelle in der IVT. Soll nun ein Interrupt an eine Service-Routine weitergeleitet werden, ja wo schreib ich dann die Weiterleitung hin? Nur die definierten Aliase mit der konstanten Adressinformation bringen Übersicht und Pflegbarkeit in das Programm. Die Alternative wäre das Maual des Controllers nach den Adressen der verschiedenen Interrupts zu durchforsten und selbst mit einer **.Org** Direktive den Aufruf eintragen. Aber was sagt schon **.Org 000C** aus? Zumindest ist nicht offensichtlich, das es die Einsprungadresse des Interrupts von Timer1 ist.

Es ist nicht immer offensichtlich, warum diese Direktiven eingesetzt werden, aber dazu mal einen kleinen Hinweis.

Angenommen, es gibt im Programm immer wieder mal einen Wert, der sich eigentlich nie ändert. Ich nenne ihn einfach Parameter1 und er hat den Wert 70. Es ist kein Problem, überall, wo dieser Parameter verwendet wird, eine 70 einzutragen. Doch, sollte sich zeigen, das 72 besser ist, muss das ganze Programm durchsucht und geändert werden. Die Lösung wäre eine Variable zu definieren und in einer Initialisierung den Wert zuzuweisen. Das belegt aber einen Speicherplatz und für den Zugriff braucht der Befehl LDS auch 2 Taktzyklen im Gegensatz zu LDI mit einem Taktzyklus. Eine Konstante Parameter1 mit .EQU Parameter1 = 70 ist mit einem Schlag überall an jeder verwendeten Stelle im Programm durch einfache Änderung der Zuweisung angepasst. Dort, wo dieser Alias mit LDI r16, Parameter1 einem Register zugewiesen wird, kommt es dem Befehl LDI r16, 70 gleich. Nachteil ist natürlich, es gibt keinen Schreibbefehl. Aber das ist bei Konstanten in der Natur, das sie nicht veränderbar sind. Zumindest nicht im laufenden Programm.

## 2.3 Unsere ersten Schritte

Das Steckbrett ist beschaltet, die Utensilien liegen bereit. Beginnen wir mit einer einfachen Aufgabe: einen Taster einlesen und eine LED blinken lassen. Dazu beschalten wir den Controllerausgang PC 5 statt mit dem Relais mit einer LED-Schaltung.




*LED an Controller*

Starten wir nun mit einem kleinem Programm. Damit der Assembler auch die richtigen Register kennt, braucht er ein externes File, in welchem die speziellen Definitionen bereits hinterlegt sind. Möglich, das dieser Aufruf bereits im Listing steht. Wenn nicht, muss das von Hand eingetragen werden.

```
.NoList
.include "m8def.inc" ; Definitionen für ATMEGA8
.List
```

Diese Zeilen sind Anweisungen für den Compiler. Der Punkt signalisiert dem Compiler seine Aufgabe.

.NoList - > es folgt die Einbindung einer Datei

.Include „m8def.inc“ - > bindet die Datei für Atmega8 ein

.List - > Einbindung abgeschlossen

Nun, damit ist schon mal einiges definiert. Wer Lust hat, der darf auch einmal einen Blick in die Datei werfen. Dazu kann man den Editor aus dem Zubehör von Windows benutzen. Die Datei befindet sich im Programmpfad unter **AVR Tools, AVRAssembler2 und Apnotes**

### 2.3.1 Auszug aus m8Def.inc Datei

```
; ***** I/O REGISTER DEFINITIONS ****************************************
; NOTE:
; Definitions marked "MEMORY MAPPED"are extended I/O ports
; and cannot be used with IN/OUT instructions
.equ SREG = 0x3f
.equ SPL = 0x3d
.equ SPH = 0x3e
.equ GICR = 0x3b
.equ GIFR = 0x3a
.equ TIMSK = 0x39
.equ TIFR = 0x38
.equ SPMCR = 0x37
.equ TWCR = 0x36
.equ MCUCR = 0x35
.equ MCUCSR = 0x34
.equ TCCR0 = 0x33
.equ TCNT0 = 0x32
.equ OSCCAL = 0x31
.equ SFIOR = 0x30
.equ TCCR1A = 0x2f
```

Auch hier wieder Compiler-Anweisungen. Der Name **SREG** wird mit der **Adresse 0x3F** verknüpft. **SREG** ist der Name vom **Statusregister**.

Der Hintergrund:

Auch ein Register ist im Controller nur eine Speicherzelle und diese werden über Adressen angesprochen. Sie haben vielleicht besondere Funktionen, aber dies spielt erst einmal keine Rolle. Man muss sich nicht alles merken, aber es ist schon sinnvoll, zu wissen, das alles, ob Speicherzelle oder Register mit einer physikalischen Adresse erreicht wird. Durch diese symbolische Namensgebung muss man sich nicht diese Adresse merken, sondern kann im Programm mit dem Namen arbeiten.

### *2.3.2 Programm- Kommentare und Gestaltung*

Wir kennen von Visual Basic Kommentarzeilen. Das sind die grün hinterlegten Texte hinter einem Hochkommata.

Auch im Assembler gibt es diese Kommentartexte und wir werden diese auch ausgiebig nutzen. Wenn in Basic der Befehl noch an der Sprache erkennbar ist, verliert sich in Assembler schnell eine programmierte Funktion in einer Unübersichtlichkeit. Nicht selten ist ein Zeilenkommentar nicht ausreichend. Zu diesem Zweck setze ich Infoboxen, wo es angebracht ist.

Assemblerlisting können und sollten wie Basicprogramme zur besseren Lesbarkeit eingerückt werden, Um Adressmarken aber nicht zu übersehen, ist es ratsam, diese immer am Zeilenanfang zu positionieren.

Ein kleines Beispiel: Eine Schleife in Basic:

```
For i = 0 to 10
        Wert(i) =i
Next i
```

Schnell ist das Gebilde durchschaut, ein Werte-Array wird mit den Zahlen 0-10 gefüllt.

Gleiches in Assembler:

```
   LDI XL,LOW(Wert) ; X-Register Anfangsadresse Wertearray Register aus R 26 und R 27
   LDI XH,HIGH(Wert)
   CLR R16              ; Register 16 auf 0 setzen
Loop:
   ST X+, R16         ; Wert von Register 16 nach der durch X adressierten Speicherzelle
                        ; schreiben, dabei X erhöhen
   INC R16              ; Register 16 erhöhen
   CPI R16,11           ; ist Wert 11 erreicht
BRLO Loop          ; BRLO = Sprung, wenn kleiner (Branch If Lower) zur Adressmarke Loop
```

Zugegeben, so unübersichtlich ist Assembler an dieser Stelle nicht, aber warten wir einmal ab.

Kommentarzeilen lassen sich auch gut für andere Zwecke einsetzen, zum Beispiel, um Programmbereiche sichtbar zu trennen, oder dem Programm die eigene Note zu verpassen.

```
;*************************************************************************
;*                        Datum 06.08.2012                               *
;*              Beispielanwendungen mit Atmega 8                         *
;* ---------------------------------------------------------------      *
;*                Ein Schulungsprojekt von xyz                           *
;*                                                                       *
;*    Die Routinen für den Atmega8 beinhalten:                           *
;*    die Kommunikation mit PC über RS 232,                              *
;* Signalerfassung von Taster und Gabellichtschranken *
;*  Ansteuerung von 7 Segmentanzeigen                                    *
;*  Ansteuerung von Relais                                               *.
;*  Copyright by xyz Germany                                             *
;*************************************************************************
```

Und es geht weiter. Natürlich ist es uns gestattet, auch eigene Namen für Register zu vergeben. Wer lieber mit bedeutsamen Namen arbeitet statt mit R 16 kann sich mit eigenen .DEF-Direktiven so richtig austoben.

```
.NoList
.include "m8def.inc"          ; Definitionen für ATMEGA8
.List

.DEF Access = R0              ; Speicherzugriffsregister
.DEF Zero = R1                ; Register für Null-Vergleich
.DEF Zehn = R2                ; Register für Vergleich mit konst. 10
.DEF Zwei = R3                ; Zeiterfassung Minute
.DEF fuenf = R4               ; Register für Vergleich mit konst. 60
.DEF hundert = R5             ; Register für Vergleich mit kont. 100
.DEF MilliSek = R6            ; Register für mSek ( schneller Zugriff)
.DEF ZehnMilli = R7           ; Register für hundertstel Sek. ( schneller Zugriff)
.Def Zehntel = R8             ; Register für  Zehntelsekunde
.DEF Ablage_A = R9            ; Zwischenspeicher A
.DEF Ablage_B = R10           ; Zwischenspeicher B
.DEF Count_L = R11            ; Counter Low 0-7
.DEF Count_H = R12            ; Counter High 8-15
.DEF Count_HL = R13           ; Counter HighLow 16-23
.DEF Count_HH = R14           ; Counter High High 24-31
. DEF SichSReg = R15          ; Sicherungsregister für Statusregister
```

Auch die Register mit erweitertem Befehlsumfang werden auf diese Weise dem eigenen Programm angepasst.

```
. DEF Reg_A = R16          ; Universalregister A
. DEF Reg_B = R17          ; Universalregister B
. DEF Reg_C = R18          ; Universalregister C
. DEF Reg_D = R19          ; Universalregister D
. DEF Reg_E = R20          ; Universalregister E
. DEF Reg_F = R21          ; Universalregister E

. DEF Work = R 22          ; allgemeines Arbeitsregister
. DEF Send_Byte = R23      ; Datenübertragung
. DEF Calc_A = R24         ; allgemeines Math. Register A
. DEF Calc_B = R25         ; allgemeines Math. Register B
; ---- Register 26- 31 sind bereits vergeben ----
; X = XL + XH = R 26 + R 27
; Y = YL + YH = R 28 + R 29
; Z = ZL + ZH = R 30 + R 31
```

Diese Compileranweisungen schreiben wir nun in ein Assemblerprojekt. Alle weiteren Versuche bauen auf einen großen Teil auf.

Nun folgt die Variablendeklaration. Wie auch bei Basic werden hier zuerst die Variablen benannt. So ganz richtig ist es nicht, denn eigentlich sage ich dem Compiler nur, das er ab Begin Datensegment sich Marken mit Adressen merken muss, an denen er später Werte abholt oder ablegt. Die Compilerdirektive .DSEG definiert diesen Bereich. Danach beginnt der Variablenspeicher. Hier habe ich ausführlich kommentiert. Der Grund, es ist einmal für die Übersicht über verwendete Bits wichtig, aber der Filter von OpenEye nutzt diese Information und erstellt dann auch die Formate der Variablen in Abhängigkeit von den Kommentaren.

Eine Variable oder Adressmarke erkennt der Compiler am Doppelpunkt, gefolgt von der Formatzuweisung **_.Byte_** und einer Zahl dahinter.

(Die Unterstriche stehen hier für Leerzeichen)

Wir beginnen mit einfachen acht Bit-Variablen und demnach steht auch **_.Byte_1**. Für die Formatauswertung im Filter ist dies allerdings nicht zu gebrauchen, daher steht das Format im Kommentar.

```
.DSEG
Trigger_In:        .Byte 1        ; Byte Debug-Variable Eingang
Trigger_Out:       .Byte 1        ; Byte Debug-Variable Ausgang
```

Die ersten beiden Variablen sind speziell für unser OpenEye-Programm. Ihre Verwendung wird zu einem späteren Zeitpunkt ausführlich erklärt. Zur Zeit haben sie aber noch keinerlei Bedeutung. Die nun Folgenden Variablen allerdings werden schon in unserem ersten Experiment erforderlich. Es soll mit einem Taster eine LED an Port C Bit 5 ein- und ausgeschaltet werden. Dazu verwenden wir zwei Variablen:

eine für die Eingänge und eine für die Ausgänge. So ist es möglich, die IO-Ebene vom Programm zu trennen und dem Prinzip **EVA** zu folgen.

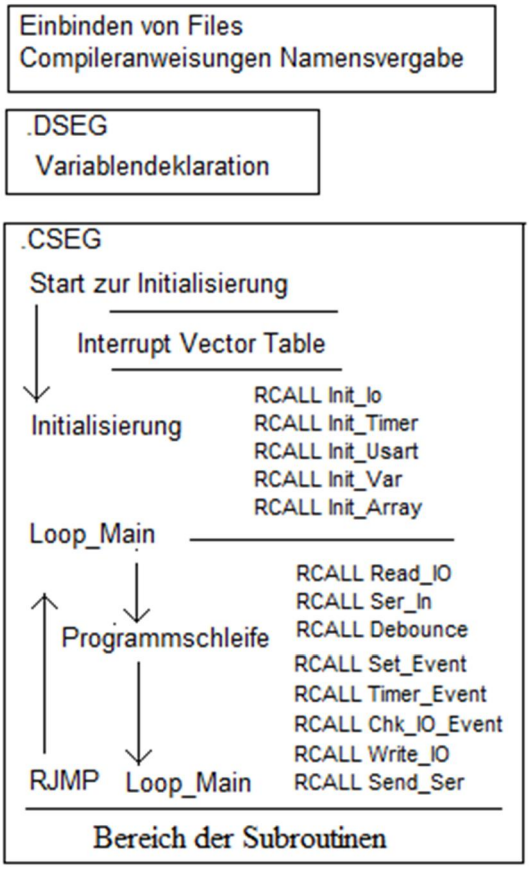
**EVA** bedeutet: **Einlesen** - **Verarbeiten** – **Ausgeben**

aber das hatten wir ja schon

```
;--------------------- Variablen IO-Ebene ---------------------
New_In: .Byte 1    ; Byte Ablage Eingänge
                   ; Bit 0 = Taster 1 Mode
                   ; Bit 1 = Taster 2 Ziffer
                   ; Bit 2 = Taster 3 auf
                    ; Bit 3 = Taster 4 ab
                    ; Bit 4 = Gabellichtschranke
                    ; Bit 5 = res
                    ; Bit 6 = res.
                    ; Bit 7 = res.

Out_Ctrl: .Byte 1  ; Byte Ablage für Ausgänge
                   ; Bit 0 = Relais 1
                   ; Bit 1 = Relais 2
                   ; Bit 2 = Relais 3
                   ; Bit 3 = Relais 4
                   ; Bit 4 = Relais 5
                   ; Bit 5 = LED 1
                   ; Bit 6 = LED 2
                   ; Bit 7 = LED 3
```

Wenn wir die Kommentare zu den einzelnen Bits eintragen, haben wir immer einen Bezug. Open_Eye kann daraus zusätzlich die Information zu den einzelnen Bits herausfiltern.

Damit unsere Experimente eine lauffähige Basis bekommt, wird auch der Programmrumpf erstellt. Dazu eine Skizze der Struktur.




*Assembler Programmaufbau*

Der Abschnitt .CSEG beinhaltet das Programm.und beginnt mit einem Sprung über die Interrupt Vector Table. Dieser Bereich wird bei den Experimenten mit den Interrupts noch näher erklärt.

```
.CSEG                    ; Speicherbereich Codesegment
.ORG 0000                ; Anfang
```

```
Reset_Point:     RJMP  Start                ; Start von hier ist Neustart
;*****************************************************************
;*          Bereich der Interrupt Vector Table            *
;*****************************************************************
          ; Sprungbefehle zu den Serviceroutinen

;*****************************************************************
;*                    Bereich Initialisierung                    *
;*****************************************************************
.ORG    INT_VECTORS_SIZE       ; Setzt die Adressmarke Start hinter die IVT

Start:
LDI      Reg_A, High(RamEnd)            ; erste Maßnahme Stack setzen
Out     SPH, Reg_A
LDI    Reg_A, Low(RamEnd)
OUT SPL, Reg_A
             ; weitere Initialisierungen
; Initialisierung IO
; Initialisierung UART
; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
; Initialisierung Variablen ( Eintragen Defaultwerte)
;*****************************************************************
;*          Programmschleife Hauptprogramm              *
;*****************************************************************
Main_Loop:
           ;  Hauptprogramm
           ; Aufruf der Subroutinen

RJMP  Main_Loop
;*****************************************************************
;*****************************************************************
;                          Subroutinen                          *
;*****************************************************************

;*****************************************************************
;*          Bereich der Interrupt Serviece Routinen      *
;*****************************************************************
```

In dieses Programmgerüst werden nun die folgenden Experimente eingebaut.

Doch bevor es an die Programmroutinen geht, ist noch die Initialisierung des Stackpointers SP-Register zu erklären.

### *2.3.3 Der Stack*

Es ist nicht erforderlich, den Stack beim Atmega8 oder Atmega16 zu setzen, doch es schadet auch nicht und deshalb führe ich ihn hier auf. Er hat eine äußerst wichtige Aufgabe im Programm und so ergibt sich die Gelegenheit, den Stack zu erklären.

Die Programmzeilen

```
LDI     Reg_A, High(RamEnd)          ; erste Maßnahme Stack setzen
Out     SPH, Reg_A
LDI     Reg_A, Low(RamEnd)
OUT SPL, Reg_A
```

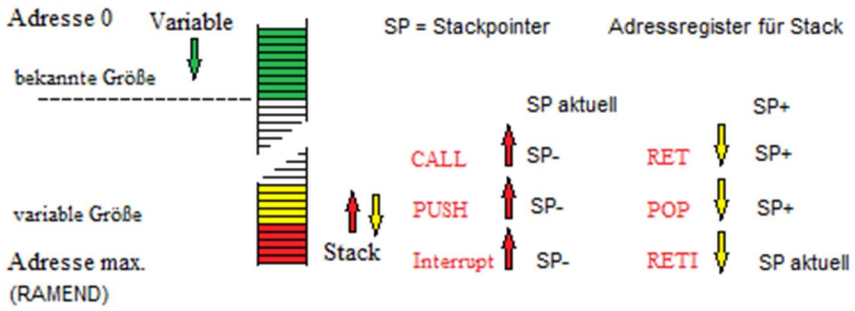
setzen das Register SP auf die letzte Adresse des Datensegmentes.

Die Ablage der Variablen geschieht im Datensegment von unten nach oben, das heißt mit aufsteigender Adresse.

```
0 Variable1
1 Variable 2
2 Variable 3
usw
```

Auch der Stack nutzt diesen Bereich, um zum Beispiel bei einem **CALL**-Befehl die Rücksprungadresse zu speichern. Dafür hat eer den Stackpointer (Register SP), welches den RAM von oben nach unten adressiert. Nach dem Aufruf des Unterprogramms soll ja in der vorherigen Befehlsfolge weiter gearbeitet werden. Auch gibt es Befehle, die ein Registerwert auf dem Stack ablegen. Das ist z.B. in Ereignisroutinen erforderlich, die über einen Interrupt aufgerufen werden. Da dies überall im Programm auftreten kann, ist ja nicht bekannt, welchen Inhalt z.B. das Statusregister hat und ob die Bits darin für den normalen Programmbetrieb wichtig sind.

Das skizzierte Schema macht die Vorgänge verdeutlich




*Prinzip Stack*

Der Variablenbereich ist durch die deklarierte Anzahl von Variablen bekannt, der Stack aber ist in ständiger Veränderung. In Assembler ist noch nachvollziehbar, welchen Platzbedarf man vorsehen muss, in Hochsprachen ist der gefürchtete Stacküberlauf keine Seltenheit, wenn darauf nicht geachtet wird. Stacküberlauf bedeutet, das der vom Stack beanspruchte Speicherplatz in den Variablenbereich wächst. Werden dann Variablen beschrieben, wird der Stackinhalt zerstört und umgekehrt durch die Eintragungen der dafür verantwortlichen Befehle werden Variableninhalte verfälscht.

Wir nehmen dies erst einmal zur Kenntnis, denn einen Stacküberlauf haben wir wirklich nicht zu befürchten, wenn wir keinen Fehler machen.

Trotzdem, einen Fehler möchte ich einmal aufzeigen:

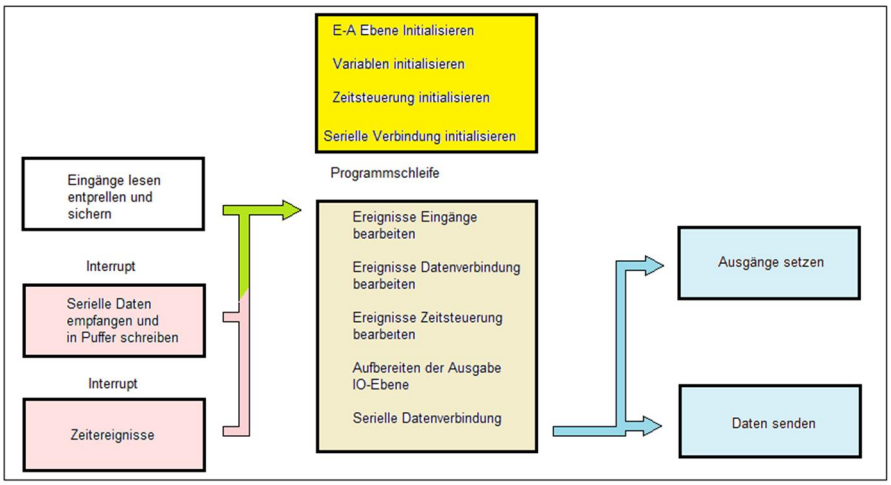
Eine Subroutine, die sich selbst aufruft und die Schachtelungstiefe nicht begrenzt.

```
Test_StackOverflow:
        RCALL Test_StackOverflow
RET
```

Ein solches Konstrukt hat binnen Sekundenbruchteilen den Stack in den Variablenbereich beschrieben. Allerdings würden wir es noch nicht einmal merken, da niemals ein RET und damit ein Sprung zum Aufruf erkannt wird.

### *2.3.4 Prinzipieller Programmaufbau*

Bei diesem experimentellen Vorgehen in der Assemblerprogrammierung ist die Verwendung einer Struktur unumgänglich. Prinzipiell folge ich EVA, Erfassen von Daten, Verarbeiten und Ausgeben.




*Prinzip EVA*

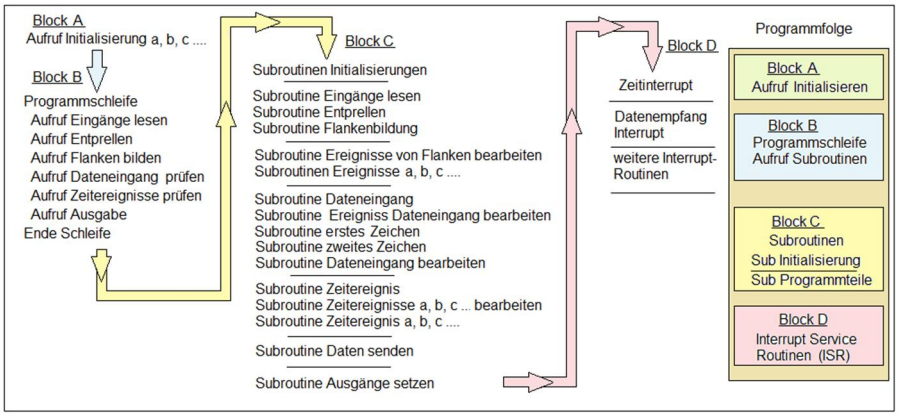
Das ist in der Regel auch richtig. Doch je weiter ein Programm fortgeschritten ist, umso unübersichtlicher wird auch die Struktur und so stellt sich oft die Frage, wohin mit dem neuen Programmblock. Es ist zwar nicht zwingend erforderlich, doch einen guten Überblick behält man durch Aufrufe von Subroutinen. So sind in der Programmschleife nur Aufrufe von Subroutinen untergebracht.

Eingänge werden gelesen und angepasst. Im zweiten Schritt entprellt und in einem dritten Schritt erfolgt die Flankenbildung, um mit den Ereignissen arbeiten zu können. Das sind die drei wichtigsten Subroutinen nur für die Erfassung von Signalen aus der Peripherie.

Zeitereignisse ausgelöst durch Timerinterrupts werden ebenfalls zur Auswertung an Subroutinen weiter geleitet.

Schließlich erfolgen Aufrufe von Subroutinen, die in jedem Programmzyklus auf Bearbeitung überprüft werden müssen. Oftmals sind Aufrufe weiterer Routinen in diesen Subroutinen eingebaut. Im Programm

sollten dann diese Programmgruppen hintereinander stehen. So sind sie schnell wieder aufzufinden und Änderungen einzubringen. Auch dafür habe ich ein Schema in einer Skizze dargestellt. In der eigenen Doku darf dann auch gezielt jeder Aufruf und jede Subroutine mit Namen in der Reihenfolge stehen.




*Ablauf Programm*

Mit einer solchen Liste hat man schnell die Stelle gefunden, wo eine neue Subroutine eingebaut oder eine vorhandene für eine Änderung zu finden ist.

## 2.4 Initialisierung IO-Port

Am Anfang des Programms stehen immer die Initialisierungen. Wir beginnen mit der IO-Ebene. Eine Subroutine schließt diese Funktion in einen leicht überschaubaren Block und läßt das Hauptprogramm übersichtlich.

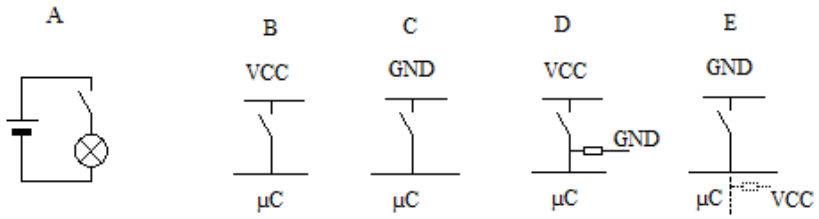
```
Init_IO:                          ; Ein-Ausgaben initialisieren
;*******************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf        *
;*    Dabei werden nur die benötigten Portbits parametriert,       *
;*    die anderen in ihrer Vorgabe belassen.                       *
;*******************************************************************************
   IN     Reg_A, DDRD              ; Port D Bits 0-1 = Serielle Schnittstelle
                                   ; Bit 2 Eingang Taster 1
                                   ; Bit 3 Eingang Taster 2
                                   ; Bit 4 Eingang Taster 3
                                   ; Bit 5 Eingang Taster 4
                                   ; Bit 6 Eingang Gabellichtschranken
                                   ; Bit 7 Ausgang Relais
   ANDI   Reg_A, 0b00000011          ; Bit 0 und 1 nicht verändern
   ORI    Reg_A, 0b10000000          ; Bit 7 auf Ausgang
   OUT    DDRD, Reg_A                ; Data Direction Register Port D beschreiben
   IN     Reg_A, PortD
   ORI    Reg_A,0b01111100  ; Port D Pull-Up Widerstände für die Eingänge einschalten
; Die anderen Portbits bleiben unverändert
   OUT    PortD, Reg_A               ; Port mit Inhalt von Register A beschreiben
   IN     Reg_A, DDRC                ; Data Direction Register lesen
   ORI    Reg_A,0b00111110           ; Port C Bits 1-5 auf Ausgang schalten
   ANDI   Reg_A, 0b11111110          ; Port C Bits ist Eingang
; Die anderen Bits bleiben unverändert
   OUT    DDRC, Reg_A                ; Data Direction Register Port C beschreiben
   In     Reg_A, PortC
   ORI    Reg_A, 0b00000001          ; Pull_Up Widerstand PC0 einschalten
   Out    PortC, Reg_A
   In     Reg_A, DDRB                ; Data Direction Register Port B lesen
   ORI    Reg_A,0b00111111           ; Portbit 0-5 Ausgang
   Out    DDRB, Reg_A                ; Data Direction Register B beschreiben
RET
;*******************************************************************************
```

Der Aufruf erfolgt im Initialisierungsteill mit RCALL Init_IO. An dieser Stelle sollte der Kommentar **Pull_UpWiderstände einschalten** auffallen.

### *2.4.1 Pull_Up-Widerstand*

Pull-Up Widerstand und Pull-Down Widerstand. Was hat es damit auf sich?

Auch dazu eine kleine Skizze.




*PullUp und PullDown*

Unter **A** ist ein einfacher Stromkreis und dort ist auch klar, das die Lampe nur leuchtet, wenn der Schalter geschlossen ist.

Bei den Schaltkreisen **B, C, D** und **E** schalte ich den Eingang eines elektronischen Bauteiles. Nun ist es für diesen Baustein, in diesem Fall ein Mikrocontroller schon wichtig, das die Signale potentialgebunden **0** (GND) oder **1** (VCC) sind. Bei **B** und **C** ist dies jedoch nur bei geschlossenem Schalter gegeben. Ist der Schalter offen, so ist das Potential unklar. Da reicht es bei offenem Schalter schon, wenn man mit seiner Hand in die Nähe der Schaltung gelangt oder das Licht einschaltet, das ein Eingang plötzlich seinen Status ändert.

Das ist natürlich nicht gewollt, das auf Störimpulse der Eingang undefiniert schaltet. Daher muß der Controllereingang an ein Potential gebunden werden. Dies geschieht mit einem hohen Widerstand. (zwischen 10000 – 50000 Ohm).

Legt man die Taster an **VCC**, so muß ein **Pull-Down-Widerstand** nach **GND** geschaltet werden. Er liefert das untere Potential (GND) bei unbetätigtem Taster.

Werden die Taster an **GND** betrieben, so kann ein interner Pull-Up Widerstand per Software zugeschaltet werden. Man spart in diesem Fall einen externen **Pull-Up-Widerstand** und bekommt das hohe Potential (VCC) an den Port-Pin geliefert.

Wer das Pollin-Board benutzt und in den Schaltplan schaut, stellt fest, dieser Bausatz hat die Taster nach VCC. Bei Verwendung der Taster auf dem Board dürfen die **Pull-Up-Widerstände nicht** zugeschaltet und der Befehl **COM nicht** programmiert werden. Hier sind externe PullDown Widerstände erforderlich.

### *2.4.2 Portbit lesen*

Treu nach dem Motto **E** wie **einlesen**, beginnen wir wieder mit einer kleinen Subroutine.

```
;---------- Lesen aller Eingänge und Signalanpassung ------------
;*************************************************
;*                    Port D                     *
;*      Bit 0 und 1   serielle Verbindung        *
;*      Bit 2 bis 5   Eingänge Taster            *
;*      Bit 6         Eingang Gabellichtschranke *
;*      Bit 7         Ausgang Anzeige            *
;*************************************************
Read_IO:                    ; Eingänge einlesen
    In Reg_A, PInD              ; Port B lesen
    COM Reg_A                   ; Bits drehen
    ANDI Reg_A, 0b01111100      ; Nur Eingänge übernehmen
    LSR Reg_A                   ; nach rechts schieben (00111110)
    LSR Reg_A                   ; nach rechts schieben (00011111)
    STS New_In, Reg_A
RET
```

Diese kleine Routine ist leicht zu überschauen. Es wird der gesamte Port in das Register A übernommen. Da die Taster an GND, also logisch 0 hängen, wird auch ein betätigter Taster ein 0 Signal und wenn nicht betätigt, bedingt durch den internen PullUp -Widerstand eine logische 1 liefern. Man kann zwar damit arbeiten, doch garantiert irrt man und assoziiert mit einem gedrückten Taster auch eine logische 1. Klar, wenn ich einen Taster an VCC anschließe, dann stimmt es ja auch, aber dann muß ich auch einen externen Pull-Down-Widerstand beschalten.

Welchen Weg man auch wählen mag, ist hier völlig unwichtig. In der Variablen New_In steht ein geschlossener Kontakt immer mit 1. So kann das Programm danach aufgebaut werden.

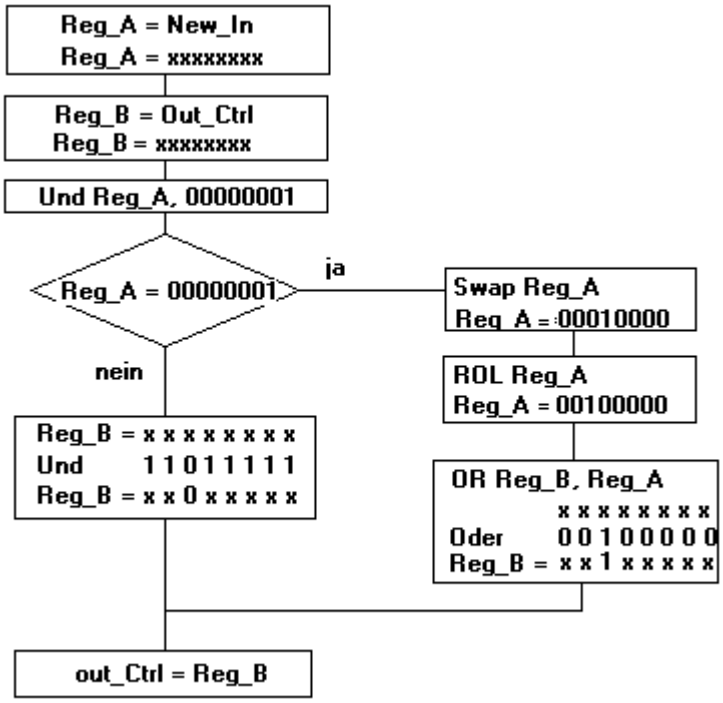
## 2.5 Die Information verarbeiten

Im Experiment wird nur **Taster 1** abgefragt. Ist also der Kontakt geschlossen, liefert die Variable auf Bit 0 eine 1. Diese soll für die Ausgabe nun auf die Variable Out_Ctrl Bit 5 umgelegt werden, damit die LED in der Ausgabe angesteuert werden kann. Dazu schreiben wir einfach eine kleine Subroutine:

```
Set_LED_Bit:
   LDS    Reg_A, New_In
   ANDI  Reg_A, 0b00000001                ; Nur Bit 1 prüfen (Ergebnis ist nicht „0“)
   BREQ  Set_LED1            ; Bit= 0, dann  auch Ergebnis 0, dann Sprung nach Marke Set_LED1
   LDS      Reg_B, Out_Ctrl                  ; Nun das Byte für den Ausgang holen
   ANDI    Reg_B, 0b11011111                ; Mit einer Und-Anweisung das Bit 5 löschen
   RJMP    End_Set_LED_Bit                    ; Sprung zum Ende der Sub-Routine
Set_LED1:                                   ; Einstieg zum Setzen des Bits
   SWAP    Reg_A    ; Befehl tauscht die unteren 4 Bits (Nibble) mit den oberen 4 Bits (Nibble)
   ROL      Reg_A       ; Schiebt den Inhalt von Register A noch einmal nach links
   LDS      Reg_B, Out_Ctrl                  ; Nun das Byte für den Ausgang holen
   OR       Reg_B, Reg_A                       ; und das 5. Bit in Register B setzen
End_Set_LED_Bit:                            ; Markiert das Ende der Subroutine
    STS      Out_Ctrl, Reg_B     ; nun noch das Register B in die Variable Out_Ctrl schreiben
RET                                         ; Und Rücksprung zum Aufruf
```

Wie bei Visual Basic werden auch hier die Zeilen besimmter Abschnitte etwas eingerückt. Das Ergebnis ist ein Bit in einer Variablen, welche den ganzen restlichen Programmdurchlauf nicht mehr verändert werden sollte. Es ist aber kein Problem, dieses Bit weiterhin in die Programmbearbeitung einzubinden und beliebig oft abzufragen. Für ein klares Programm ist es sinnvoll, ein Bit nicht mehrfach im Programm zu verändern. Die Zuweisung an mehreren Stellen erschwert eine Fehlersuche ungemein.

Damit diese Funktion deutlicher wird, eine kleine Skizze




*PAP Bit umschaufeln*

In unserem kleinen Programm ist ein bedingter Sprung hinterlegt: **BREQ**, was einen Sprung auslöst, wenn zwei Werte gleich sind. Der Wortlaut: **Branch if equal**. Wie aber kann eine Und-Verknüpfung im Ergebnis gleich sein? Hier ist es etwas irreführend, aber lesen wir doch einmal, wie ein Vergleich ausgeführt wird. Es wird ein Ergebnis einer Pseudo-Subtraktion ausgewertet und in Statusbits hinterlegt. Bei Gleichheit wäre das Ergebnis 0 und somit auch das Zero-Bit gesetzt. Dieses Bit wird aber auch von binären Verknüpfungen beeinflusst und wenn bei einem Und-Befehl ein Ergebnis =0 ist, dann ist auch dieses Bit gesetzt. Daher hilft es, sich den Befehl auch mit **Branch if Zero** zu merken.

Werfen wir einen Blick in das Datenblatt in die Rubrik **Instruction Set Summary**, suchen den Befehl **BREQ** und in der Beschreibung finden wir, wann er ausgelöst wird.

Mit dieser Subroutine und der bereits beschriebenen Routine Taster_Status haben wir schon **EV** zusammen. Fehlt nur noch das **A**, um **EVA** komplett zu machen.

### *2.5.1 Einen Ausgang zuweisen*

Das A steht für **Ausgabe** und die werden wir nun programmieren.

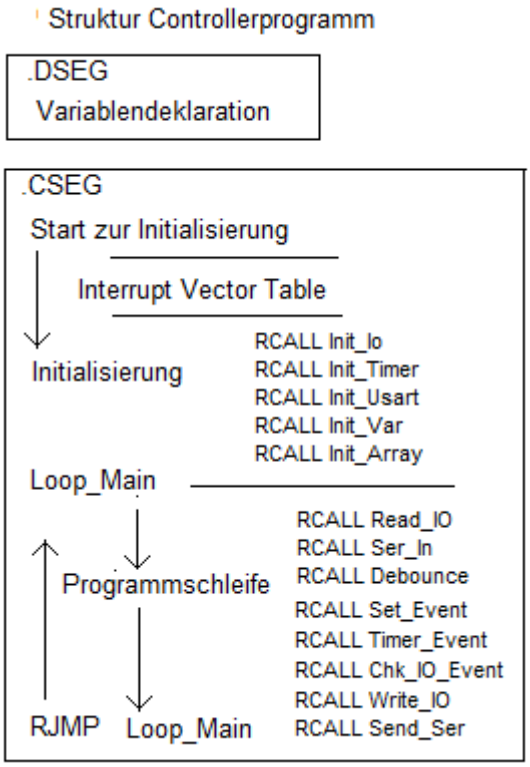
Es gilt zuerst einmal, den Port-Pin PC 5 zu setzen. Dafür haben wir in der Verarbeitung bereits das Bit 5 in der Variablen Out_Ctrl gesetzt. Nun gilt es, dieses dem Port zuzuweisen. Wieder schreiben wir eine kleine Sub-Routine, die diese Aufgabe erfüllt.

```
;*********************************************
;*      Ausgabe an die IO-Ebene        *
;*      Port C Bit 5 = Relais / LED     *
;*********************************************
Write_IO:
   IN Reg_B, PinC                  ; gesamten Port C lesen
   LDS Reg_A, Out_Ctrl             ; Variable für Ausgang holen
   ANDI Reg_A, 0b00100000          ; Bit 5 Prüfen
   BREQ Rel_Aus                    ; Wenn Bit =0 dann Ausgang aus
   ORI Reg_B, 0b00100000           ; Ausgang zuschalten, Nur Bit 5 setzen
   RJMP Set_Ausgang
Rel_Aus:
   ANDI Reg_B,0b11011111           ; Ausgang abschalten, nur Bit 5 löschen
Set_Ausgang:
   OUT PortC, Reg_A                ; Port mit Reg_C beschreiben
RET
```

Nun können wir das Programm zusammensetzen.

## 2.6 Das erste Programm

Um die Programmstruktur noch einmal in das Gedächtnis zu rufen, eine kleine Skizze. Sie soll der Leitfaden sein, die in den folgenden Kapiteln angesprochenen Subroutinen richtig in das Programm zu ordnen.




*Struktur Controllerprogramm*

Zuerst die Variablendeklaration. In diesem Bereich müssen immer wieder Variablen nachgetragen werden. Die Dokumentation darf auch schon Aufschluss über Format und Funktion Auskunft geben. So ist es ein Leichtes, diese Variablen anschließend in Open_Eye in den Filter zu koperen und mit geringen Anpassungen eine Visualisierung zu bekommen.

```
;***********************************
;*                Programmtest 1              *
;***********************************
.NoList
.Include "m8def.inc"
.List
.DEF ….                        ;  Namen der Register
.
.DSEG
Trigger_In: .Byte 1            ; Byte Debug-Variable Eingang
Trigger_Out: .Byte 1           ; Byte Debug-Variable Ausgang
;---------------------- Variablen IO-Ebene ----------------------
New_In: .Byte 1                ; Byte Ablage Eingänge
                               ; Bit 0 = Taster 1 Mode
                               ; Bit 1 = Taster 2 Ziffer
                               ; Bit 2 = Taster 3 auf
                               ; Bit 3 = Taster 4 ab
                               ; Bit 4 = Gabellichtschranke
                               ; Bit 5 = res
                               ; Bit 6 = res.
                               ; Bit 7 = res.

Out_Ctrl: .Byte 1              ; Byte Ablage für Ausgänge
                               ; Bit 0 = res
                               ; Bit 1 = res
                               ; Bit 2 = res.
                               ; Bit 3 = res.
                               ; Bit 4 = res.
                               ; Bit 5 = LED 1
                               ; Bit 6 = LED 2
                               ; Bit 7 = LED 3
```

Nun beginnt das Codesegment mit dem Programmteil. Die Anweisung ORG ist eine Compilerdirektive und weist den Compiler an den Start des Programms auf die Adresse 0000 im Flashspeicher zu setzen. Dort erfolgt zuerst ein Sprung über die IVT (Interrupt Vector Table) Den Begriff erkläre ich noch und was dahinter steckt.

Programmeintritt und Programmschleife

```
.CSEG
.ORG 0000                          ; Setzt die Anfangsadresse auf 0
Reset: RJMP   Start                ; Sprung über die IVT
;*****************************************************
;* Bereich der Interrupt Vector Table *
;*****************************************************
.ORG    INT_VECTORS_SIZE       ; Setzt die Adressmarke Start hinter die IVT
Start :
;*****************************************************
;*          Bereich Initialisierung, beginnend mit dem Stack          *
;*          Anschließend initialisierung der Hardware                 *
;*            ( IO-Ebene)                                             *
;*****************************************************
   LDI       Reg_A ,high(RAMEND)              ; Stack Pointer setzen
   OUT       SPH, Reg_A                       ; RAMEND ist in m8def.inc (s.o.) festgelegt
   LDI       Reg_A, low(RAMEND)
   OUT       SPL, Reg_A
   RCALL   INIT_IO                    ; erst jetzt ist ein CALL erlaubt, da der Stack nun zugewiesen ist
;SEI                              ; Masterfreigabe Interrupts
```

```
;*****************************************************
;*                    Schleife Hauptprogramm                    *
;*****************************************************
Loop:
    RCALL Read_IO                 ; Eingänge lesen „E“
    RCALL Set_LED_Bit             ; Bearbeiten „V“
    RCALL Write_IO                ; und ausgeben „A“
RJMP Loop
```

Aus dem Programm werden nun die bereits erstellten Subroutinen aufgerufen.

### *2.6.1 Programm LED mit Taster*

```
;*********************************************************************
;*                        Datum 06.08.2012                           *
;*              Beispielanwendungen mit Atmega 8                     *
;* ------------------------------------------------------------------ *
;*        Ein Experimente mit PC und Mikrocontroller                 *
;*                                                                   *
;*   Die Routinen für den Atmega8 beinhalten:                        *
;*      Standart  IO                                                 *
;*      Signalerfassung von Taster                                   *
;*      Ansteuerung von LED                                          *.
;*  Copyright by Martin Vogel   Germany                              *
;*********************************************************************
.NoList
.include   "m8def.inc"      ; Definitionen für ATMEGA8
.List

.DEF Access      = R0      ; Speicherzugriffsregister
.DEF Zero        = R1      ; Register für Null-Vergleich
.DEF Zehn        = R2      ; Register für Vergleich mit konst. 10
.DEF Zwei        = R3      ; Zeiterfassung Minute
.DEF fuenf       = R4      ; Register für Vergleich mit konst. 60
.DEF hundert     = R5      ; Register für Vergleich mit kont. 100
.DEF MilliSek    = R6      ; Register für mSek ( schneller Zugriff)
.DEF ZehnMilli   = R7      ; Register für hundertstel Sek. ( schneller Zugriff)
.Def Zehntel     = R8      ; Register für  Zehntelsekunde
.DEF Ablage_A    = R9      ; Zwischenspeicher A
.DEF Ablage_B    = R10     ; Zwischenspeicher B
.DEF Count_L     = R11     ; Counter Low 0-7
.DEF Count_H     = R12     ; Counter High 8-15
.DEF Count_HL    = R13     ; Counter HighLow 16-23
.DEF Count_HH    = R14     ; Counter High High 24-31
.Def SichSREG    = R15     ; Zwischenspeicher f. Statusregister
.DEF Reg_A       = R16     ; Universalregister A
.DEF Reg_B       = R17     ; Universalregister B
.DEF Reg_C       = R18     ; Universalregister C
.DEF Reg_D       = R19     ; Universalregister D
.DEF Reg_E       = R20     ; Universalregister E
.DEF Reg_F       = R21     ; Universalregister E
.DEF Work        = R22     ; Arbeitsregister
.DEF Send_Byte   = R23     ; Datenübertragung
.DEF Calc_A      = R24     ; Math.-Register A
```

```
.DEF Calc_B        = R25      ; Math. Register B
; ---- Register 26- 31 sind bereits vergeben ----
; X = XL + XH = R 26 + R 27
; Y = YL + YH = R 28 + R 29
; Z = ZL + ZH = R 30 + R 31
.DSEG
Trigger_In:        .Byte 1      ; Byte Debug-Variable Eingang
Trigger_Out:       .Byte 1      ; Byte Debug-Variable Ausgang

;--------------------- Variablen IO-Ebene ----------------------
New_In:            .Byte 1  ; Byte Ablage Eingänge
                            ; Bit 0 = Taster 1 Mode
                            ; Bit 1 = Taster 2 Ziffer
                            ; Bit 2 = Taster 3 auf
                            ; Bit 3 = Taster 4 ab
                            ; Bit 4 = Gabellichtschranke
                            ; Bit 5 = res
                            ; Bit 6 = res.
                            ; Bit 7 = res.

Out_Ctrl:          .Byte 1  ; Byte Ablage für Ausgänge
                           ; Bit 0 = Relais 1
                           ; Bit 1 = Relais 2
                           ; Bit 2 = Relais 3
                           ; Bit 3 = Relais 4
                           ; Bit 4 = Relais 5
                           ; Bit 5 = LED 1
                           ; Bit 6 = LED 2
                           ; Bit 7 = LED 3

.CSEG
.ORG 0000
Reset_Point:   RJMP  Start    ; Start von hier ist Neustart
;*****************************************************************
;*          Bereich der Interrupt Vector Table              *
;*****************************************************************
          ; Sprungbefehle zu den Serviceroutinen

;*****************************************************************
;*                  Bereich Initialisierung                     *
;*****************************************************************
.ORG    INT_VECTORS_SIZE       ; Setzt die Adressmarke Start hinter die IVT
Start:
  LDI     Reg_A, High(RamEnd)   ; erste Maßnahme Stack setzen
  Out     SPH, Reg_A
```

```
 LDI     Reg_A, Low(RamEnd)
 OUT     SPL, Reg_A
                        ; weitere Initialisierungen
 RCALL   Init_IO            ; Initialisierung IO
; Initialisierung UART
; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
; Initialisierung Variablen ( Eintragen Defaultwerte)
; SEI                              ; Masterfreigabe von Interrupts
;*****************************************************************
;*            Programmschleife Hauptprogramm          *
;*****************************************************************
Main_Loop:
           ;  Hauptprogramm
           ; Aufruf der Subroutinen
 RCALL    Read_IO
 RCALL    Set_LED_Bit
 RCALL    Write_IO
RJMP  Main_Loop
;-----------------------------------------------------------------
;*****************************************************************
;                          Subroutinen                          *
;*****************************************************************

Init_IO:                   ; Ein-Ausgaben initialisieren
;*******************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf      *
;*   Dabei werden nur die benötigten Portbits parametriert,      *
;*   die anderen in ihrer Vorgabe belassen.                      *
;*******************************************************************************
  IN     Reg_A, DDRD              ; Port D Bits 0-1 = Serielle Schnittstelle
                                   ; Bit 2 Eingang Taster 1
                                   ; Bit 3 Eingang Taster 2
                                   ; Bit 4 Eingang Taster 3
                                   ; Bit 5 Eingang Taster 4
                                   ; Bit 6 Eingang Gabellichtschranken
                                   ; Bit 7 Ausgang Relais
  ANDI   Reg_A, 0b00000011       ; Bit 0 und 1 nicht verändern
  ORI    Reg_A, 0b10000000       ; Bit 7 auf Ausgang
  OUT    DDRD, Reg_A           ; Data Direction Register PortD beschreiben
  IN     Reg_A, PortD
  ORI    Reg_A,0b01111100        ; PortD Pull-Up für die Eingänge einschalten
                          ; Die anderen Portbits bleiben unverändert
```

```
  OUT    PortD, Reg_A            ; Port mit Inhalt von Register A beschreiben
  IN     Reg_A, DDRC             ; Data Direction Register lesen
  ORI    Reg_A,0b00111110        ; Port C Bits 1-5 auf Ausgang schalten
  ANDI   Reg_A, 0b11111110       ; Port C Bits ist Eingang
                                 ; Die anderen Bits bleiben unverändert
  OUT    DDRC, Reg_A             ; Data Direction Register Port C beschreiben
  In     Reg_A, PortC
  ORI    Reg_A, 0b00000001       ; Pull_Up Widerstand PC0 einschalten
  Out    PortC, Reg_A
  In     Reg_A, DDRB             ; Data Direction Register Port B lesen
  ORI    Reg_A,0b00111111        ; Portbit 0-5 Ausgang
  Out    DDRB, Reg_A             ; Data Direction Register B beschreiben
RET
;-----------------------------------------------------------------------------

;********** Lesen aller Eingänge und Signalanpassung **************
;*     Port D                                                      *
;*       Bit 0 und 1    serielle Verbindung                        *
;*       Bit 2 bis 5    Eingänge Taster                            *
;*       Bit 6          Eingang Gabellichtschranke                 *
;*       Bit 7          Ausgang Anzeige                            *
;*****************************************************************************
Read_IO:                       ; Eingänge einlesen
  In     Reg_A, PInD             ; Port B lesen
  COM    Reg_A                   ; Bits drehen
  ANDI   Reg_A, 0b01111100       ; Nur Eingänge übernehmen
  LSR    Reg_A                   ; nach rechts schieben (00111110)
  LSR    Reg_A                   ; nach rechts schieben (00011111)
  STS    New_In, Reg_A
RET


;********** Eingabe bearbeiten und Ausgabe vorbereiten ************
;*     Variable  New_In enthält Status der Eingänge                *
;*                                                                 *
;*     Variable Out_Ctrl enthält Ergebnis aus der Bearbeitung      *
;*     für die Ausgabe                                             *
;*****************************************************************************
Set_LED_Bit:                   ; LED einschalten
  LDS    Reg_A, New_In
  ANDI   Reg_A, 0b00000001       ; Nur Bit 1 prüfen (Ergebnis ist nicht „0“)
  BRNE   Set_LED1                ; Bit=1, dann  Sprung nach Marke Set_LED1
  LDS    Reg_B, Out_Ctrl         ; Nun das Byte für den Ausgang holen
  ANDI   Reg_B, 0b11011111       ; Mit einer Und-Anweisung das Bit 5 löschen
  RJMP   End_Set_LED_Bit         ; Sprung zum Ende der Sub-Routine
```

```
Set_LED1:                          ; Einstieg zum Setzen des Bits
  SWAP  Reg_A                  ; tauscht die unteren 4 Bits mit den oberen 4 Bits (Nibble)
  ROL   Reg_A                 ; Schiebt den Inhalt von Register A noch einmal nach links
  LDS   Reg_B, Out_Ctrl        ; Nun das Byte für den Ausgang holen
  OR    Reg_B, Reg_A           ; und das 5. Bit in Register B setzen
End_Set_LED_Bit:               ; Markiert das Ende der Subroutine
  STS   Out_Ctrl, Reg_B        ; nun noch das Register B in die Variable Out_Ctrl schreiben
RET                            ; Und Rücksprung zum Aufruf
;-----------------------------------------------------------------

;*******************************************************************
;*                Ausgabe an die IO-Ebene                          *
;*              Port C Bit 5 = Relais / LED                        *
;*******************************************************************
Write_IO:
  IN      Reg_B, PinC                ; gesamten Port C lesen
  LDS     Reg_A, Out_Ctrl            ; Variable für Ausgang holen
  ANDI    Reg_A, 0b00100000           ; Bit 5 Prüfen
  BREQ Rel_Aus                       ; Wenn Bit =0 dann Ausgang aus
  ORI     Reg_B, 0b00100000          ; Ausgang zuschalten, Nur Bit 5 setzen
  RJMP Set_Ausgang
Rel_Aus:
  ANDI    Reg_B,0b11011111            ; Ausgang abschalten, nur Bit 5 löschen
Set_Ausgang:
  OUT   PortC, Reg_A                 ; Port mit Reg_C beschreiben
RET
;-----------------------------------------------------------------

;*******************************************************************
;*            Bereich der Interrupt Serviece Routinen              *
;*******************************************************************
```

### *2.6.2 Blinkende LED*

Für eine einfache Kontrolle sollte eine Anzeige her, um in unserem Programm auch einen Fehler entdecken zu können. Wir werden eine LED in einer kleinen Routine blinken lassen. Läuft das Programm nicht wie beabsichtigt, kann dieser **Blinker** an der Programmstelle aufgerufen werden. So erkennen wir, ob ein Programmteil überhaupt durchlaufen wird. Dafür benötigen wir zwei weitere Variablen, um den Wechsel zu verlangsamen. Ist er zu schnell, ist die LED scheinbar immer an.

Nehmen wir für einen 16 Bit Zähler die Variablen **Counter_0** und **Counter_1**

Fügen wir der Variablendeklaration hinzu:

```
Counter_0: .Byte 1                    ; 16 Bit-Zähler LowByte
Counter_1: .Byte 1                    ; 16 Bit-Zähler HighByte
```

Die Routine ist schnell erstellt.

```
Blinker:
      LDS Reg_A,Counter_0             ;inneren Schleifenzähler
      INC Reg_A
      STS Counter_0, Reg_A
      CPI Reg_A, 250                  ; bis 100 zählen, dann von vorn
      BRLO End_Blinker
         CLR Reg_A
         STS Counter_0, Reg_A         ; mit 100 multiplizieren
         LDS Reg_A, Counter_1
         INC Reg_A
         STS Counter_1, Reg_A
         CPI Reg_A, 250
     BRLO End_Blinker
         CLR Reg_A                    ; bei 10000 wieder von vorn anfangen
         STS Counter_1, Reg_A
         LDS Reg_A, Out_Ctrl
         LDI Reg_B, 0b00100000
         EOR Reg_A, Reg_B
         STS Out_Ctrl, Reg_A
End_Blinker:
RET
```

Auf Bit 5 von Out_Ctrl haben wir den Ausgangspin PC 5 rangiert. Dort schließen wir eine LED an.

Zu diesem Zeitpunkt sollte der µC noch mit dem internen Takt von 1 MHz laufen, so dass sich Blinken der LED bei ca. 1 Hz ergibt. Den **Blinker** prüfen wir durch einen Aufruf aus der Programmschleife. Danach wird nur der Aufruf entfernt.

```
;*****************************************************
;*                    Schleife Hauptprogramm                    *
;*****************************************************
Loop:
    RCALL Read_IO              ; Eingänge lesen „E“
    ;RCALL Set_LED_Bit         ; Bearbeiten „V“
    RCALL Blinker              ; Blinkerbit bilden
    RCALL Write_IO             ; und ausgeben „A“
RJMP Loop
```

Der Programmteil Blinker muss funktionieren, denn er wird später zu Testzwecken im Programm an problematischen Stellen aufgerufen, um den Durchlauf zu kontrollieren. Wenn später das Ergebnis unserer Programmbearbeitung nicht das gewünschte Ergebnis liefert, kann durch Einsetzen des Befehles **RCALL Blinker** geprüft werden, ob dieser Programmabschnitt überhaupt durchlaufen wird.

### 2.6.3 Programm Blinker

```
;*********************************************************************
;*                    Datum 06.08.2012                              *
;*            Beispielanwendungen mit Atmega 8                     *
;* ------------------------------------------------------------- *
;*      Ein Experimente mit PC und Mikrocontroller                 *
;*                                                                 *
;*  Die Routinen für den Atmega8 beinhalten:                       *
;*     Standart  IO                                                *
;*     Signalerfassung von Taster                                  *
;*    Ansteuerung von LED                                          *.
;*  Copyright by Martin Vogel   Germany                            *
;*********************************************************************
.NoList
.include   "m8def.inc"      ; Definitionen für ATMEGA8
.List

.DEF Access      = R0      ; Speicherzugriffsregister
.DEF Zero        = R1      ; Register für Null-Vergleich
.DEF Zehn        = R2      ; Register für Vergleich mit konst. 10
.DEF Zwei        = R3      ; Zeiterfassung Minute
.DEF fuenf       = R4      ; Register für Vergleich mit konst. 60
.DEF hundert     = R5      ; Register für Vergleich mit kont. 100
.DEF MilliSek    = R6      ; Register für mSek ( schneller Zugriff)
.DEF ZehnMilli  = R7      ; Register für hundertstel Sek. ( schneller Zugriff)
.Def Zehntel     = R8      ; Register für  Zehntelsekunde
.DEF Ablage_A   = R9      ; Zwischenspeicher A
.DEF Ablage_B   = R10     ; Zwischenspeicher B
.DEF Count_L     = R11     ; Counter Low 0-7
.DEF Count_H     = R12     ; Counter High 8-15
.DEF Count_HL  = R13     ; Counter HighLow 16-23
.DEF Count_HH  = R14     ; Counter High High 24-31
.Def SichSREG    = R15     ; Zwischenspeicher f. Statusregister
.DEF Reg_A       = R16     ; Universalregister A
.DEF Reg_B       = R17     ; Universalregister B
.DEF Reg_C       = R18     ; Universalregister C
.DEF Reg_D       = R19     ; Universalregister D
.DEF Reg_E       = R20     ; Universalregister E
.DEF Reg_F       = R21     ; Universalregister E
.DEF Work        = R22     ; Arbeitsregister
.DEF Send_Byte  = R23     ; Datenübertragung
.DEF Calc_A      = R24     ; Math.-Register A
.DEF Calc_B      = R25     ; Math. Register B
; ---- Register 26- 31 sind bereits vergeben ----
```

```
; X = XL + XH = R 26 + R 27
; Y = YL + YH = R 28 + R 29
; Z = ZL + ZH = R 30 + R 31
.DSEG
Trigger_In:        .Byte 1      ; Byte Debug-Variable Eingang
Trigger_Out:       .Byte 1      ; Byte Debug-Variable Ausgang

;--------------------- Variablen IO-Ebene ---------------------
New_In:            .Byte 1  ; Byte Ablage Eingänge
                            ; Bit 0 = Taster 1 Mode
                            ; Bit 1 = Taster 2 Ziffer
                            ; Bit 2 = Taster 3 auf
                            ; Bit 3 = Taster 4 ab
                            ; Bit 4 = Gabellichtschranke
                            ; Bit 5 = res
                            ; Bit 6 = res.
                            ; Bit 7 = res.

Out_Ctrl:          .Byte 1  ; Byte Ablage für Ausgänge
                            ; Bit 0 = Relais 1
                            ; Bit 1 = Relais 2
                            ; Bit 2 = Relais 3
                            ; Bit 3 = Relais 4
                            ; Bit 4 = Relais 5
                            ; Bit 5 = LED 1
                            ; Bit 6 = LED 2
                            ; Bit 7 = LED 3
Counter_0:         .Byte 1      ; 16 Bit-Zähler LowByte
Counter_1:         .Byte 1      ; 16 Bit-Zähler HighByte

.CSEG
.ORG 0000
Reset_Point:   RJMP  Start    ; Start von hier ist Neustart
;*****************************************************************
;*          Bereich der Interrupt Vector Table              *
;*****************************************************************
          ; Sprungbefehle zu den Serviceroutinen

;*****************************************************************
;*                  Bereich Initialisierung                     *
;*****************************************************************
.ORG    INT_VECTORS_SIZE       ; Setzt die Adressmarke Start hinter die IVT
Start:
```

```
 LDI     Reg_A, High(RamEnd)   ; erste Maßnahme Stack setzen
 Out     SPH, Reg_A
 LDI     Reg_A, Low(RamEnd)
 OUT      SPL, Reg_A
                                        ; weitere Initialisierungen
 RCALL   Init_IO                        ; Initialisierung IO
; Initialisierung UART
; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
; Initialisierung Variablen ( Eintragen Defaultwerte)
; SEI                                    ; Masterfreigabe von Interrupts
;*****************************************************************
;*            Programmschleife Hauptprogramm            *
;*****************************************************************
Main_Loop:
           ;  Hauptprogramm
           ; Aufruf der Subroutinen
 RCALL    Read_IO
 ; RCALL    Set_LED_Bit
 RCALL    Blinker
 RCALL    Write_IO
RJMP  Main_Loop
;-----------------------------------------------------------------
;*****************************************************************
;                        Subroutinen                              *
;*****************************************************************

Init_IO:                    ; Ein-Ausgaben initialisieren
;*******************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf       *
;*   Dabei werden nur die benötigten Portbits parametriert,       *
;*   die anderen in ihrer Vorgabe belassen.                       *
;*******************************************************************************
  IN     Reg_A, DDRD       ; Port D Bits 0-1 = Serielle Schnittstelle
                           ; Bit 2 Eingang Taster 1
                           ; Bit 3 Eingang Taster 2
                           ; Bit 4 Eingang Taster 3
                           ; Bit 5 Eingang Taster 4
                           ; Bit 6 Eingang Gabellichtschranken
                           ; Bit 7 Ausgang Relais
  ANDI   Reg_A, 0b00000011       ; Bit 0 und 1 nicht verändern
  ORI    Reg_A, 0b10000000       ; Bit 7 auf Ausgang
  OUT    DDRD, Reg_A             ; Data Direction Register PortD beschreiben
  IN       Reg_A, PortD
  ORI      Reg_A,0b01111100        ; PortD Pull-Up für die Eingänge einschalten
```

```
                        ; Die anderen Portbits bleiben unverändert
  OUT   PortD, Reg_A          ; Port mit Inhalt von Register A beschreiben
  IN    Reg_A, DDRC           ; Data Direction Register lesen
  ORI   Reg_A,0b00111110      ; Port C Bits 1-5 auf Ausgang schalten
  ANDI  Reg_A, 0b11111110     ; Port C Bits ist Eingang
                        ; Die anderen Bits bleiben unverändert
  OUT   DDRC, Reg_A         ; Data Direction Register Port C beschreiben
  In    Reg_A, PortC
  ORI   Reg_A, 0b00000001     ; Pull_Up Widerstand PC0 einschalten
  Out   PortC, Reg_A
  In    Reg_A, DDRB           ; Data Direction Register Port B lesen
  ORI   Reg_A,0b00111111      ; Portbit 0-5 Ausgang
  Out   DDRB, Reg_A           ; Data Direction Register B beschreiben
RET
;-----------------------------------------------------------------------------

;********** Lesen aller Eingänge und Signalanpassung *************
;*    Port D                                                      *
;*      Bit 0 und 1   serielle Verbindung                         *
;*      Bit 2 bis 5   Eingänge Taster                             *
;*      Bit 6         Eingang Gabellichtschranke                  *
;*      Bit 7         Ausgang Anzeige                             *
;*****************************************************************************
Read_IO:                    ; Eingänge einlesen
  In    Reg_A, PInD           ; Port B lesen
  COM   Reg_A                 ; Bits drehen
  ANDI  Reg_A, 0b01111100     ; Nur Eingänge übernehmen
  LSR   Reg_A                 ; nach rechts schieben (00111110)
  LSR   Reg_A                 ; nach rechts schieben (00011111)
  STS   New_In, Reg_A
RET
;-----------------------------------------------------------------------------

;********** Eingabe bearbeiten und Ausgabe vorbereiten ************
;*    Variable  New_In enthält Status der Eingänge                *
;*                                                                *
;*    Variable Out_Ctrl enthält Ergebnis aus der Bearbeitung      *
;*    für die Ausgabe                                             *
;*****************************************************************************
Set_LED_Bit:                ; LED einschalten
  LDS   Reg_A, New_In
  ANDI  Reg_A, 0b00000001     ; Nur Bit 1 prüfen (Ergebnis ist nicht "0")
  BRNE  Set_LED1              ; Bit= 1, dann  Sprung nach Marke Set_LED1
```

```
  LDS    Reg_B, Out_Ctrl          ; Nun das Byte für den Ausgang holen
  ANDI   Reg_B, 0b11011111        ; Mit einer Und-Anweisung das Bit 5 löschen
  RJMP   End_Set_LED_Bit          ; Sprung zum Ende der Sub-Routine
Set_LED1:                         ; Einstieg zum Setzen des Bits
  SWAP  Reg_A                     ; tauscht die unteren 4 Bits mit den oberen 4 Bits (Nibble)
  ROL   Reg_A                     ; Schiebt den Inhalt von Register A noch einmal nach links
  LDS   Reg_B, Out_Ctrl           ; Nun das Byte für den Ausgang holen
  OR    Reg_B, Reg_A              ; und das 5. Bit in Register B setzen
End_Set_LED_Bit:                  ; Markiert das Ende der Subroutine
  STS   Out_Ctrl, Reg_B           ; nun noch das Register B in die Variable Out_Ctrl schreiben
RET                               ; Und Rücksprung zum Aufruf
;-----------------------------------------------------------------

;*****************************************************************
;*                Eine LED blinken lassen                        *
;*****************************************************************

Blinker:
  LDS     Reg_A,Counter_0         ;inneren Schleifenzähler
  INC     Reg_A
  STS     Counter_0, Reg_A
  CPI     Reg_A, 100              ; bis 100 zählen, dann von vorn
  BRLO    End_Blinker
  CLR     Reg_A
  STS     Counter_0, Reg_A        ; mit 100 multiplizieren
  LDS     Reg_A, Counter_1
  INC     Reg_A
  STS     Counter_1, Reg_A
  CPI     Reg_A, 100
  BRLO    End_Blinker
  CLR     Reg_A                   ; bei 10000 wieder von vorn anfangen
  STS     Counter_1, Reg_A
  LDS     Reg_A, Out_Ctrl
  LDI     Reg_B, 0b00100000
  EOR     Reg_A, Reg_B
  STS     Out_Ctrl, Reg_A
End_Blinker:
RET
;-----------------------------------------------------------------

;*****************************************************************
;*              Ausgabe an die IO-Ebene                          *
;*            Port C Bit 5 = Relais / LED                        *
;*****************************************************************
Write_IO:
```

```
  IN      Reg_B, PinC            ; gesamten Port C lesen
  LDS     Reg_A, Out_Ctrl        ; Variable für Ausgang holen
  ANDI    Reg_A, 0b00100000      ; Bit 5 Prüfen
  BREQ    Rel_Aus                ; Wenn Bit =0 dann Ausgang aus
  ORI     Reg_B, 0b00100000      ; Ausgang zuschalten, Nur Bit 5 setzen
  RJMP    Set_Ausgang
Rel_Aus:
  ANDI    Reg_B,0b11011111       ; Ausgang abschalten, nur Bit 5 löschen
Set_Ausgang:
  OUT     PortC, Reg_A           ; Port mit Reg_C beschreiben
RET
;-----------------------------------------------------------------


;*****************************************************************
;*          Bereich der Interrupt Serviece Routinen      *
;*****************************************************************
```

## 2.7 Ereignissteuerung (Event)

### *2.7.1 Flankenerkennung*

Wenn unser kleines Programm so funktioniert, wie wir es möchten, können wir den nächsten Schritt wagen: eine LED bei jedem Tastendruck an- oder auszuschalten. Sozusagen eine Stromstoßschaltung.

Dazu brauchen wir eine Flankenerkennung des Eingangssignales. Wie funktioniert sowas? Nun, alles was dafür benötigt wird ist der letzte gelesene Zustand vom Port und wenn unsere Variable New_In benannt ist, sollte der letzte Zustand in Old_In abgelegt werden. Die Variable Old_In existiert allerdings noch nicht und mussin den Deklarationsteil eingefügt werden. Dazu kopieren wir die Deklaration von New_In mit allen Kommentaren und ändern nur den Namen ab. So behalten wir die Kommentare zur Auswertung in Open_Eye.

```
old_In: .Byte 1          ; Byte Ablage zuletzt gelesene Eingänge
                          ; Bit 0 = Taster 1 Mode
                          ; Bit 1 = Taster 2 Ziffer
                          ; Bit 2 = Taster 3 auf
                          ; Bit 3 = Taster 4 ab
                          ; Bit 4 = Gabellichtschranke
                          ; Bit 5 = res
                          ; Bit 6 = res.
                          ; Bit 7 = res.
```

Nun brauchen wir noch eine Variable, in der wir uns merken, welches Bit geändert wurde. Dazu kopieren wir ein weiteres Mal den Inhalt von New_In und ändern den Namen in

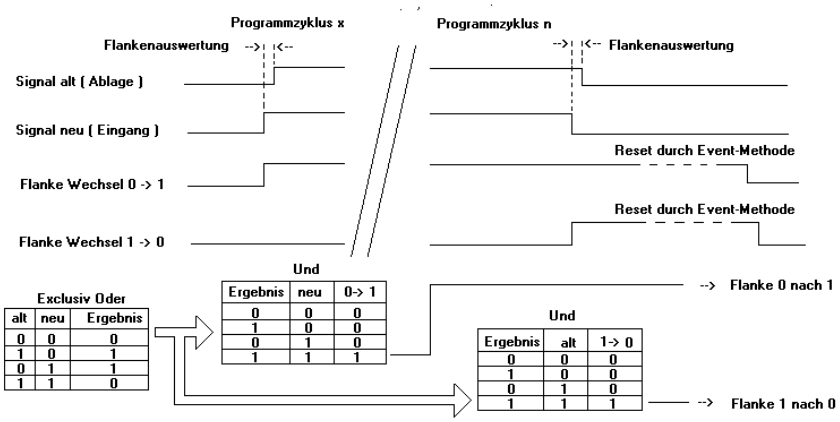
```
Event_To_High: .Byte 1          ; Byte Ablage Eingänge
                                  ; Bit 0 = Taster 1 Mode
                                  ; Bit 1 = Taster 2 Ziffer
                                  ; Bit 2 = Taster 3 auf
                                  ; Bit 3 = Taster 4 ab
                                  ; Bit 4 = Gabellichtschranke
                                  ; Bit 5 = res
                                   ; Bit 6 = res.
                                  ; Bit 7 = res.
```

```
Event_To_Low: .Byte 1          ; Byte Ablage Eingänge
                                 ; Bit 0 = Taster 1 Mode
                                 ; Bit 1 = Taster 2 Ziffer
                                 ; Bit 2 = Taster 3 auf
                                 ; Bit 3 = Taster 4 ab
                                 ; Bit 4 = Gabellichtschranke
                                 ; Bit 5 = res
                                  ; Bit 6 = res.
                                 ; Bit 7 = res.
```

An der Flankenerkennung sind nun insgesamt drei Variablen beteiligt. Da ist die bereits verwendete **New_In**, die mit den Status davor **Old_In** und eine Variable, die das Ereignis **Taster gedrückt** oder eben nur Signalwechsel nach **1** speichert. Da es sich immer um die gleichen Bits handelt, bleiben die Kommentare gültig.

Wir brauchen nun eine Routine, die uns das Event, oder zu deutsch, das Ereignis liefert. Man kann es in die Routine Read_IO mit einbauen, aber wir bleiben bei kleinen übersichtlichen Programmblöcken und nennen die neue Subroutine IO_Event.

Um Flanken zu verstehen, ist auch hier eine Skizze zum Signalverlauf angebracht:




*Signalwechsel Flankenerkennung*

Nach der Flankenauswertung wird die Ablage mit dem neuen Signal überschrieben.

Flankenauswertung ist ein geeignetes Mittel, Programme im Zyklus schnell zu machen. Wenn ein Programm nur auf Ereignisse reagieren muss, braucht nicht jeder Bearbeitungsschritt xMal mit dem gleichen Ergebnis bearbeitet werden. Tritt ein Ereignis auf, wird bearbeitet, sonst nicht. Ein weiterer Vorteil sind kleine überschaubare und schnell anpassbare Programmteile. Im Laufe der Tests werden wir die Vorteile noch erkennen.

### 2.7.2 *Ein Ereignis erfassen*

Immer, wenn ein Signalwechsel vorliegt, ist ein Ereignis aufgetreten. Bei Visual Basic haben wir davon ausgiebig Gebrauch gemacht. Warum nicht auch in einem Controllerprogramm nur auf Ereignisse reagieren. Nur, wenn eine Änderung eingetreten ist, ist auch eine Bearbeitung erforderlich. Andernfalls gibt es keinen Grund, ständig festzustellen, das keine anderen Ergebnisse in der Bearbeitung erzielt werden.

Nun hat so eine Ereignisbearbeitung auch ein relativ einfaches Schema. Das Ereignis wird erkannt und markiert. Das Programm wertet irgendwann diese Markierung aus und entscheidet, ob eine Bearbeitung erforderlich ist. Dabei wird die Markierung gelöscht. So wird diese Bearbeitung nur einmal vorgenommen. Erst, wenn eine neue Markierung gesetzt ist, wird wieder etwas durchgeführt. Das Programm eines Stromstoßrelais eignet sich hervorragend, um diese Technik zu verdeutlichen. Beginnen wir nun mit dem Aufbau einer Ereigniserfassung.

```
;*********************************************
;*   Ereignis Signalwechsel nach „1“ am Eingabe-Port.     *
;*********************************************
IO_Event_To_1:
  LDS Reg_A, Old_In              ; letzter gelesener Wert der Eingänge
  LDS Reg_B, New_In              ; neu gelesener Wert der Eingänge
  EOR Reg_A, Reg_B               ; EOR= Exclusiv-Oder Verknüpfung. 1 nur bei Unterschied
  AND Reg_A, Reg_B               ; Unterschied in Reg.A, Reg_B alter Wert
  BREQ End_Event_To_1            ; nur wenn ein Wechsel von 0 nach 1 erfolgt ist
  LDS  Reg_C, Event_To_High      ; alte Bits erhalten
  OR     Reg_A, Reg_C            ; neues Bit hinzufügen
  STS Event_To_High, Reg_A       ; und eintragen
  STS Old_In, Reg_B              ; Neuen Wert in Ablage „old_In“ kopieren
RET
```

Und das soll nun schon alles sein? Nun, im Prinzip ja. In der Skizze sind die Werte der Register einmal eingetragen und der Verlauf bis zum Ergebniswert aufgezeichnet.

| | | | |
|---|---|---|---|
| Wert alt | ⇨ | Register A | 10011100 |
| Wert neu | ⇨ | Register B | 01110011 |
| EOR Register A ; Register B | | | 11101111 |
| | | | |
| Ergebnis in | | Register A | 11101111 |
| | | Register B | 01110011 |
| UND Register A ; Register B | | | 01100011 |
| | | | |
| Ergebnis in Register A | | | 01100011 |
| Event_To_High | ⇦ | Register A | 01100011 |
| Old_In | ⇦ | Register B | 01110011 |

*Bearbeitungsfolge Flankenerfassung*

Während der gesamten Bearbeitung hat sich der Wert in Register B nicht verändert. Da es der neue Status der Eingänge ist, kann somit Old_In direkt wieder beschrieben werden.

Die Logik ist folgende:

Zuerst wird der Unterschied in **Old_In** und **New_In** mit der Exclusiv-Verknüpfung festgestellt. Das Ergebnis steht immer im erstgenannten Register, also Register A. Da in Register B die neuen Werte sind und dieser Inhalt mit den geänderten Bits logisch Und-Verknüpft wird, stehen im Register A nur die Bits, die vorher eine 0 hatten und zur 1 geworden sind. Damit haben wir Flankenbits für die Eingangssignale, wenn sie einen Wechsel von **0** nach **1** haben. Das entspricht nach der Aufarbeitung in Read_IO T**aster gedrückt**.

Natürlich ist auf gleichem Weg eine Abfrage nach fallenden Flanken möglich. Dazu muss der Code nur etwas umgestellt werden.

```
;*********************************************************************
;*    Ereignis Signalwechsel nach 0 am Eingabe-Port.    *
;*********************************************************************
IO_Event_To_0:
  LDS   Reg_B, Old_In            ; letzter gelesener Wert der Eingänge
  LDS   Reg_A, New_In            ; neu gelesener Wert der Eingänge
  STS   Old_In, Reg_A            ; Neuen Wert in Ablage „old_In“ kopieren,
; ist hier bereits erforderlich, da Reg_A überschrieben wird
  EOR   Reg_A, Reg_B     ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
  AND   Reg_A, Reg_B     ; Unterschied in Reg.A, alter Wert in Reg_B
  BREQ End_Event_To_0            ; nur wenn ein Wechsel von 0 nach 1 erfolgt ist
```

```
  LDS  Reg_B, Event_To_High    ; alte Bits erhalten
  OR     Reg_A, Reg_B              ; neues Bit hinzufügen
 STS   Event_To_LOW, Reg_A     ; alten Wert liefert die Bits mit Änderung von „1“ nach „0“
End_Event_To_0:
RET
```

Sollen aber beide Flanken in einem Programm Verwendung finden, müssen die Routinen in eine einzige zusammengefasst werden. Der Grund ist die Übernahme des Neuen Wertes in die Ablage. In der zweiten Routine steht dann natürlich nicht mehr der Unterschied zur Verfügung. Doch beide Routinen in einer einzigen zusammengefasst, ist auch kein Problem.

```
;**********************************************
;*          Ereignisse Signalwechsel am Eingabe-Port.      *
;**********************************************
IO_Event_Flanke:
  LDS Reg_A, Old_In              ; letzter gelesener Wert der Eingänge
  MOV Ablage, Reg_A              ; alten Wert zwischenspeichern
  LDS Reg_B, New_In              ; neu gelesener Wert der Eingänge
  EOR Reg_A, Reg_B               ; EOR= Exclusiv-Oder Verknüpfung. 1 nur bei Unterschied
  AND Reg_A, Reg_B               ; Unterschied in Reg.A, Reg_B alter Wert
  BREQ End_Event_High            ; nur wenn ein Wechsel von 0 nach 1 erfolgt ist
  LDS  Reg_C, Event_To_High      ; alte Bits erhalten
  OR     Reg_A, Reg_C            ; neues Bit hinzufügen
  STS Event_To_High, Reg_A       ; und eintragen
End_Event_High :
  MOV   Reg_B, Ablage            ; Zwischenspeicher lesen
  LDS   Reg_A, New_In            ; neu gelesener Wert der Eingänge
  STS   Old_In, Reg_A            ; Neuen Wert in Ablage „old_In“ kopieren,
; ist hier bereits erforderlich, da Reg_A überschrieben wird
 EOR   Reg_A, Reg_B     ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
 AND   Reg_A, Reg_B     ; Unterschied in Reg.A, alter Wert in Reg_B
 BREQ End_Event_Flanke          ; nur wenn ein Wechsel von 0 nach 1 erfolgt ist
  LDS  Reg_B, Event_To_Low      ; alte Bits erhalten
  OR     Reg_A, Reg_B            ; neues Bit hinzufügen
 STS   Event_To_LOW, Reg_A     ; und eintragen
End_Event_Flanke:
RET
```

Auch diese Bearbeitung verliert nicht an Übersichtlichkeit. Flanken, die nicht verwendet werden können unbearbeitet bleiben. In Open_Eye werden sie dann eben als anstehendes Ereignis markiert.

### 2.7.3 *Ereignisse auswerten und bearbeiten*

Mit der Flankenerfassung haben wir die Grundlage für Signalereignisse von der Peripherie. Mit der Routine Taster_Event wird auf diese Ereignisse reagiert.

```
*****************************************************
;*                    Tastersignal prüfen                    *
;*****************************************************
Taster_Event:
    LDS     Reg_A, Event_To_High        ; Ereignisbits laden
    ANDI    Reg_A, 0b00000010           ; Nur Bit 1 prüfen (Ergebnis ist nicht 0)
    BREQ    End_Taster_Event            ; Ergebnis 0, dann Sprung nach Marke Status_Ende
    LDS     Reg_A, Out_Ctrl             ; Nun das Byte für den Ausgang holen
    LDI     Reg_B, 0b00000010           ; Ist für eine EOR Anweisung erforderlich
    EOR     Reg_A, Reg_B                ; dreht das Bit 1 um, 1 nach 0 und 0 nach 1
    STS     Out_Ctrl, Reg_A     ; nun noch das Register B in die Variable Out_Ctrl schreiben
    LDS     Reg_A, Event_To_High        ; Ereignisbits laden
    ANDI    Reg_A, 0b11111101           ; Ereignisbit 0 löschen
    STS     Event_To_High, Reg_A        ; Und Ereignisse zurückschreiben
End_Taster_Event:                       ; Markiert das Ende der Subroutine
RET
```

Damit wir Ereignisse bekommen, bauen wir die neue Routine in die Programmschleife ein:

```
;*****************************************************
;*        Schleife Hauptprogramm              *
;*****************************************************
Loop:
   RCALL Read_IO              ; Eingänge lesen
   RCALL Set_LED_Bit          ; Bearbeiten „V“
   RCALL Blinker              ; Blinkerbit bilden
   RCALL IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
 ; alternativ auch      RCALL IO_Event_Flanke
  RCALL Taster_Event          ; Bearbeiten
   RCALL Write_IO             ; und ausgeben
RJMP Loop
```

Nun haben wir Flankenbits, deren Bearbeitung nur einmalig erforderlich ist. Dazu ändern wir nun die Routine **Taster_Status** und passen sie dem Ereignis an. Dabei setzen wir für die LED einen anderen Ausgang ein,

denn PC5 ist ja der Blinker, den wir zur Fehlersuche noch brauchen werden. PC 1 ist ebenfalls als Ausgang parametriert, also schließen wir dort eine LED mit Vorwiderstand nach GND an.

Dieses Ereignis wird bei einem Tastendruck nur einmal durchlaufen. Die EOR-Anweisung dreht deshalb auch nur einmal das Bit pro Tastendruck. Eine Änderung wird erst mit einem erneuten Signalwechsel durchgeführt. Die klassische Stromstoßschaltung. Unsere Ausgaberoutine Write_IO muss jetzt auch angepasst werden, damit auch die anderen Portpins durchgereicht werden.

```
;*********************************************
;*      Ausgabe an die IO-Ebene        *
;*      Port C Bit 5 = Relais / LED      *
;*********************************************
Write_IO:
   IN      Reg_B, PinC                ; gesamten Port C lesen
   ANDI    Reg_B, 0b11000001           ;  alle Ausgabebits auf 0
   LDS     Reg_A, Out_Ctrl            ; Variable für Ausgang holen
   ANDI    Reg_A, 0b00111110           ;  Nur Ausgabebits maskieren
   OR      Reg_B, Reg_A                ;  Register a zu B hinzufügen
   OUT     PortC, Reg_A               ; Port mit Reg_C beschreiben
RET
```

Dabei ist die Routine noch viel einfacher geworden. Klar, wenn man weis mit Verknüpfungen  umzugehen, ist es einfach. Im ersten Schritt wird Port C gelesen und die benutzen Portpins zur Ausgabe auf 0 gesetzt. Am Port selbst hat sich noch nichts geändert. Nun wird in der Variablen Out_Ctrl ein Filter auf die Bits 1-5 gesetzt. Nur in diesem Bereich können jetzt noch Bits mit dem Status 1 gesetzt sein, Waren sie vorher 0, behalten sie natürlich die 0.

Schließlich fügt die Oder-Verknüpfung die beiden Register A und B zusammen. Die Bits 0, 6 und 7 bleiben dem Port unverändert erhalten, lediglich im Bereich Bit 1 bis 5 werden nun die auf 1 gesetzten Bits von Register A übernommen.

### *2.7.4 Programm FlippFlopp*

```
;*********************************************************************
;*                    Datum 06.08.2012                               *
;*            Beispielanwendungen mit Atmega 8                       *
;* ------------------------------------------------------------------ *
;*      Ein Experimente mit PC und Mikrocontroller                   *
;*                                                                   *
;*  Die Routinen für den Atmega8 beinhalten:                         *
;*     Standart  IO                                                  *
;*     Signalerfassung von Taster                                    *
;*    Ansteuerung von LED                                            *.
;*  Copyright by Martin Vogel   Germany                              *
;*********************************************************************
.NoList
.include   "m8def.inc"      ; Definitionen für ATMEGA8
.List

.DEF Access      = R0      ; Speicherzugriffsregister
.DEF Zero        = R1      ; Register für Null-Vergleich
.DEF Zehn        = R2      ; Register für Vergleich mit konst. 10
.DEF Zwei        = R3      ; Zeiterfassung Minute
.DEF fuenf       = R4      ; Register für Vergleich mit konst. 60
.DEF hundert     = R5      ; Register für Vergleich mit kont. 100
.DEF MilliSek    = R6      ; Register für mSek ( schneller Zugriff)
.DEF ZehnMilli  = R7      ; Register für hundertstel Sek. ( schneller Zugriff)
.Def Zehntel     = R8      ; Register für  Zehntelsekunde
.DEF Ablage_A   = R9      ; Zwischenspeicher A
.DEF Ablage_B   = R10     ; Zwischenspeicher B
.DEF Count_L     = R11     ; Counter Low 0-7
.DEF Count_H     = R12     ; Counter High 8-15
.DEF Count_HL  = R13     ; Counter HighLow 16-23
.DEF Count_HH  = R14     ; Counter High High 24-31
.Def SichSREG    = R15     ; Zwischenspeicher f. Statusregister
.DEF Reg_A       = R16     ; Universalregister A
.DEF Reg_B       = R17     ; Universalregister B
.DEF Reg_C       = R18     ; Universalregister C
.DEF Reg_D       = R19     ; Universalregister D
.DEF Reg_E       = R20     ; Universalregister E
.DEF Reg_F       = R21     ; Universalregister E
.DEF Work        = R22     ;  Arbeitsregister
.DEF Send_Byte  = R23     ; Datenübertragung
.DEF Calc_A      = R24     ; Math.-Register A
```

```
.DEF Calc_B        = R25      ; Math. Register B
; ---- Register 26- 31 sind bereits vergeben ----
; X = XL + XH = R 26 + R 27
; Y = YL + YH = R 28 + R 29
; Z = ZL + ZH = R 30 + R 31
.DSEG
Trigger_In:        .Byte 1      ; Byte Debug-Variable Eingang
Trigger_Out:       .Byte 1      ; Byte Debug-Variable Ausgang

;--------------------- Variablen IO-Ebene ----------------------
New_In:            .Byte 1  ; Byte Ablage Eingänge
                            ; Bit 0 = Taster 1 Mode
                            ; Bit 1 = Taster 2 Ziffer
                            ; Bit 2 = Taster 3 auf
                            ; Bit 3 = Taster 4 ab
                            ; Bit 4 = Gabellichtschranke
                            ; Bit 5 = res
                            ; Bit 6 = res.
                            ; Bit 7 = res.

Out_Ctrl:          .Byte 1  ; Byte Ablage für Ausgänge
                           ; Bit 0 = Relais 1
                           ; Bit 1 = Relais 2
                           ; Bit 2 = Relais 3
                           ; Bit 3 = Relais 4
                           ; Bit 4 = Relais 5
                           ; Bit 5 = LED 1
                           ; Bit 6 = LED 2
                           ; Bit 7 = LED 3
Counter_0:         .Byte 1      ; 16 Bit-Zähler LowByte
Counter_1:         .Byte 1      ; 16 Bit-Zähler HighByte
old_In:            .Byte 1      ; Byte Ablage zuletzt gelesene Eingänge
                           ; Bit 0 = Taster 1 Mode
                           ; Bit 1 = Taster 2 Ziffer
                           ; Bit 2 = Taster 3 auf
                           ; Bit 3 = Taster 4 ab
                           ; Bit 4 = Gabellichtschranke
                           ; Bit 5 = res
                           ; Bit 6 = res.
                           ; Bit 7 = res.
Event_To_High:     .Byte 1      ; Byte Ablage Eingänge
                           ; Bit 0 = Taster 1 Mode
                           ; Bit 1 = Taster 2 Ziffer
                           ; Bit 2 = Taster 3 auf
                           ; Bit 3 = Taster 4 ab
```

```
                        ; Bit 4 = Gabellichtschranke
                        ; Bit 5 = res
                        ; Bit 6 = res.
                        ; Bit 7 = res.

Event_To_Low:   .Byte 1     ; Byte Ablage Eingänge
                        ; Bit 0 = Taster 1 Mode
                        ; Bit 1 = Taster 2 Ziffer
                        ; Bit 2 = Taster 3 auf
                        ; Bit 3 = Taster 4 ab
                        ; Bit 4 = Gabellichtschranke
                        ; Bit 5 = res
                        ; Bit 6 = res.
                        ; Bit 7 = res.


.CSEG
.ORG 0000
Reset_Point:   RJMP  Start    ; Start von hier ist Neustart
;*******************************************************************
;*          Bereich der Interrupt Vector Table              *
;*******************************************************************
          ; Sprungbefehle zu den Serviceroutinen

;*******************************************************************
;*                  Bereich Initialisierung                     *
;*******************************************************************
.ORG    INT_VECTORS_SIZE      ; Setzt die Adressmarke Start hinter die IVT

Start:
  LDI     Reg_A, High(RamEnd)   ; erste Maßnahme Stack setzen
  Out     SPH, Reg_A
  LDI     Reg_A, Low(RamEnd)
  OUT      SPL, Reg_A
                                ; weitere Initialisierungen
  RCALL   Init_IO               ; Initialisierung IO
; Initialisierung UART
; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
; Initialisierung Variablen ( Eintragen Defaultwerte)
; SEI                           ; globale Freigabe von Interrupts
;*******************************************************************
;*            Programmschleife Hauptprogramm            *
```

```
;*****************************************************************
Main_Loop:
            ;  Hauptprogramm
            ; Aufruf der Subroutinen
  RCALL    Read_IO
 ; RCALL    Set_LED_Bit
  RCALL    Blinker
  RCALL    IO_Event_To_1     ; Ereignisbits bilden. Ergebnis in Event_To_High
  RCALL    Taster_Event      ; Bearbeiten
  RCALL    Write_IO
RJMP  Main_Loop
;-----------------------------------------------------------------
;*****************************************************************
;                        Subroutinen                            *
;*****************************************************************

Init_IO:                    ; Ein-Ausgaben initialisieren
;*******************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf      *
;*   Dabei werden nur die benötigten Portbits parametriert,      *
;*   die anderen in ihrer Vorgabe belassen.                      *
;*******************************************************************************
;*******************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf      *
;*   Dabei werden nur die benötigten Portbits parametriert,      *
;*   die anderen in ihrer Vorgabe belassen.                      *
;*******************************************************************************
   IN     Reg_A, DDRD       ; Port D Bits 0-1 = Serielle Schnittstelle
                            ; Bit 2 Eingang Taster 1
                            ; Bit 3 Eingang Taster 2
                            ; Bit 4 Eingang Taster 3
                            ; Bit 5 Eingang Taster 4
                            ; Bit 6 Eingang Gabellichtschranken
                            ; Bit 7 Ausgang Relais
   ANDI   Reg_A, 0b00000011       ; Bit 0 und 1 nicht verändern
   ORI    Reg_A, 0b10000000       ; Bit 7 auf Ausgang
   OUT    DDRD, Reg_A             ; Data Direction Register PortD beschreiben
   IN     Reg_A, PortD
   ORI    Reg_A,0b01111100        ; PortD Pull-Up für die Eingänge einschalten
                            ; Die anderen Portbits bleiben unverändert
   OUT    PortD, Reg_A            ; Port mit Inhalt von Register A beschreiben
   IN     Reg_A, DDRC             ; Data Direction Register lesen
   ORI    Reg_A,0b00111110        ; Port C Bits 1-5 auf Ausgang schalten
   ANDI   Reg_A, 0b11111110       ; Port C Bits ist Eingang
                            ; Die anderen Bits bleiben unverändert
```

```
  OUT    DDRC, Reg_A          ; Data Direction Register Port C beschreiben
  In     Reg_A, PortC
  ORI    Reg_A, 0b00000001      ; Pull_Up Widerstand PC0 einschalten
  Out    PortC, Reg_A
  In     Reg_A, DDRB            ; Data Direction Register Port B lesen
  ORI    Reg_A,0b00111111       ; Portbit 0-5 Ausgang
  Out    DDRB, Reg_A            ; Data Direction Register B beschreiben
RET
;-----------------------------------------------------------------------------

;********** Lesen aller Eingänge und Signalanpassung *************
;*    Port D                                                          *
;*      Bit 0 und 1    serielle Verbindung                            *
;*      Bit 2 bis 5    Eingänge Taster                                *
;*      Bit 6          Eingang Gabellichtschranke                     *
;*      Bit 7          Ausgang Anzeige                                *
;*****************************************************************************
Read_IO:                    ; Eingänge einlesen
 In     Reg_A, PInD                ; Port B lesen
 COM    Reg_A                      ; Bits drehen
 ANDI   Reg_A, 0b01111100          ; Nur Eingänge übernehmen
 LSR    Reg_A                      ; nach rechts schieben (00111110)
 LSR    Reg_A                      ; nach rechts schieben (00011111)
 STS    New_In, Reg_A
RET
;-----------------------------------------------------------------------------

;*****************************************************************************
;*     Ereignis Signalwechsel nach "1" am Eingabe-Port.        *
;*****************************************************************************
IO_Event_To_1:
 LDS    Reg_A, Old_In      ; letzter gelesener Wert der Eingänge
 LDS    Reg_B, New_In      ; neu gelesener Wert der Eingänge
 EOR    Reg_A, Reg_B       ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
 AND    Reg_A, Reg_B       ; eine Und Verknüpfung mit dem neuen Status
 STS    Event_To_High, Reg_A ; liefert die Bits mit Änderung von "0" nach "1"
 STS    Old_In, Reg_B      ; Neuen Wert in Ablage "old_In" kopieren
RET
;-----------------------------------------------------------------------------

;*****************************************************************************
;*        Ereignis Signalwechsel nach 0 am Eingabe-Port.        *
;*****************************************************************************
```

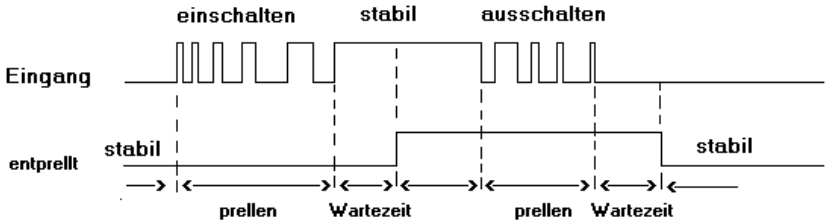
```
IO_Event_To_0:
  LDS   Reg_B, Old_In       ; letzter gelesener Wert der Eingänge
  LDS   Reg_A, New_In       ; neu gelesener Wert der Eingänge
  STS   Old_In, Reg_A       ; Neuen Wert in Ablage "old_In" kopieren,
; ist hier bereits erforderlich, da Reg_A überschrieben wird
  EOR   Reg_A, Reg_B        ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
  AND   Reg_A, Reg_B        ; Und Verküpfung mit dem alten Status
  STS   Event_To_LOW, Reg_A     ; liefert die Bits mit Änderung von "1" nach "0"
RET
;-----------------------------------------------------------------

;*********************************************************************
;*                          Tastersignal prüfen                      *
;*********************************************************************
Taster_Event:
  LDS      Reg_A, Event_To_High      ; Ereignisbits laden
  ANDI     Reg_A, 0b00000001           ; Nur Bit 1 prüfen (Ergebnis ist nicht 0)
  BREQ     End_Taster_Event          ; Ergebnis 0, dann Sprung nach Marke Status_Ende
  LDS      Reg_A, Out_Ctrl             ; Nun das Byte für den Ausgang holen
  LDI      Reg_B, 0b00000010         ; Ist für eine EOR Anweisung erforderlich
  EOR      Reg_A, Reg_B              ; dreht das Bit 1 um, 1 nach 0 und 0 nach 1
  STS      Out_Ctrl, Reg_A      ; nun noch das Register B in die Variable Out_Ctrl schreiben
  LDS      Reg_A, Event_To_High      ; Ereignisbits laden
  ANDI     Reg_A, 0b11111110           ; Ereignisbit 0 löschen
  STS    Event_To_High, Reg_A       ; Und Ereignisse zurückschreiben
End_Taster_Event:                    ; Markiert das Ende der Subroutine
RET
;-----------------------------------------------------------------------------

;********** Eingabe bearbeiten und Ausgabe vorbereiten ************
;*     Variable  New_In enthält Status der Eingänge                    *
;*                                                                       *
;*     Variable Out_Ctrl enthält Ergebnis aus der Bearbeitung      *
;*     für die Ausgabe                                                   *
;*******************************************************************************
Set_LED_Bit:                         ; LED einschalten
   LDS    Reg_A, New_In
   ANDI   Reg_A, 0b00000001     ; Nur Bit 1 prüfen (Ergebnis ist nicht "0")
   BRNE   Set_LED1                  ; Bit= 1, dann  Sprung nach Marke Set_LED1
   LDS    Reg_B, Out_Ctrl           ; Nun das Byte für den Ausgang holen
   ANDI   Reg_B, 0b11011111       ; Mit einer Und-Anweisung das Bit 5 löschen
   RJMP   End_Set_LED_Bit       ; Sprung zum Ende der Sub-Routine
Set_LED1:                            ; Einstieg zum Setzen des Bits
   SWAP  Reg_A                 ; tauscht die unteren 4 Bits mit den oberen 4 Bits (Nibble)
   ROL   Reg_A                 ; Schiebt den Inhalt von Register A noch einmal nach links
```

```
  LDS   Reg_B, Out_Ctrl       ; Nun das Byte für den Ausgang holen
  OR    Reg_B, Reg_A          ; und das 5. Bit in Register B setzen
End_Set_LED_Bit:              ; Markiert das Ende der Subroutine
  STS   Out_Ctrl, Reg_B       ; nun noch das Register B in die Variable Out_Ctrl schreiben
RET                           ; Und Rücksprung zum Aufruf
;-----------------------------------------------------------------

;*******************************************************************
;*                  Eine LED blinken lassen                        *
;*******************************************************************

Blinker:
  LDS     Reg_A,Counter_0       ;inneren Schleifenzähler
  INC     Reg_A
  STS     Counter_0, Reg_A
  CPI     Reg_A, 100            ; bis 100 zählen, dann von vorn
  BRLO    End_Blinker
  CLR     Reg_A
  STS     Counter_0, Reg_A      ; mit 100 multiplizieren
  LDS     Reg_A, Counter_1
  INC     Reg_A
  STS     Counter_1, Reg_A
  CPI     Reg_A, 100
  BRLO    End_Blinker
  CLR     Reg_A                 ; bei 10000 wieder von vorn anfangen
  STS     Counter_1, Reg_A
  LDS     Reg_A, Out_Ctrl
  LDI     Reg_B, 0b00100000
  EOR     Reg_A, Reg_B
  STS     Out_Ctrl, Reg_A
End_Blinker:
RET
;-----------------------------------------------------------------

;*******************************************************************
;*                Ausgabe an die IO-Ebene                          *
;*             Port C Bit 1- 5 = Relais / LED                      *
;*                  ab jetzt geändert !                            *
;*******************************************************************
Write_IO:
  IN      Reg_B, PinC           ; gesamten Port C lesen
  ANDI    Reg_B, 0b11000001     ; alle Ausgänge auf 0
  LDS     Reg_A, Out_Ctrl       ; Variable für Ausgang holen
```

```
  ANDI    Reg_A, 0b00111110       ; nur Bit 1-5 gültig
  OR      Reg_B, Reg_A            ; Ausgänge zuschalten, (1 in Out_Ctrl)
  OUT     PortC, Reg_B            ; Port mit Reg_C beschreiben
RET
;-----------------------------------------------------------------


;*****************************************************************
;*          Bereich der Interrupt Serviece Routinen       *
;*****************************************************************
```

## 2.8 Kontaktprellen

Das ist schnell programmiert und in den µC geladen. Nun sollte die Stromstoßschaltung funktionieren. Und jetzt ist der Punkt, wo Theorie und Praxis verschiedene Wege gehen. Das Ergebnis scheint zufällig zu sein. Mal wechselt die LED, mal nicht. Was haben wir falsch gemacht? Nun, der Elektroniker wird's wissen, vielleicht auch der Elektriker. Kein Kontakt schaltet ohne Kontaktprellen. Den Signalverlauf kann man sich in etwa so vorstellen:




*Signalbild Kontaktprellen*

Das geht so schnell, das es bei einer direkt angesteuerten Leuchte nicht auffällt, aber in der Ereignissteuerung wird für jeden erkannten Signalwechsel die Auswerteroutine durchlaufen und das zu bearbeitende Bit hin und her gedreht. Ist der Signalpegel dann endlich im stabilen Zustand angekommen, hat sich das Bit schon mehrfach gedreht. Ist in der Auswertung ein Zähler für diesen Impuls oder eine bistabile Relaisstufe (Stromstoßrelais) programmiert, dann ist die endgültige Position ist eher zufällig. Das ist aber nicht das, was wir wollen. So eine unzuverlässige Schaltung braucht kein Mensch. Nun, wir würden nicht mit Mikrocontrollern arbeiten, wenn es für diese Fehlfunktion, die durch externe Signalgeber ausgelöst wird, keine Abhilfe gäbe. Entprellen heißt das Zauberwort. Aber was versteht man darunter?

So ein Signal flattert nicht ewig, sondern beruhigt sich nach ein paar Millisekunden. In dieser Zeit hat ein Controller aber sein Programm schon mehrfach durchlaufen. Daher setzen wir einen Zähler. Jedes Mal, wenn die Routine **Read_IO** einen neuen Wert liefert, wird er ebenfalls mit einem vorherigen geprüft. Wir wissen schon, die **EOR**-Anweisung. Allerdings ist dies nicht die Flankenerkennung, sondern lediglich das Feststellen einer Signaländerung am Eingang. Wird diese erkannt, setzen wir einen Zähler auf bspw. 50. Ist kein Unterschied, wird der Zähler herab gezählt. Landen

wir bei 0, dann ist die Signallage stabil und der gelesene Wert kann in **New_IN** eingetragen werden.

Schauen wir doch dieser Funktion einmal in den Aufbau. Zuerst, ja, wir brauchen wieder drei neue Variablen:

```
Akt_In: .Byte 1            ; Aktuell gelesener Portwert
In_Debounce: .Byte 1       ; zuletzt gelesener Portwert
Debounce_Cnt: .Byte 1      ; Zähler für Wartezeit
```

Und nun ändern wir in der Routine **Read_IO** die Zuweisung von **New_In** auf **Akt_In**.

Die weitere Bearbeitung erfolgt in einem weiteren Schritt.

### 2.8.1 Eingänge entprellen

Wie ihr euch schon denken könnt, auch dafür wird eine eigene Subroutine geschrieben.

```
;*****************************************************
;*          Entprellen von Eingängen          *
;*****************************************************
IO_Debounce:
   LDS     Reg_A, Akt_In            ; Aktuellen Portwert laden
   LDS     Reg_B, In_Debounce   ; Vergleichswert laden
   EOR     Reg_B, Reg_A             ; Ergebnis in Register B, Register A nicht verändern
   BREQ   Chk_Debounce_Time ; Wenn beide gleich, Prellzeit prüfen
   STS     In_Debounce, Reg_A,   ; aktuellen Wert für nächsten Vergleich ablegen
   LDI      Reg_A, 50                  ; Überwachungszeit neu setzen
   STS      DebounceCnt, Reg_A    ; und in Zähler eintragen
   RJMP    End_Debounce
Chk_Debounce_Time:
   LDS     Reg_A, Debounce_Cnt
   CPI      Reg_A, 0                   ; Vergleich ist wichtig, Ladeanweisung setzt kein Zerobit.
   BREQ  End_Debounce           ; Wenn zähler auf 0, keine weitere Aktion
   LDS      Reg_B, Debounce_Cnt    ; Register A hat noch den Wert vom gelesenen Port
   Dec      Reg_B
   STS      Debounce_Cnt, Reg_B
   BRNE    End_Debounce
   STS      New_In, Reg_A              ; Zähler hat bis 0 gezählt. Der neue Wert ist gültig
End_Debounce:
RET
```

Es gibt in diesem kleinen Programm eigentlich drei Bereiche:

Der Zähler ist auf 0, es gibt keine neuen Änderungen am Port und das Programm wird verlassen.

Der Zähler ist nicht 0, dann wird der aktuell eingelesene Wert mit einem hinterlegten verglichen. Bei einem Unterschied wird der Zähler neu gesetzt und das Programm verlassen.

Ist aktuell gelesener Wert und hinterlegter Wert gleich, wird bei jedem Durchlauf der Zähler herunter gezählt. Erreicht er die 0, ist der Eingang stabil und der aktuell gelesene Wert kann an die Variable New_In übergeben werden. Wie ihr seht, wird auch diese Routine nur

durchlaufen. Es gibt keine Wartezeiten, und damit bleibt das Programm immer bedienbar.

Der Aufruf wird direkt hinter Read_IO in die Programmschleife eingefügt.

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO              ; Eingänge lesen
   RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit          ; Bearbeiten „V“
   RCALL    Blinker              ; Blinkerbit bilden
   RCALL    IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL    Taster_Event         ; Bearbeiten
   RCALL    Write_IO             ; und ausgeben
RJMP Loop
```

Obwohl hier eine Wartezeit programmiert ist, bleibt die Programmschleife weiterhin in Betrieb. Die IO ist bedienbar und reagiert auch sofort. Allerdings werden die Tests keine zufrieden stellende Ergebnisse liefern. Je nach verwendeten Tastern ist sogar noch bei einem Maximalwert von 255 gelegentlich ein Prellen bemerkbar. Das liegt daran, das mit der Zykluszeit keine gleichmäßige Wartezeit generiert werden kann.

### *2.8.2 Programm Eingänge entprellt*

```
;*********************************************************************
;*                    Datum 06.08.2012                               *
;*            Beispielanwendungen mit Atmega 8                       *
;* ------------------------------------------------------------------ *
;*      Ein Experimente mit PC und Mikrocontroller                   *
;*                                                                   *
;*  Die Routinen für den Atmega8 beinhalten:                         *
;*     Standart  IO                                                  *
;*     Signalerfassung von Taster                                    *
;*    Ansteuerung von LED                                            *.
;*  Copyright by Martin Vogel   Germany                              *
;*********************************************************************
.NoList
.include   "m8def.inc"      ; Definitionen für ATMEGA8
.List

.DEF Access      = R0      ; Speicherzugriffsregister
.DEF Zero        = R1      ; Register für Null-Vergleich
.DEF Zehn        = R2      ; Register für Vergleich mit konst. 10
.DEF Zwei        = R3      ; Zeiterfassung Minute
.DEF fuenf       = R4      ; Register für Vergleich mit konst. 60
.DEF hundert     = R5      ; Register für Vergleich mit kont. 100
.DEF MilliSek    = R6      ; Register für mSek ( schneller Zugriff)
.DEF ZehnMilli  = R7      ; Register für hundertstel Sek. ( schneller Zugriff)
.Def Zehntel     = R8      ; Register für  Zehntelsekunde
.DEF Ablage_A   = R9      ; Zwischenspeicher A
.DEF Ablage_B   = R10     ; Zwischenspeicher B
.DEF Count_L     = R11     ; Counter Low 0-7
.DEF Count_H     = R12     ; Counter High 8-15
.DEF Count_HL  = R13     ; Counter HighLow 16-23
.DEF Count_HH  = R14     ; Counter High High 24-31
.Def SichSREG    = R15     ; Zwischenspeicher f. Statusregister
.DEF Reg_A       = R16     ; Universalregister A
.DEF Reg_B       = R17     ; Universalregister B
.DEF Reg_C       = R18     ; Universalregister C
.DEF Reg_D       = R19     ; Universalregister D
.DEF Reg_E       = R20     ; Universalregister E
.DEF Reg_F       = R21     ; Universalregister E
.DEF Work        = R22     ; Arbeitsregister
.DEF Send_Byte  = R23     ; Datenübertragung
.DEF Calc_A      = R24     ; Math.-Register A
```

```
.DEF Calc_B        = R25      ; Math. Register B
; ---- Register 26- 31 sind bereits vergeben ----
; X = XL + XH = R 26 + R 27
; Y = YL + YH = R 28 + R 29
; Z = ZL + ZH = R 30 + R 31
.DSEG
Trigger_In:        .Byte 1      ; Byte Debug-Variable Eingang
Trigger_Out:       .Byte 1      ; Byte Debug-Variable Ausgang

;--------------------- Variablen IO-Ebene ----------------------
New_In:            .Byte 1  ; Byte Ablage Eingänge
                            ; Bit 0 = Taster 1 Mode
                            ; Bit 1 = Taster 2 Ziffer
                            ; Bit 2 = Taster 3 auf
                            ; Bit 3 = Taster 4 ab
                            ; Bit 4 = Gabellichtschranke
                            ; Bit 5 = res
                            ; Bit 6 = res.
                            ; Bit 7 = res.

Out_Ctrl:          .Byte 1  ; Byte Ablage für Ausgänge
                           ; Bit 0 = Relais 1
                           ; Bit 1 = Relais 2
                           ; Bit 2 = Relais 3
                           ; Bit 3 = Relais 4
                           ; Bit 4 = Relais 5
                           ; Bit 5 = LED 1
                           ; Bit 6 = LED 2
                           ; Bit 7 = LED 3
Counter_0:         .Byte 1      ; 16 Bit-Zähler LowByte
Counter_1:         .Byte 1      ; 16 Bit-Zähler HighByte
old_In:            .Byte 1      ; Byte Ablage zuletzt gelesene Eingänge
                           ; Bit 0 = Taster 1 Mode
                           ; Bit 1 = Taster 2 Ziffer
                           ; Bit 2 = Taster 3 auf
                           ; Bit 3 = Taster 4 ab
                           ; Bit 4 = Gabellichtschranke
                           ; Bit 5 = res
                           ; Bit 6 = res.
                           ; Bit 7 = res.
Event_To_High:     .Byte 1      ; Byte Ablage Eingänge
                           ; Bit 0 = Taster 1 Mode
                           ; Bit 1 = Taster 2 Ziffer
                           ; Bit 2 = Taster 3 auf
                           ; Bit 3 = Taster 4 ab
```

```
                        ; Bit 4 = Gabellichtschranke
                        ; Bit 5 = res
                        ; Bit 6 = res.
                        ; Bit 7 = res.

Event_To_Low:   .Byte 1     ; Byte Ablage Eingänge
                        ; Bit 0 = Taster 1 Mode
                        ; Bit 1 = Taster 2 Ziffer
                        ; Bit 2 = Taster 3 auf
                        ; Bit 3 = Taster 4 ab
                        ; Bit 4 = Gabellichtschranke
                        ; Bit 5 = res
                        ; Bit 6 = res.
                        ; Bit 7 = res.
Akt_In:             .Byte 1         ; Aktuell gelesener Portwert
In_Debounce:    .Byte 1         ; zuletzt gelesener Portwert
Debounce_Cnt: .Byte 1         ; Zähler für Wartezeit

.CSEG
.ORG 0000
Reset_Point:   RJMP  Start    ; Start von hier ist Neustart
;*****************************************************************
;*          Bereich der Interrupt Vector Table              *
;*****************************************************************
          ; Sprungbefehle zu den Serviceroutinen

;*****************************************************************
;*                  Bereich Initialisierung                     *
;*****************************************************************
.ORG    INT_VECTORS_SIZE      ; Setzt die Adressmarke Start hinter die IVT

Start:
  LDI     Reg_A, High(RamEnd)   ; erste Maßnahme Stack setzen
  Out     SPH, Reg_A
  LDI     Reg_A, Low(RamEnd)
  OUT      SPL, Reg_A
                                ; weitere Initialisierungen
  RCALL   Init_IO               ; Initialisierung IO
; Initialisierung UART
; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
; Initialisierung Variablen ( Eintragen Defaultwerte)
; SEI                           ; globale Freigabe von Interrupts
```

```
;*****************************************************************
;*            Programmschleife Hauptprogramm            *
;*****************************************************************
Main_Loop:
            ;  Hauptprogramm
            ; Aufruf der Subroutinen
  RCALL    Read_IO
  RCALL    IO_Debounce
 ; RCALL    Set_LED_Bit
  RCALL    Blinker
  RCALL    IO_Event_To_1     ; Ereignisbits bilden. Ergebnis in Event_To_High
  RCALL    Taster_Event      ; Bearbeiten
  RCALL    Write_IO
RJMP  Main_Loop
;-----------------------------------------------------------------
;*****************************************************************
;                        Subroutinen                            *
;*****************************************************************

Init_IO:                    ; Ein-Ausgaben initialisieren
;*******************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf       *
;*   Dabei werden nur die benötigten Portbits parametriert,       *
;*   die anderen in ihrer Vorgabe belassen.                       *
;*******************************************************************************
;*******************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf       *
;*   Dabei werden nur die benötigten Portbits parametriert,       *
;*   die anderen in ihrer Vorgabe belassen.                       *
;*******************************************************************************
   IN     Reg_A, DDRD       ; Port D Bits 0-1 = Serielle Schnittstelle
                            ; Bit 2 Eingang Taster 1
                            ; Bit 3 Eingang Taster 2
                            ; Bit 4 Eingang Taster 3
                            ; Bit 5 Eingang Taster 4
                            ; Bit 6 Eingang Gabellichtschranken
                            ; Bit 7 Ausgang Relais
   ANDI   Reg_A, 0b00000011       ; Bit 0 und 1 nicht verändern
   ORI    Reg_A, 0b10000000       ; Bit 7 auf Ausgang
   OUT    DDRD, Reg_A             ; Data Direction Register PortD beschreiben
   IN     Reg_A, PortD
   ORI    Reg_A,0b01111100        ; PortD Pull-Up für die Eingänge einschalten
                            ; Die anderen Portbits bleiben unverändert
   OUT    PortD, Reg_A            ; Port mit Inhalt von Register A beschreiben
   IN     Reg_A, DDRC             ; Data Direction Register lesen
```

```
  ORI    Reg_A,0b00111110       ; Port C Bits 1-5 auf Ausgang schalten
  ANDI   Reg_A, 0b11111110      ; Port C Bits ist Eingang
                                ; Die anderen Bits bleiben unverändert
  OUT    DDRC, Reg_A            ; Data Direction Register Port C beschreiben
  In     Reg_A, PortC
  ORI    Reg_A, 0b00000001      ; Pull_Up Widerstand PC0 einschalten
  Out    PortC, Reg_A
  In     Reg_A, DDRB            ; Data Direction Register Port B lesen
  ORI    Reg_A,0b00111111       ; Portbit 0-5 Ausgang
  Out    DDRB, Reg_A            ; Data Direction Register B beschreiben
RET
;-----------------------------------------------------------------------------

;********** Lesen aller Eingänge und Signalanpassung *************
;*    Port D                                                      *
;*      Bit 0 und 1    serielle Verbindung                        *
;*      Bit 2 bis 5    Eingänge Taster                            *
;*      Bit 6          Eingang Gabellichtschranke                 *
;*      Bit 7          Ausgang Anzeige                            *
;*****************************************************************************
Read_IO:                        ; Eingänge einlesen
  In     Reg_A, PInD            ; Port B lesen
  COM    Reg_A                  ; Bits drehen
  ANDI   Reg_A, 0b01111100      ; Nur Eingänge übernehmen
  LSR    Reg_A                  ; nach rechts schieben (00111110)
  LSR    Reg_A                  ; nach rechts schieben (00011111)
  STS    New_In, Reg_A
RET
;-----------------------------------------------------------------------------

;*****************************************************************************
;*                    Entprellen von Eingängen                               *
;*****************************************************************************
IO_Debounce:
  LDS    Reg_A, Debounce_Cnt
  CPI    Reg_A, 0               ; Vergleich ist wichtig, Ladeanweisung setzt kein Zerobit.
  BREQ   End_Debounce           ; Wenn zähler auf 0, keine weitere Aktion
  LDS    Reg_A, Akt_In          ; Aktuellen Portwert laden
  LDS    Reg_B, In_Debounce     ; Vergleichswert laden
  EOR    Reg_B, Reg_A           ; Ergebnis in Register B, Register A nicht verändern
  BREQ   Chk_Debounce_Time      ; Wenn beide gleich, Prellzeit prüfen
  STS    In_Debounce, Reg_A     ; aktuellen Wert für nächsten Vergleich ablegen
  LDI    Reg_A, 50              ; Überwachungszeit neu setzen  (max.255)
```

```
  STS      Debounce_Cnt, Reg_A     ; und in Zähler eintragen
  RJMP    End_Debounce
Chk_Debounce_Time:
  LDS      Reg_B, Debounce_Cnt      ; Register A hat noch den Wert vom gelesenen Port
  Dec      Reg_B
  STS      Debounce_Cnt, Reg_B
  BRNE   End_Debounce
  STS     New_In, Reg_A      ; Zähler hat bis 0 gezählt. Der neue Wert ist gültig
End_Debounce:
RET
;-----------------------------------------------------------------


;*****************************************************************************
;*     Ereignis Signalwechsel nach "1" am Eingabe-Port.        *
;*****************************************************************************
IO_Event_To_1:
  LDS    Reg_A, Old_In      ; letzter gelesener Wert der Eingänge
  LDS    Reg_B, New_In      ; neu gelesener Wert der Eingänge
  EOR   Reg_A, Reg_B        ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
  AND   Reg_A, Reg_B        ; eine Und Verknüpfung mit dem neuen Status
  STS    Event_To_High, Reg_A  ; liefert die Bits mit Änderung von "0" nach "1"
  STS    Old_In, Reg_B        ; Neuen Wert in Ablage "old_In" kopieren
RET
;-----------------------------------------------------------------------------

;*****************************************************************************
;*        Ereignis Signalwechsel nach 0 am Eingabe-Port.        *
;*****************************************************************************
IO_Event_To_0:
  LDS   Reg_B, Old_In       ; letzter gelesener Wert der Eingänge
  LDS   Reg_A, New_In       ; neu gelesener Wert der Eingänge
  STS   Old_In, Reg_A       ; Neuen Wert in Ablage "old_In" kopieren,
; ist hier bereits erforderlich, da Reg_A überschrieben wird
  EOR   Reg_A, Reg_B        ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
  AND   Reg_A, Reg_B        ; Und Verküpfung mit dem alten Status
  STS   Event_To_LOW, Reg_A     ; liefert die Bits mit Änderung von "1" nach "0"
RET
;-------------------------------------------------------------------

;*********************************************************************
;*                        Tastersignal prüfen                        *
;*********************************************************************
Taster_Event:
  LDS      Reg_A, Event_To_High      ; Ereignisbits laden
```

```
  ANDI    Reg_A, 0b00000001          ; Nur Bit 1 prüfen (Ergebnis ist nicht 0)
  BREQ    End_Taster_Event          ; Ergebnis 0, dann Sprung nach Marke Status_Ende
  LDS     Reg_A, Out_Ctrl           ; Nun das Byte für den Ausgang holen
  LDI     Reg_B, 0b00000010         ; Ist für eine EOR Anweisung erforderlich
  EOR     Reg_A, Reg_B              ; dreht das Bit 1 um, 1 nach 0 und 0 nach 1
  STS     Out_Ctrl, Reg_A      ; nun noch das Register B in die Variable Out_Ctrl schreiben
  LDS     Reg_A, Event_To_High      ; Ereignisbits laden
  ANDI    Reg_A, 0b11111110          ; Ereignisbit 0 löschen
  STS     Event_To_High, Reg_A      ; Und Ereignisse zurückschreiben
End_Taster_Event:                   ; Markiert das Ende der Subroutine
RET
;-----------------------------------------------------------------------------

;********** Eingabe bearbeiten und Ausgabe vorbereiten ************
;*     Variable  New_In enthält Status der Eingänge                    *
;*                                                                       *
;*     Variable Out_Ctrl enthält Ergebnis aus der Bearbeitung      *
;*     für die Ausgabe                                                   *
;*****************************************************************************
Set_LED_Bit:                        ; LED einschalten
  LDS     Reg_A, New_In
  ANDI    Reg_A, 0b00000001         ; Nur Bit 1 prüfen (Ergebnis ist nicht "0")
  BRNE    Set_LED1                  ; Bit= 1, dann  Sprung nach Marke Set_LED1
  LDS     Reg_B, Out_Ctrl           ; Nun das Byte für den Ausgang holen
  ANDI    Reg_B, 0b11011111         ; Mit einer Und-Anweisung das Bit 5 löschen
  RJMP    End_Set_LED_Bit           ; Sprung zum Ende der Sub-Routine
Set_LED1:                           ; Einstieg zum Setzen des Bits
  SWAP  Reg_A                       ; tauscht die unteren 4 Bits mit den oberen 4 Bits (Nibble)
  ROL   Reg_A                       ; Schiebt den Inhalt von Register A noch einmal nach links
  LDS   Reg_B, Out_Ctrl             ; Nun das Byte für den Ausgang holen
  OR    Reg_B, Reg_A                ; und das 5. Bit in Register B setzen
End_Set_LED_Bit:                    ; Markiert das Ende der Subroutine
  STS   Out_Ctrl, Reg_B             ; nun noch das Register B in die Variable Out_Ctrl schreiben
RET                                 ; Und Rücksprung zum Aufruf
;-----------------------------------------------------------------

;*******************************************************************
;*                    Eine LED blinken lassen                      *
;*******************************************************************

Blinker:
  LDS     Reg_A,Counter_0           ;inneren Schleifenzähler
  INC     Reg_A
```

```
  STS     Counter_0, Reg_A
  CPI     Reg_A, 100            ; bis 100 zählen, dann von vorn
  BRLO    End_Blinker
  CLR     Reg_A
  STS     Counter_0, Reg_A      ; mit 100 multiplizieren
  LDS     Reg_A, Counter_1
  INC     Reg_A
  STS     Counter_1, Reg_A
  CPI     Reg_A, 100
  BRLO    End_Blinker
  CLR     Reg_A                 ; bei 10000 wieder von vorn anfangen
  STS     Counter_1, Reg_A
  LDS     Reg_A, Out_Ctrl
  LDI     Reg_B, 0b00100000
  EOR     Reg_A, Reg_B
  STS     Out_Ctrl, Reg_A
End_Blinker:
RET
;-----------------------------------------------------------------

;*****************************************************************
;*                Ausgabe an die IO-Ebene                        *
;*              Port C Bit 1- 5 = Relais / LED                   *
;*                    ab jetzt geändert !                        *
;*****************************************************************
Write_IO:
  IN      Reg_B, PinC           ; gesamten Port C lesen
  ANDI    Reg_B, 0b11000001     ; alle Ausgänge auf 0
  LDS     Reg_A, Out_Ctrl       ; Variable für Ausgang holen
  ANDI    Reg_A, 0b00111110     ; nur Bit 1-5 gültig
  OR      Reg_B, Reg_A          ; Ausgänge zuschalten, (1 in Out_Ctrl)
  OUT     PortC, Reg_B          ; Port mit Reg_C beschreiben
RET
;-----------------------------------------------------------------


;*****************************************************************
;*           Bereich der Interrupt Serviece Routinen             *
;*****************************************************************
```

## 2.9 Laufzeit berechnen

Wollen wir nun wissen, wie lange so ein Eingang stabil bleiben muss, damit er gültig wird, müssen wir die Befehlsfolge untersuchen und ein wenig rechnen.

Unser Controller ist immer noch im Auslieferzustand, das heißt, er arbeitet mit 1 MHz Taktfrequenz. Ein Befehl nimmt im Durchschnitt zwei Takte in Anspruch. Na, nun fangen wir mal an mit Erbsenzählen...

Die Programmschleife hat 6 Befehle, denn **RJMP Loop** müssen wir mitrechnen

**Read_IO** hat 8 Befehle

**IO_Debounce** hat je nach Verlauf bis zu 12 Befehle

**Event_To_High** hat 7 Befehle

**Taster_Status** hat bis zu 11 Befehle

**Write_IO** hat bis zu 7 Befehle,

das sind insgesamt bis zu 51 Befehle und somit ca. 102 Takte.

Ein Takt ist 1/1000000 Sekunden.

Also dauert ein Programmdurchlauf ca. 102 µSek. Bei 50 Programmdurchläufen würde ein gültiger Eingang nach ca. 5 mSek. Verfügbar sein.

Allerdings ist unser Programm ja noch im Aufbau und man kann davon ausgehen, dass noch ein paar Befehle eingefügt werden. Besteht da nicht die Gefahr, das die Entprellzeit zu lange dauert und der Controller auf kurze Tasterbetätigung unter Umständen nicht reagiert ?

Natürlich besteht diese Gefahr. Es werden ja auch noch Programmteile eingefügt, die zum Beispiel in einer Schleife 10 oder gar 20 Werte umkopieren. Daraus schließen wir, das es andere Maßnahmen geben muss, denn die Vorgabezahl in der Variablen **Debounce_Cnt** zu

verringern, kann auch tückisch sein. Ein Programm durchläuft nicht alle Befehle und so kann die Programmzykluszeit mal kurz und mal lang sein.

Doch bevor wir die Lösung im **Timer** finden, möchte ich noch etwas anderes angehen.

Es ist nun Zeit, dass wir bei der Programmierung und den Tests nicht mehr auf ein paar behelfsmäßige LED zugreifen müssen. Auch wenn nun ein schwieriges Kapitel der seriellen Datenübertragung ansteht und es sicherlich noch einfache Funktionen zu besprechen gibt, wir haben **Open_Eye** speziell für Programmentwicklung mit µC geschrieben. Also sollten wir nun den Weg öffnen und die Kommunikation aufbauen.

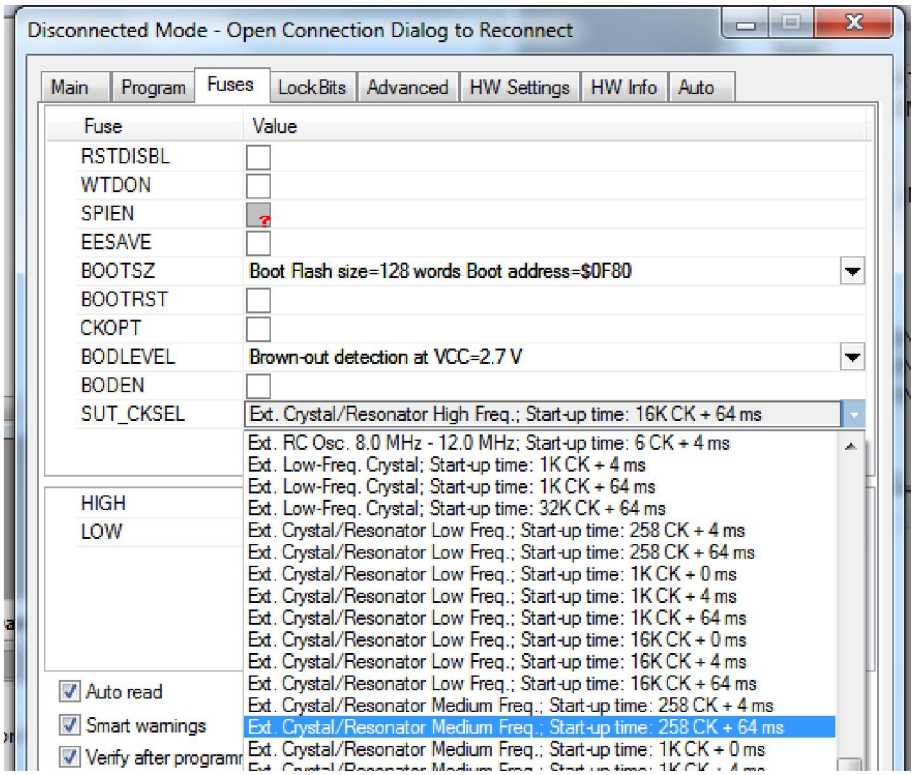
## 2.10 Fuse Bits für externen Quarz setzen

Die Autoren verweisen an dieser Stelle immer, und das nicht ohne Grund, auf die Verwendung eines externen Quarzes oder Taktgebers. Die Frequenz des internen Taktgebers ist nicht stabil genug. Mag sein, dass mit einer Kalibrierung ein etwas besseres Verhalten hinbekommt, aber bei einem Temperaturwechsel kann es dann schon wieder vorbei sein. Wir möchten schon da ein wenig Verlass auf die Hardware haben.

Daher werde ich nun den µC an einen externen Quarz binden. Der Atmega8 kann lt. Datenblatt mit 16 Mhz getaktet werden. Auf dem Atmel Evaluationsboard ist ein solcher verbaut, bei anderen Schaltungen muss dies überprüft werden. Es ist sehr wichtig, das ein Quarz am Controller angeschlossen ist, denn sonst ist er nach der Programmierung der Fuse-Bits nicht mehr erreichbar. Fehlt ihm der Takt nach dem Setzen der entsprechenden Fuses, dann kann er nicht mehr arbeiten. Man hat den Eindruck, der Controller ist defekt und daher ist hier äußerste Genauigkeit in der Vorgehensweise angesagt.

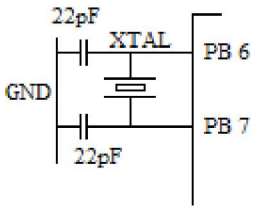
Wer sich mit PonyProg abplagt, sollte genauestens die Anweisungen befolgen. Ich verweise daher auf die Tutorials in **Mikrokontroller.NET**. Sie beschreiben die Thematik ausführlich.

In AVR Studio ist die Einstellung ungleich einfacher. Einfach bei verbundenem Controller die Menüwahl **Tools → Programm AVR → Auto Connect** aufrufen und man erhält die Seite **Fuse**. Bevor irgendetwas an den Einstellungen geändert wird, müssen unbedingt die Fuses gelesen werden. Erst dann sollte man in der Spalte **SUT_CKSEL** die Einstellung vornehmen.




*Einstellung Fuse für externen Quarz*

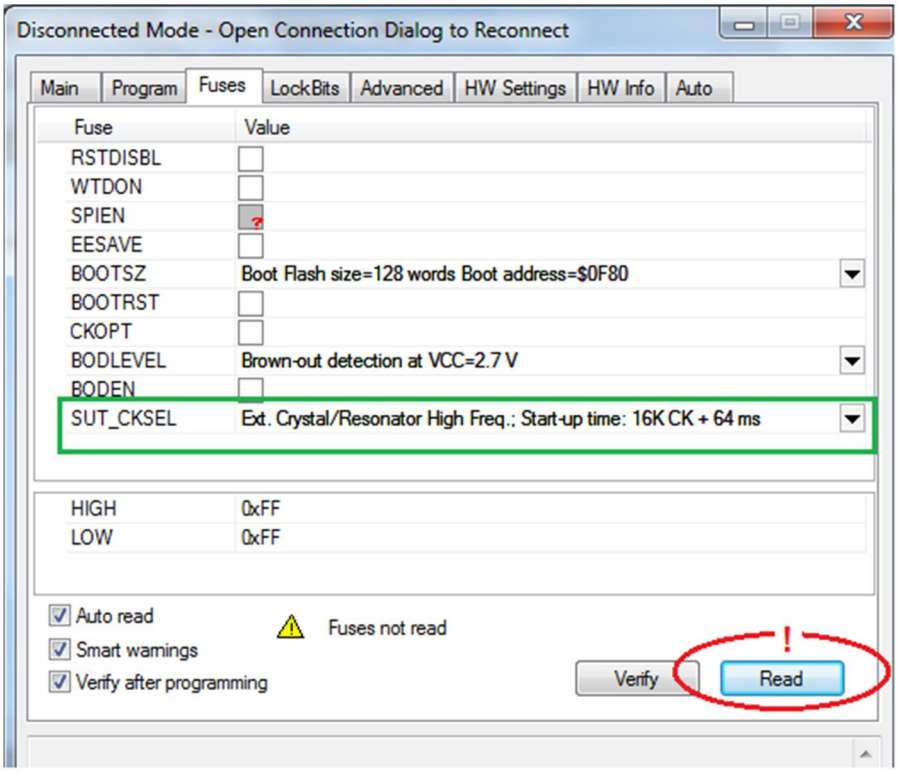
Ist diese Einstellung ausgewählt, muss der Controller eine Beschaltung mit einem Quarz besitzen. Der Quarz ist beim Atmega8 an den Portpins PB6 und PB 7 vorzunehmen. Hier ein die Skizze der Beschaltung.




*Beschaltung mit  Quarz*

Erst, wenn sichergestellt ist, das der Controller mit einem Quarz beschaltet ist, kann diese Änderung in den Controller übertragen werden.

Diesen Hinweis kann man nicht oft genug geben!




*FuseBits setzen*

Wenn wir nun einen Quarz als Taktquelle haben, können wir auch an eine serielle Kommunikation denken.

## 2.11 Serielle Kommunikation einrichten

Zur Bestimmung der Baudrate hilft und der Compiler. Er ist in der Lage, ein paar einfache Anweisungen durchzuführen. Im Assembler-Tutorial bei Mikrocontroller.Net ist eine kleine Hilfe aufgezeigt, die es ermöglicht, die gewünschte Baudrate mit dem aktuellen Systemtakt auf Fehler zu prüfen und ein paar Werte bereit stellt, die wir bei der Initialisierung brauchen.

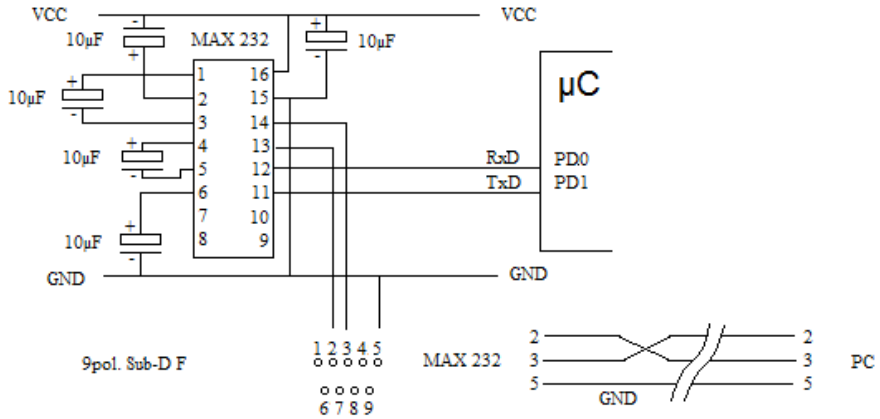
Auszug einer Assembler-Direktive aus dem AVR-Tutorial „UART“ von Mikrocontroller.net

```
;*******************************************************************************
;*    Die folgenden Zeilen definieren ein paar Constanten.          *
;*   Diese Werte werden nur einmal vom Compiler zugewiesen.   *
;*   Danach werden sie nicht mehr verändert.                          *
;*******************************************************************************

.EQU F_CPU = 16000000                ; Systemtakt in Hz ( im Pollin Board 16 MHz)
.EQU BAUD = 9600                     ; Baudrate ; Berechnungen
.EQU UBRR_VAL = ((F_CPU+BAUD*8)/(BAUD*16)-1)            ; runden
.EQU BAUD_REAL = (F_CPU/(16*(UBRR_VAL+1)))               ; Reale Baudrate
.EQU BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)      ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10)) ; max. +/-10 Promille Fehler
.error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu hoch!"
.endif
```

Nun erweitern wir die Hardware mit einer seriellen Schnittstelle. Wer das Atmel Evaluationsboard hat, besitzt bereits einen seriellen Anschluss. Ist aber keine Schnittstelle vorhanden, muss der **Atmega8** entsprechend beschaltet werden.

Der Atmega8 / Atmega16 hat zwei Portpins mit der Bezeichnung **TxD** und **RxD**. Es handelt sich um die Portpins **PD0** und **PD1**. Es reicht aber nicht, diese einfach mit dem PC zu verbinden, weil ja ein Controller an seinen Portpins gerade einmal etwas weniger als **VCC** abgeben kann. Eine **RS 232** arbeitet aber mit +12V und -12 V, um eine höhere Signaldifferenz zu bekommen. Und dafür benötigen wir einen Pegelumsetzer. Diese gibt es für verschiedene Schnittstellen. Wir wollen eine **RS 232** und dafür gibt es den Baustein **MAX 232**.

Der Schaltplan für eine solche Pegelanpassung ist auch nichts Besonderes:




*Schaltung MAX 232*

Die Elkos können auch mit **1µF** beschaltet sein. Es ist vielleicht ungewöhnlich, **das Elkos mit „+“ an GND angeschlossen werden**, doch das ist für die negative Pegelgewinnung wichtig. Daher ist beim Bestücken exakt dieser Schaltplan einzuhalten. Eine Suchanfrage im Internet wird dies bestätigen.

Das Verbindungskabel zum PC benötigt nur 3 Adern, **GND** und die beiden Signalleitungen. Es ist darauf zu achten, wenn es selbst angefertigt wird, dass die beiden Signalleitungen gekreuzt werden müssen.

Wer nur **USB**-Anschlüsse an seinem PC besitzt, kann hier einen **USB-RS 232 Wandler** einsetzen.

### *2.11.1 Fehlersuche*

An dieser Stelle entrümpeln wir erst einmal den Blinker, da er möglicherweise zur Störungssuche eingesetzt werden muss. Noch haben wir die Verbindung zu Open_Eye nicht und so ganz ohne Hilfsmittel ein nicht funktionierendes Programm untersuchen ist alles andere als sinnvoll. Der Blinker wird nun so eingerichtet, das er an jeder Programmstelle aufgerufen werden kann, also auch in einer ISR und bei jedem Durchlauf soll die LED umgesteuert werden.

```
;*****************************************************************
;*                    LED zur Fehlersuche nutzen                 *
;*****************************************************************
Blinker:
  Push Reg_A          ; sichern für den Einsatz in ISR
  Push Reg_B
  IN   SichSREg, SREG        ; Statusregister auch sichern, aber nicht auf Stack
  LDS  Reg_A, Out_Ctrl
  LDI  Reg_B, 0b00100000
  EOR  Reg_A, Reg_B          ; verändert das Statusregister
  STS  Out_Ctrl, Reg_A
  OUT  SREG, SichSReg
  POP  Reg_B
  POP  Reg_A
RET
```

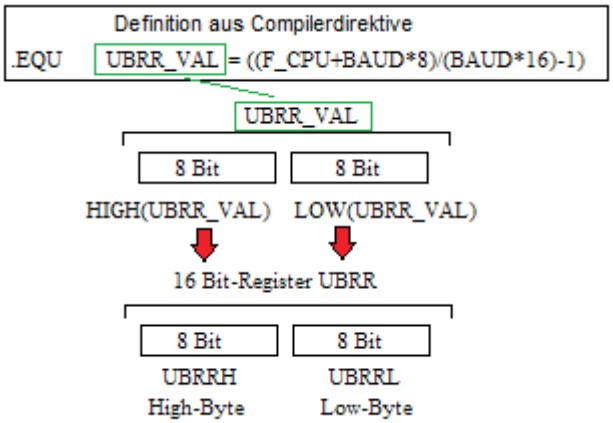
Beim Einsatz dieses kleinen Programms bedenkt aber, das die LED immer wechselt. Wird sie mehrfach aufgerufen, ist bei einem gradzahligen Aufruf nichts zu sehen. Ein Beispiel wird es später verdeutlichen.

### *2.11.2 Die Initialisierung des USART*

Damit ist die erforderliche Hardware am µC vollständig. Die beiden Portpins PD 0 und PD 1 sind direkt mit dem USART verbunden und müssen nun im Controller parametriert werden. Dazu folgt eine kleine Subroutine. Hier wird auch auf den ermittelten Wert in **UBRR_VAL** zugegriffen. Erinnern wir uns: der Wert steckt in der Formel (F_CPU+BAUD*8)/(BAUD*16)-1

```
;------------------- Serielle Schnittstelle parametrieren ----------------
;*****************************************************************************
;*    Die Parametrierung ist mit dem Koppelpartner abzustimmen.  *
;*****************************************************************************
Init_USART:
   LDI Reg_A, HIGH(UBRR_VAL)                    ; Baudrate einstellen
   OUT UBRRH, Reg_A
   LDI Reg_A, LOW(UBRR_VAL)
   OUT UBRRL, Reg_A
                                          ; Übertragungsformat: 8 Daten-, 2 Stoppbit
   LDI REG_A, (1<<URSEL)|(1<<USBS)|(3<<UCSZ0)
   OUT UCSRC,REG_A
                                          ; Empfangen und senden freigeben
   LDI REG_A, (1<<RXEN)|(1<<TXEN)
   OUT UCSRB,REG_A
   SBI UCSRB, RXCIE                         ; RX (Empfang) Interrupt aktivieren
RET
```

In dem Listing von **INIT_USART** wird zuerst in einem 16 Bit-Register die Baudrate angegeben. So ein 16 Bit-Register wird uns noch öfter begegnen und daher werde ich einmal die Art des Zugriffs etwas deutlicher machen.




*Definition UBRR_VAL*

Mathematisch läßt sich diese Formel natürlich auch ausrechnen. Nehmen wir eine Baudrate von 2400 an.

((8000000Hz+2400*8)/(2400*16)-1) =207.833333333

Die Konstante **UBRR_VAL** ist nicht Bestandteil des Programmes, sondern wird nur beim Compilerlauf erzeugt und der Wert zugewiesen. Dieser Wert steht später weder im **DSEG**, **CSEG** oder **ESEG** des Controllers. Aber im Register **UBRR**, welches den Wert zur Bildung der Baudrate beinhaltet, ist der Wert eingetragen.

Betrachten wir einmal die Zuweisung an das USART Control- und Statusregister **UCSRC**. Dieses Register ist 8 Bit groß. Um es zu verstehen, braucht man das Datenblatt des Controllers. Die Bits sind dort einzeln benannt und in ihrer Funktion beschrieben.

**RXEN** Empfang freigeben

**TXEN** Senden freigeben

Kommen wir nun zu den etwas ungewöhnlichen Zuweisungen der Bits an das Register mit der Anweisung

```
LDI REG_A, (1<<URSEL)|(1<<USBS)|(3<<UCSZ0)
```

Dazu werfen wir einmal einen Blick in die Datei **m8der.inc**

```
; UCSRC - USART Control and Status Register C
.equ UCPOL = 0          ; Clock Polarity
.equ UCSZ0 = 1          ; Character Size
.equ UCSZ1 = 2          ; Character Size
.equ USBS = 3           ; Stop Bit Select
.equ UPM0 = 4           ; Parity Mode Bit 0
.equ UPM1 = 5           ; Parity Mode Bit 1
.equ UMSEL = 6          ; USART Mode Select
.equ URSEL = 7          ; Register Select
```

Hier stellen wir fest, dass **URSEL** den Wert 7 hat und die **1** Bit 7 zugeordnet wird

Der Assembler fügt nun die Anweisung folgendermaßen aus. Die Anweisung **LDI** bedeutet, das der Wert direkt als Konstante in das Register eingetragen wird. Aber dazu muss diese Konstante erst zusammengestellt werden. Wie mit einer bitweisen Oder-Verknüpfung setzt der Assembler nun Bit 3 (**USBS**) und durch den Wert 3, das entspricht einer binären Zahl **11**, noch Bit 1 (**UCSZ0**) und Bit 2 (**UCSZ1**). Bei der Zählung von Bits immer daran denken, dass sie bei 0 beginnt.

Dazu ein Blick auf das Register UCSRC, wie es im Datenblatt abgebildet ist.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Register UCSRC | URSEL | UMSEL | UPM1 | UPM0 | USBS | UCSZ1 | UZSZ0 | UCPOL |

*Register UCSRC*

Und so ist es nach der Bearbeitung des Befehles gesetzt.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Register UCSRC | URSEL<br>1 | UMSEL<br>0 | UPM1<br>0 | UPM0<br>0 | USBS<br>1 | UCSZ1<br>1 | UZSZ0<br>1 | UCPOL<br>0 |

*Register UCSRC gesetzt*

Man könnte die Befehlsfolge auch so zusammensetzen:

```
        10000000
oder   00001000
oder   00000110
----------------------
=       10001110
```

Dieser Wert wird dann in das Control- und Statusregister **UCSRC** übertragen.

Eigentlich könnte man auch gleich folgende Anweisungen schreiben und das gleiche bewirken:

```
CLR    Reg_A
ORI    Reg_A, 0b1000110
OUT    UCSRC, Reg_A
```

Der Programmierer zieht allerdings die erstere Schreibweise vor. Zum einen hat er über den Namen einen Bezug zum Bit, zum anderen sind diese Bits auch im Datenblatt so definiert und so sind auch andere Programmierer sofort informiert, was da gemeint ist.

Nun könnte die Frage auftauchen: „Warum werden nicht UCSZ1 und UCSZ0 einzeln zugewiesen oder warum wird nicht gleich bei drei aufeinander folgenden Bits die Zahl 7 eingesetzt“. Klar geht das, aber das widerspricht dem Sinn. UCSZ0 und UCSZ1 sind Bits, die eine Wertigkeit von 0 bis drei besitzen können. Das Datenblatt gibt mit Table 58 darüber Auskunft.

**Table 58.** UCSZ Bits Settings

| UCSZ2 | UCSZ1 | UCSZ0 | Character Size |
|---|---|---|---|
| 0 | 0 | 0 | 5-bit |
| 0 | 0 | 1 | 6-bit |
| 0 | 1 | 0 | 7-bit |
| 0 | 1 | 1 | 8-bit |
| 1 | 0 | 0 | Reserved |
| 1 | 0 | 1 | Reserved |
| 1 | 1 | 0 | Reserved |
| 1 | 1 | 1 | 9-bit |

Befindet sich in Register UCSRB

*UCSZ-Bits im Register UCSRC*

Damit ist klar, die Bits definieren den Rahmen der Datenbits bei der übertragung. Nicht immer sind automatisch 8 Bit Datenlänge eingerichtet. Für uns ist dies erst einmal nicht relevant, denn wir arbeiten mit 8 Bit Daten. Somit sollte auch klar sein, warum diese beiden Bits mit der Zahl 3 zusammengefasst werden können. Das gesetzte **USBS**-Bit hat aber die Bedeutung, das zwei Stoppbits erwartet werden und sollte deshalb auch Mit dieser Einstellung haben wir einen Übertragungsrahmen von:

Asynchroner Betrieb, kein Parity, 8 Datenbit und 2 Stoppbit.

Das Bit **URSEL** muss für den Zugriff auf dieses Register gesetzt sein. Das die Zuweisung an das Control-und Statusregister mit **OUT** erfolgt, liegt in der Art der Adressierung. Einige Register liegen im IO-Bereich und daher ist der Zugriff nur mit OUT oder IN möglich.

Betrachten wir nun die weitere Befehlsfolge

```
LDI REG_A, (1<<RXEN)|(1<<TXEN)
```

Diese beiden Bits befinden sich im Register UCSRB. Auch hier ist schnell klar, wie die Bezeichnungen **RXEN** und **TXEN** zu deuten sind. Ein Blick in die **m8def**-Datei beseitigt letzte Zweifel. Mit **RXEN** kann nur **Freigabe Empfangen** und mit **TXEN Freigabe Senden** gemeint sein.

```
; UCSRB - USART Control and Status Register B
.equ UCR = UCSRB                ; For compatibility
.equ TXB8 = 0                   ; Transmit Data Bit 8
.equ RXB8 = 1                   ; Receive Data Bit 8
.equ UCSZ2 = 2                  ; Character Size
.equ CHR9 = UCSZ2               ; For compatibility
.equ TXEN = 3                   ; Transmitter Enable
.equ RXEN = 4                   ; Receiver Enable
.equ UDRIE = 5                  ; USART Data register Empty Interrupt Enable
.equ TXCIE = 6                  ; TX Complete Interrupt Enable
.equ RXCIE = 7                  ; RX Complete Interrupt Enable
```

Doch noch einmal kurz zurück zur Initialisierung des USART. Es nützt nichts, allein diese Routine zu schreiben, sie muss auch dem Programm beigefügt werden. Dazu tragen wir **im Initialisierungsteil vor** der Programmschleife den Aufruf dieser Routine ein:

```
;*****************************************************************
;*                    Bereich Initialisierung                    *
;*****************************************************************
Start:
LDI     Reg_A, High(RamEnd)          ; erste Maßnahme Stack setzen
Out     SPH, Reg_A
LDI     Reg_A, Low(RamEnd)
OUT SPL, Reg_A
            ; weitere Initialisierungen
RCALL   Init_IO
RCALL   Init_ USART                  ; serielle Schnittstelle parametrieren
; weitere
; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
; Initialisierung Variablen ( Eintragen Defaultwerte)
```

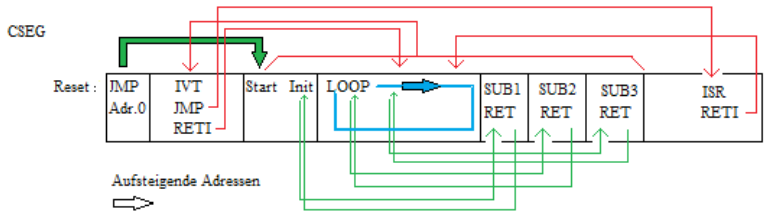
### *2.11.3 Datenempfang mit Interrupt*

Damit ist der USART doch schon einigermaßen eingerichtet. Nun möchten wir aber wissen, wann ein USART Daten empfängt und nicht jedes Mal nachsehen wollen. Viele Anfänger tun dies nicht selten aus unberechigter Angst vor dem Namen Interrupt. Die Programmierart nennt sich pollen. Sinnvoller ist doch, wenn in einem Ereignis angezeigt wird, das Daten eingetroffen sind. Schön, hier von einem Ereignis zu reden, aber zutreffender ist es, dieses Ereignis **Interrupt** zu nennen.

Ein **Interrupt** unterbricht die normale Programmbearbeitung an jeder beliebigen Stelle und setzt den Adresszähler für die Befehle auf eine fest zugewiesene Adresse. Übrigends, der Name Adresszähler wird in diesem Fall mit **Programmcounter** definiert und im Datenblatt mit **PC** abgekürzt.

Diese fest zugewiesene Adresse befindet sich in der **Interrupt Vektor Tabelle (IVT)**. Schon Anfangs hatte ich diese erwähnt. Die Aufgabe dieser Tabelle ist es, genau einen Befehl an der Adresse aufzunehmen. Nun kann hier einfach nur ein **RETI** -also **Return from Interrupt** stehen und die Bearbeitung wird normal fortgesetzt, ohne auf diesen Interrupt irgendwie mit einem Programm zu reagieren oder es steht ein Sprung zu einer Programmmarke, einem sogenannten **Adresspointer**. Dort befindet sich dann ein Unterprogramm, welches dafür sorgt, dass der Interrupt etwas Nützliches ausführt. Diese Art eines Unterprogrammes nennt man **Interrupt Service Routine (ISR).**

In der folgenden Skizze ist der Ablauf aufgezeichnet.




*Ablauf Interrupts*

Das normale Programm folgt den grünen Linien. Am Strtpunkt wird die Interrupt Vektor Tabelle übersprungen und landet nach der Initialisierung in der Programmschleife. Ein Aufruf von einem Unterprogramm, auch grüne Linien, landet nach Bearbeitung hinter dem aufrufenden Befehl.

Hier ist nicht nur die Rücksprungadresse bekannt, sondern auch der Inhalt aller Register und Statusbits.

Bei einem Interrupt ist das anders. Er kann überall im Programm auftreten, nachdem er freigegeben ist. Dann erfolgt der Sprung zur IVT mit zur fest vorgebenen Adresse mit dem nächsten Befehl.

Was bedeutet es für uns ? Ja, wir müssen Vorkehrungen treffen, dass in einer ISR nicht ein Ergebnis verfälscht wird. Dazu ein kleines Beispiel im Ablauf eines normalen Programmes:

```
…....
    LDS      Reg_A, New_In
   ANDI    Reg_A,0b00000001
   ; hier tritt jetzt ein Interrupt auf
    BREQ   Next ; Statusbit richtig ???
….…
```

Dazu die aufgerufene ISR

```
ISR:
   LDS Reg_A, Counter
   Dec Counter
   STS Counter
RETI
```

Es ist nicht sicher, ob das Ergebnis der Und-Verknüpfung und damit das Statusregister richtige Information hat, denn es wird durch den DEC-Befehl beeinflusst. Außerdem ist der Inhalt von Register A überschrieben und somit nicht mehr gültig. Damit ist bei jeder ISR Pflicht, die darin verwendeten Register und zusätzlich das Statusregister **SREG** zu sichern. Und auch wenn **SREG** nicht von uns definiert ist, so ist es ein Register, welches (fast) immer beeinflusst wird. Die Methode ist einfach:

Die Register werden am Anfang einer ISR auf den Stack gelegt und am Ende wieder abgeholt. Dazu dienen die Befehle PUSH und POP. Da das Statusregister im IO-Bereich liegt, muss es erst in ein anderes Register geholt werden.

```
ISR:
   Push Reg_A              ; Inhalt von Register A sichern
```

```
   IN Reg_A, SREG        ; Statusregister in Register A holen
   Push Reg_A            ; und Register A wieder sichern
   LDS Reg_A, Counter
   Dec Counter
   STS Counter
   POP Reg_A             ; Register A mit dem Inhalt vom Statusregister zurückholen
   OUT SREG, Reg_A       ; Status wieder herstellen
   POP Reg_A             ; ursprünglichen Inhalt von Register A zurückholen
RETI
```

Dieses Schema stellt sicher, das keine Werte von Registerinhalten durch einen Interrupt verloren gehen. Dies sollte einleitend zum Einstieg in die Programmierung von Interrupt Service Routinen unbedingt verstanden sein, denn mit dem letzten Befehl in der Initialisierung des USART geben wir den Interrupt beim Empfang frei.

```
SBI UCSRB, RXCIE                ; RX (Empfang) Interrupt aktivieren
```

### 2.11.3.1 *Die Interrupt Service Routine (ISR)*

Bevor wir an die ISR gehen, stelle ich erst einmal die Interrupt Vector Table vor. Beim Einstieg in dieses Programm habe ich diese Tabelle bereits erwähnt.

Ich setze sie immer komplett in mein Programm, da ein vergessener Interrupt sonst in einer falschen ISR oder gar im Programm landet. Es sollte zwar nicht passieren, da alle Interrupts auch freigegeben werden müssen, doch es schadet nicht und belegt auch keinen Speicher. Und vielleicht entdeckt man einem interessanten Interrupt für eine eigene Anwendung.

```
.org INT0addr          ;RJMP EXT_INT0 ; External Interrupt0 Vector Address
RETI
.org INT1addr          ; External Interrupt1 Vector Address
RETI
.org OC2addr           ; Output Compare2 Interrupt Vector Address
RETI
.org OVF2addr          ; Overflow2 Interrupt Vector Address
RETI
.org ICP1addr          ; Input Capture1 Interrupt Vector Address
RETI
.org OC1Aaddr RJMP isrTimer1 ; Einsprungadresse ISR Timer1
;RET I
.org OC1Baddr          ; Output Compare1B Interrupt Vector Address
RETI
.org OVF1addr          ; Overflow1 Interrupt Vector Address
RETI
.org OVF0addr          ; Overflow0 Interrupt Vector Address
RETI
.org SPIaddr                    ; SPI Interrupt Vector Address
RETI
.org URXCaddr RJMP ISR_Rec    ; USART Receive Complete Interrupt Vector Address
;RETI
.org UDREaddr                   ; USART Data Register Empty Interrupt Vector Address
RETI
.org UTXCaddr   ;RJMP USART_TXC    ; USART Transmit Complete Interrupt Vector Address
RETI
.org ADCCaddr                   ; ADC Interrupt Vector Address
RETI
.org ERDYaddr                   ; EEPROM Interrupt Vector Address
RETI
.org ACIaddr                    ; Analog Comparator Interrupt Vector Address
RETI
```

```
.org TWIaddr    ;RJMP TWSI      ; Irq. vector address for Two-Wire Interface
RETI
.org SPMRaddr                   ; SPM complete Interrupt Vector Address
RETI
```

Nicht benützte Interrupts sind erst einmal auskommentiert und somit ist das **RETI** wirksam. Werden diese Interrupts allerdings genutzt, so ist das **RETI** zu entfernen oder einfach mit einem Semikolon unwirksam zu machen.

Werfen wir nun einen Blick auf den Kommentar

```
USART Receive Complete Interrupt Vector Address
```

Die Adresse wird dem Compiler mit

```
.org URXCaddr RJMP ISR_REC
```

mitgeteilt und der Befehl dahinter ist ein Sprung in die ISR. Iim Gegensatz zu anderen Unterprogrammen hat eine ISR keinen Aufruf im Hauptprogramm.

```
;*************************************************************************
;*      ISR Datenempfang über serielle Schnittstelle.        *
;*      erforderliche Variablen:                              *
;*      Ringpuffer      Com_Buff    20 Bytes                  *
;*     Schreizeiger    Write_Pos     1 Byte                  *
;*************************************************************************
ISR_REC:
   PUSH    Reg_A                    ; Register A auf dem Stack sichern
   IN      Reg_A, sreg              ; SREG sichern
   PUSH    Reg_A                    ; Statusbits auf dem Stack sichern
   Push    XL                       ; Adressregister XL sichern
   Push    XH                       ; Adressregister XH sichern
   Push    Work                     ; Arbeitsregister sichern
   IN      Work, UDR                ; USART-Empfangsregister in Arbeitsregister
   LDS     Reg_A, Write_Pos         ; Schreibzeiger in Register A
   LDI     XL,LOW(Com_Buff)         ; -XPointer auf Empfangspuffer
   LDI     XH,HIGH(Com_Buff)
   ADD     XL, Reg_A                ; Schreibzeiger + Anfangsadresse Ringpuffer
   ADC     XH, Zero                 ; 16 Bit-Addition, daher Carrybit addieren (Addition mit 0)
```

```
   ST      X, Work              ; Inhalt Arbeitsregister in Ringpuffer schreiben
   INC     Reg_A                ; Register A mit Inhalt Schreibzeiger erhöhen
   CPI     Reg_A, 20            ; Grenzwert abfragen
   BRLO    End_rxc              ; wenn kleiner, dann ISR beenden
   CLR     Reg_A                ; Schreibzeiger auf 0 setzen
End_rxc:                        ; Ausstieg aus ISR
   STS     Write_Pos, Reg_A     ; Zuerst Schreibzeiger speichern
   POP     Work                 ; Dann alten Wert von Arbeitsregister zurückholen
   POP     XH                   ; X-Register wieder herstellen
   POP     XL
   POP     Reg_A                ; Register A mit Statusbits laden
   OUT     sreg, Reg_A          ; und SREG wieder herstellen
   POP     Reg_A                ; Register A wieder herstellen
RETI
```

Damit dieses Programm auch assemblierfähig ist, müssen wir noch die erforderlichen Variablen nachtragen. Dazu fügen wir bei der Variablendeklaration folgende Zeilen hinzu:

```
Value_Cnt:      .Byte 1         ; Int8  Zähler für angelegte Variablen
Com_Ctrl:       .Byte 1         ; Byte  Statusflags vom Programm
Read_Pos:       .Byte 1         ; Int8   Lesezeiger
Write_Pos:      .Byte 1         ; Int8   Schreibzeiger
Com_Buff:       .Byte 20        ; Ringpuffer Empfang
```

Im Initialisierungabschnitt rufen wir eine weitere kleine Routine auf, die uns die Variablen und Register vorbesetzt.

```
;*****************************************************************************
;*  Neue Initialisierungsroutine. Muß vor der Programmschleife   *
;*  in Initialisierungsteil aufgerufen werden.                   *
;*****************************************************************************
Init_Var:
   CLR Zero                     ; Register „Zero“ auf 0 setzen
   STS Read_Pos, Zero           ; des Lese-
   STS Write_Pos, Zero          ; und des Schreibzeigers
   STS Com_Ctrl, Zero           ; Wichtig, sonst fehlerhafte Kommunikation
   LDS  Reg_A, 37               ; Hier wird der Wert eingetragen, den Open_Eye
   STS  Value_Cnt, Reg_A        ; aus dem Variablenfilter ermittelt hat.
```

```
RET
```

An dieser Stelle ein Hinweis auf die Variable Value_Cnt. Wenn wir den Variablenbereich dieses Assemblerlistings kopieren und in Open_Eye einfügen, ermittelt das Visual Basic Prograsmm die Anzahl der Variablen. Diese sind hier einzutragen, damit die Sendeschleife die Anzahl der Variablen auch kennt und entsprechend viele Bytes verschickt.

```
;*******************************************************************
;*                    Bereich Initialisierung                      *
;*******************************************************************
Start:
   LDI     Reg_A, High(RamEnd)            ; erste Maßnahme Stack setzen
   Out     SPH, Reg_A
   LDI    Reg_A, Low(RamEnd)
   OUT SPL, Reg_A
            ; weitere Initialisierungen
   RCALL    Init_IO
   RCALL    Init_ USART                   ; serielle Schnittstelle parametrieren
; weitere
; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
   RCALL    Init_Var          ;Initialisierung Variablen ( Eintragen Defaultwerte)
```

Der Moment ist nun gekommen, die Verbindung zu testen. Dazu muss aber noch der **SEI**-Befehl im Initialisierungsteil freigegeben werden. Das geschieht einfach durch Löschen des davor stehenden Semikolons. Dieses **SEI** gibt die gesamte Interruptebene frei. Das ist sozusagen der Hauptschalter für die Interrupts.

Dennoch wäre ein Ergebnis nicht sichtbar. Daher werden wir folgendes tun, wir schicken die Inhalte der Variablen zurück, sobald Lese- und Schreibzeiger unterschiedlich sind. Das können wir im laufenden Programm prüfen, denn die ISR setzt den Schreibzeiger auf eine andere Position, wenn ein Zeichen eingetroffen ist.

### *2.11.4 Prüfung Datenempfang*

Die Routine dafür benennen wir erst einmal **Chk_Receive**.

```
;*********************************************
;*        Empfangene Daten auswerten                    *
; *   empfangenes Byte in Variable     Is_Data_Rec      *
;*    ablegen und in Controll-und Steuervariable        *
;*    Com_Ctrl    den Empfang mit Bit 0 anzeigen         *
;*********************************************
Chk_Receive:
    LDS      Reg_A, Read_Pos            ; Lesezeiger holen
    LDS      Reg_B, Write_Pos           ; Schreibzeiger holen
    CP       Reg_A, Reg_B               ; und vergleichen
    BREQ     End_Chk_Receive            ; Wenn gleich, dann keine Daten und Ende Subroutine
;*********************************************
;*        empfangenes Byte in Arbeitsregister              *
;*          und Schreibzeiger nachführen                   *
;*********************************************
      LDI     XL,LOW(Com_Buff)          ; -XPointer auf Empfangspuffer
      LDI     XH,HIGH(Com_Buff)
      ADD     XL, Reg_A          ; Schreibzeiger auf Anfangsadresse Ringpuffer addieren
       ADC      XH, Zero            ; 16 Bit-Addition, daher noch Carrybit addieren (Addition mit 0)
       LD      Work, X            ; Inhalt Arbeitsregister in Ringpuffer schreiben
       INC     Reg_A              ; Register A mit Inhalt Schreibzeiger erhöhen
       CPI     Reg_A, 20          ; Grenzwert abfragen
       BRLO    Set_Read_Pos       ; wenn kleiner, dann Wert in Schreibzeiger ablegen
       CLR     Reg_A              ; Schreibzeiger auf 0 setzen
 Set_Read_Pos:
       STS     Read_Pos, Reg_A            ; Schreibzeiger übernehmen
       STS     Is_Data_Rec, Work          ; Empfangenes Byte in Variable schreiben
       LDS      Reg_B, Com_Ctrl           ; Programmcontrollbits
       LDI     Reg_A, 0b00000001          ; Signalbit setzen
       Orr     Reg_A, Reg_B
       STS     Com_Ctrl, Reg_A            ; und in Control- und Steuervariable eintragen
End_Chr_Receive:
RET
```

Natürlich ist dies noch nicht alles, aber diesen Bereich möchte ich zuerst einmal erklären. Diese kleine Routine stellt lediglich fest, es sind Daten eingetroffen und das Byte, welches der Lesezeiger adressiert, wird zur Prüfung in eine Variable geschrieben. Damit das laufende Programm weiß, es liegen Daten zur Prüfung vor, habe ich ein Byte als Variable Com_Ctrl definiert, welches Signal-oder Jobbits beinhaltet. Nun bekommt

auch die Bearbeitung von seriellen Daten eine Art **Ereignissteuerung**. Das Ereignis wird durch das **Bit 0** in der Variablen **Com_Ctrl** signalisiert. Es zeigt, das ein Byte eingetroffen ist. Es ist selbstverständlich, dass die benutzten Variablen natürlich im Variablenbereich auch deklariert werden müssen.

```
Is_Data_Rec:    .Byte 1          ; Ablage für empfangenes Byte zur Bearbeitung
Com_Ctrl:       .Byte 1          ; Control- und Steuervariable für Datenempfang
```

Es ist zwar nicht erforderlich, nun für jede Teilaufgabe eine eigene Routine zu schreiben, aber es ist durchaus ratsam. Der geringe Zeitverlust in der Programmzykluszeit ist nichts gegen den Gewinn, ein überschaubares und nachträglich anpassbares Programm zu erhalten. Wenn nicht unbedingt die Zykluszeit minimal sein muß, dann ist dies der bessere Weg. Binden wir nun diese Auswertung in die Main_Loop des Programms.

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL   Read_IO          ; Eingänge lesen
   RCAll   IO_Debounce      ; Eingänge entprellen, gültig in Variablen New_In
   RCALL   Set_LED_Bit      ; Bearbeiten „V“
   RCALL   Blinker          ; Blinkerbit bilden
   RCALL   IO_Event_To_1    ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL   Taster_Event     ; Bearbeiten
   RCALL   Chk_Receive      ; Datenempfang prüfen
   RCALL   Write_IO         ; und ausgeben
RJMP Loop
```

Betrachten wir nun einmal unsere Daten, die wir empfangen wollen und die ausgewertet werden müssen. Der einfachste Weg wäre nur ein Byte zu senden und dieses auszuwerten. Immerhin könnte man 255 verschiedene Aufgaben erledigen. Aber das ist nicht gerade übersichtlich und wir möchten doch dahin gehend bestens vorarbeiten. Außerdem, die Trigger würden uns da ein klein wenig stören, denn da sollen ja Bits ausgewertet werden. Meine Überlegung geht da folgenden Weg:

Ich nehme 2 Bytes, die ausgewertet werden müssen. Der Aufbau gestaltet sich dann wie folgt:

1. Empfang von Byte ist mit Bit 0 in Com_Ctrl markiert. Prüfen, ob ein Bit im Bereich 2-7 gesetzt ist, dann ist das das zweite Byte und hier abbrechen.
2. Erstes Byte: Befehl und auswerten und in Com_Ctrl markieren. Bit 0 löschen.
3. Zweites Byte erkannt. Befehl auswerten und Com_Ctrl komplett löschen. Dadurch Bereitschaft für nächsten Befehl herstellen.

Beispiele:

V für Value -> Anforderung der internen Werte in den Variablen. Zweites Byte ohne Bedeutung.

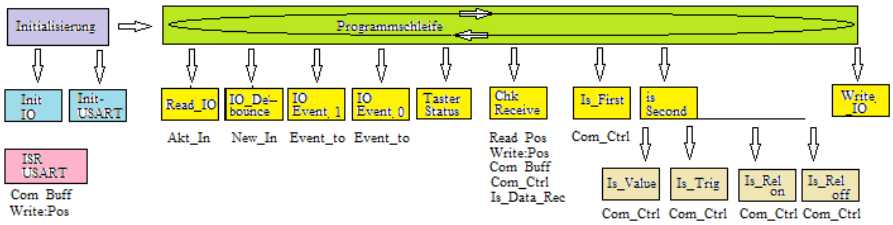
R und r für Relais, ein oder aus. Zweites Byte enthält Relaisnummer

T für Trigger. Zweites Byte enthält Triggerbit, t setzt Triggerbit zurück

Sollte sich kein Erfolg abzeichnen, nun, wir haben die Routine Blinker. Mit **RCALL Blinker** können wir versuchen, Programmteile zu testen. Fügen wir den Aufruf der Routine Blinker doch einmal ein.

```
Chk_Receive:
    LDS     Reg_A, Read_Pos         ; Lesezeiger holen
    LDS     Reg_B, Write_Pos        ; Schreibzeiger holen
    CP      Reg_A, Reg_B            ; und vergleichen
    BREQ    End_Chk_Receive         ; Wenn gleich, dann keine Daten und Ende Subroutine
    RCALL   Blinker                 ; für Testzwecke
;*********************************************
;*      empfangenes Byte in Arbeitsregister          *
;*        und Schreibzeiger nachführen               *
;*********************************************
….
```

Das sind doch schon ein paar Ansätze, die durchaus für eigene Projekte abgewandelt werden können. Nun ist auch klar, das zu dieser Struktur eine besondere Bearbeitung erforderlich ist. Ich nutze einfach die Variable **Com_Ctrl**, um mit weiteren Bits der Anforderung gerecht zu werden.



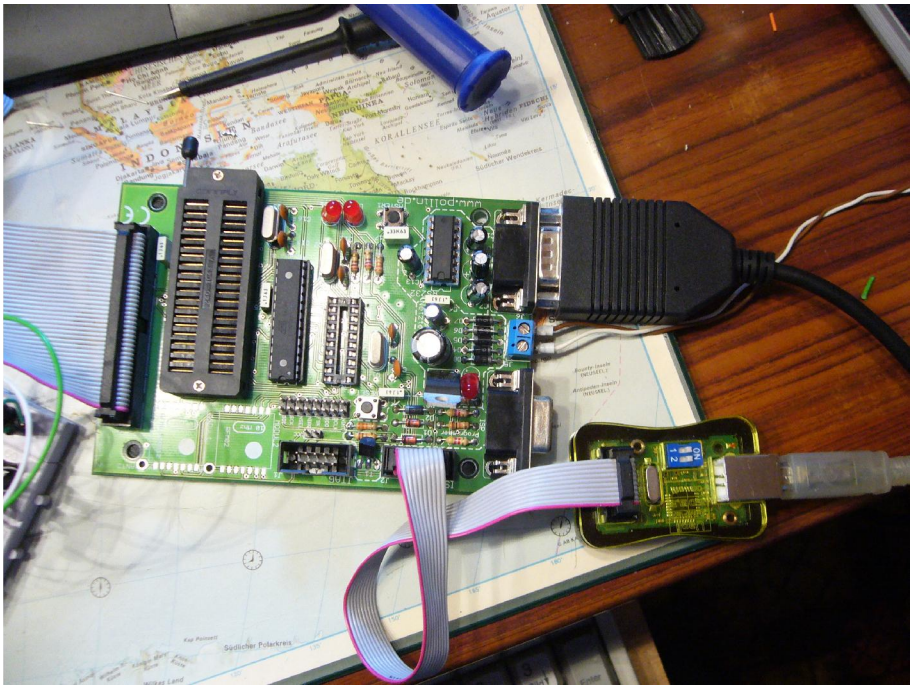

*Blockbild einer Programmstruktur*

Wie auf der Skizze ersichtlich, hat die **ISR** der **USART** keinerlei Verbindung zum Programm. Sie bedient lediglich den Ringpuffer. Das Programm selbst stellt in der Routine **Chk_Receive** einen Unterschied der Schreib- und Lesezeiger fest und kopiert ein empfangenes Byte in eine Arbeitsvariable. Zusätzlich wird der Empfang mit einem gesetzten Bit in der Variablen **Com_Ctrl** angezeigt.

Nun ist es zeit für einen Versuch mit unserem Visual Basic Programm Open_Eye

### *2.11.5 Open_Eye und Mikrocontroller*

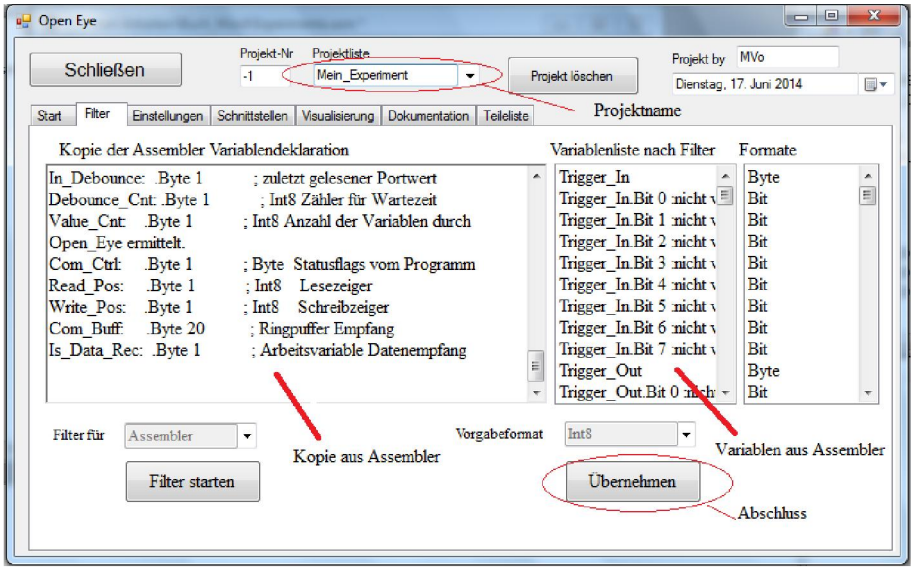
Dazu starten wir nun Open_Eye und verbinden den Controller über die serielle Schnittstelle mit dem PC. Auch ein Seriell-USB-Wandler ist ohne Probleme einsetzbar.



*Verbindung µC mit PC*

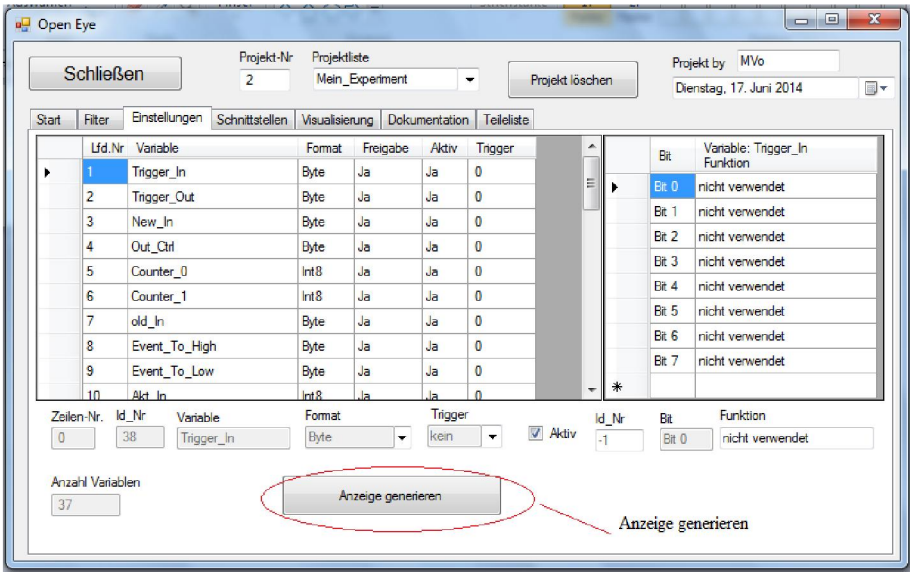
Dann kopieren wir den Variablenbereich im Assemblerprogramm in AVR Studio und übertragen diese Kopie nach Open_Eye in den Filter.

Folgen wir nun einfach den Screenshots.



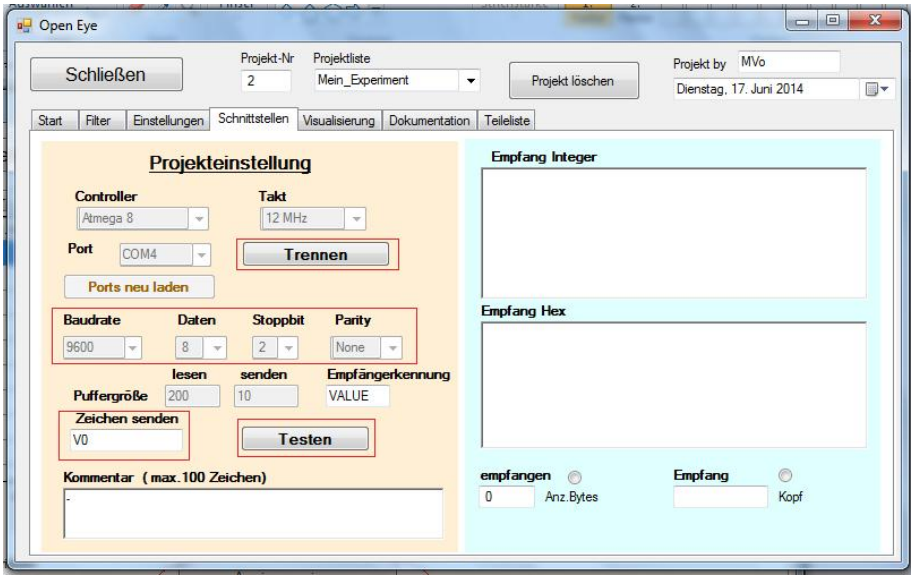

*OpenEye und µC Variablenfilter*

Nach der Filterung vergeben wir einen Namen und übernehmen. Es öffnet sich die Seite Einstellungen.




*OpenEye und µC Bearbeiten Variablen*

Mit dem Button Anzeige generieren geht es gleich weiter und es öffnet sich die Seite für die Einstellung der Schnittstelle.



*OpenEye und µC Schnittstelle*

Solange noch keine Verbindung zum Controller besteht, sind die Schnittstellenparameter einzustellen. Dabei ist die Baudrate, Datenbreite, Stopp- und Paritybit von Bedeutung. Danach kann die Verbindung aktiviert werden. Wenn wir sehen wollen, ob auf der Seite des Controllers ein Byte eingetroffen ist, darf nur ein Zeichen gesendet werden, ob V oder 0 ist egal, aber nicht beide zusammen. Dann ist an der LED nichts zu sehen. Sollte keine Reaktion erfolgen, dann gilt es folgendes zu kontrollieren:

Ist die LED richtig angeschlossen. Nach unserem Programm wird PC5 beschaltet.

Die nächste Kontrolle ist die Prüfung im Controller. Stimmt die eingetragene CPU-Frequenz und sind auch die Fuses für einen externen Takt geschaltet. Passen die restlichen Parameter mit den Parametern in Open_Eye überein.

C:\Daten\Arbeiten\Buch_Word\Experimente.asm *

```
;*****************************************************************
;*    Die folgenden Zeilen definieren ein paar Constanten.       *
;*   Diese Werte werden nur einmal vom Compiler zugewiesen.      *
;*   Danach werden sie nicht mehr verändert.                     *
;*****************************************************************

.EQU F_CPU = 12000000                          ; Systemtakt in H
.EQU BAUD = 9600                               ; Baudrate ; Bere


.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))     ; Reale Baudrat
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Pro
```

C:\Daten\Arbeiten\Buch_Word\Experimente.asm *

```
;-------------- Serielle Schnittstelle parametrieren ----------
;*****************************************************************
;*    Die Parametrierung ist mit dem Koppelpartner abzustimmen.
;*****************************************************************
Init_USART:
  LDI   Reg_A, HIGH(UBRR_VAL)        ; Baudrate einstellen
  OUT UBRRH, Reg_A
  LDI Reg_A, LOW(UBRR_VAL)
  OUT UBRRL, Reg_A
                ; Übertragungsformat: 8 Daten-, 2 Stoppbit
  LDI REG_A, (1<<URSEL)|(1<<USBS)|(3<<UCSZ0)
  OUT UCSRC,REG_A
                ; Empfangen und senden freigeben
  LDI REG_A, (1<<RXEN)|(1<<TXEN)
  OUT UCSRB,REG_A
  SBI UCSRB, RXCIE        ; RX (Empfang) Interrupt aktivieren
RET
```

*Abgleich RS232 Parameter*

Stimmt alles, sollte auch die LED bei jedem Betätigen vom Button Test an- und ausgehen. Damit ist bewiesen, das der PC mit dem Controller kommuniziert. Nun ist der Weg Controller zum PC zu programmieren.

Die Routine Chk_Receive erkennt einen Datenempfang und stellt die Information in der Variablen Is_Data_Rec der weiteren Auswertung zur Verfügung. Zusätzlich wird in einem Bit in Com_Control der Eingang eines unverarbeiteten Bytes angezeigt. Dieser Zwischenschritt ist nicht unbedingt erforderlich, doch so behalten wir den Überblick und müssen uns nnicht durch ellenlange Programmabschnitte kämpfen.

### *2.11.6 Verarbeiten der Information vom PC*

Bis hierhin sind die Routinen geschrieben. Nun folgt eine Auswertung der Information. Die Routine Is_First ist zuständig, den Befehlscode zu entschlüsseln und die Info abzugeben, das ein Befehl erkannt ist. Wenn wir den Befehlen V, R, r, T und t jeweils ein Bit in der Variablen Com_Ctrl zuordnen, brauchen wir keine weiteren Variablen für diesen Job. Im ersten Schritt der Prüfung testen wir das Bit 0, ob eine unverarbeitete Information vorliegt.

```
;*****************************************************************************
;*    Prüfen, ob bereits ein Befehl erkannt ist und                  *
;*    gegebenenfalls ein Steuerbit in „Com_Ctrl“ setzen          *
;*****************************************************************************
Is_First:
      LDS      Reg_A, Com_Ctrl          ; Kontroll- und Steuerbyte serieller Empfang
      ANDI     Reg_A, 0b00000001        ; Ereignis ein Byte eingetroffen und
      BREQ     End_Is_First
      LDS      Reg_A, Com_Ctrl          ; liegt in der Variablen  Is_Data_Rec zur Bearbeitung
      ANDI     Reg_A, 0b11111100        ; Ist ein Bit gesetzt, dann ist dies das zweite Byte
      BRNE     End_Is_First
      LDS      Reg_B, Is_Data_Rec       ; Noch kein Bit im oberen Nibble, Befehl decodieren
      LDI      Reg_A, 0b00000100        ; Bit für „Relais an“ setzen
      CPI      Reg_B, „R“               ; Befehl „Relais ein?
      BREQ     Store_Order              ; ja, dann Befehlsbit ablegen
      LDI      Reg_A, 0b00001000        ; sonst Bit für „Relais aus“ setzen
      CPI      Reg_B, „r“               ; Befehl „Relais aus“?
      BREQ     Store_Order              ; ja, dann Befehlsbit ablegen
      LDI      Reg_A, 0b00010000        ; sonst Bit für Trigger Ein setzen
      CPI      Reg_B, „V“               ; Befehl ist Trigger ?
      BREQ     Store_Order              ; ja, dann Befehlsbit ablegen
      LDI      Reg_A, 0b00100000        ; sonst Bit für Trigger Aus setzen
      CPI      Reg_B, „v“               ; Befehl ist Trigger reset ?
      BREQ     Store_Order              ; ja, dann Befehlsbit ablegen
      LDI      Reg_A, 0b10000000        ; Bit für Werteabfrage der internen Variablen
Store_Order:
      STS      Com_Ctrl, Reg_A
 ; Kontroll- und Steuerwort beschreiben. Dabei wird Bit 0 gelöscht
End_Is_First:
RET
```

Im zweiten Schritt werden die oberen 6 Bits von **Com_Ctrl** auf eine **1** geprüft. Ist eine **1** vorhanden, dann ist bereits ein Befehl erkannt und es ist das 2. Byte einer Befehlsfolge. Diese Bearbeitung findet in der

nächsten Routine **Is_Second** statt. Ist noch kein Bit gesetzt, so wird eine Befehlsauswertung durchgeführt. Soll ein Relais geschaltet werden oder ist das nächste Byte ein Trigger. Hier werden die Weichen gestellt.

Die Auswertung des zweiten Bytes erfolgt ähnlich. Dazu werten wir nun die Bits 2-7 aus und verweigen in passende Subroutinen, z. B. Set_Relais _On. Dort wird dann auch das entsprechende Relais mit der Information aus dem zweiten empfangenen Byte zugeordnet. Im Moment aber ist das noch nicht von Bedeutung und so werden lediglich die Programmrümpfe geschrieben, damit das Programm compilierbar und lauffähig bleibt. Für Testzwecke kann dort auch der Blinker installiert werden.

```
;************************ Relais einschalten *********************
Set_Relais_On:
  RCALL Blinker
RET

;************************ Relais ausschalten *********************
Set_Relais_Off:

RET
```

Hier ist nun das Programm zur vollständigen Auswertung des Befehls und einem anschließenden Aufruf zur Übertragung der Variableninhalte. Dazu werft einmal einen Blick in den Bereich Set_Internes.

```
;*******************************************************************
;*     Zweites Byte eines Befehles auswerten              *
;*******************************************************************
Is_Second:
   LDS      Reg_A, Com_Ctrl        ; Zuerst wieder abfragen, ob Daten eingegangen
   ANDI     Reg_A, 0b00000001      ; Wenn ja, dann Abfragen, ob 2. Byte
   BREQ     End_Is_Second
   LDS      Reg_A, Com_Ctrl        ; Daten sind eingegangen, ist ein Bit 2-7 gesetzt
   ANDI     Reg_A, 0b11111100      ; Wenn ja, dann ist 2. Byte eingetroffen.
   BREQ     End_Is_Second          ; wenn kein Bit gesetzt, dann nicht 2. Byte
   LDS      Reg_A, Com_Ctrl        ; 2.Byte bearbeiten
   ANDI     Reg_A, 0b00000100      ; Bit für „Relais An“ gesetzt
   BREQ     Chk_Rel_Aus
   RCALL    Set_Relais_On          ; Subroutine für Relais ein, Rel.Nr ist in Work_A
   LDS      Reg_A, Com_Ctrl        ; nun nur noch das Steuerbit löschen
   ANDI     Reg_A, 0b11111011
```

```
   STS      Com_Ctrl.  Reg_A
Chk_Rel_Aus:
   LDS        Reg_A, Com_Ctrl
   ANDI       Reg_A, 0b00001000      ; Bit für Relais aus gesetzt
   BREQ       Chk_Trigger
   RCALL      Set_Relais_Off         ; Subroutine für Relais aus, Rel.Nr ist in Work_A
   LDS        Reg_A, Com_Ctrl        ; nun nur noch das Steuerbit löschen
   ANDI       Reg_A, 0b11110111
   STS        Com_Ctrl.  Reg_A
Chk_Trigger:
   LDS        Reg_A, Com_Ctrl
   ANDI       Reg_A, 0b00010000      ; Bit für Trigger gesetzt
   BREQ       Chk_TrigerReset
   LDS        Reg_A, IS_Data_Rec     ; dann den Wert des 2. Bytes
   CPI        Reg_A, 0
   BRNE       ReSet_Trig_Ein
   LDI        Reg_B, 0b10000000      ; Bit für Werteabfrage der internen Variablen
   STS        Com_Ctrl, Reg_B
   RJMP       Set_Internes
ReSet_Trig_Ein:
   LDS        Reg_B, Trigger_In      ;  Variable Trigger in_laden
   Or         Reg_B, Reg_A           ; Bit hinzufügen
   STS        Trigger_In, Reg_B      ; in die Variable „Trigger_In“
   LDS        Reg_A, Com_Ctrl        ; nun nur noch das Steuerbit löschen
   ANDI       Reg_A, 0b11101111
   STS      Com_Ctrl.  Reg_A
Chk_TriggerReset:
   LDS        Reg_A, Com_Ctrl
   ANDI       Reg_A, 0b00100000      ; Bit für Trigger gesetzt
   BREQ       Set_Internes
   LDS        Reg_A, IS_Data_Rec     ; dann den Wert des 2. Bytes
   CPI        Reg_A, 0
   BRNE       ReSet_Trig_Aus
   LDI        Reg_B, 0b10000000      ; Bit für Werteabfrage der internen Variablen
   STS        Com_Ctrl, Reg_B
   RJMP       Set_Internes
ReSet_Trig_Aus:
   COM        Reg_ A                 ; Bits invertieren
   LDS        Reg_B, Trigger_In      ;  Variable Trigger_in laden
   And        Reg_A , Reg_B          ;  Bit löschen“
   STS        Trigger_In, Reg_A      ;  in Variable „Trigger_In“ ablegen
   LDS        Reg_A, Com_Ctrl        ; nun nur noch das Steuerbit löschen
   ANDI       Reg_A, 0b11011111
   STS      Com_Ctrl.  Reg_A
Set_Internes:
```

```
   LDS      Reg_A, Com_Ctrl
   ANDI     Reg_A, 0b10000000        ; Bit für Werteabfrage gesetzt
   BREQ     End_Is_Second
   RCALL    Send_Value               ; Senderoutine für die Variablenwerte
   CLR      Reg_A
   STS      Com_Ctrl.  Reg_A
End_Is_Second:
RET
```

Die Bits in **Com_Ctrl** sind wie Flankenbits der Taster zu betrachten. Sie signalisieren ein Ereignis. Ist dieses abgearbeitet, wird das entsprechende Bit gelöscht. Bit 7 ist für die Übermittlung der Variablen zuständig. Die Subroutine Send_Value wird ebenfalls erst einmal nur als Gerüst geschrieben und mit dem Blinker ausgetestet. So können wir eine saubere Erkennung und Bearbeitung der Kommandos vom PC überprüfen. Hierbei ist auch klar, das beide Zeichen, also Befehl und Nummer ausgewertet werden und dabei nur ein Wechsel im Status der LED stattfindet

```
Send_Value:
  RCALL Blinker
RET
```

Auch diese beiden Routinen Is_First und Is_Second werden in der Main_Loop eingebunden

```
;*****************************************************
;*                Schleife Hauptprogramm                *
;*****************************************************
Loop:
   RCALL    Read_IO          ; Eingänge lesen
   RCAll    IO_Debounce      ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit      ; Bearbeiten „V“
   RCALL    Blinker          ; Blinkerbit bilden
   RCALL    IO_Event_To_1    ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL    Taster_Event     ; Bearbeiten
   RCALL    Chk_Receive      ; Datenempfang prüfen
   RCALL    Is_First         ; erstes empfangenesByte prüfen und markieren
   RCALL    Is_Second        ; Datenempfang endgültig auswerten
```

```
   RCALL    Write_IO              ; und ausgeben
RJMP Loop
```

Bisher habe ich immer versucht, das **EVA-Prinzip** zu befolgen. Doch wie ist es mit einem Datenempfang. Auch das ist eine Eingabe, allerdings nicht über den Programmzyklus, sondern im Interrupt. Betrachten wir es einmal etwas anders.

Die Differenz zwischen **Lesezeiger** und **Schreibzeiger** ist Bearbeitung, doch das Bit 0, welches den Datenempfang und ein Byte zur Bearbeitung ankündigt, ist eigentlich das Ereignis der Schnittstelle, welches den Empfang signalisiert. Also passt dieser Bereich noch zum **E**. Erst die Auswertung auf Befehl und Parameter ist eine Bearbeitung und passt in den Bereich **V** und letztlich die Aufbereitung des Ausgabebytes in der Variablen **Out_Ctrl** sowie der Aufruf **Send_Values** läßt sich als Ausgabe definieren und dem **A** zuordnen.

Übertragen wir nun die Routinen und beseitigen auftretende Fehler. In der Regel fehlen Variablendeklarationen oder es sind Aufrufe von nicht erstellten Subroutinen.

Beispiel:

```
  C:\Daten\Arbeiten\Buch_Word\Experimente.asm(18): Including file 'C:\Program Files\Atmel Studio\AvrAssembler2
● C:\Daten\Arbeiten\Buch_Word\Experimente.asm(518): error: Undefined symbol: Set_Relais_Off
  C:\Daten\Arbeiten\Buch_Word\Experimente.asm(690): No EEPROM data, deleting C:\Daten\Arbeiten\Buch_Word\Exper
```

*Assemblerfehler fehlendes Unterprogramm*

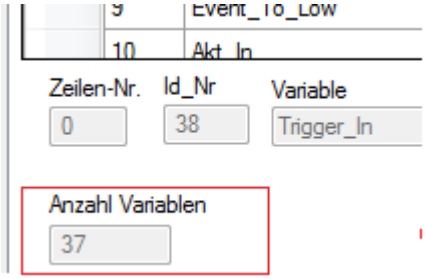
### *2.11.7 Variablenwerte zum PC senden*

Im Aufbau unseres Controllerprogrammes sind wir nun am interessantesten Punkt angekommen. Die Datenübertragung vom Controller zum PC. Alle vorbereitenden Maßnahmen sind durchgesprochen und die Routinen, die diese Kommunikation anstoßen sollen sind geschrieben. Der Assembliervorgang ist fehlerfrei abgeschlossen. Ob das Programm aber fehlerfrei läuft, das wissen wir noch nicht. Der Controller hat nicht sehr viel Peripherie, mit der er uns über die Funktionalität des Programms in Kenntnis setzen könnte. Also atmen wir noch einmal tief durch und nehmen uns die Routine **Send_Value** vor.

```
;*********************************************************************
;*  diese Routine sendet die Werte der Variablen        *
;*  an einen PC. Das Visual Basic Programm kann         *
;*  aus dem Assemblerlisting die Variablennamen         *
;*  herausfiltern und entsprechend Anzeigeobjekte      *
;*  generieren. Besteht eine serielle Verbindung        *
;*  zum Controller, werden die aktuellen Werte          *
;*  angezeigt.                                            *
;*********************************************************************
Send_Value:
   LDI      Send_Byte , 'V'          ; Telegrammkennung senden
   RCALL    Ser_Out
   LDI      Send_Byte , 'A'
   RCALL    Ser_Out
   LDI      Send_Byte , 'L'
   RCALL    Ser_Out
   LDI      Send_Byte , 'U'
   RCALL    Ser_Out
   LDI      Send_Byte , 'E'
   RCALL    Ser_Out
   LDS      Reg_A, Value_Cnt         ; Anzahl der Variablen
; Value_Cnt muß im Initialisierungsteil gesetzt werden. Ergebnis aus Open_Eye-Filter
   LDI      XL,LOW(Trigger_In)       ; -XPointer auf Anfang der Variablen
   LDI      XH,HIGH(Trigger_In)
Send_Loop:
   LD       Send_Byte , +X
 ; die Durch X adressierte Variable an Send_Byte übergeben, dabei X um 1 erhöhen
   RCALL    Send_Data                ; und Senderoutine aufrufen
   Dec      Reg_A                    ; Schleifenzähler erniedrigen
   BRNE     Send_Loop                ; wenn nicht 0, dann nächste Variable
```

```
RET
```

In dieser Routine ist die Variable Value_Cnt von Bedeutung. Den Wert erhalten wir in Open_Eye auf der Seite Einstellungen.




*Anzahl Variablen*

Ihr seht, niemand muss die Variablen zählen. Das erledigt der Filter für euch. Es muss lediglich der Wert im Controller programmiert werden.

Schließlich fehlt noch eine kleine Subroutine **Send_Data** Diese ist aus Gründen der Übersichtlichkeit sowie der weiteren Verwendung separat geschrieben.

```
;---------------- serielle Ausgabe -----------------
;************************************
;*    Diese Routine sendet ein Byte über        *
;*    die serielle Schnittstelle                *
;************************************
Send_Data:
   SBIS    UCSRA, UDRE          ; Warten Freigabe UDR
   RJMP    Send_Data
   OUT     UDR, Send_Byte
RET
```

Die Routine **Ser_Out** hat einen kleinen Nachteil. Schaut einmal in die zweite Zeile. Dort steht ein Sprung zum Einstieg in die Routine. Nein, keine Angst, der Stack könnte überlaufen. So eine Adressmarke ist für alles Mögliche gut und bei einem **RJMP**-Befehl gibt es keine Rücksprungadresse wie bei einem **RCALL**-Befehl, die auf dem Stack hinterlegt werden muss. Dennoch wird hier auf ein gesetztes Bit im **USART Control**- und **Statusregister** gewartet und solange läuft das

Programm nicht weiter. Auch wenn nur auf Freigabe gewartet wird, bedeutet es einen Anstieg der Zykluszeit. Im Übrigen gibt das Datenblatt Auskunft. Ist der **Transmit-Buffer UDR** bereit für neue Daten, ist dieses Bit gesetzt und der **RJMP**-Befehl wird übersprungen. **SBIS** kann nur einen Befehl überspringen, aber das reicht an dieser Stelle. !!!

### *2.11.8 Senden mit Interrupt*

Bei kleinen Datenmengen spielt diese kleine Wartezeit keine Rolle, aber wenn wie in unserem Fall doch schnell ein paar zig Bytes zusammenkommen, wird das Programm doch sichtbar ausgebremst. Doch wir brauchen uns damit ja nicht abzufinden. So wie es einen Interrupt für das Empfangen von Zeichen gibt, ist es auch möglich, ein Zeichen zu senden und dann weiterzuarbeiten. Der USART meldet sich dann irgendwann, dass er die Aufgabe erledigt hat und für einen neuen Auftrag bereit ist. Allerdings müssen wir dann die Sendedaten etwas anders aufarbeiten. Zuerst benötigen wir einen Zeichenzähler außerhalb der Senderoutine. Er wird im Variablenbereich deklariert. Ich habe ihn mit Send_Cnt definiert. Der Grund sollte auch klar sein, wir senden nur das erste Byte bewußt, danach übernimmt der Interrupt die Aufgabe und da ist nicht bekannt, wieviele Zeichen bereits versendet wurden, Alles ist weg, auch der Zeiger auf den Sendeblock. Aber sehen wir uns erst einmal an, welche Schritte für eine Interrupt gesteuerte Sendung erforderlich ist. In der IVT gibt es den Eintrag

```
.org UTXCaddr      RETI  ; USART Transmit Complete Interrupt Vector Address
```

Ah ja, das ist der Interrupt, der bei einem Sendebefehl auslöst, wenn die Bereitschaft für den nächsten Sendeauftrag vorliegt. Ein RJMP in eine entsprechende ISR und das nächste Zeichen wird übertragen. Leider ist diese Annahme falsch, aber gehen wir ruhig diesen Weg erst einmal weiter.

Was braucht diese ISR alles, damit das funktioniert und ein Datenblock zusammenhängend übertragen wird? Beginnen wir mit dem erforderliche Sichern der Register und einem ***Brainstorm***.

Zeiger auf das Datenfeld. Wert mit einem Offset zum nächsten Datum. Grenzwertabfrage und entsprechende Abschaltung des Interrupts. Natürlich sind in einer ISR auch alle verwendeten Register zu sichern. Beginnen wir einfach mal mit dem Gerüst:

```
ISR_Send_Date:
  IN      Ablage_SReg_SREG      ; Sichern aller Register
  PUSH   XL
  PUSH   XH
  PUSH   Reg_A
  Push   Reg_B

  POP     Reg_B                 ; Register wieder herstellen
  POP     Reg_A
  POP     XH
```

```
  POP     XL
  OUT    SREG, Ablage_SReg
RETI
```

Wenn der Rahmen zuerst erstellt wird, sind die Push und POP Befehle besser übersichtlich und es fällt leichter, die richtige Reihenfolge zu beachten. Das Programm setzen wir dann zwichen den letzten Push- und den ersten POP- Befehl.

```
  Push    Reg_B
  LDI     XL, Low(Trigger_In)        ; Basisadresse Sendeblock
  LDI     XH, High(Trigger_In)
  LDS     Reg_A, Send_Cnt            ; Datenzeiger
  ADD     XL, Reg_A                  ; addieren
  ADC,    Zero                       ; Zero reserviertes  Null-Register
  LD      Send_Byte, X               ; Send_Byte reservierten Register Daten senden
  OUT     UDR, Send_Byte             ; in Sendepuffer eintragen
  INC     Reg_A                      ; Datenzeiger hochzählen
  STS     Send_Cnt, Reg_A            ; zurückspeichern
  LDS     Reg_B, Value_Cnt           ; Anzahl Sendedaten erreicht?
  CP      Reg_A, Reg_B
  BRLO    End_ISR_Send               ; wenn nicht, dann Ende ISR
  CBI     UCSRB,   TXCIE             ; USART Controlregister Interrupt abschalten
  LDS     Reg_A,  Prg_Ctrl
  ANDI    Reg_A, 0b01111111          ; Sendung läuft quittieren
  STS     Prg_Ctrl, Reg_A
End_ISR_Send:                        ; fertig

  POP     Reg_B                      ; Register wieder herstellen
```

An dieser Stelle sei bemerkt, dass der Sendeblock nicht nur die Werte der Variablen enthalten muss. Es können auch andere Bereiche oder Datenpuffer zusammengestellt und übertragen werden. Allerdings ist dieser Interrupt nicht in der Initialisierung USART freizugeben. Wir wollen ja vorher noch die ASCII-Zeichen „V“, „A“, „L“, „U“ und „E“ zur Kontrolle an den PC senden und erst dann den Interrupt freigeben. Somit bleibt dieser Programmteil erst einmal mit ein paar Änderungen bestehen. Bis zum Zeichen „U“ belassen wir die alte Sendemethode. Nun kommen noch zwei Aufgaben, die zu erledigen sind. Zuerst muss der Datenzähler zurückgesetzt werden. Das ist kein Problem. Der Befehl STS Send_Cnt, Zero erledigt das. Da wir aber einen Automatismus der Datenübertragung anstoßen, müssen wir verhindern, das eine weitere Datenübertragung

den Aufruf Send_Value anstößt. Dies darf erst passieren, wenn eine Übertragung vollständig beendet ist. Eine Triggerfunktion könnte uns sonst die Daten komplett durcheinander bringen. Aber wir haben ja noch das Byte Prg_Ctrl. Nehmen wir doch einfach ein freies Bit um einen laufenden Datentransfer zu markieren. In der ISR wird dann mit der Abschaltung des Interrupts auch dieses Bit wieder gelöscht. Fügen wir der ISR zuerst diese Option hinzu. Hier die ganze ISR

```
ISR_Send_Data:
  IN      Ablage_SReg_SREG      ; Sichern aller Register
  PUSH   XL
  PUSH   XH
  PUSH   Reg_A
  Push   Reg_B

  LDI     XL, Low(Trigger_In)       ; Basisadresse Sendeblock
  LDI     XH, High(Trigger_In)
  LDS     Reg_A, Send_Cnt           ; Datenzeiger
  ADD     XL, Reg_A                 ; addieren
  ADC,    Zero                      ; Zero reserviertes  Null-Register
  LD      Send_Byte, X              ; Send_Byte reservierten Register Daten senden
  OUT     UDR, Send_Byte            ; in Sendepuffer eintragen
  INC     Reg_A                     ; Datenzeiger hochzählen
  STS     Send_Cnt, Reg_A           ; zurückspeichern
  LDS     Reg_B, Value_Cnt          ; Anzahl Sendedaten erreicht?
  CP      Reg_A, Reg_B
  BRLO    End_ISR_Send              ; wenn nicht, dann Ende ISR
  CBI     UCSRB, TXCIE              ; Interruptbit  Senden löschen
  LDS     Reg_A, Prg_Ctrl           ; Programmkontrolle Datenübertragung läuft
  ANDI    Reg_A, 0b01111111         ; Bit 7 abschalten = Übertragung beendet
  STS     Prg_Ctrl, Reg_A
End_ISR_Send:                       ; fertig

   POP     Reg_B                        ; Register wieder herstellen
   POP     Reg_A
   POP     XH
   POP     XL
   OUT     SREG, Ablage_SReg
RETI
```

Nun zur Senderoutine Send_Value. Das alte Unterprogramm belassen wir in seiner Form und schreinben eine Neue. Es ist eine Kopie, die mit dem Namen Send_Value_Int aufgerufen wird. So kann jederzeit auf funktionierende Programmteile zurückgeschaltet werden, wenn mal etwas

schief läuft. Die Aufrufe im Programm Send_Value werden einfach auskommentiert und statt dessen der Befehl RCALL Send_Value_Int eingetragen. Auch brauchen wir kein X –Register sichern, da es in dieser Routine nicht verwendet wird. Also kann auch ein Trigger diese Routine innerhalb einer Schleife anstoßen. Lediglich das Register A könnte bei unachtsamer Verwendung verfälscht werden und deshalb wird es auf dem Stak gesichert. In dieser Routine beginnen wir mit der Prüfung, ob ein Sendeauftrag bereits läuft.und geben eine neue Sendung nur frei, wenn dies nicht der Fall ist.

```
Send_Value_Int:
  PUSH    Reg_A
  LDS     Reg_A, Prg_Ctrl          ; Programmkontrolle
  ANDI    Reg_A, 0b10000000        ; Bit 7  Übertragung läuft?
  BRNE    End_Send_Value
  LDS     Reg_A, Prg_Ctrl          ; Programmkontrolle
  ORI     Reg_A, 0b10000000        ; setze Bit 7  Übertragung läuft
  STS     Prg_Ctrl. Reg_A
  LDI     Send_Byte, "V"           ; Telegrammkennung senden
  RCALL   Send_Data
  LDI     Send_Byte, "A"
  RCALL   Send_Data
  LDI     Send_Byte, "L"
  RCALL   Send_Data
  LDI     Send_Byte, "U"
  RCALL   Send_Data
  LDI     Send_Byte, "E"
  RCALL   Send_Data
  STS     Send_Cnt, Zero           ; Nutzdatenzähler auf Anfang setzen
  SBI     UCSRB , TXCIE            ; Interruptsteuerung aktivieren
End_Send_Value:
  POP     Reg_A
RET
```

Bleibt noch, die ISR in die IVT eizutragen.

```
.org UTXCaddr     RJMP   ISR_Send_Data ; USART Transmit Complete Interrupt
;RETI
```

Starten wir nun eine Übertragung zu unserem Open_Eye-Programm. Irgendwie läuft da etwas falsch. Die Daten sind vielleicht beim ersten

Anfordern richtig, aber danach läuft es irgendwie aus dem Ruder. Das liegt daran, das bei Freigabe des Interrupts sofort ein Zeichen in das Senderegister übertragen wird, ohne das durch das Bit UDRE im Register UCSRA signalisiert wird, das der USART für ein neues Byte bereit ist. Klar, man könnte nun so eine kleine Warteschleife beim Eintritt in die ISR programmieren, wie sie auch in der kleinen Routine Send_Data eingebaut ist, doch das widerspricht dem Sinn eines Interrupts. Allerdings gibt es auch einen Interrupt, der vom Bit UDRE ausgelöst wird. Das zugehörige Interruptbit ist nicht TXCIE sondern UDRIE. Auch dieses Bit löst einen Sprung zur IVT zu einer eigenen Adresse aus. Tauschen wir doch einmal TXCIE gegen UDRIE aus und ändern den Aufruf in der IVT.

```
.org UDREaddr    RJMP   ISR_Send_Data  ; USART Data Register Empty Interrupt
   ;RETI
.org UTXCaddr      ;RJMP  ISR_Send_Data    ; USART Transmit Complete Interrupt
   RETI
```

Jetzt ist es perfekt. Von einer Datenübertragung bekommt das Programm so gut wie nichts mit. Das bedeutet auch, dass Signale an den Eingängen wieder erfasst werden und nicht durch eine Schleife beim Senden Von Daten verloren gehen könnten. Es ist auch kein Problem über hundert Variable an den PC zur Visualisierung zu schicken. Das Flackern der Anzeige tritt nicht mehr auf.

### *2.11.9 Anzahl Variablen von PC Anforderung erfassen*

Bisher haben wir die Anzahl der im Filter aufbereiteten Variablen immer von Hand in unser Assemblerprogramm eingetragen und so manche Datenübertragung lief nicht wie erwartet, weil dieser Wert nicht eingetragen oder angepasst war. Das können wir besser. Bisher übertragen wir V0 zur Anforderung der Variableninhalte. Natürlich ist es auch möglich, statt der 0 den Wert in dem Textfeld TB_Anzahl_Var zu übertragen und somit die Variable Value_Cnt im Controller zu besetzen. Schließlich werten wir bei einem empfangenen V das zweite Byte gar nicht aus.

Dazu ändern wir erst im Visual- Basic Programm die Übertragungsroutine. Die Variable Werte_Feld habe ich in Send_Feld umbenannt.

```
Public Sub Send_Befehl(ByVal Order As String)
   Dim Befehl As String
   Dim Anz As Integer
   Befehl = Trim(Order)
   Anz = Len(Befehl)
   If Anz > 0 Then
     If InStr(Befehl, "V") > 0 Then
        Send_Feld(0) = Asc("V")
        Send_Feld(1) = Val(TB_Anzahl_Var.Text)
     Else
        Send_Feld(0) = Mid(Befehl, 1, 1)
        Send_Feld(1) = Mid(Befehl, 2, 1)
     End If
     SerialPort1.Write(Send_Feld, 0, 2)
  End If
 End Sub
```

Der Befehl SerialPort1.Write wird statt mit einem String dem Array Send_Feld durchgeführt, weil wir ja nicht wissen, ob die Anzahl der herausgefilterten Variablen einem ASCII-Zeichen entspricht. Der alte Befehl

```
If Anz > 0 Then
        SerialPort1.Write(Befehl)
End If
```

Könnte durch den String verfälscht werden. Dazu muss aber auch erfragt werden, ob es eine Anforderung der Variableninhalte sein soll, denn auch der Trigger hat eine Nummer und auch der Befehl, ein Relais zu schalten. Beim Relais steht aber die Nummer direkt hinter dem Kennbuchstaben. Also wird nur bei einen V-Befehl die Anzahl der Variablen in das Send_Feld(1) eingetragen.

Im Assembler ist nun ebenfalls eine kleine Änderung erforderlich, um dieses zweite Byte auszuwerten und die Anzahl in Value_Cnt einzutragen. Die Änderung erfolgt in der Subroutine Is_Second.

```
;*****************************************************************
;*     Zweites Byte eines Befehles auswerten              *
;*****************************************************************
Is_Second:
   LDS       Reg_A, Com_Ctrl          ; Zuerst wieder abfragen, ob Daten eingegangen
   ANDI      Reg_A, 0b00000001        ; Wenn ja, dann Abfragen, ob 2. Byte
   BRNE      Second_Chk:
MARK_1:                                ; Weil Sprungadresse zu weit liegt
  RJMP      End_Is_Second
Second_Chk:
   LDS       Reg_A, Com_Ctrl          ; Daten sind eingegangen, ist ein Bit 2-7 gesetzt
   ANDI      Reg_A, 0b11111100        ; Wenn ja, dann ist 2. Byte eingetroffen.
   BREQ      Mark_1                   ; wenn kein Bit gesetzt, dann nicht 2. Byte
   LDS       Reg_A, Com_Ctrl          ; 2.Byte bearbeiten
   ANDI      Reg_A, 0b00000100        ; Bit für „Relais An“ gesetzt
   BREQ      Chk_Rel_Aus
  RCALL     Set_Relais_On             ; Subroutine für Relais ein, Rel.Nr ist in Work_A
   LDS       Reg_A, Com_Ctrl          ; nun nur noch das Steuerbit löschen
   ANDI      Reg_A, 0b11111010
   STS       Com_Ctrl.  Reg_A
  RJMP      End_Is_Second
Chk_Rel_Aus:
   LDS       Reg_A, Com_Ctrl
   ANDI      Reg_A, 0b00001000        ; Bit für Relais aus gesetzt
   BREQ      Chk_Trigger
   RCALL     Set_Relais_Off           ; Subroutine für Relais aus, Rel.Nr ist in Work_A
   LDS       Reg_A, Com_Ctrl          ; nun nur noch das Steuerbit löschen
   ANDI      Reg_A, 0b11110110
   STS       Com_Ctrl.  Reg_A
   RJMP      End_Is_Second
Chk_Trigger:
   LDS       Reg_A, Com_Ctrl
   ANDI      Reg_A, 0b00010000        ; Bit für Trigger gesetzt
   BREQ      Chk_TrigerReset
```

```
   LDS        Reg_A, IS_Data_Rec       ; dann den Wert des 2. Bytes
   ;LDS       Reg_B, Trigger_In        ;  Variable Trigger in_laden
   ;Or        Reg_A, Reg_B             ; Bit hinzufügen
   ;STS       Ampelzeit, Reg_A
   STS        Trigger_In, Reg_A        ; in die Variable „Trigger_In“
   ; LDI      Reg_A, 88
   ; STS      Ampelphase, Reg_A
   LDS        Reg_A, Com_Ctrl          ; nun nur noch das Steuerbit löschen
   ANDI       Reg_A, 0b11101110
   STS        Com_Ctrl.  Reg_A
   RJMP       Set_Internes
 Chk_TriggerReset:
   LDS        Reg_A, Com_Ctrl
   ANDI       Reg_A, 0b00100000        ; Bit für Trigger gesetzt
;********************  geändert ******************
   BREQ       Chk_Order_V
   STS        Trigger_In, Zero         ;  in Variable "Trigger_In" ablegen
   STS        Trigger_Out, Zero
  ; LDI       Reg_A, 55
  ; STS       Ampelphase, Reg_A
   LDS        Reg_A, Com_Ctrl          ; nun nur noch das Steuerbit löschen
   ANDI       Reg_A, 0b11011110
   ORI        Reg_A, 0b10000000
   STS        Com_Ctrl,  Reg_A
   RJMP       Set_Internes
Chk_Order_V:
   LDS        Reg_A, Com_Ctrl          ; nun nur noch das Steuerbit löschen
   ANDI       Reg_A, 0b10111110
   ORI        Reg_A, 0b10000000
   STS        Com_Ctrl,  Reg_A
   LDS        Reg_A, IS_Data_Rec       ; dann den Wert des 2. Bytes
   STS        Value_Cnt,Reg_A
Set_Internes:
   LDS        Reg_A, Com_Ctrl
   ANDI       Reg_A, 0b10000000        ; Bit für Trigger gesetzt
   BREQ       End_Is_Second
   RCALL      Send_Value               ; Senderoutine für die Variablenwerte
   RCALL      Send_Value_Int           ; neue Routine senden mit Interrupt
   CLR        Reg_A
   STS        Com_Ctrl.  Reg_A
End_Is_Second:
RET
```

Eines gilt es jedoch zu beachten: Bevor ein Trigger zum Controller geschickt wird muss mindestens einmal ein Variablenstatus normal angefordert werden, damit die Variable Value_Cnt besetzt wird. Diesen Wert behält sie solange der Controller unter Spannung steht.

### *2.11.10 Programm Senden und Empfangen*

```
;*********************************************************************
;*                    Datum 06.08.2012                              *
;*            Beispielanwendungen mit Atmega 8                     *
;* ------------------------------------------------------------- *
;*      Ein Experimente mit PC und Mikrocontroller                 *
;*                                                                 *
;*  Die Routinen für den Atmega8 beinhalten:                       *
;*    Standart  IO                                                 *
;*    Signalerfassung von Taster                                   *
;*    Ansteuerung von LED                                          *.
;*  Copyright by Martin Vogel   Germany                            *
;*********************************************************************
.NoList
.include   "m8def.inc"      ; Definitionen für ATMEGA8
.List
.DEF Access      = R0       ; Speicherzugriffsregister
.DEF Zero        = R1       ; Register für Null-Vergleich
.DEF Zehn        = R2       ; Register für Vergleich mit konst. 10
.DEF Zwei        = R3       ; Zeiterfassung Minute
.DEF fuenf       = R4       ; Register für Vergleich mit konst. 60
.DEF hundert     = R5       ; Register für Vergleich mit kont. 100
.DEF MilliSek    = R6       ; Register für mSek ( schneller Zugriff)
.DEF ZehnMilli   = R7       ; Register für hundertstel Sek. ( schneller Zugriff)
.Def Zehntel     = R8       ; Register für  Zehntelsekunde
.DEF Ablage_A    = R9       ; Zwischenspeicher A
.DEF Ablage_B    = R10      ; Zwischenspeicher B
.DEF Count_L     = R11      ; Counter Low 0-7
.DEF Count_H     = R12      ; Counter High 8-15
.DEF Count_HL    = R13      ; Counter HighLow 16-23
.DEF Count_HH    = R14      ; Counter High High 24-31
.Def SichSREG    = R15      ; Zwischenspeicher f. Statusregister
.DEF Reg_A       = R16      ; Universalregister A
.DEF Reg_B       = R17      ; Universalregister B
.DEF Reg_C       = R18      ; Universalregister C
.DEF Reg_D       = R19      ; Universalregister D
.DEF Reg_E       = R20      ; Universalregister E
.DEF Reg_F       = R21      ; Universalregister E
```

```
.DEF Work          = R22     ; Arbeitsregister
.DEF Send_Byte  = R23      ; Datenübertragung
.DEF Calc_A        = R24      ; Math.-Register A
.DEF Calc_B        = R25      ; Math. Register B
; ---- Register 26- 31 sind bereits vergeben ----
; X = XL + XH = R 26 + R 27
; Y = YL + YH = R 28 + R 29
; Z = ZL + ZH = R 30 + R 31
.DSEG
Trigger_In:         .Byte 1       ; Byte Debug-Variable Eingang
Trigger_Out:       .Byte 1       ; Byte Debug-Variable Ausgang

;---------------------- Variablen IO-Ebene ----------------------
New_In:              .Byte 1  ; Byte Ablage Eingänge
                                  ; Bit 0 = Taster 1 Mode
                                  ; Bit 1 = Taster 2 Ziffer
                                  ; Bit 2 = Taster 3 auf
                                  ; Bit 3 = Taster 4 ab
                                  ; Bit 4 = Gabellichtschranke
                                  ; Bit 5 = res
                                  ; Bit 6 = res.
                                  ; Bit 7 = res.

Out_Ctrl:            .Byte 1  ; Byte Ablage für Ausgänge
                                 ; Bit 0 = Relais 1
                                 ; Bit 1 = Relais 2
                                 ; Bit 2 = Relais 3
                                 ; Bit 3 = Relais 4
                                 ; Bit 4 = Relais 5
                                 ; Bit 5 = LED 1
                                 ; Bit 6 = LED 2
                                 ; Bit 7 = LED 3
Counter_0:       .Byte 1      ; 16 Bit-Zähler LowByte
Counter_1:       .Byte 1      ; 16 Bit-Zähler HighByte
old_In:             .Byte 1      ; Byte Ablage zuletzt gelesene Eingänge
                              ; Bit 0 = Taster 1 Mode
                              ; Bit 1 = Taster 2 Ziffer
                              ; Bit 2 = Taster 3 auf
                              ; Bit 3 = Taster 4 ab
                              ; Bit 4 = Gabellichtschranke
                              ; Bit 5 = res
                              ; Bit 6 = res.
                              ; Bit 7 = res.
Event_To_High:   .Byte 1      ; Byte Ablage Eingänge
                              ; Bit 0 = Taster 1 Mode
```

```
                        ; Bit 1 = Taster 2 Ziffer
                        ; Bit 2 = Taster 3 auf
                        ; Bit 3 = Taster 4 ab
                        ; Bit 4 = Gabellichtschranke
                        ; Bit 5 = res
                        ; Bit 6 = res.
                        ; Bit 7 = res.

Event_To_Low:   .Byte 1      ; Byte Ablage Eingänge
                        ; Bit 0 = Taster 1 Mode
                        ; Bit 1 = Taster 2 Ziffer
                        ; Bit 2 = Taster 3 auf
                        ; Bit 3 = Taster 4 ab
                        ; Bit 4 = Gabellichtschranke
                        ; Bit 5 = res
                        ; Bit 6 = res.
                        ; Bit 7 = res.
Akt_In:             .Byte 1          ; Aktuell gelesener Portwert
In_Debounce:    .Byte 1          ; zuletzt gelesener Portwert
Debounce_Cnt: .Byte 1          ; Zähler für Wartezeit
Com_Ctrl:         .Byte 1         ; Byte  Statusflags vom Programm
Read_Pos:        .Byte 1         ; Int8    Lesezeiger
Write_Pos:       .Byte 1         ; Int8    Schreibzeiger
Com_Buff:        .Byte 20         ; Ringpuffer Empfang
Is_Data_Rec:    .Byte 1          ; Arbeitsvariable Datenempfang

;*******************************************************************************
;*    Die folgenden Zeilen definieren ein paar Constanten.          *
;*   Diese Werte werden nur einmal vom Compiler zugewiesen.   *
;*   Danach werden sie nicht mehr verändert.                          *
;*******************************************************************************

.EQU F_CPU = 16000000                  ; Systemtakt in Hz ( im Pollin Board 16 MHz)
.EQU BAUD = 9600                       ; Baudrate ; Berechnungen
.EQU UBRR_VAL = ((F_CPU+BAUD*8)/(BAUD*16)-1)            ; runden
.EQU BAUD_REAL = (F_CPU/(16*(UBRR_VAL+1)))               ; Reale Baudrate
.EQU BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)      ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10)) ; max. +/-10 Promille Fehler
.error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu hoch!"
.endif

.CSEG
```

```
.ORG 0000
Reset_Point:   RJMP  Start    ; Start von hier ist Neustart
;*****************************************************************
;*            Bereich der Interrupt Vector Table              *
;*****************************************************************
           ; Sprungbefehle zu den Serviceroutinen
.org INT0addr            ;RJMP EXT_INT0 ; External Interrupt0 Vector Address
RETI
.org INT1addr            ; External Interrupt1 Vector Address
RETI
.org OC2addr             ; Output Compare2 Interrupt Vector Address
RETI
.org OVF2addr            ; Overflow2 Interrupt Vector Address
RETI
.org ICP1addr            ; Input Capture1 Interrupt Vector Address
RETI
.org OC1Aaddr        RJMP   isrTimer1 ; Einsprungadresse ISR Timer1
;RET I
.org OC1Baddr            ; Output Compare1B Interrupt Vector Address
RETI
.org OVF1addr            ; Overflow1 Interrupt Vector Address
RETI
.org OVF0addr            ; Overflow0 Interrupt Vector Address
RETI
.org SPIaddr                          ; SPI Interrupt Vector Address
RETI
.org URXCaddr      RJMP ISR_REC      ; USART Receive Complete Interrupt
                      ;geändert !      von int_rxc  auf ISR_REC !
;RETI
.org UDREaddr      RJMP  ISR_Send_Data    ; USART Data Register Empty Interrupt
;RETI
.org UTXCaddr                       ; USART Transmit Complete Interrupt Vector Address
RETI
.org ADCCaddr                       ; ADC Interrupt Vector Address
RETI
.org ERDYaddr                       ; EEPROM Interrupt Vector Address
RETI
.org ACIaddr                        ; Analog Comparator Interrupt Vector Address
RETI
.org TWIaddr                        ; Irq. vector address for Two-Wire Interface
RETI
.org SPMRaddr                       ; SPM complete Interrupt Vector Address
RETI

;*****************************************************************
```

```
;*                    Bereich Initialisierung                    *
;*****************************************************************
.ORG    INT_VECTORS_SIZE       ; Setzt die Adressmarke Start hinter die IVT

Start:
  LDI     Reg_A, High(RamEnd)   ; erste Maßnahme Stack setzen
  Out     SPH, Reg_A
  LDI     Reg_A, Low(RamEnd)
  OUT      SPL, Reg_A
                                ; weitere Initialisierungen
  RCALL   Init_IO               ; Initialisierung IO
  RCALL   Init_USART            ; Initialisierung UART
; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
  RCALL   Init_Var              ; Initialisierung Variablen ( Eintragen Defaultwerte)
; SEI                           ; globale Freigabe von Interrupts
;*****************************************************************
;*              Programmschleife Hauptprogramm              *
;*****************************************************************
Main_Loop:
            ;  Hauptprogramm
            ; Aufruf der Subroutinen
  RCALL    Read_IO
  RCALL    IO_Debounce
 ; RCALL    Set_LED_Bit
  RCALL   Chk_Receive
  RCALL   Is_First
  RCALL   Is_Second
  RCALL    Blinker
  RCALL    IO_Event_To_1     ; Ereignisbits bilden. Ergebnis in Event_To_High
  RCALL    Taster_Event      ; Bearbeiten
  RCALL    Write_IO
RJMP  Main_Loop
;-----------------------------------------------------------------
;*****************************************************************
;                        Subroutinen                            *
;*****************************************************************

Init_IO:                    ; Ein-Ausgaben initialisieren
;*******************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf       *
;*   Dabei werden nur die benötigten Portbits parametriert,       *
;*   die anderen in ihrer Vorgabe belassen.                       *
```

```
;*****************************************************************************
;*****************************************************************************
;*   Auf die Parametrierung der Ports baut die Hardware auf       *
;*   Dabei werden nur die benötigten Portbits parametriert,       *
;*   die anderen in ihrer Vorgabe belassen.                       *
;*****************************************************************************
  IN     Reg_A, DDRD       ; Port D Bits 0-1 = Serielle Schnittstelle
                           ; Bit 2 Eingang Taster 1
                           ; Bit 3 Eingang Taster 2
                           ; Bit 4 Eingang Taster 3
                           ; Bit 5 Eingang Taster 4
                           ; Bit 6 Eingang Gabellichtschranken
                           ; Bit 7 Ausgang Relais
  ANDI   Reg_A, 0b00000011       ; Bit 0 und 1 nicht verändern
  ORI    Reg_A, 0b10000000       ; Bit 7 auf Ausgang
  OUT    DDRD, Reg_A             ; Data Direction Register PortD beschreiben
  IN     Reg_A, PortD
  ORI    Reg_A,0b01111100        ; PortD Pull-Up für die Eingänge einschalten
                           ; Die anderen Portbits bleiben unverändert
  OUT    PortD, Reg_A            ; Port mit Inhalt von Register A beschreiben
  IN     Reg_A, DDRC             ; Data Direction Register lesen
  ORI    Reg_A,0b00111110        ; Port C Bits 1-5 auf Ausgang schalten
  ANDI   Reg_A, 0b11111110       ; Port C Bits ist Eingang
                           ; Die anderen Bits bleiben unverändert
  OUT    DDRC, Reg_A             ; Data Direction Register Port C beschreiben
  In     Reg_A, PortC
  ORI    Reg_A, 0b00000001       ; Pull_Up Widerstand PC0 einschalten
  Out    PortC, Reg_A
  In     Reg_A, DDRB             ; Data Direction Register Port B lesen
  ORI    Reg_A,0b00111111        ; Portbit 0-5 Ausgang
  Out    DDRB, Reg_A             ; Data Direction Register B beschreiben
RET
;-----------------------------------------------------------------------------

;------------------- Serielle Schnittstelle parametrieren ----------------
;*****************************************************************************
;*   Die Parametrierung ist mit dem Koppelpartner abzustimmen.  *
;*****************************************************************************
Init_USART:
   LDI Reg_A, HIGH(UBRR_VAL)                    ; Baudrate einstellen
   OUT UBRRH, Reg_A
   LDI Reg_A, LOW(UBRR_VAL)
   OUT UBRRL, Reg_A
                                                ; Übertragungsformat: 8 Daten-, 2 Stoppbit
   LDI REG_A, (1<<URSEL)|(1<<USBS)|(3<<UCSZ0)
```

```
   OUT UCSRC,REG_A
                                          ; Empfangen und senden freigeben
   LDI REG_A, (1<<RXEN)|(1<<TXEN)
   OUT UCSRB,REG_A
   SBI UCSRB, RXCIE                       ; RX (Empfang) Interrupt aktivieren
RET
;-----------------------------------------------------------------------------

;*****************************************************************************
;*  Neue Initialisierungsroutine. Muß vor der Programmschleife   *
;*  in Initialisierungsteil aufgerufen werden.                   *
;*****************************************************************************
Init_Var:
   CLR Zero                       ; Register „Zero“ auf 0 setzen
   STS Read_Pos, Reg_1            ; des Lese-
   STS Write_Pos, Reg_1           ; und des Schreibzeigers
   STS Com_Ctrl, Reg_1            ; Wichtig, sonst fehlerhafte Kommunikation
RET
;-----------------------------------------------------------------------------



;********** Lesen aller Eingänge und Signalanpassung *************
;*    Port D                                                      *
;*      Bit 0 und 1    serielle Verbindung                        *
;*      Bit 2 bis 5    Eingänge Taster                            *
;*      Bit 6          Eingang Gabellichtschranke                 *
;*      Bit 7          Ausgang Anzeige                            *
;*****************************************************************************
Read_IO:                          ; Eingänge einlesen
  In     Reg_A, PInD              ; Port B lesen
  COM    Reg_A                    ; Bits drehen
  ANDI   Reg_A, 0b01111100        ; Nur Eingänge übernehmen
  LSR    Reg_A                    ; nach rechts schieben (00111110)
  LSR    Reg_A                    ; nach rechts schieben (00011111)
  STS    New_In, Reg_A
RET
;-----------------------------------------------------------------------------

;*****************************************************************************
;*                          Entprellen von Eingängen                         *
;*****************************************************************************
IO_Debounce:
```

```
  LDS     Reg_A, Debounce_Cnt
  CPI     Reg_A, 0              ; Vergleich ist wichtig, Ladeanweisung setzt kein Zerobit.
  BREQ    End_Debounce          ; Wenn zähler auf 0, keine weitere Aktion
  LDS     Reg_A, Akt_In         ; Aktuellen Portwert laden
  LDS     Reg_B, In_Debounce    ; Vergleichswert laden
  EOR     Reg_B, Reg_A          ; Ergebnis in Register B, Register A nicht verändern
  BREQ    Chk_Debounce_Time     ; Wenn beide gleich, Prellzeit prüfen
  STS     In_Debounce, Reg_A    ; aktuellen Wert für nächsten Vergleich ablegen
  LDI     Reg_A, 50             ; Überwachungszeit neu setzen  (max.255)
  STS     Debounce_Cnt, Reg_A   ; und in Zähler eintragen
  RJMP    End_Debounce
Chk_Debounce_Time:
  LDS     Reg_B, Debounce_Cnt   ; Register A hat noch den Wert vom gelesenen Port
  Dec     Reg_B
  STS     Debounce_Cnt, Reg_B
  BRNE    End_Debounce
  STS     New_In, Reg_A         ; Zähler hat bis 0 gezählt. Der neue Wert ist gültig
End_Debounce:
RET
;-----------------------------------------------------------------


;*****************************************************************************
;*     Ereignis Signalwechsel nach "1" am Eingabe-Port.        *
;*****************************************************************************
IO_Event_To_1:
  LDS     Reg_A, Old_In         ; letzter gelesener Wert der Eingänge
  LDS     Reg_B, New_In         ; neu gelesener Wert der Eingänge
  EOR     Reg_A, Reg_B          ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
  AND     Reg_A, Reg_B          ; eine Und Verknüpfung mit dem neuen Status
  STS     Event_To_High, Reg_A  ; liefert die Bits mit Änderung von "0" nach "1"
  STS     Old_In, Reg_B         ; Neuen Wert in Ablage "old_In" kopieren
RET
;-----------------------------------------------------------------------------


;*****************************************************************************
;*        Ereignis Signalwechsel nach 0 am Eingabe-Port.        *
;*****************************************************************************
IO_Event_To_0:
  LDS     Reg_B, Old_In         ; letzter gelesener Wert der Eingänge
  LDS     Reg_A, New_In         ; neu gelesener Wert der Eingänge
  STS     Old_In, Reg_A         ; Neuen Wert in Ablage "old_In" kopieren,
; ist hier bereits erforderlich, da Reg_A überschrieben wird
  EOR     Reg_A, Reg_B          ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
  AND     Reg_A, Reg_B          ; Und Verküpfung mit dem alten Status
```

```
  STS   Event_To_LOW, Reg_A     ; liefert die Bits mit Änderung von "1" nach "0"
RET
;-----------------------------------------------------------------------

;*********************************************************************
;*                          Tastersignal prüfen                      *
;*********************************************************************
Taster_Event:
  LDS      Reg_A, Event_To_High      ; Ereignisbits laden
  ANDI    Reg_A, 0b00000001           ; Nur Bit 1 prüfen (Ergebnis ist nicht 0)
  BREQ   End_Taster_Event          ; Ergebnis 0, dann Sprung nach Marke Status_Ende
  LDS      Reg_A, Out_Ctrl             ; Nun das Byte für den Ausgang holen
  LDI      Reg_B, 0b00000010        ; Ist für eine EOR Anweisung erforderlich
  EOR     Reg_A, Reg_B               ; dreht das Bit 1 um, 1 nach 0 und 0 nach 1
  STS      Out_Ctrl, Reg_A     ; nun noch das Register B in die Variable Out_Ctrl schreiben
  LDS      Reg_A, Event_To_High     ; Ereignisbits laden
  ANDI     Reg_A, 0b11111110           ; Ereignisbit 0 löschen
  STS    Event_To_High, Reg_A      ; Und Ereignisse zurückschreiben
End_Taster_Event:                     ; Markiert das Ende der Subroutine
RET
;-----------------------------------------------------------------------------

;********** Eingabe bearbeiten und Ausgabe vorbereiten ************
;*      Variable  New_In enthält Status der Eingänge                    *
;*                                                                        *
;*      Variable Out_Ctrl enthält Ergebnis aus der Bearbeitung      *
;*      für die Ausgabe                                                   *
;*******************************************************************************
Set_LED_Bit:                          ; LED einschalten
   LDS    Reg_A, New_In
   ANDI   Reg_A, 0b00000001     ; Nur Bit 1 prüfen (Ergebnis ist nicht "0")
   BRNE   Set_LED1                  ; Bit= 1, dann  Sprung nach Marke Set_LED1
   LDS    Reg_B, Out_Ctrl           ; Nun das Byte für den Ausgang holen
   ANDI   Reg_B, 0b11011111       ; Mit einer Und-Anweisung das Bit 5 löschen
   RJMP   End_Set_LED_Bit       ; Sprung zum Ende der Sub-Routine
Set_LED1:                             ; Einstieg zum Setzen des Bits
   SWAP  Reg_A                 ; tauscht die unteren 4 Bits mit den oberen 4 Bits (Nibble)
   ROL   Reg_A                 ; Schiebt den Inhalt von Register A noch einmal nach links
   LDS   Reg_B, Out_Ctrl        ; Nun das Byte für den Ausgang holen
   OR    Reg_B, Reg_A           ; und das 5. Bit in Register B setzen
End_Set_LED_Bit:               ; Markiert das Ende der Subroutine
   STS   Out_Ctrl, Reg_B        ; nun noch das Register B in die Variable Out_Ctrl schreiben
RET                               ; Und Rücksprung zum Aufruf
```

```
;-----------------------------------------------------------------

;*********************************************
;*        Empfangene Daten auswerten                          *
; *    empfangenes Byte in Variable     Is_Data_Rec          *
;*     ablegen und in Controll-und Steuervariable             *
;*     Com_Ctrl    den Empfang mit Bit 0 anzeigen             *
;*********************************************
Chk_Receive:
     LDS       Reg_A, Read_Pos          ; Lesezeiger holen
     LDS       Reg_B, Write_Pos         ; Schreibzeiger holen
     CP        Reg_A, Reg_B             ; und vergleichen
     BREQ   End_Chk_Receive             ; Wenn gleich, dann keine Daten und Ende Subroutine
;*********************************************
;*        empfangenes Byte in Arbeitsregister                 *
;*          und Schreibzeiger nachführen                      *
;*********************************************
        LDI      XL,LOW(Com_Buff)        ; -XPointer auf Empfangspuffer
        LDI      XH,HIGH(Com_Buff)
        ADD      XL, Reg_A          ; Schreibzeiger auf Anfangsadresse Ringpuffer addieren
        ADC       XH, Zero           ; 16 Bit-Addition, daher noch Carrybit addieren (Addition mit 0)
        LD       Work, X           ; Inhalt Arbeitsregister in Ringpuffer schreiben
        INC      Reg_A              ; Register A mit Inhalt Schreibzeiger erhöhen
        CPI      Reg_A, 20          ; Grenzwert abfragen
        BRLO   Set_Read_Pos    ; wenn kleiner, dann Wert in Schreibzeiger ablegen
        CLR      Reg_A             ; Schreibzeiger auf 0 setzen
 Set_Read_Pos:
        STS      Read_Pos, Reg_A              ; Schreibzeiger übernehmen
        STS      Is_Data_Rec, Work         ; Empfangenes Byte in Variable schreiben
        LDI      Reg_A, 0b00000001           ; Signalbit setzen
        STS      Com_Ctrl, Reg_A              ; und in Control- und Steuervariable eintragen
End_Chr_Receive:
RET
;-----------------------------------------------------------------

;*****************************************************************************
;*     Prüfen, ob bereits ein Befehl erkannt ist und                  *
;*     gegebenenfalls ein Steuerbit in „Com_Ctrl“ setzen          *
;*****************************************************************************
Is_First:
       LDS       Reg_A, Com_Ctrl           ; Kontroll- und Steuerbyte serieller Empfang
       ANDI     Reg_A, 0b00000001       ; Ereignis ein Byte eingetroffen und
       BREQ   End_Is_First
       LDS       Reg_A, Com_Ctrl           ; liegt in der Variablen  Is_Data_Rec zur Bearbeitung
       ANDI     Reg_A, 0b11111100       ; Ist ein Bit gesetzt, dann ist dies das zweite Byte
```

```
        BRNE    End_Is_First
        LDS     Reg_B, Is_Data_Rec      ; Noch kein Bit im oberen Nibble, Befehl decodieren
        LDI     Reg_A, 0b00000100       ; Bit für „Relais an“ setzen
        CPI     Reg_B, „R“              ; Befehl „Relais ein?
        BREQ    Store_Order             ; ja, dann Befehlsbit ablegen
        LDI     Reg_A, 0b00001000       ; sonst Bit für „Relais aus“ setzen
        CPI     Reg_B, „r“              ; Befehl „Relais aus“?
        BREQ    Store_Order             ; ja, dann Befehlsbit ablegen
        LDI     Reg_A, 0b00010000       ; sonst Bit für Trigger setzen
        CPI     Reg_B, „V“              ; Befehl ist Trigger ?
        BREQ    Store_Order             ; ja, dann Befehlsbit ablegen
        LDI     Reg_A, 0b00100000       ; sonst Bit für Trigger setzen
        CPI     Reg_B, „v“              ; Befehl ist Trigger reset ?
        BREQ    Store_Order             ; ja, dann Befehlsbit ablegen
        LDI     Reg_A, 0b10000000       ; Bit für Werteabfrage der internen Variablen
Store_Order:
        STS     Com_Ctrl, Reg_A
 ; Kontroll- und Steuerwort beschreiben. Dabei wird Bit 0 gelöscht
End_Is_First:
RET
;_____________________________________________________
;*****************************************************************
;*      Zweites Byte eines Befehles auswerten           *
;*****************************************************************
Is_Second:
    LDS     Reg_A, Com_Ctrl         ; Zuerst wieder abfragen, ob Daten eingegangen
    ANDI    Reg_A, 0b00000001       ; Wenn ja, dann Abfragen, ob 2. Byte
    BRNE    Second_Chk:
MARK_1:                             ; Weil Sprungadresse zu weit liegt
   RJMP     End_Is_Second
Second_Chk:
    LDS     Reg_A, Com_Ctrl         ; Daten sind eingegangen, ist ein Bit 2-7 gesetzt
    ANDI    Reg_A, 0b11111100       ; Wenn ja, dann ist 2. Byte eingetroffen.
    BREQ    Mark_1                  ; wenn kein Bit gesetzt, dann nicht 2. Byte
    LDS     Reg_A, Com_Ctrl         ; 2.Byte bearbeiten
    ANDI    Reg_A, 0b00000100       ; Bit für „Relais An“ gesetzt
    BREQ    Chk_Rel_Aus
   RCALL    Set_Relais_On           ; Subroutine für Relais ein, Rel.Nr ist in Work_A
    LDS     Reg_A, Com_Ctrl         ; nun nur noch das Steuerbit löschen
    ANDI    Reg_A, 0b11111010
    STS     Com_Ctrl.  Reg_A
   RJMP     End_Is_Second
Chk_Rel_Aus:
```

```
   LDS        Reg_A, Com_Ctrl
   ANDI       Reg_A, 0b00001000      ; Bit für Relais aus gesetzt
   BREQ       Chk_Trigger
   RCALL      Set_Relais_Off         ; Subroutine für Relais aus, Rel.Nr ist in Work_A
   LDS        Reg_A, Com_Ctrl        ; nun nur noch das Steuerbit löschen
   ANDI       Reg_A, 0b11110110
   STS        Com_Ctrl.  Reg_A
   RJMP       End_Is_Second
Chk_Trigger:
   LDS        Reg_A, Com_Ctrl
   ANDI       Reg_A, 0b00010000       ; Bit für Trigger gesetzt
   BREQ       Chk_TrigerReset
   LDS        Reg_A, IS_Data_Rec      ; dann den Wert des 2. Bytes
   ;LDS       Reg_B, Trigger_In       ;  Variable Trigger in_laden
   ;Or        Reg_A, Reg_B            ; Bit hinzufügen
   ;STS       Ampelzeit, Reg_A
   STS        Trigger_In, Reg_A       ; in die Variable „Trigger_In“
   ; LDI      Reg_A, 88
   ; STS      Ampelphase, Reg_A
   LDS        Reg_A, Com_Ctrl        ; nun nur noch das Steuerbit löschen
   ANDI       Reg_A, 0b11101110
   STS        Com_Ctrl.  Reg_A
   RJMP       Set_Internes
 Chk_TriggerReset:
   LDS        Reg_A, Com_Ctrl
   ANDI       Reg_A, 0b00100000          ; Bit für Trigger gesetzt
;******************  geändert *****************
   BREQ     Chk_Order_V
   STS      Trigger_In, Zero              ;  in Variable "Trigger_In" ablegen
   STS      Trigger_Out, Zero
  ; LDI     Reg_A, 55
  ; STS     Ampelphase, Reg_A
   LDS      Reg_A, Com_Ctrl               ; nun nur noch das Steuerbit löschen
   ANDI     Reg_A, 0b11011110
   ORI      Reg_A, 0b10000000
   STS      Com_Ctrl,  Reg_A
   RJMP     Set_Internes
Chk_Order_V:
   LDS       Reg_A, Com_Ctrl              ; nun nur noch das Steuerbit löschen
   ANDI     Reg_A, 0b10111110
   ORI      Reg_A, 0b10000000
   STS      Com_Ctrl,  Reg_A
   LDS      Reg_A, IS_Data_Rec      ; dann den Wert des 2. Bytes
   STS      Value_Cnt,Reg_A
Set_Internes:
```

```
   LDS      Reg_A, Com_Ctrl
   ANDI     Reg_A, 0b10000000        ; Bit für Trigger gesetzt
   BREQ     End_Is_Second
   RCALL    Send_Value               ; Senderoutine für die Variablenwerte
   RCALL    Send_Value_Int           ; neue Routine senden mit Interrupt
   CLR      Reg_A
   STS      Com_Ctrl.  Reg_A
End_Is_Second:
RET
;-----------------------------------------------------------------

;*****************************************************************
;*                Eine LED blinken lassen                        *
;*****************************************************************

Blinker:
  LDS     Reg_A,Counter_0        ;inneren Schleifenzähler
  INC     Reg_A
  STS     Counter_0, Reg_A
  CPI     Reg_A, 100             ; bis 100 zählen, dann von vorn
  BRLO    End_Blinker
  CLR     Reg_A
  STS     Counter_0, Reg_A       ; mit 100 multiplizieren
  LDS     Reg_A, Counter_1
  INC     Reg_A
  STS     Counter_1, Reg_A
  CPI     Reg_A, 100
  BRLO    End_Blinker
  CLR     Reg_A                  ; bei 10000 wieder von vorn anfangen
  STS     Counter_1, Reg_A
  LDS     Reg_A, Out_Ctrl
  LDI     Reg_B, 0b00100000
  EOR     Reg_A, Reg_B
  STS     Out_Ctrl, Reg_A
End_Blinker:
RET
;-----------------------------------------------------------------
;*****************************************************************
;*                Bereich für die Ausgaben                       *
;*****************************************************************
;******************* Daten an PC senden *******************
;*  diese Routine sendet die Werte der Variablen                *
;*  an einen PC. Das Visual Basic Programm kann                 *
```

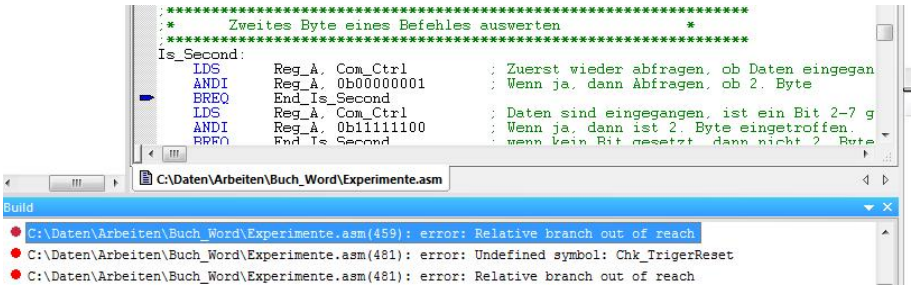
```
;*  aus dem Assemblerlisting die Variablennamen        *
;*  herausfiltern und entsprechend Anzeigeobjekte     *
;*  generieren. Besteht eine serielle Verbindung        *
;*  zum Controller, werden die aktuellen Werte          *
;*  angezeigt.                                            *
;*********************************************************************
Send_Value:
   LDI        Send_Byte , 'V'              ; Telegrammkennung senden
   RCALL   Ser_Out
   LDI        Send_Byte , 'A'
   RCALL   Ser_Out
   LDI        Send_Byte , 'L'
   RCALL   Ser_Out
   LDI        Send_Byte , 'U'
   RCALL    Ser_Out
   LDI         Send_Byte , 'E'
   RCALL    Ser_Out
   LDS        Reg_A, Value_Cnt
; Value_Cnt muß im Initialisierungsteil mit Anzahl der Variablen gesetzt werden
   LDI         XL,LOW(Trigger_In)            ; -XPointer auf Anfang der Variablen
   LDI         XH,HIGH(Trigger_In)
Send_Loop:
   LD          Send_Byte , +X
 ; die Durch X adressierte Variable an Send_Byte übergeben, dabei X um 1 erhöhen
   RCALL     Send_Data                      ; und Senderoutine aufrufen
   Dec         Reg_A                          ; Schleifenzähler erniedrigen
   BRNE       Send_Loop                     ; wenn nicht 0, dann nächste Variable
RET
;-----------------------------------------------------------------------
;*************************************************************************
;*              Bei Verwendung Sendeinterrupt                        *
;*                  Alternativ zu Send_Value                          *
;*************************************************************************
Send_Value_Int:
  PUSH    Reg_A
  LDS      Reg_A, Prg_Ctrl              ; Programmkontrolle
  ANDI     Reg_A, 0b10000000         ; Bit 7  Übertragung läuft?
  BRNE    End_Send_Value
  LDS      Reg_A, Prg_Ctrl              ; Programmkontrolle
  ORI      Reg_A, 0b10000000         ; setze Bit 7  Übertragung läuft
  STS      Prg_Ctrl. Reg_A
  LDI      Send_Byte, “V”               ; Telegrammkennung senden
  RCALL   Send_Data
  LDI      Send_Byte, “A”
  RCALL   Send_Data
```

```
  LDI       Send_Byte, "L"
  RCALL   Send_Data
  LDI       Send_Byte, "U"
  RCALL   Send_Data
  LDI       Send_Byte, "E"
  RCALL   Send_Data
  STS        Send_Cnt, Zero          ; Nutzdatenzähler auf Anfang setzen
  SBI       UCSRB , TXCIE            ; Interruptsteuerung aktivieren
End_Send_Value:
  POP       Reg_A
RET

;---------------- serielle Ausgabe -----------------
;*************************************
;*    Diese Routine sendet ein Byte über        *
;*    die serielle Schnittstelle                *
;*************************************
Send_Data:
   SBIS    UCSRA, UDRE          ; Warten Freigabe UDR
   RJMP    Send_Data
   OUT     UDR, Send_Byte
RET

;*****************************************************************
;*                Ausgabe an die IO-Ebene                        *
;*              Port C Bit 1- 5 = Relais / LED                   *
;*                    ab jetzt geändert !                        *
;*****************************************************************
Write_IO:
  IN        Reg_B, PinC              ; gesamten Port C lesen
  ANDI    Reg_B, 0b11000001      ; alle Ausgänge auf 0
  LDS     Reg_A, Out_Ctrl          ; Variable für Ausgang holen
  ANDI    Reg_A, 0b00111110      ; nur Bit 1-5 gültig
  OR      Reg_B, Reg_A             ; Ausgänge zuschalten, (1 in Out_Ctrl)
  OUT     PortC, Reg_B             ; Port mit Reg_C beschreiben
RET
;-----------------------------------------------------------------


;*******************************************************************
;*            Bereich der Interrupt Serviece Routinen              *
;*******************************************************************
;***********************************************************************
```

```
;*      ISR Datenempfang über serielle Schnittstelle.        *
;*      erforderliche Variablen:                              *
;*      Ringpuffer     Com_Buff    20 Bytes                   *
;*     Schreizeiger    Write_Pos     1 Byte                   *
;*************************************************************************
ISR_REC:
   PUSH    Reg_A                    ; Register A auf dem Stack sichern
   IN      Reg_A, sreg              ; SREG sichern
   PUSH    Reg_A                    ; Statusbits auf dem Stack sichern
   Push    XL                       ; Adressregister XL sichern
   Push    XH                       ; Adressregister XH sichern
   Push    Work                     ; Arbeitsregister sichern
   IN      Work, UDR                ; USART-Empfangsregister in Arbeitsregister
   LDS     Reg_A, Write_Pos         ; Schreibzeiger in Register A
   LDI     XL,LOW(Com_Buff)         ; -XPointer auf Empfangspuffer
   LDI     XH,HIGH(Com_Buff)
   ADD     XL, Reg_A                ; Schreibzeiger + Anfangsadresse Ringpuffer
   ADC     XH, Zero                 ; 16 Bit-Addition, daher Carrybit addieren (Addition mit 0)
   ST      X, Work                  ; Inhalt Arbeitsregister in Ringpuffer schreiben
   INC     Reg_A                    ; Register A mit Inhalt Schreibzeiger erhöhen
   CPI     Reg_A, 20                ; Grenzwert abfragen
   BRLO    End_rxc                  ; wenn kleiner, dann ISR beenden
   CLR     Reg_A                    ; Schreibzeiger auf 0 setzen
End_rxc:                            ; Ausstieg aus ISR
   STS     Write_Pos, Reg_A         ; Zuerst Schreibzeiger speichern
   POP     Work                     ; Dann alten Wert von Arbeitsregister zurückholen
   POP     XH                       ; X-Register wieder herstellen
   POP     XL
   POP     Reg_A                    ; Register A mit Statusbits laden
   OUT     sreg, Reg_A              ; und SREG wieder herstellen
   POP     Reg_A                    ; Register A wieder herstellen
RETI
;-------------------------------------------------------------------------
;*************************************************************************
;*                    Senden mit Interrupt                               *
;*************************************************************************
ISR_Send_Data:
  IN      Ablage_SReg_SREG          ; Sichern aller Register
  PUSH    XL
  PUSH    XH
  PUSH    Reg_A
  Push    Reg_B

  LDI     XL, Low(Trigger_In)       ; Basisadresse Sendeblock
  LDI     XH, High(Trigger_In)
```

```
  LDS     Reg_A, Send_Cnt          ; Datenzeiger
  ADD     XL, Reg_A                ; addieren
  ADC,    Zero                     ; Zero reserviertes  Null-Register
  LD      Send_Byte, X             ; Send_Byte reservierten Register Daten senden
  OUT     UDR, Send_Byte           ; in Sendepuffer eintragen
  INC     Reg_A                    ; Datenzeiger hochzählen
  STS     Send_Cnt, Reg_A          ; zurückspeichern
  LDS     Reg_B, Value_Cnt         ; Anzahl Sendedaten erreicht?
  CP      Reg_A, Reg_B
  BRLO    End_ISR_Send             ; wenn nicht, dann Ende ISR
  CBI     UCSRB, TXCIE             ; Interruptbit  Senden löschen
  LDS     Reg_A, Prg_Ctrl          ; Programmkontrolle Datenübertragung läuft
  ANDI    Reg_A, 0b01111111        ; Bit 7 abschalten = Übertragung beendet
  STS     Prg_Ctrl, Reg_A
End_ISR_Send:                      ; fertig

  POP     Reg_B                    ; Register wieder herstellen
  POP     Reg_A
  POP     XH
  POP     XL
  OUT     SREG, Ablage_SReg
RETI
```

Bei der Übersetzung wird ein Fehler erzeugt Relative branch out of reach.



*Compilerfehler out of reach*

Solche Fehlermeldungen werden euch bei eigenen Entwicklungen öfter begegnen. Ein BREQ oder BRNE also Branch (Sprung) geht nur über die Distanz eines viertel Bytewertes, und das vorwärts als auch rückwärts. Ein viertel Byte sind 63 Adressen vor und 64 Adressen rückwärts und wenn die angegebene Adresse nicht in diesem Bereich liegt, muss eine andere

Lösung her. So ist mit einem kürzeren Sprung bei einer anderen Prüfung das Problem erledigt. Statt mit BREQ einfach BRNE und einen Befehl überspringen. Nicht grad elegant, aber wirkungsvoll.

```
    LDS     Reg_A, Com_Ctrl         ; Zuerst wieder abfragen, ob Daten eingegangen
    ANDI    Reg_A, 0b00000001       ; Wenn ja, dann Abfragen, ob 2. Byte
    BRNE    Second_Chk              ; Abfrage geändert und kurzer Sprung !
    RJMP    End_Is_Second           ; ein Sprungbefehl erreicht sein Ziel.
Second_Chk:
    LDS     Reg_A, Com_Ctrl         ; Daten sind eingegangen, ist ein Bit 2-7 gesetzt
```

## 2.12 Verbinden Open_Eye und µC

Nun ist es endlich soweit. Das Programm sollte nun zusammengestellt sein. Der Assemblerlauf ist auch fehlerfrei ausgeführt und die Anzahl der Variablen in **Value_Cnt** eingetragen, die mit Hilfe von Open_Eye ermittelt wurden. Dazu starten wir **Open_Eye** und kopieren den gesamten Variablenbereich ab **Trigger_In** in den Filter von **Open_Eye**.

Mein **Open_Eye** hat 37 Variablen erkannt. Also trage ich in der Initialisierung der Register und Variablen entsprechend den Wert in die Variable **Value_Cnt**

```
;*****************************************************************************
;*  Neue Initialisierungsroutine. Muß vor der Programmschleife    *
;*  in Initialisierungsteil aufgerufen werden.                    *
;*****************************************************************************
Init_Var:
    CLR     Zero                ; Register „Zero“ auf 0 setzen
    STS     Read_Pos, Zero      ; des Lese-
    STS     Write_Pos, Zero     ; und des Schreibzeigers
    STS     Com_Ctrl, Zero      ; Wichtig, sonst fehlerhafte Kommunikation
    LDI     Reg_A, 37           ; Wert vom Filter aus Open_Eye ** nachgetragen
    STS     Value_Cnt, Regf_A   ; Anzahl der Variablen für Sendeauftrag  nachgetragen
RET
```

Nun nur noch in Open_Eye die Schnittstellenparameter einstellen und die Verbindung sollte stehen. Mit dem **Testen-Button** kann der Datenempfang überprüft werden.

Sollte dies nicht auf Anhieb gelingen, dann geht es an die Fehlersuche mit dem Blinker. Begonnen wird in der ISR Datenempfang.

Wenn hier schon keine Reaktion erfolgt, dann haben wir vielleicht vergessen, im Initialisierungsteil den Hauptschalter für die Interrupts mit dem **SEI**-Befehl zu setzen.

Auch ein Hardwarefehler ist denkbar.

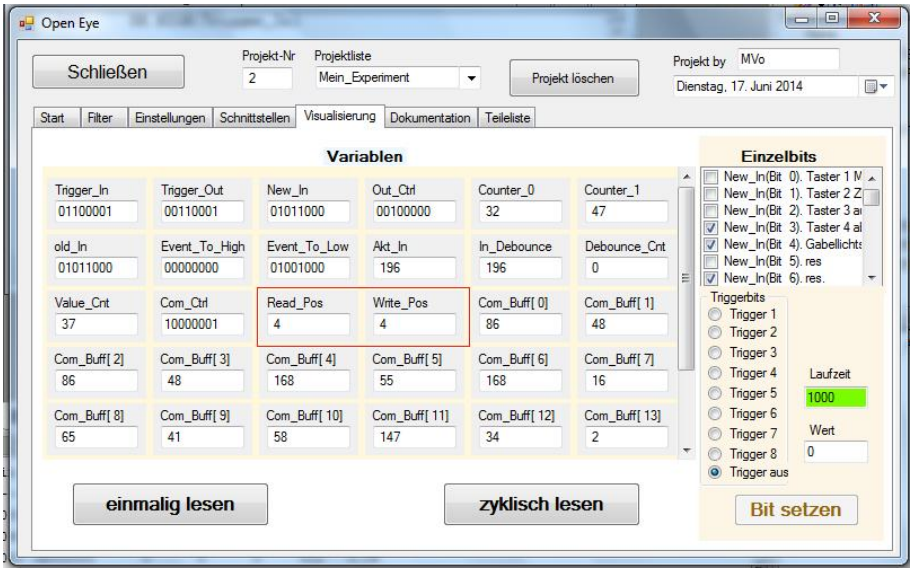
Ist die serielle Verbindung PC zum µC hergestellt?

Sind die richtigen Stecker am Board gesteckt?

Ist alles richtig, und es werden auch Daten empfangen, was man anhand der LED beobachten kann, aber keine Daten an den PC zurückgesendet, kann es auch sein, dass ein falsches Zeichen gesendet wurde. Nur ein V und ein weiteres beliebiges Zeichen schickt Daten zurück.

Sind Abschnitte im Programm geprüft, werden die Testzeilen wieder entfernt. Wer will, kommentiert sie nur aus. Dadurch wird das Programm nicht größer, denn Kommentarzeilen landen nicht im µC-Programm. Doch trotzdem sollte man es mit auskommentierten Befehlen nicht übertreiben, um die Übersichtlichkeit nicht zu verlieren.

Ist es nun endlich gelungen, mit dem Test-Button auch eine Antwort zu erhalten und wird auch die Telegrammkennung korrekt angezeigt, dann können wir auch die Variablen auf der Ansichtsseite beobachten und die Werte auslesen. Zuerst bringt ein Blick auf die Variablen **Read_Pos** und **Write_Pos** Gewissheit, dass es sich um Werte aus dem Controller handelt. Sie werden bei jedem Anfordern von Daten um 2 größer bis max. 20. Dann beginnen sie wieder von 0 in 2er Schritten zu zählen.



*OpenEye aktiv*

Ist ja auch klar, wir werten erst das zweite empfangene Byte für den Sendeauftrag aus.

Betrachten wir einmal die Information der Variableninhalte. Der erste wirklich passende Eintrag ist in Value_Cnt. Auch Com_Ctrl ist interessant. Dort steht, das ein Byte Empfangen und das Bit 7 zeigt, das es ein V war und Read_Pos sowie Write_Pos werden sich bei jeder Datenanforderung verändern, aber den gleichen Wert besitzen. Ab jetzt sind wir in der Lage, unser Programm auch über die Variableninhalte zu kontrollieren und das korrekte Arbeiten des Controllerprogramms auch anhand von Werten überprüfen.

### *2.12.1 Testen der Triggerfunktion.*

Schritt für Schritt werden nun die Programmteile getestet. Sind die Testanfragen positiv, gehen wir zur Visualisierung der Variablenwerte. Hier haben wir ein besonderes Bonbon eingebaut, den Trigger. Auch diesen sollten wir testen. Dabei ist zu bedenken, dass wir ja gar keine Überwachung mit **Trigger** bisher eingebaut haben. Das holen wir jetzt nach.

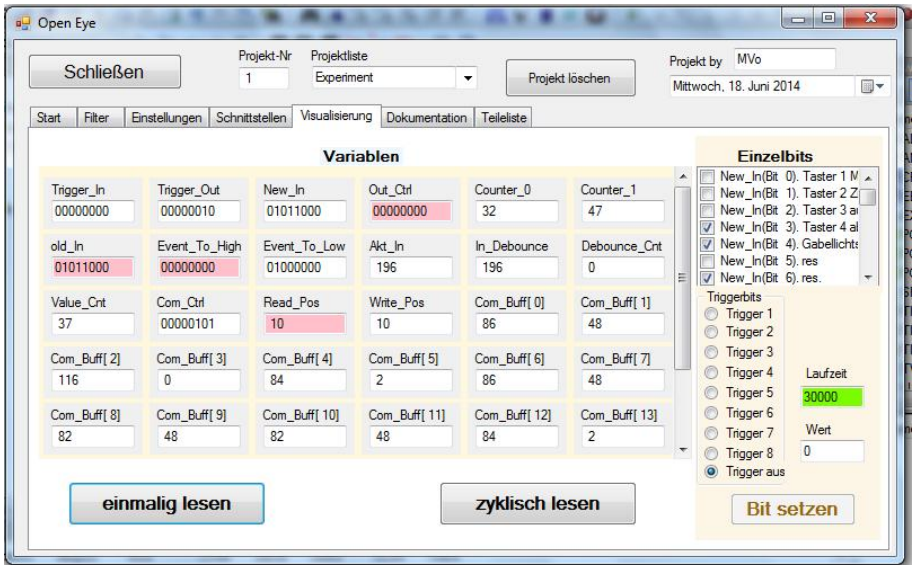
Wir wissen, welchen Weg die Programmbearbeitung geht, also bauen wir in diesen Weg irgendwo diesen **Trigger** ein. Viel Auswahl haben wir nicht, denn in die Empfang- und Empfangsauswerteroutine einzugreifen macht nicht viel Sinn. Oder vielleicht doch. So sind z. B. Die **Relais_On** und **Relais_Off** -Routinen noch völlig leer. Testen wir doch einmal, ob es uns gelingt, getriggerte Werte zu erhalten. Dafür erweitern wir die Routine **Set_Relais_On**

```
;*************************************************************
;*    Relais per Befehl vom PC einschalten     *
;*************************************************************
Set_Relais_On:
   LDS      Reg_A, Trigger_In                ; Variable Trigger_In prüfen
   ANDI     Reg_A, 0b00000010                ; Ist Bit 1 gesetzt ?
   BREQ     Go_Set_Rel                       ; Wenn nicht, dann normal weiter
   STS      Trigger_Out, Reg_A               ; Trigger zurückschicken
   LDS      Reg_A, Trigger_In                ; Variable Trigger_In prüfen
   ANDI     Reg_A, 0b11111101                ; Bit 1 löschen
   STS      Trigger_In, Reg_A                ; und Triggerbit in Trigger_In quittieren
   RCALL    Send_Value                       ; Anstoß Senden an Open_Eye
Go_Set_Rel:
;-------------------------------------------------------
; normale Bearbeitung
RET
```

Nun werden wir ein paar Variablen auf dieses Ereignis ansetzen. Dafür setzen wir in der Einstellung der Variablen für die Variablen **Read_Pos**, **Event_To_High** und **Out_Ctrl** den **Trigger** auf **2**. Anschließend generieren wir die Ausgabe der Variablenwerte neu, damit die Trigger in die Ausgabeobjekte übernommen werden. Nun testen wir das Ergebnis, indem wir den Trigger auf der Visualisierung vorgeben

Es passiert erst einmal nichts, da die Routine ja nicht angesprochen wird. Die Überwachungszeit läuft ab und der Trigger wird gelöscht. Setzen wir die die Überwachungszeit auf 30 Sekunden. Das bringt uns Zeit, einen

Relaisbefehl einzugeben. Nun sollte die Routine Set_Relais_On durchlaufen werden. Dabei wird geprüft, ob das Triggerbit 2 gesetzt ist. Ist dies der Fall, werden sofort die Variableninhalte zum PC versendet. Im Programm **Open_Eye** erkennen wir dies an den eingefärbten Variablenfeldern, die speziell auf dieses Triggerereignis reagieren sollen.



*Test mit Trigger*

Sind die Ergebnisse wie gewünscht, werden wir nun ein wenig mit der Variablen **Out_CTRL** spielen. Sie soll ja nach der Bestimmung mit Bit 0-4 fünf Relais schalten. Dazu haben wir bestimmt, dass die Befehle Vom PC **Rn** lauten, wobei **n** für **1-5** steht.

### *2.12.2 Relais ein- und ausschalten*

Diese Routinen werden im Unterprogramm **Is_Second** aufgerufen und sind nicht in der Main_Loop zu finden.

```
;*************************************************************
;*    Relais per Befehl vom PC einschalten        *
;*************************************************************
Set_Relais_On:
      LDS       Reg_A, Is_Data_Rec                ; Zuerst die Relaisnummer laden
      LDS       Reg_B, Out_Ctrl                   ; und die Variable Out_Ctrl
      CPI       Reg_A, '1'                        ; Dann die Relasinummer prüfen
      BRNE    Chk_Rel_2_On
      ORI       Reg_B, 0b00000010      ; entprechend zur Relaisnummer Bit setzen
      RJMP     Set_Out_On
Chk_Rel_2_On:
      CPI       Reg_A, '2'
      BRNE     Chk_Rel_3_On
      ORI       Reg_B, 0b00000100      ; entprechend zur Relaisnummer Bit setzen
      RJMP     Set_Out_On
Chk_Rel_3_On:
      CPI       Reg_A, '3'
      BRNE     Chk_Rel_4_On
      ORI       Reg_B, 0b00001000     ; entprechend zur Relaisnummer Bit setzen
      RJMP     Set_Out_On
Chk_Rel_4_On:
      CPI       Reg_A, '4'
      BRNE     Chk_Rel_5_On
      ORI       Reg_B, 0b00010000     ; entprechend zur Relaisnummer Bit setzen
      RJMP    Set_Out_On
Chk_Rel_5_On:
      CPI       Reg_A, '5'
      BRNE    End_Rel_On                ; keine gültige Relaisnummer erkannt
      ORI       Reg_B, 0b00100000       ; entprechend zur Relaisnummer Bit setzen
Set_Out_On:
      STS       Out_Ctrl, Reg_B
End_Rel_On:
      RCALL   Send_Value
; Zu Testzwecken eingefügt. Bei verdrahteter Hardware entfernen
RET
```

Damit wir auch die Relais wieder abschalten können, gleich die Ausschaltroutine dazu.

```
;*****************************************************************
;*      Relais per Befehl vom PC ausschalten              *
;*****************************************************************
 Set_Relais_Off:
      LDS       Reg_A, Is_Data_Rec                ; Zuerst die Relaisnummer laden
      LDS       Reg_B, Out_Ctrl                   ; und die Variable Out_Ctrl
      CPI       Reg_A, '1'                         ; Dann die Relasinummer prüfen
      BRNE      Chk_Rel_2_Off
      ANDI      Reg_B, 0b11111101          ; entprechend zur Relaisnummer Bit löschen
       RJMP     Set_Out_Off
Chk_Rel_2_Off:
      CPI        Reg_A, '2'
      BRNE       Chk_Rel_3_Off
      ANDI       Reg_B, 0b11111011          ; entprechend zur Relaisnummer Bit löschen
      RJMP       Set_Out_Off
Chk_Rel_3_Off:
      CPI        Reg_A, '3'
      BRNE       Chk_Rel_4_Off
      ANDI       Reg_B, 0b11110111          ; entprechend zur Relaisnummer Bit löschen
      RJMP       Set_Out_Off
Chk_Rel_4_Off:
      CPI        Reg_A, '4'
      BRNE       Chk_Rel_5_Off
      ANDI       Reg_B, 0b11101111            ; entprechend zur Relaisnummer Bit löschen
      RJMP       Set_Out_Off
Chk_Rel_5_Off:
      CPI          Reg_A, '5'
      BRNE         End_Rel_Off                  ; keine gültige Relaisnummer erkannt
      ANDI         Reg_B, 0b11011111              ; entprechend zur Relaisnummer Bit löschen
Set_Out_Off:
      STS          Out_Ctrl, Reg_B
End_Rel_Off:
      RCALL        Send_Value
 ; Zu Testzwecken eingefügt. Bei verdrahteter Hardware entfernen
RET
```

Der Unterschied beider Routinen liegt in der Art der Verknüpfung. Ein einzelnes Bit in **Out_Ctrl** wird mit einer **Oder**-Verknüpfung gesetzt. Dabei ist nur dieses eine relevante Bit in der Konstanten 1, alle anderen sind **0**

Der Umkehrschluss, ein Bit wieder abschalten, geschieht mit einer **UND**-Verknüpfung. Dabei ist nur das eine relevante Bit in der Konstanten **0**, alle anderen sind **1**.

Der Aufruf der Sub-Routine **Send_Value** ist hier nur für Testzwecke eingefügt. Wir haben ja keine 5 Relais, um die Wirkung zu signalisieren. Dies werden wir erst aufbauen, wenn die Variable **Out_Ctrl** richtig arbeitet. Dann wird auch der Aufruf RCALL **Send_Value** aus beiden Routinen entfernt und wieder die Triggerprüfung eingesetzt.

Doch noch haben wir im Bereich der Software viel abzuarbeiten.

Testen wir nun die beiden Routinen. Dazu müssen wir uns in **Open_Eye** im Visualisierungsfenster befinden. Allerdings, und das ist etwas unpraktisch, wird der Sendebefehl in der Seite der Schnittstelleneinstellung festgelegt. Wir haben nun zwei Möglichkeiten:

Ohne Änderung von **Open_Eye** die Seite wechseln, den Sendebefehl ändern und zurück zur Visualisierung. Dann einmalig die Daten anfordern. Die Variable **Out_Ctrl** liefert das Ergebnis.

Der andere Weg ist, **Open_Eye** anzupassen und den Sendebefehl für **Daten Anfordern** auch auf die Seite der Visualisierung zu bringen. Da wir **Open_Eye** selbst geschrieben haben, sollte diese Anpassung kein Problem sein. Einfach eine TextBox unter dem Button **einmalig lesen** platzieren, einen Namen vergeben und in der Ereignisroutine **Click** des Buttons den Inhalt vom Sendeauftrag anpassen.

Dabei gilt die einzige Überlegung, ob der Inhalt vom Einstellfenster, der ja in der Datenbank abgespeichert wird, nicht grundsätzlich der Default - Wert für Sendeaufträge bleibt. Dieser gespeicherte Wert bliebe für das Test-Button auf der Schnittstellenseite weiterhin aktuell. Lediglich beim Laden eines Projektes muss dann die neue TextBox mit diesem Defaultwert geladen werden.

Versucht, diese Anpassung selbst durchzuführen.

Der Weg:

**Open_Eye** Programm stoppen

In die Entwurfsansicht gehen

TextBox auf die Seite **Visualisierung** einfügen

Ereignisroutine von Button **einmalig lesen** ändern und Text des neuen Textfeldes dem Textfeld **TB_Auftrag** zuweisen.

Nun braucht nicht mehr zwischen den Seiten gewechselt werden. Allerdings, und daran sollte man auch denken, hat diese Vorgehensweise einen kleinen Haken. Es funktioniert nur solange, bis der Aufruf der Routine **Send_Value** aus den Relaisroutinen herausgenommen wird. Danach reagiert die Visualisierung nur noch auf einen mit **Vx** eingegebenen Befehl, wobei **x** für beliebiges Zeichen steht, oder auf einen Trigger. Deshalb sollten wir der TextBox noch ein Label verpassen **nur für Testzwecke**

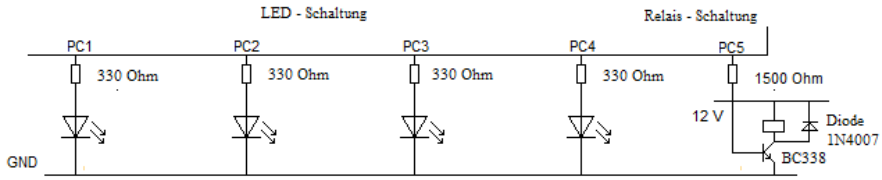
Ansonsten sollte die Voreinstellung zum Abruf der Variablenwerte eingetragen sein. Im Moment aber hilft es unser Programm zu prüfen und die Bits in der Variablen **Out_Ctrl** zu beeinflussen.

## 2.13 IO- Pin Ausgang zuweisen

Doch damit ist noch kein Ausgang gesetzt. Dies wollen wir nun angehen und die 5 Relais auf den Ausgängen verteilen. Dazu erweitern wir die bereits vorhandene Routine Write_IO. Statt Relais lassen sich auch LED für das Experiment einsetzen.

Zuerst fertigen wir einen Schaltplan an und verdrahten ein paar LEDs mit Vorwiderstand zwischen 270 und 330 Ohm oder, wenn vorhanden, auch Relais. Aber dann nicht die Freilauf-Diode vergessen! Auch im Schaltplan ist sie wichtig. Ein Blick auf unseren Schaltplan von Seite 573 zeigt, das schon Ein-und Ausgänge geplant sind. #Schaltplan Ziel der Experimente

In der Initialisierung der IO-Ports ist dies ja auch schon berücksichtigt. Also beschalten wir die vorgesehenen Ausgänge.



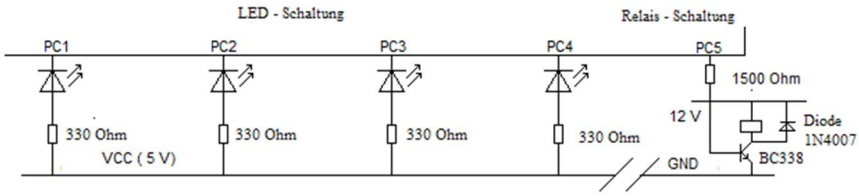

*Bechaltung mit LED und Relais*

```
;*************************************************************
;*          Ausgabe an die IO-Ebene                  *
;*        Port C Bit 1-5 = Relais / LED              *
;*************************************************************
Write_IO:
   IN      Reg_B, PortC                ; gesamten Port C lesen
   ANDI    Reg_B, 11000001             ; Bit 1-5 löschen
   LDS     Reg_A, Out_Ctrl             ; Variable für Ausgang holen
   ANDI    Reg_A, 00111110             ; Bit 0, 6 und 7 ausmaskieren
   Or      Reg_B, Reg_A                ; Register a und B zusammenführen
   OUT     PortC, Reg_B                ; Port beschreiben
RET
```

Es ist nicht zwingend erforderlich, alles zu beschalten, aber so kann jeder vorgesehene Ausgang von Port C schon einmal geprüft werden.

### *2.13.1 Inverse Ausgänge zuweisen*

Wenn wir uns unsere Schaltung genauer ansehen, ist ein 7 Segment-Anzeige mit gemeinsamer Kathode eingezeichnet. Eine gemeinsame Kathode wird mit einem NPN Transistor und der Signallage 0 geschaltet, also lägen die LED genau anders herum und der **GND** käme vom Controller. Bauen wir die Schaltung einmal anders auf, so wie die Anzeige später ihre LED verschaltet hat. Das Relais bleibt.




*Bechaltung mit LED invers und Relais*

Nun liegen die Signale an den LED genau invers. Eine leuchtende LED wird mit Status 0 erreicht. Für unser logisches Denken ist dies eine Herausforderung. Wenn hier einfach die Verarbeitung umgestellt wird, stimmt irgendwann die Zuordnung nicht mehr, denn Licht an = 1. So ist unser Denken. Deshalb wird auch in der Verarbeitung weiter mit diesem Denken gearbeitet und die Anpassung in der Ausgabe Write_IO erledigt. Nehmen wir uns diese Routine noch einmal vor. Was ist zu tun? Nach wie vor müssen wir den Status vom Port lesen. Da jetzt die Bits1 bis 4 0 aktiv sind,, müssen wir sie umdrehen. Dann erfolgt die bisherige Verarbeitung und vor der Ausgabe werden die Bits 1-4 wieder umgedreht. Dazu brauchen wir noch ein Register für eine EOR Anweisung (XOR)

```
;*************************************************************
;*          Ausgabe an die IO-Ebene                          *
;*        Port C Bit 1-5 = Relais / LED                      *
;*************************************************************
Write_IO:
  IN      Reg_B, PortC              ; gesamten Port C lesen
  LDI     Reg_C, 0b00011110         ; Maske für EOR Bit 1 - 4
  EOR     Reg_B, Reg_C              ; nur Bits 1 bis 4 werden gedreht
  ANDI    Reg_B, 0b11000001         ; Bit 1-5 löschen
  LDS     Reg_A, Out_Ctrl           ; Variable für Ausgang holen
  ANDI    Reg_A, 0b00111110         ; Bit 0, 6 und 7 ausmaskieren
  Or      Reg_B, Reg_A              ; Register a und B zusammenführen
  EOR     Reg_B, Reg_C              ; nur Bits 1 bis 4 werden zurück gedreht
```

```
  OUT    PortC, Reg_B                ; Port beschreiben
RET
```

Mit diesen zusätzlichen Anweisungen ist es überhaupt kein Problem, die Verarbeitung mit der Annahme Licht an = 1 durchzuführen. Das ist eine prima Möglichkeit, Denkfehler zu vermeiden und die Logik bleibt auch einfach. So helfen kleine Programme, die Anpassung nur an einer Stelle vorzunehmen. Das ist dann schnell geprüft und wenn fehlerfrei, dann ab zu den Akten. Ab jetzt sind Fehler außerhalb dieser Routine zu suchen.

Sind wir bis hierher erfolgreich vorgedrungen, dann steht uns nun :

a) eine Kontrolle der internen Vorgänge des Controllers zur Verfügung.

b) wir können Relais vom PC aus schalten

c) wir können im PC Werte vom Controller verarbeiten (z.B. visualisieren)

Wir haben Tastereingänge auf einen Ausgang gelegt, aber nach dem Entprellen und der Flankenbildung abgebrochen. Der Grund war, dass nun unser Programm im Controller doch schon ein paar nette Aufgaben erledigen soll und wir gern sehen möchten, wenn unsere Überlegungen falsch sind. Das Werkzeug dafür können wir nun dank funktionierender serieller Kommunikation auch nutzen. Bevor ich nun mit der anfangs vorgestellten Schaltung beginne, ist noch eine wichtige Hürde zu nehmen. Es ist bereits beim Entprellen aufgefallen, dass die Wartezeit nicht zuverlässig gleich ist, wenn sie an den Programmzyklus gebunden wird. Andererseits, eine berechnete Warteschleife blockiert das gesamte Programm, so dass dieser Weg auch keinen Erfolg verspricht. Welche Möglichkeiten gibt es denn sonst noch, eine zuverlässige gleichbleibende vom Programm unabhängige Entprellzeit zu bekommen. Das Stichwort: der Timer.

## 2.14 Der Timer

Ich weiß nicht warum, doch viele Anfänger scheuen sich, dieses geniale Teil eines Controllers zu nutzen. Vielleicht ist es Angst vor Interrupts, davor etwas zu benutzen, was nicht gleich verständlich ist. Dieser Abschnitt macht Schluss, mit dieser Hürde. Der **Timer** ist eines der wichtigsten Bestandteile eines Controllers. Seine umfangreichen Funktionen lassen ihn vielleicht etwas schwierig erscheinen, aber wir müssen ja nicht alle Funktionen bis ins kleinste Detail kennen. Wir müssen erst einmal anfangen, ihn zu einzusetzen. Werden später, wenn man es gelernt hat, die Datenblätter anzuwenden, neue Funktionalitäten eines **Timers** erforderlich, so ist dies gewiss kein großes Problem mehr, wenn schon eine Arbeit damit erfolgreich war.

Uns wird der **Timer** helfen, die Signale von Schaltern zu Entprellen und er wird uns Zeittakte liefern, für Aufgaben, die zeitrelevant sind.

So denke ich im banalsten Fall an eine blinkende **LED**, etwas weiter an eine Zeitanzeige oder eine Stoppuhr. Weitere Aufgaben können sich noch ergeben. Lassen wir uns überraschen.

Der **Timer** ist eigentlich nur ein Zähler, der direkt am **Takt** des **Controllers** hängt und die eingehenden **Takte** zählt. Dafür hat er ein Register und wenn dieses voll ist, dann fängt er von vorn an. Das allein nutzt uns noch nichts, denn wir möchten schon eine exakte Zeitbasis haben. Die bekommen wir bei der Betrachtung, wie lang ein Takt ist. Also bei f=16 MHz ist die Taktzeit exakt 1/ f = 1/16000000 und das entspricht 0,0000000625 Sekunden oder 62,5 nSek.

Rein theoretisch müsste ich, um eine mSek. zu erhalten, bis 16 000 zählen. Dies ist nur mit **Timer1** möglich, denn er besitzt ein 16 Bit-Zählregister, während **Timer0** und **Timer2** nur 8 Bit Zählregister haben.

Wär ja nun Klasse, wenn dann dieser **Timer** nach 16000 Takten ein Signal gibt und von vorn beginnt.

Ja, man kann ihn entsprechend einstellen. Auch hier ist das Ergebnis ein **Interrupt**, der bei einem Zählwert ausgelöst wird.

Dazu muss der **Timer** wie auch der **USART** erst einmal mit einer Initialisierung vorbereitet werden. Aufgrund der verschiedenen Funktionen ist bei der Vorbereitung auch einiges zu beachten. So kann ein Vorteiler

schon die Taktfrequenz, wie der Name sagt, teilen. Dann reicht zum Beispiel bei einem Vorteiler von 8 ein Vergleichswert von 2000 statt 16000, um jede Millisekunde einen Interrupt zu erhalten. Für unsere Anforderung muss der Timer im **CTC** -Mode laufen. Das ist genau die Funktion die wir brauchen **Clear Timer to Compare Match** so steht es im Datenblatt und das bedeutet, er setzt den Zählwert bei Erreichen des Vergleichswertes auf 0. Somit beginnt die Zählerei von vorn. Da der Zählvorgang bei 0 beginnt, sind 2000 Zählimpulse nicht bei 2000 erreicht, sondern bei 1999. Dies mag etwas ungewöhnlich sein, aber im Dezimalsystem rechnen wir exakt mit 10 Ziffern → 0 bis 9. Wir vergessen dies, und zählen immer von 1 bis 10. In der Elektronik müssen wir uns aber an die mathematischen Vorgaben halten und dahingehend umdenken, dass die 0 immer mitgezählt wird.

### *2.14.1 Die Timer-Initialisierung*

```
;---------------------- Timer 1 Parametrierung ------------------
;*************************************************************************
;*   Timer 1 ist die Zeitbasis für alle zeitabhängigen          *
;*   Ereignisse. Er erzeugt einen Interrupt je mSek In          *
;*   den Zeitregistern weden diese mSek. aufaddiert und  *
;*   die Zeitbasis gebildet. Ist vielleicht mal für eine          *
;*   andere Anwendung von Interesse. Aber die Zeitbasis  *
;*   ist abhängig vom Takt. In diesem Fall muß die            *
;*   Taktfrequenz 8 MHz sein.                                     *
;*************************************************************************
Init_Timer1:                                ; Einstellung 8 MHz
   LDI     Reg_A, high( 1000 - 1 )          ; Set Compare Value für eine msek
   OUT     OCR1AH, Reg_A                    ; bei 16 MHz 2000-1 eintragen
   LDI     Reg_A, low( 1000 - 1 )
   OUT     OCR1AL, Reg_A
;++++++++++++++++++++++++++++++++++++++++++++
;* Vorteiler CS12 CS11 CS10 *
;* 0 0 0 No Timer *
;* 0 0 1 Teiler 1 *
;* 0 1 0 Teiler 8 *
;* 0 1 1 Teiler 64 *
;* 1 0 0 Teiler 256 *
;* 1 0 1 Teiler 1024 *
;* 1 1 0 Zähler 1-0 *
;* 1 1 1 Zähler 0-1 *
;*********************************************
; CTC Modus einschalten Vorteiler auf 8
   LDI     Reg_A, ( 1 << WGM12 ) | ( 1 << CS11 )     ; Bit WGM 12 ist CTC Modus ein
   OUT     TCCR1B, Reg_A                             ; Bit CS11 ist Teiler 8 vorgeschaltet
; OCIE1A: Interrupt bei Timer Compare
   LDI     Reg_A, 1 << OCIE1A
   OUT     TIMSK, Reg_A
RET
```

Den Sprung zu dieser Routine tragen wir im Initialisierungsteil vor der Programmschleife ein. Sie soll ja nur ein einziges Mal durchlaufen werden.

Wir sehen hier in den Kommentarzeilen, welche Vorteiler es gibt. Der Referenzwert ist dem Takt des Controllers entsprechend anzupassen. Diese Initialisierung eines **Timers** zeigt, dass es gar nicht so viel Programm ist, wenn man weiß, welche Funktion man braucht. Daher ist es wichtig, sich immer das Datenblatt des verwendeten Controllers vorzulegen und nachzulesen. Nur so kann entschieden werden, welche Controllerfunktion zur eigenen Anwendung erforderlich ist und eingesetzt wird.

Diesen Programmblock können wir fast ohne Änderung in anderen Anwendungen immer wieder einsetzen und somit die Zeitbasis innerhalb einer Anwendung generieren. Eventuell ist der Vorteiler an den Takt des Controllers anzupassen.

Den Aufruf der Initialisierung schreiben wir in den Initialisierungsteil hinzu

```
;*****************************************************************
;*                    Bereich Initialisierung                    *
;*****************************************************************
Start:
LDI   Reg_A, High(RamEnd)          ; erste Maßnahme Stack setzen
OUT   SPH, Reg_A
LDI   Reg_A, Low(RamEnd)
OUT   SPL, Reg_A
          ; weitere Initialisierungen
RCALL   IniT_IO
RCALL   Ini_USART                  ; serielle Schnittstelle parametrieren
RCALL   Init_Timer1                ; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
RCALL   Init_Var        ;Initialisierung Variablen ( Eintragen Defaultwerte)
```

### *2.14.2 Die Timer ISR*

Wie bereits beim **USART** gibt es auch hier in der **IVT** eine Stelle, wo ein **Interrupt** landet. Hier müssen wir nun den **RETI**-Befehl entfernen und einen **RJMP**-Befehl zur **ISR**, also zur **Interrupt Service Routine** setzen.

Dazu brauchen wir aber zuerst die **ISR**.

Wir kennen bereits aus der **ISR** vom Datenempfang, das die verwendeten Register erst einmal auf dem Stack zu sichern sind Des Weiteren wissen wir, wie mit Signalbits gearbeitet wird, z.B. mit der Variablen **Com_Ctrl**. Daher werden wir auch eine Variable einrichten, die verschiedene Zeitinformation an das Hauptprogramm liefert. Ich nenne sie Time_Flag

```
Time_Flag: .Byte 1      ; Byte für Zeitsteuerung
                        ; Bit 0 1 mSek
                        ; Bit 1 10 msek
                        ; Bit 2 20 mSek
                        ; Bit 3 50 mSek
                        ; Bit 4 100 mS
                        ; Bit 5 500 mSek
                        ; Bit 6 1 Sek
                        ; Bit 7 res.
```

Wenn die Variable so deklariert wird, kann sie von **Open_Eye** auch im Einzelbitstatus betrachtet werden.

**Aber Achtung: Es ist eine weitere Variable dazugekommen und das muss in der Variableninitialisierung auch eingetragen werden. Daher immer darauf achten, das die Anzahl der Variablen, die durch den Filter berechnet wurden, auch in der Initialisierung eingetragen ist.**

Alles, was wir nun in der **Timer-ISR** zu tun haben, ist, diese Signalbits zu setzen. Das hat auch seinen guten Grund:

Eine **ISR** darf nicht so viel Programm bearbeiten, denn dies belastet die Zykluszeit ungemein. Außerdem besteht die Gefahr, speziell beim **Timer**, das das nächste Zeitereignis eintritt und die **ISR** noch gar nicht fertig ist. So dauert bei 8 MHz und ca. 2 Takte pro Befehl die Bearbeitung 1/8000000*2 also ca 250 nSek. Das ist immer noch schnell, aber das bedeutet auch, dass in 1 mSek nur ca. 4000 Befehle abgearbeitet werden können. Na ja, auch noch eine ganze Menge, aber der Teufel steckt im

Detail. Wir werden diese Thematik bei der Ansteuerung der Siebensegmentanzeige noch einmal aufgreifen.

Außerdem brauchen wir noch Variablen für die Zähler. Damit es schneller geht, habe ich Register aus dem unteren Adressbereich genommen. Die können auch fast alles, allerdings keine direkte Bearbeitung mit Konstanten. Deshalb sind Register für festen Vergleich mit Defaultwerten besetzt und dienen hier als Konstante. Die Vorbesetzung tragen wir in der Register- und Variableninitialisierung ein.

```
;*******************************************************************************
;*  Neue Initialisierungsroutine. Muß vor der Programmschleife  *
;*   in Initialisierungsteil aufgerufen werden.                  *
;*******************************************************************************
Init_Var:
    CLR    Zero                    ; Register „Zero“ auf 0 setzen
    STS    Read_Pos, Zero          ; des Lese-
    STS    Write_Pos, Zero         ; und des Schreibzeigers
    STS    Com_Ctrl, Zero          ; Wichtig, sonst fehlerhafte Kommunikation
    LDI    Reg_A, 2                ; Konstante 10 laden ** nachtragen
    MOV    Zwei, Reg_A             ; In Referenzregister übertragen ** nachtragen
    LDI    Reg_A, 5                ; Konstante 10 laden ** nachtragen
    MOV    fuenf, Reg_A            ; In Referenzregister übertragen ** nachtragen
    LDI    Reg_A, 10               ; Konstante 10 laden ** nachtragen
    MOV    Zehn, Reg_A             ; In Referenzregister übertragen ** nachtragen
    LDI    Reg_A, 38               ; Wert vom Filter aus Open_Eye
    STS    Value_Cnt, Regf_A       ; Anzahl der Variablen für Sendeauftrag

RET
```

Die Register, die wir für die Zählwerte und Vergleichskonstanten gewählt haben, dürfen nicht auf dem Stack gesichert werden. Sie sollen ja die Werte aus der ISR heraus beibehalten und werden sonst nirgends im Programm verändert.

### *2.14.3 Das Programm der Timer ISR*

```
;---------------- Interruptroutine Timer 1 für Zeitbasis ------------------
;*****************************************************************************
;*   Die Aufgabe dieser Routine ist eine Zeitbasis für das                 *
;*   Programm zu liefern. Alle hier und in den von hier                    *
;*   aufgerufenen Unterprogrammen benutzten Registerinhalte                *
;*   müssen gesichert werden. Die Zeiteinheit sind  abgestuft              *
;*   MilliSekunden bis Sekunden.  Größere  Zeiteinheiten werden  *
;*   im Programm  abgeleitet. Damit sind alle Zeitgesteuerten              *
;*   Aktionen  abgedeckt.                                                  *
;*****************************************************************************
Isr_Timer1:
   PUSH    Reg_A                          ; Register sichern
   IN      Ablage_SReg, SREG              ; SREG in reservierten Register sichern
   LDS     Reg_A, Time_Flag
;************************* 1 msek*************************
   ORI     Reg_A, 0b00000001              ; Zeitflag 1 mSek. setzen
   INC     MilliSek
   CP      MilliSek, Zehn
   BRLO    End_ISR_Timer1
;************************* 10 msek *************************
   CLR     MilliSek
   ORI     Reg_A, 0b00000010              ; Zeitflag 10 mSek. setzen
   INC     Zehn_Milli
   CP      Zehn_Milli, zehn
   BRLO    End_ISR_Timer1
;************************* 100 msek ************************
   CLR     Zehn_Milli
   ORI     Reg_A, 0b00000100              ; Zeitflag 100 mSek setzen
   Inc     Zehntel
   CP      Zehntel, zehn
   BRLO    End_ISR_Timer1
;************************* 1 Sek *************************
   ORI     Reg_A, 0b00001000              ; Zeitflag 1 Sek setzen
   CLR     Zehntel
End_ISR_Timer1:
   STS     Time_Flag, Reg_A
   POP     Reg_A
   OUT     SREG, Ablage_SReg
RETI
```

Nun können wir die Auskommentierung in der **IVT** aufheben und den Interrupt freigeben mit dem **Befehl RJMP ISR_Timer1**.

Das Ergebnis ist in **Open_Eye** deutlich sichtbar. Allerdings müssen wir den Filter noch einmal starten und die Anzahl der Variablen anpassen. Ein paar Einstellungen werden zwar verloren gehen, aber zum jetzigen Zeitpunkt ist dies nicht so schlimm. Die Parameter sind schnell wieder eingestellt.

### *2.14.3.1 Zeitereignisse ( Timer Events)*

Alles ist im **Open_Eye** leider nicht sichtbar, da die Datenübertragung sowie die Umsetzung der Werte im PC auch Zeit erfordern, ist natürlich immer nur ein Schnappschuß der Variablenwerte möglich. So werden wir feststellen, das die Variable **Time_Flags** mit **1** gefüllt ist, bevor wir eine Chance haben, diese Zeitstaffelung uns anzusehen. Damit wir aber trotzdem diese Zeitbits kontrollieren können, basteln wir mal wieder eine kleine Subroutine und setzen den Blinker ein.

```
;*********************** Zeitereignisse prüfen **********************
;*     Variable  New_In enthält Status der Eingänge            *
;*                                                                 *
;*      Ruft Blinker auf, um Sekundentakt zu testen             *
;*****************************************************************************
Chk_Time_Flags:                     ; Zeitbits testen
  LDS     Reg_A, Time_Flag
  ANDI    Reg_A, 0b00001000         ; ein beliebiges Bit prüfen
  BREQ    End_Chk_Time_Flags        ; Bit= 0, dann  kein Zeitereignis
  LDS     Reg_A, Time_Flag          ; Zeitflag quittieren
  ANDI    Reg_A, 0b11110111         ; Mit einer Und-Anweisung das Bit löschen
  STS     Time_Flag, Reg_A
  RCALL   Blinker                   ; Aufruf Blinker zum Testen
End_Chk_Time_Flags:                 ; Markiert das Ende der Subroutine

RET                                 ; Und Rücksprung zum Aufruf
```

Zeitflags machen auch nur Sinn, wenn diese bearbeitet werden und nun kommen wir schon zu einem sinnvolleren Einsatz der Triggerung. Doch zuerst bauen wir einen Verteiler, der die Zeitflags auswertet und diese zurück setzt.

Die Timeflags sind mit Hilfe des Blinkers getestet. Es folgt nun die vollständige Erfassung aller Zeitereignisse in einer Routine.

```
;******************** Zeitereignisse prüfen **********************
;*     Die Zeitereignisse werden auf die Zeitjobs verteilt    *
;*****************************************************************************
Chk_Time_Flags:                     ; Zeitbits testen
  LDS     Reg_A, Time_Flag
  ANDI    Reg_A,0b00000001          ; Zeitevent 1 mSek
  BREQ    Chk_msek_10
```

```
  RCALL   Event_1msek              ; Zeitevent bearbeiten
Chk_msek_10:
  LDS     Reg_A, Time_Flag
  ANDI    Reg_A,0b00000010          ; Zeitevent 10 mSek
  BREQ    Chk_msek_100
  RCALL   Event_10msek             ; Zeitevent bearbeiten
Chk_msek_100:
  LDS     Reg_A, Time_Flag
  ANDI    Reg_A,0b00000100          ; Zeitevent 100 mSek
  BREQ    Chk_sekunde
  RCALL   Event_100msek             ; Zeitevent bearbeiten
Chk_sekunde:
  LDS     Reg_A, Time_Flag
  ANDI    Reg_A,0b00001000          ; Zeitevent Sekunde
  BREQ    End_Chk_Time_Flags
  RCALL   Event_sekunde            ; Zeitevent bearbeiten

End_Chk_Time_Flags:               ; Markiert das Ende der Subroutine
RET
```

Diese Routine rufen wir nun in unserer Programmschleife auf.

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO              ; Eingänge lesen
   RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit          ; Bearbeiten „V“
   RCALL    Blinker              ; Blinkerbit bilden
   RCALL    IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL    Taster_Event         ; Tasterereignisse bearbeiten
   RCALL    Chk_Time_Flag        ; Zeitereignisse bearbeiten
   RCALL    Chk_Receive          ; Datenempfang prüfen
   RCALL    Is_First             ; erstes empfangenesByte prüfen und markieren
   RCALL    Is_Second            ; Datenempfang endgültig auswerten
   RCALL    Write_IO             ; und ausgeben
RJMP Loop
```

Aber, bevor das Programm assemblierbar ist, müssen wir nun die Subroutinen der Zeitereignisse schreiben, die hier aufgerufen werden. Die Routinen sind in der Art vom Rumpf her ziemlich gleich und so genügt es, eine Subroutine zu schreiben, diese zu kopieren und anzupassen. In jedem Zeitereignis setzen wir das Bit wieder zurück.

```
Event_1mSek::
;------ Bereich der Zeiterreignisbearbeitung –1 mSek---------
;-------------------------- Zeitflag quittieren -----------------------
    LDS     Reg_A, Time_Flags          ; Variable mit Zeitflags holen
    ANDI    Reg_A, 0b11111110          ; Flag für 1 mSek löschen
    STS     Time_Flags, Reg_A          ; Variable zurückschreiben
; Zeitjob
RET
```

```
Event_10mSek::
;------ Bereich der Zeiterreignisbearbeitung –10 mSek---------
;-------------------------- Zeitflag quittieren -----------------------
    LDS     Reg_A, Time_Flags          ; Variable mit Zeitflags holen
    ANDI    Reg_A, 0b11111101          ; Flag für 10 mSek löschen
    STS     Time_Flags, Reg_A          ; Variable zurückschreiben
; Zeitjob
RET
```

```
Event_100mSek::
;------ Bereich der Zeiterreignisbearbeitung –100 mSek---------
;-------------------------- Zeitflag quittieren -----------------------
    LDS     Reg_A, Time_Flags          ; Variable mit Zeitflags holen
    ANDI    Reg_A, 0b11111011          ; Flag für 100 mSek löschen
    STS     Time_Flags, Reg_A          ; Variable zurückschreiben
; Zeitjob
RET
```

```
Event_Sekunde::
;------ Bereich der Zeiterreignisbearbeitung –Sekunde---------
;-------------------------- Zeitflag quittieren -----------------------
    LDS     Reg_A, Time_Flags          ; Variable mit Zeitflags holen
    ANDI    Reg_A, 0b111110111         ; Flag für Sekunde löschen
    STS     Time_Flags, Reg_A          ; Variable zurückschreiben
; Zeitjob
```

```
RET
```

Nun fügen wir zum Testen des Triggers in verschiedenen Zeitereignissen ein Triggerereignis ein. Beginnen wird mit dem Test in dem Zeitereignis 1 mSek.

```
Event_1mSek::
;------ Bereich der Zeiterreignisbearbeitung –1 mSek---------
;--------------------------- Zeitflag quittieren -----------------------
    LDS     Reg_A, Time_Flags          ; Variable mit Zeitflags holen
    ANDI    Reg_A, 0b11111110          ; Flag für 1 mSek löschen
    STS     Time_Flags, Reg_A          ; Variable zurückschreiben
                ; Zeitjob                        ; Aufgabe abarbeiten
    LDS     Reg_A, Trigger_In          ; Triggerbyte lesen
    ANDI    Reg_A, 0b00001000          ; Trigger 4 ?
    BREQ    End_ Job_1_mSek
    STS     Trigger_Out, Reg_A         ; Nur dieses Triggerbit zurücksenden
   LDS      Reg_A, Trigger_In          ; Triggerbyte erneut lesen
    ANDI    Reg_A, 011110111           ; Triggerbit löschen
    RCALL   Send_Value                 ; Datenübertragung anstoßen
End_Job_1_mSek:
 RET
```

Der Sendeauftrag für die Variableninhalte wird hier nach der Bearbeitung der Routine erledigt, so dass die Werte die Ergebnisse nach der Bearbeitung enthalten. Dieser Programmabschnitt kann in jeder Subroutine untergebracht werden und entweder aus dem vorherigen gelöscht oder mit einem anderen Trigger aufgerufen werden. Man braucht keine Bedenken zu haben, das nun im mSek. Takt Daten an den PC gesendet werden. Der Trigger schickt nur ein einziges Mal die Daten ab.

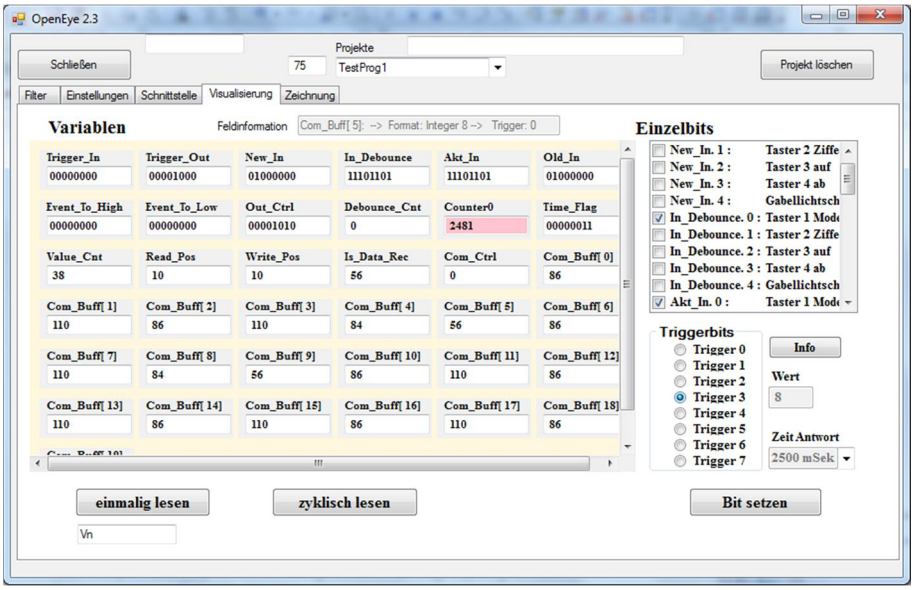
Lassen wir einmal einen 16 Bit-Zähler mitlaufen, der z. B. In die Zeitroutine 10 mSek. Eingebaut wird. Dazu benutzen wir die Variablen **Counter0** und **Counter1**, die noch deklariert werden müssen. Zusätzlich fügen wir einen Aufruf des Triggers ein, wenn ein bestimmter Zählerwert erreicht ist.

```
Counter0:       .Byte 1             ; Variable 16 Bit Zähler Low
Counter1:       .Byte 1             ; Variable 16 Bit Zähler High
Event_1msek:
;------------------------ Zeitflag quittieren -----------------------
```

```
   LDS      Reg_A, Time_Flags          ; Variable mit Zeitflags holen
   ANDI     Reg_A, 0b11111101          ; Flag für 10 mSek löschen
   STS      Time_Flags, Reg_A          ; Variable zurückschreiben
;--------- Bereich der Zeiterreignisbearbeitung –10 mSek---------
   LDS      Reg_A, Counter0            ; LowByte des 16 Bit Zählers
   INC      Reg_A                      ; erhöhen
   STS      Counter0, Reg_A            ; und ablegen
   OR       Reg_A, Reg_A               ; Zerobit setzen wenn Überlauf
   BRNE     Next_Job_1mSek             ; wenn nicht 0 (Überlauf), dann ende Zeitereignis
   LDS      Reg_A, Counter1            ; Ansonsten Highbyte laden
   INC      Reg_A                      ; erhöhen
   STS      Counter1, Reg_A            ; und zurückspeichern
   CPI      Reg_A, 37                  ; Hier einen Triggeraufruf einbauen
   BRNE     SecondJob_1mSek
   LDS      Reg_A, Trigger_In          ; Triggerbyte lesen
   ANDi     Reg_A, 0b00001000          ; Trigger 4 ?
   BREQ     SecondJob_1mSek            ; Kein Trigger
   STS      Trigger_Out, Reg_A         ; Nur dieses Triggerbit zurücksenden
   LDS      Reg_A, Trigger_In          ; Triggerbyte erneut lesen
   ANDI     Reg_A, 0b11110111          ; Triggerbit löschen
   STS      Trigger_In, Reg_A
   RCALL    Send_Value                 ; Datenübertragung anstoßen
   STS      Trigger_Out, Zero          ; Bit in Trigger_Out löschen
Next_Job_1mSek:                       ; weitere Jobs
End_Event_1mSek:
RET
```

Nun wird der Variablenbereich noch einmal in den Filter von Open_Eye übertragen und die Anzahl der zu übertragenden Bytes korrigiert.

Wie dieser Aufbau zeigt, wird das Programm allein durch die Bearbeitung der Zeitflags in der Zykluszeit ansteigen. Trotzdem ist immer noch genug Zeit, in den Ereignissen Programm einzubauen. Gut, je kürzer die Zeit, umso kleiner sollten die Aufgaben sein. Testen wir nun erst einmal das Ergebnis, bevor wir uns wieder den Eingängen zuwenden.




*OpenEye mit Daten vom Controller*

An dieser Stelle sind wir mit dem Ergebnis erst einmal zufrieden. Auch ein Funktionstest mit dem Button zyklisch lesen liefert den laufenden Zählerstand in der Variablen Counter0, der für diesen Zweck mit dem Format Int16 eingerichtet ist. Der 16 Bit Zähler zeigt auch plausible Werte an und das ist wichtig. Leicht hätte es hier zu einer Vertauschung von High- und Lowbyte in der Auswertung kommen können. Auch ein angelegter Trigger auf die farblich markierten Felder hat beim Erreichen des Vergleichswertes ausgelöst. Dafür habe ich die Überwachungszeit auf 30 Sekunden eingestellt. Die Routine ist im Zeitereignis 1 mSek eingebaut. Der Wert im oberen Register entspricht aber nicht 37 000, sondern ist das 256fache von 37, also 9472. Das ist wichtig, weil sonst die Trigger-Überwachungszeit ohne Ergebnis abläuft.

Betrachten wir noch ein paar andere Werte und beginnen mit **Trigger_In** und **Trigger_Out**. Es zeigt sich, das **Trigger_Out** das auslösende Triggerbit zurückliefert, während der Wert in **Trigger_In** wieder 0 ist. Das ist auch so gewollt, dadurch ist erkennbar, dass der Trigger auch sauber bearbeitet und nur einmalig ausgeführt wurde.

Die Werte in **New_In** und **Old_In** sind gleich, allerdings sind die Variablen **In_Debounce** und **Akt_In** noch nicht eingebunden. Die Inhalte passen noch nicht ins Bild.

Diesen Part hatte ich zwar schon im Programm eingebunden und wenn nun ein Taster betätigt werden sollte, so wird auch in diesen Variablen etwas verändert. Es sollte ja immer noch der Taster angeschlossen sein, testen wir doch einmal die Auswirkung. Sicher, wir sehen den Taster in **New_In** und **Old_In**, aber nicht in **Akt_In** und **In_Debounce**. Dazu sehen wir uns **Read_IO** an und stellen dort fest, dass die Zuweisung der eingehenden Daten nicht nach **Akt_In** geleitet wird, sondern nach **New_In.** Da ist noch etwas unklar und so setzen wir erst einmal mit der Programmierung der Eingänge die Arbeit fort.

### 2.14.4 *Eingänge lesen, Entprellen und Ereignis erfassen (Flanken bilden)*

Zurück zur Leseroutine Read_IO

```
Read_IO:                                ; Eingänge einlesen
    In      Reg_A, PInD                     ; Port B lesen
    COM     Reg_A                       ; Bits drehen
    ANDI    Reg_A, 0b01111100           ; Nur Eingänge übernehmen
    LSR     Reg_A                       ; nach rechts schieben (00111110)
    LSR     Reg_A                       ; nach rechts schieben (00011111)
    STS     New_In, Reg_A               ; ändern in Sts Akt_In, Reg_A
RET
```

Die nächste Routine erledigt das Entprellen der Eingangssignale. Diese Routine hatten wir bereits abhängig vom Zyklus programmiert. Nun setzen wir eine stabile Zeitbasis dafür ein und die liefert uns der Timer.

Um diese Änderung nun umzusetzen, trennen wir den Part mit der Prellzeit einfach ab und schreiben das in den Bereich der mSek-Bearbeitung.

```
IO_Debounce_T:
    LDS     Reg_A, Akt_In               ; neu eingelesene Eingänge
    LDS     Reg_B, In_Debounce          ; Ablage Kontrolle Kontaktprellen
    EOR     Reg_B, Reg_A                ; Registerwert A erhalten
    BREQ    End_IO_Debounce_T:              ; Register A und B gleich, keine Änderung
    STS     In_Debounce, Reg_A,         ; Ablage zur Kontrolle Kontaktprellen
    LDI     Reg_B, 50                   ; Überwachungszeit neu setzen
    STS     Debounce_Cnt, Reg_B         ; und in Zähler eintragen
End_IO_Debounce_T:
RET ; der Rest kommt in die ISR vom Timer
```

```
Event_1msek:
;------ Bereich der Zeiterreignisbearbeitung –1 mSek---------
    LDS     Reg_B, Debounce_Cnt
    CPI     Reg_B, 0        ; Vergleich ist wichtig, da eine Ladeanweisung kein Zerobit setzt.
    BREQ    Quit_Event_1mSek            ; Wenn zähler auf 0, keine weitere Aktion
    Dec     Reg_B
    STS     Debounce_Cnt, Reg_B
    BRNE    Quit_Event_1mSek
```

```
    LDS        Reg_A, Akt_In
    STS        New_In, Reg_A
 ; Zähler hat bis 0 gezählt. Der neugelesene Wert ist gültig und wird übernommen
;---------------------- Zeitflag quittieren ------------------------
Quit_Event_1mSek:
    LDS        Reg_A, Time_Flags                ; Variable mit Zeitflags holen
    ANDI       Reg_A, 0b11111110                ; Flag für 1 mSek löschen
    STS        Time_Flags, Reg_A                ; Variable zurückschreiben
End_Event_1mSek:
RET
```

Nun ist ein Eingang gültig, wenn er 50 mSek nicht geändert wurde. Diese Zeit lässt sich aber auch noch variieren.

### *2.14.4.1 Ein Stromstoßschalter*

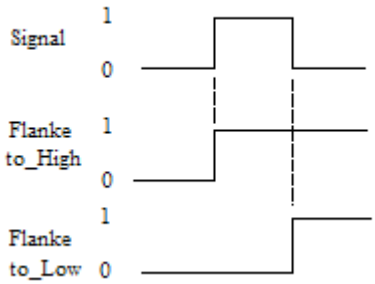
Dazu legen wir folgendes kleines Programm in der Subroutine Taster_Status ab:

```
;*****************************************************
;*              Tastersignal prüfen              *
;*****************************************************
Taster_Event:
    LDS     Reg_A, Event_To_high      ; Ereignisbit laden
    ANDI    Reg_A, 0b00000001         ; Nur Bit 0 prüfen (Ergebnis ist nicht 0)
    BREQ    End_Taster_Event          ; wenn Ergebnis 0, dann kein Tasterereignis
    LDS     Reg_A, Out_Ctrl           ; Out_Ctrl laden
    LDI     Reg_B,0b00100000          ; Maske zum Bit drehen
    EOR     Reg_A, Reg_B              ; Bit 5 drehen
    STS     Out_Ctrl, Reg_A           ; und zurückschreiben
    LDS     Reg_A, Event_To_ high     ; Ereignisbit laden
    ANDI    Reg_A, 0b11111110         ; Nur Bit 0 auf 0 setzen
    STS     Event_To_high, Reg_A      ; und Eventbit löschen
End_Taster_event:
RET
```

Nun testen wir das Ergebnis. Dazu benutzen wir die zyklische Variablenabfrage. Es sollte das Bit 5 seinen Status bei jedem Tastendruck ändern. Würden wir nun an diesen Port ein Relais schalten, wäre der Stromstoßschalter perfekt. Aber nicht nur ein Stromstoßschalter ist machbar. Wie wär es einmal mit einem Zeitrelais. Bit einschalten und nach Ablauf einer Zeit wieder abschalten. Nun, das Handwerkszeug dazu besitzen wir.

### 2.14.4.1 *Flankenbildung mit entprellten Eingängen*

Kommen wir nun zu den Flanken der Signale. Das Thema hatten wir bereits angesprochen. Diesmal sind nicht direkt die Eingänge, sondern die bereits entprellten Informationen in der Variablen New_In beteiligt. Ein Eingang hat eine steigende Flanke, wenn ein Kontakt geschlossen und eine fallende Flanke, wenn er geöffnet wird. So ist es in unseren Köpfen und deshalb haben wir auch, bedingt, das die Taster an GND hängen, die Eingangssignale dahingehend gedreht. Die abgelegte Signallage in der Variablen New_In entspricht unserer Denkweise und darauf legen wir auch die Signallage der Flankenbildungl. Damit deutlich wird, wie das mit den Flanken funktioniert, hier auch noch eine kleine Skizze:




*Signalverlauf*

Dazu die Programmroutinen. Auch diese werden in der Programmschleife zyklisch aufgerufen.

```
IO_Event_To_1:
    LDS    Reg_A, Old_In          ; letzter gelesener Wert der Eingänge
    LDS    Reg_B, New_In          ; neu gelesener Wert der Eingänge
    EOR    Reg_A, Reg_B   ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
    BREQ   End_Event_to_1
    AND    Reg_A, Reg_B   ; Der Unterschied steht in Reg.A, eine Und Verküpfung mit dem
    LDS    Reg_B, Event_To_High       ; Flankenbits laden
    OR     Reg_A, Reg_B                ; neues Ergebnis hinzufügen
    STS    Event_To_High, Reg_A   ; neuen Wert liefert die Bits mit Änderung von 0 nach 1
    STS    Old_In, Reg_B          ; Neuen Wert in Ablage „old_In“ kopieren
End_Event_to_1:
RET
```

```
IO_Event_To_0:
    LDS     Reg_B, Old_In                   ; letzter gelesener Wert der Eingänge
    LDS     Reg_A, New_In                   ; neu gelesener Wert der Eingänge
    STS     Old_In, Reg_A                   ; Neuen Wert in Ablage old_In kopieren,
; ist hier bereits erforderlich, da Reg_A überschrieben wird
    EOR     Reg_A, Reg_B            ; EOR= Exclusiv-Oder. Eine 1 nur bei Unterschied
    BREQ    End_Event_to_0
    AND     Reg_A, Reg_B                    ; Ergebnis in Reg.A, Und mit dem
    LDS     Reg_B, Event_To_Low             ; Flankenbits laden
    OR      Reg_A, Reg_B                    ; neues Ergebnis hinzufügen
    STS     Event_To_LOW, Reg_A     ; alten Wert liefert die Bits mit Änderung von 1 nach 0
End_Event_to_0:
RET
```

Die Trennung beider Routinen hat einen Nachteil. Es wird entweder eine steigende oder eine fallende Flanke erkannt, niemals beide, wenn sie auf verschiedenen Bits gleichzeitig eintreffen. Der Grund liegt in der Übernahme der neuen Information in die Ablage beim Durchlauf der ersten Subroutine. Sollen beide Flanken erfasst werden, ist eine Zusammenlegung der beiden Flankenbildnern unumgänglich.

```
IO_Event_Flanke:                            ; beginnen mit der Flanke nach logisch 1
    LDS     Reg_A, Old_In               ; letzter gelesener Wert der Eingänge
    MOV     Ablage, Reg_A
    LDS     Reg_B, New_In               ; neu gelesener Wert der Eingänge
    EOR     Reg_A, Reg_B                ; EOR= Exclusiv-Oder. 1 bei Unterschied
    AND     Reg_A, Reg_B                ; Egebnis in Reg.A, Und mit New_In
    BREQ    End_Event_To_High           ; keine Änderung, kein Eintrag
    LDS     Reg_C, Event_To_High        ; Flankenbits laden
    OR      Reg_A, Reg_C                ; neues Ergebnis hinzufügen
    STS     Event_To_High, Reg_A        ; Änderung von 0 nach 1
End_Event_To_High:                          ; dann Flanke nach logisch 0 prüfen
    MOV     Reg_B, Ablage               ; letzter gelesener Wert der Eingänge
    LDS     Reg_A, New_In               ; neu gelesener Wert der Eingänge
    STS     Old_In, Reg_A               ; Neuen Wert in Ablage old_In kopieren,
; ist hier bereits erforderlich, da Reg_A überschrieben wird
    EOR     Reg_A, Reg_B                ; EOR= Exclusiv-Oder. 1 nur bei Unterschied
    AND     Reg_A, Reg_B                ; Ergebnis in Reg.A, Und mit Old_In
    BREQ    End_Event_Flanke            ; keine Änderung, kein Eintrag
    LDS     Reg_B, Event_To_Low         ; Flankenbits laden
    OR      Reg_A, Reg_B                ; neues Ergebnis hinzufügen
    STS     Event_To_LOW, Reg_A         ; Bits mit Änderung von 1 nach 0
```

```
End_Event_Flanke:
RET
```

Dank dieser Flankenbits können wir nun bei einem Tastendruck eine einmalige Programmbearbeitung durchführen und erst beim nächsten Tastendruck wieder darauf reagieren.

Wozu ist dies wichtig? Nun, beginnen wir mit den Ein- und Ausschalten eines Relais oder einer LED. Wenn wir einfach nur einen Taster abfragen, ob gedrückt, also den Signalpegel, dann wird dieser Programmabschnitt immer durchlaufen und der Wechsel ständig stattfinden. Wir möchten aber eine **Ein-Aus-Ei**n Funktion.

Wenn wir nun bei jedem Flankenwechsel dieses Programm durchlaufen und nach der Bearbeitung dieses Flankenbit löschen, so ist das kein Problem mehr. Der Taster muss erst losgelassen und erneut gedrückt werden, damit dieses Bit wieder gesetzt wird. Fügen wir nun die neue Routine IO_Event in unsereProgrammschleife und entfernen die Routine IO_Event_To_1

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO              ; Eingänge lesen
   RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    IO_Event_Flanke      ; Ereignisbits bilden Flanken nach 0 und 1
   RCALL    Taster_Event         ; Tasterereignisse bearbeiten
   RCALL    Chk_Time_Flag        ; Zeitereignisse bearbeiten
   RCALL    Chk_Receive          ; Datenempfang prüfen
   RCALL    Is_First             ; erstes empfangenesByte prüfen und markieren
   RCALL    Is_Second            ; Datenempfang endgültig auswerten
   RCALL    Write_IO             ; und ausgeben
RJMP Loop
```

Testen wir doch gleich mit unserer **Out_Ctrl**-Variable und **Open_Eye** diese Funktion. Dazu sollten die Taster verdrahtet sein. Bei Betätigen werden die Bits in den Flankenspeicher Event_To_High und Event_To_Low abgelegt. Ändern wir nun die Subroutine Tasterevent und lassen bei jedem Tastendruck ein Bit in Out_Ctrl setzen und mit Loslassen

wieder löschen. Aber nicht direkt durch das Tastersignal, sondern nur durch die Flanken.

```
;*************************************************************
;*                    Tastersignal prüfen                    *
;*************************************************************
Taster_Event:
  LDS    Reg_A, Event_To_High   ; Ereignisbits laden
  ANDI   Reg_A, 0b00000001      ; Nur Bit 1 prüfen (Ergebnis ist nicht 0)
  BREQ   Chk_Taster_Low         ; Ergebnis 0, dann Sprung nach Marke Status_Ende
  LDS    Reg_B, Out_Ctrl        ; Nun das Byte für den Ausgang holen
  OR     Reg_A, Reg_B
  STS    Out_Ctrl, Reg_A        ; nun noch das Register B in die Variable Out_Ctrl schreiben
  LDS    Reg_A, Event_To_High   ; Ereignisbits laden
  ANDI   Reg_A, 0b11111110      ; Ereignisbit 0 löschen
  STS    Event_To_High, Reg_A   ; Und Ereignisse zurückschreiben
Chk_Taster_Low:                 ; hier kommt die fallende Flanke zur Bearbeitung
  LDS    Reg_A, Event_To_Low    ; Ereignisbits laden
  ANDI   Reg_A, 0b00000001      ; Nur Bit 1 prüfen (Ergebnis ist nicht 0)
  BREQ   End_Taster_Event       ; Ergebnis 0, dann Sprung nach Marke Status_Ende
  LDS    Reg_B, Out_Ctrl        ; Nun das Byte für den Ausgang holen
  COM    Reg_A                  ; Invertiert alle Bits, getestetes Bit wird 0
  AND    Reg_A, Reg_B           ; Bit in Out_Ctrl wird ausgeblendet
  STS    Out_Ctrl, Reg_A        ; und zurückgeschrieben
  LDS    Reg_A, Event_To_Low    ; Ereignisbits laden
  ANDI   Reg_A, 0b11111110      ; Ereignisbit 0 löschen
  STS    Event_To_Low, Reg_A    ; Und Ereignisse zurückschreiben
End_Taster_Event:               ; Markiert das Ende der Subroutine
RET
;-------------------------------------------------------------
```

Der Test wird zeigen, das das Bit 0 in Out_Ctrl gesetzt, aber nicht zurückgesetzt wird. Erst, wenn ein entsprechendes Ereignis einer fallenden Flanke in dem Abschnitt Chk_Taster_Low eingebaut wird, schaltet das Bit auch wieder ab. Das Bit verhält sich nun wie der Taster. Doch es kann nicht der Sinn eines Controllers sein, einen Ausgang wie hier beschrieben zu schalten. Dennoch ist diese Übung der Ausgangspunkt für viele Abwandlungen.

### *2.14.4.2 Ein Zeitrelais*

Allerdings benötigen wir dazu wieder eine Variable und damit wir nicht neu Filtern müssen, trennen wir einfach **Counter0** und **Counter1** wieder auf. Die 16 Bit Zählvariable hat ja erst einmal ihren Dienst getan. Die Variable **Counter0** setzen wir dafür in der Zeitbearbeitung von 1 Sekunde ein.

Zuerst wird das Zeitevent von einer Sekunde bearbeitet. Nach der Prüfung, ob die Zählvariable Counter0 abgelaufen ist, wird entweder in einen weiteren Job verzweigt oder der Zähler herunter gezählt. Erreicht er dann 0, wird das Bit 5 in Out_Ctrl wieder gelöscht.

```
Event_1sek:
;------------------------- Zeitflag quittieren ------------------------
    LDS     Reg_A, Time_Flag          ; Variable mit Zeitflags holen
    ANDI    Reg_A, 0b11110111         ; Flag für Sekunde löschen
    STS     Time_Flag, Reg_A          ; Variable zurückschreiben
;--------- Bereich der Zeiterreignisbearbeitung –1 Sek---------
    LDS     Reg_A, Counter0           ; Zähler laden
    CPI     Reg_A, 0                  ; ist Counter0 auf 0
    BREQ    Next_Job_Sek              ; wenn nicht 0 dan nächster Job
    Dec     Reg_A                     ; Counter runterzählen
    STS     Counter0, Reg_A           ; und Zähler wieder  ablegen
    BRNE    Next_Job_Sek              ; wenn nicht 0, dann Ende Zeitereignis
    LDS     Reg_A, Out_Ctrl           ; AusgabeByte holen
    ANDI    Reg_A, Ob11011111         ; Bit 5 abschalten
    STS     Out_Ctrl, Reg_A           ; und zurückspeichern
Next_Job_Sek:
End_Event_1sek:
RET
```

Die Tasterbearbeitung ändern wir nun ein wenig und setzen nicht nur das Bit 5 in Out_Ctrl direkt, sondern weisen auch der Variablen einen Zählwert zu. Eigentlich ist es ein ähnliches Verfahren, wie beim Entprellen der Taster.

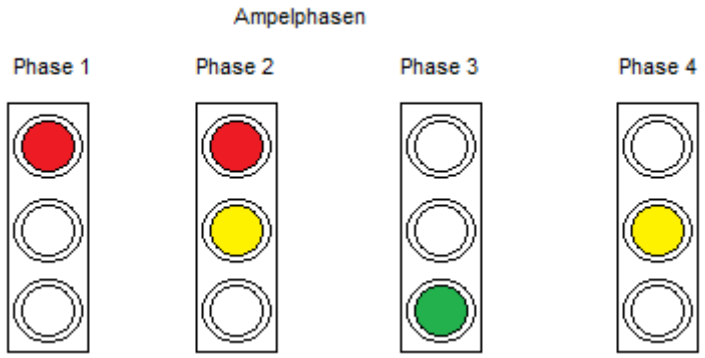
```
;*******************************************************
;*              Tastersignal prüfen                    *
;*******************************************************
Taster_Event:
```

```
    LDS     Reg_A, Event_To_high          ; Ereignisbit laden
    ANDI    Reg_A, 0b00000001             ; Nur Bit 0 prüfen
    BREQ    End_Taster_Status             ; wenn Ergebnis 0, dann kein Tasterereignis
    LDS     Reg_A, Out_Ctrl               ; Out_Ctrl laden
    ORI     Reg_A,0b00100000              ; Maske zum Bit drehen
    STS     Out_Ctrl, Reg_A               ; und zurückschreiben
    LDI     Reg_A, 7                      ; Zählvariable Zeitwert zuweisen
    STS     Counter0, Reg_A    ; und ablegen. Den Rest erledigt die Zeitbearbeitung
    LDS     Reg_A, Event_To_ high         ; Ereignisbit laden
    ANDI    Reg_A, 0b11111110             ; Nur Bit 0 auf 0 setzen
    STS     Event_To_high, Reg_A          ; und Eventbit löschen
End_Taster_Event:
RET
```

Damit haben wir einen Zeitschalter. Ich denke, wer bis hierher mitgehen und sein Programm auch kontrollieren konnte, der wird heiß sein, einen Schritt weiter zu kommen.

### *2.14.5 Ampelschaltung*

Eine Ampelschaltung ist schon eine kleine Herausforderung. Wie schon bei Programmentwicklung un Visual Basic oft geübt, ist auch hier das ordnen der Gedanken erste Pflicht. Wie funktioniert eine Ampel? Rot, RotGelb,Grün und Gelb. Die nächste Phase beginnt mit Rot. Abgezählt sind das vier Schritte. Skizzieren wir es einmal um zu verstehen.




*Ampelphasen*

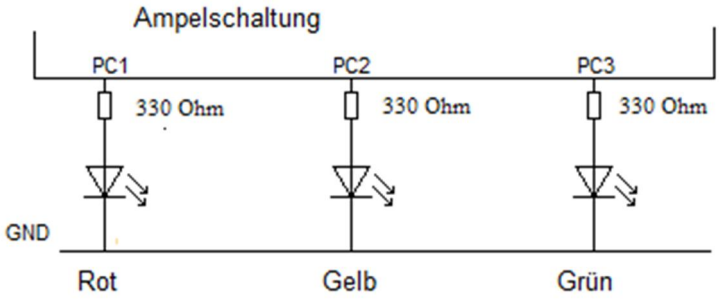
Deklarieren wir eine Variable Ampel und bestimmen die Bits für Rot, Gelb und Grün

```
Ampel:    .Byte    ; Arbeitsbyte für Ampel
                   ; Bit 0 Rot
                   ; Bit 1 Gelb
                   ; Bit 2 Grün
```

Dieser Variablen müssen wir nun in den vier Zeitabschnitten die Werte

0b00000001, 0b00000011, 0b00000100 und 0b00000010 zuweisen.

Verlassen wir uns erst einmal auf den Inhalt und weisen sie den Bits PC1 – PC3 in der Write_IO Ausgängen zu. Ändern wir dazu den Schaltplan von 2.4.17.1




*Ampelschaltung*

Die Routine **Write_IO** wird nun abgeändert. Statt **Out_Ctrl** wird die die Variable **Ampel** geladen und der Inhalt einmal nach links geschoben. Es sind ja anfangs die Bits 0-2 für die Ampelfarben vergeben. Der Ausgang PC0 steht aber nicht zur Verfügung, so dass erst PC1 beschaltet werden konnte.Durch den Befehl ROL (Rotate Left) passen wir den Inhalt von Reg_A an. Dann ändern wir die Bitmaske so, das nur die 3 Bits für **Rot**, **Gelb** und **Grün** durchgereicht werden und alles andere den Status behält.

```
;*************************************************************
;*          Ausgabe an die IO-Ebene                  *
;*         Port C Bit 1-5 = Relais / LED             *
;*************************************************************
Write_IO:
   IN      Reg_B, PortC              ; gesamten Port C lesen
   ANDI    Reg_B, 11110001           ; Bit 1-5 löschen
   LDS     Reg_A, Ampel              ; Variable für LED holen
   LSL     Reg_A                     ; einmal link schieben
   ANDI    Reg_A, 00001110           ; Bit 0 und Bit 4-7  ausmaskieren
   Or      Reg_B, Reg_A              ; Register a und B zusammenführen
   OUT     PortC, Reg_B              ; Port beschreiben
RET
```

Nun müssen wir im Programm nur noch die Variable Ampel setzen, um die Ampelpasen auszugeben.

Bekannt sind 4 Schritte, die wir mit einem Zähler von 0 bis 3 abbilden können. In der ISR Datenempfang haben wir mit einer Zeigervariablen einen Ringpuffer abgebildet. Diese Technik können wir hier auch benutzen. Dazu brauchen wir ein Array Ampelphasen.

```
Ampelphasen:   .Byte 4      ; Abbild Ampelbits
                            ; Phase 1 (Schritt 0)  = 00000001
                            ; Phase 2 (Schritt 1)  = 00000011
                            ; Phase 3 (Schritt 2)  = 00000100
                            ; Phase 4 (Schritt 3)  = 00000010
```

Dieses Array müssen wir vorbesetzen, bevor wir auf die Inhalte zugreifen können. In diesem Fall können wir aber nur mit einer Zeigervariable auf die Arrayelemente zugreifen.

```
;*********************** Initialisierung Ampel **************************
;*     jeder denkbare Zustand einer Ampel bekommt ein          *
;*     Byte für die Ausgabe.                                   *
;*****************************************************************************
Init_Ampelphasen:
   LDI XL,LOW(AmpelPhase)    ; X-Pointer auf Anfang der Ampelphasen
   LDI XH,HIGH(AmpelPhase)
    LDI Reg_A,0b00000001       ; Phase 1 Rot (Schritt 0)
    ST  X+, Reg_A
    LDI Reg_A,0b00000011       ; Phase 2 (Schritt 1)
    ST  X+, Reg_A
    LDI Reg_A,0b00000100       ; Phase 3 (Schritt 2)
   ST  X+, Reg_A
   LDI Reg_A,0b00000010       ; Phase 4 (Schritt 3)
   ST  X+, Reg_A
    LDI  Reg_A, 4
   STS Ampelzeit, Reg_A        ; Ampelzeit setzen
   STS Ampelschritt, Zero      ; Beginnen mit Schritt 0
RET
```

Diese Initialisierung kommt natürlich auch in den Innitialisierungsteil.

```
;*******************************************************************
;*                    Bereich Initialisierung                      *
;*******************************************************************
Start:
```

```
LDI    Reg_A, High(RamEnd)          ; erste Maßnahme Stack setzen
OUT  SPH, Reg_A
LDI    Reg_A, Low(RamEnd)
OUT  SPL, Reg_A
            ; weitere Initialisierungen
RCALL    IniT_IO
RCALL    Ini_USART                  ; serielle Schnittstelle parametrieren
RCALL    Init_Timer1                 ; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
RCALL    Init_Var                   ;Initialisierung Variablen ( Eintragen Defaultwerte)
RCALL    Init_Ampelphasen          ;Initialisierung Variablen ( Eintragen Defaultwerte)
```

Nicht zu empfehlen ist die Vergabe einzelner Namen für die Ampelphasen wie nachstehend beschrieben. Das ist zwar auch nicht mehr Platzverbrauch, aber nicht erweiterungsfähig, falls die Ampelschaltung ausgebaut werden soll..

```
AmpelRot:        .Byte 1          ;Rotphase
AmpelRotGelb:   .Byte 1           ;Rot – Gelbphase
AmpelGruen:     .Byte 1           ;Grünphase
AmpelGelb:       .Byte 1          ;Gelbphase
```

Zusätzlich nehmen wir noch eine Variable für die Schritte und eine Variable für die Zeit der Ampelpasen.

```
Ampelschritt:  .Byte 1            ; Zähler der Ampelschritte
AmpelZeit:       .Byte 1           ; Zeit für die Ampelphase
```

Nun werden wir unserer Ampel mal mit einer Ereignissteuerung Leben einhauchen. Dazu benutzen wir ein **Zeitflag**, das die Timer-ISR bereits jede Sekunde bereitstellt und das in der Routine Event_Sekunde bereits bearbeitet und auch schon getestet wurde. Damit diese Routine nicht allzu groß wird, rufen wir eine weitere Routine **Ampel** auf.

```
AmpelSteuerung:
    LDS     Reg_A, Ampelzeit        ; Zeit einer Ampelphase
    Dec     Reg_A
    STS     AmpelZeit, Reg_A
    BRNE    End_Ampel               ; Zeit abgelaufen, dann nächster Schritt
    LDS      Reg_B, Ampelschritt
    INC      Reg_B
```

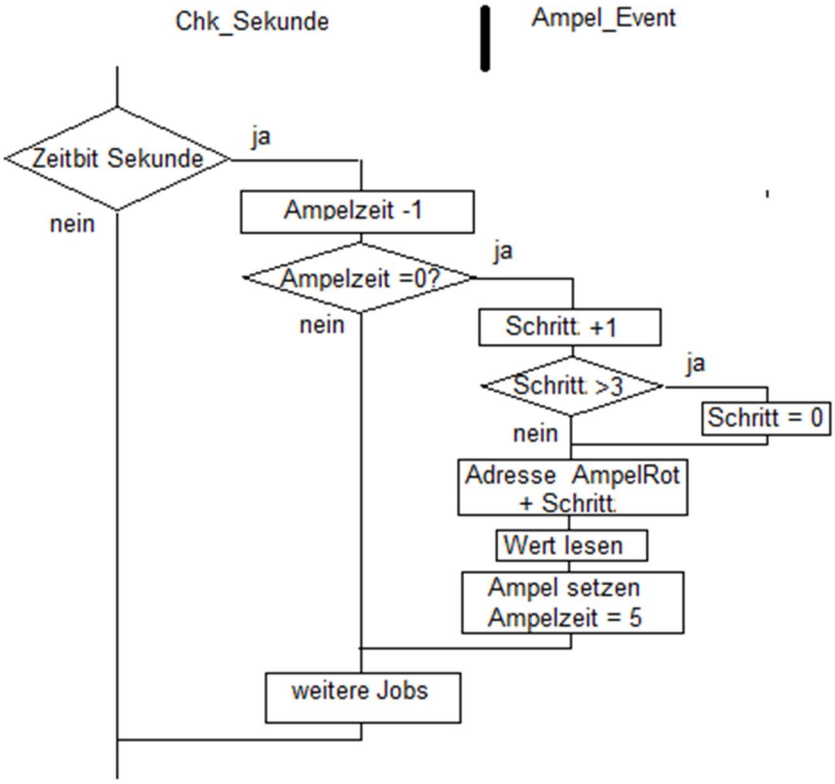
```
    CPI     Reg_B,4                 ; Grenzwert erreicht, dann von vorn
    BRNE    Set_Schritt
    CLR     Reg_B
Set_Schritt:
    STS     Ampelschritt, Reg_B
    LDI     Reg_A, 4                ; Neue Ampelzeit eintragen
    STS     AmpelZeit, Reg_A
    LDI     XL,LOW(AmpelPhase)      ; X-Pointer auf Anfang der Ampelphasen
    LDI     XH,HIGH(AmpelPhase)
    ADD     XL, Reg_B               ; Schritt auf Basisadresse dazu zählen
    ADC     XH, Zero
    LD      Reg_A, X                ; Muster der Ampelphase holen
    STS     Ampel, Reg_A            ; und ausgeben
End_Ampel:
RET;
```

Diese Routine wird nun in das Ereignis Sekunde eingetragen.

```
Event_1sek:
;------------------------- Zeitflag quittieren ------------------------
    LDS     Reg_A, Time_Flag        ; Variable mit Zeitflags holen
    ANDI    Reg_A, 0b11110111       ; Flag für Sekunde löschen
    STS     Time_Flag, Reg_A        ; Variable zurückschreiben
;--------- Bereich der Zeiterreignisbearbeitung –1 Sek---------
    LDS     Reg_A, Counter0         ; Zähler laden
    CPI     Reg_A, 0                ; ist Counter0 auf 0
    BREQ    Next_Job_Sek            ; wenn nicht 0 dan nächster Job
    Dec     Reg_A                   ; Counter runterzählen
    STS     Counter0, Reg_A         ; und Zähler wieder  ablegen
    BRNE    Next_Job_Sek            ; wenn nicht 0, dann Ende Zeitereignis
    LDS     Reg_A, Out_Ctrl         ; AusgabeByte holen
    ANDI    Reg_A, Ob11011111       ; Bit 5 abschalten
    STS     Out_Ctrl, Reg_A         ; und zurückspeichern
Next_Job_Sek:
    RCALL Ampelsteuerung            ; Aufruf Ampelsteuerung
End_Event_1sek:
RET
```

Damit haben wir den ersten Schritt für eine unabhängige Ereignisbearbeitung erledigt. Den eigentlichen Job haben wir weiter gegeben an die Routine **Ampelsteuerung.**

Ein PAP macht den Ablauf etwas deutliche.




*PAP Ampel*

Zuerst reduzieren wir die Ampelzeit und prüfen ,ob sie abgelaufen ist. Wenn das der Fall ist, setzen wir den Schritt hoch und prüfen, ob der Grenzwert 3 überschritten ist. Dann beginnt die Sequenz erneut bei 0. Im nächsten Schritt holen wir uns die Basisadresse der Ampelphasen, das war die Adresse der Variablen AmpelRot, in ein Adressregister. Auf dieses addieren wir nun den Schrittzähler,, holen den hinterlegten Wert aus dem Array AmpelRot und weisen ihn der Variablen Ampel zu. Zum Schluss wird noch der neue Zeitwert wieder in die Variable Ampelzeit eingetragen. Mit einem PAP doch eigentlich ziemlich einfach. Nun, das sollte erst einmal ausprobiert werden. Dabei sollte auffallen, dass ist so ziemlich unrealistisch. Alle Ampelphasen sind gleich lang. Dabei ist Gelb und Rotgelb zumindest wesentlich kürzer. Mit einer Zeittabelle passend zu den Ampelphasen ist das auch kein Problem. Lassen wir die Ampelzeit

weiterhin für die aktuelle Zeit und legen ein Zeitfeld zur Auswahl dahinter an.

```
AmpelZeit1:   .Byte 1
AmpelZeit2:   .Byte 1
AmpelZeit3:   .Byte1
AmpelZeit4:   .Byte1
```

Die Basisadresse für die Tabelle ist Ampelzeit1. In der Initialisierung setzen wir dann auch die Zeiten. AmpelZeit ist bereits mit 5 Sekunden vorbesetzt.

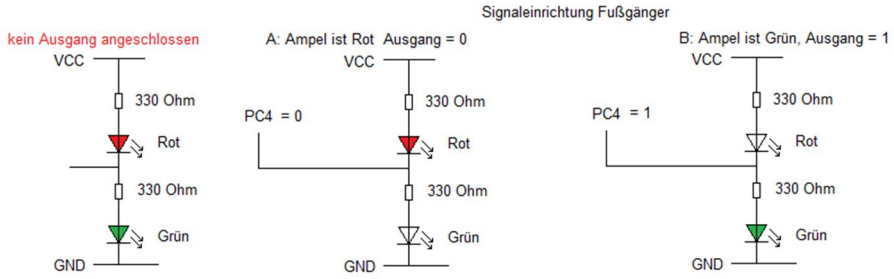
```
Init_Ampelphasen:
   LDI     Reg_A, 0b00000010
   STS     AmpelRot, Reg_A          ; Basisadresse für Ampelphasen
   LDI     Reg_A, 0b00000110
   STS     AmpelRotGelb, Reg_A      ; Basisadresse +1 für Ampelphase RotGelb
   LDI     Reg_A, 0b00001000
   STS     AmpelGruen, Reg_A        ; Basisadresse +2  für Ampelphase Grün
   LDI     Reg_A, 0b00000100
   STS     AmpelGelb, Reg_A         ; Basisadresse+3  für Ampelphase Gelb
   LDI     Reg_A, 5                 ; erste Zeit für Rotphase  = 5 Sekunden
   STS     AmpelZeit, Reg_A
   LDI     Reg_A, 5                 ; 5 Sekunde für Rot
   STS     AmpelZeit1, Reg_A
   LDI     Reg_A, 2                 ; 2 Sekunden für RotGelb
   STS     AmpelZeit2, Reg_A
   LDI     Reg_A, 6                 ; 6Sekunde für Grün
   STS     AmpelZeit3, Reg_A
   LDI     Reg_A, 2                 ; 2 Sekunde für Gelb
   STS     AmpelZeit4, Reg_A
   STS     Ampelschritt, Zero       ; und mit Schritt 0 beginnen
RET
```

Entsprechend wird in der Subroutine Ampelsteuerung nun die Zeit über das Adressregister geladen.

```
AmpelSteuerung:
   LDS     Reg_A, AmpelSchritt      ; Schritt laden
   INC     Reg_A                    ; Schritt+1
   CPI     Reg_A,4                  ; < 4 ok
```

```
  BRLO   Set_Ampel              ; überspringe
  CLR    Reg_A                   ; Schritt =0
Set_Ampel:
  STS    AmpelSchritt, Reg_A     ; Schritt speichern
  LDI    XL,Low(AmpelPhase)        ; Adresse von AmpelRot
  LDI    XH,High(AmpelPhase)       ; nach Doppelregister X
  ADD    XL, Reg_A               ; Schritt dazu addieren
  ADC    XH, Zero                ; 16 Bit-Addition
  LD     Reg_A, X                ; Lade Inhalt von adressiertr Speicherzelle
  STS    Ampel, Reg_A            ; und speichere in Ampel ab
  LDI    XL,Low(AmpelZeit1)      ; Adresse von AmpelZeit
  LDI    XH,High(AmpelZeit1)     ; nach Doppelregister X
  ADD    XL, Reg_A               ; Schritt dazu addieren
  ADC    XH, Zero                ; 16 Bit-Addition
  LD     Reg_A, X                ; Lade Inhalt von adressiertr Speicherzelle
  STS    AmpelZeit, Reg_A        ; und speichere in AmpelZeit ab
RET
```

War doch gar nicht so schwer, oder? Mit ein paar wenigen Änderungen schon ein völlig anderes Verhalten herbeizaubern ist halt die kleine Kunst. Deshalb ist es auch wichtig, sich ab und zu mal einen Ablauf zu skizzieren und die Möglichkeiten dabei erkennen. Durch das geänderte Zeitverhalten der einzelnen Schritte ist die Ampel in ihrer Funktion schon so ziemlich realistisch. Erweitern wir nun sie Funktion zu einer Fußgängerampel. Bei der Signalanlage für Fußgänger wechselt Rot direkt ohne eine Gelbphase zu Grün. Dafür reicht ein einziges Bit. Und das möchte ich mit einer Skizze erst einmal erklären.




*Ampel Fußgäner*

Wenn zwei LEDs in Reihe geschaltet werden, dann würden sie wie in der Skizze links dargestellt, beide leuchten. Wird aber zwischen diese beiden LEDs ein Controllerausgang angeschlossen, so leuchtet entweder Rot oder grün, je nach Ausgangssignal des Controllers. Dies machen wir uns zunutze und benutzen das nächste Bit, also Bit 3 für die Signaleinrichtung

der Fußgänger. Dazu erweitern wir die Initialisierung und ergänzen die Ampel und Zeitentabelle.

```
Ampelphasen:    .Byte 7          ; Offset 0 =  Rotphase
                                  ; Offset 1 = Rot – Gelbphase
                                  ; Offset 2 = Grünphase
                                  ; Offset 3 = Gelbphase
                                  ; Offset 4 = Wartezeit Fußgängerampel grün
                                  ; Offset 5 = Fußgänger
```

```
Ampelzeit1:    .Byte 1
AmpelZeit2:    .Byte 1
AmpelZeit3:    .Byte1
AmpelZeit4:    .Byte 1
AmpelZeit5:    .Byte 1
AmpelZeit6:    .Byte 1
```

Die Initialisierung für die Ampelphasen muss nun den neuen Umständen angepasst werden. Also sind auch diesmal Schritte zu erfassen. Danach ist es ein Leichtes, die Variablen zu initialisieren.

| | | |
|---|---|---|
| Schritt 0 | Fahrbahnampel Rot | Fußgängerampel Rot |
| Schritt 1 | Fahrbahnampel RotGelb | Fußgängerampel Rot |
| Schritt 2 | Fahrbahnampel Grün | Fußgängerampel Rot |
| Schritt 3 | Fahrbahnampel Gelb | Fußgängerampel Rot |
| Schritt 4 | Fahrbahnampel Rot | Fußgängerampel Rot |
| Schritt 5 | Fahrbahnampel Rot | Fußgängerampel Grün |

Dann beginnt alles wieder von vorn. In der Initialisierung sind nun die Bits und die Zeiten zu setzen. Nach dem Aufstellen der Tabelle für die einzelnen Schritte ist das überhaupt kein Problem

| | | | Ampelbits | Zeiten |
|---|---|---|---|---|
| Schritt 0 | Fahrbahn Rot | Fußgänger Rot | 00000001 | 2 Sek |
| Schritt 1 | Fahrbahn RotGelb | Fußgänger Rot | 00000011 | 1 Sek |
| Schritt 2 | Fahrbahn Grün | Fußgänger Rot | 00000100 | 7 Sek |
| Schritt 3 | Fahrbahn Gelb | Fußgänger Rot | 00000010 | 2 Sek |
| Schritt 4 | Fahrbahn Rot | Fußgänger Rot | 00000001 | 2 Sek |

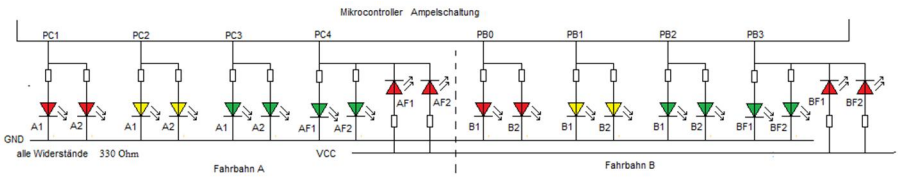
| | | | | |
|---|---|---|---|---|
| Schritt 5 | Fahrbahn Rot | Fußgänger Grün | 00001001 | 5 Sek |

Selbst eine Kreuzung mit allem drum und dran ist nach einem solchen Schema kein Problem. Nehmen wir zum Abschluss noch eine Kreuzung mit Fußgänger. Das passt grad so in das Byte Ampel.

Auch hier wieder zuerst die Tabelle für die einzelnen Schritte.

| | Fahrbahn A | Person A | Fahrbahn B | Person B | Ampelbits | Zeiten |
|---|---|---|---|---|---|---|
| Schritt 0 | Rot | Rot | Rot | Rot | 00010001 | 2 Sek |
| Schritt 1 | RotGelb | Rot | Rot | Grün | 10010011 | 1 Sek |
| Schritt 2 | Grün | Rot | Rot | Grün | 10010100 | 7 Sek |
| Schritt 3 | Grün | Rot | Rot | Rot | 00010100 | 2 Sek |
| Schritt 4 | Gelb | Rot | Rot | Rot | 00010010 | 2 Sek |
| Schritt 5 | Rot | Rot | Rot | Rot | 00010001 | 2 Sek |
| Schritt 6 | Rot | Grün | RotGelb | Rot | 00111001 | 1 Sek |
| Schritt 7 | Rot | Grün | Grün | Rot | 01001100 | 7 Sek |
| Schritt 8 | Rot | Rot | Grün | Rot | 01000001 | 2 Sek |
| Schritt 9 | Rot | Rot | Gelb | Rot | 00100001 | 2 Sek |

Zuerst die Hardware. Da sind vier weitere LEDs erforderlich. Setzen wir nun Bit 4 wieder für Rot, Bit 5 für das zweite Gelb, Bit 6 für Grün und Bit 7 für die Fußgängerampel vom Byte Ampel dazu. Ein Blick auf das Schaltbild 2.1.5 zeigt, das für die Ausgänge nun PB0 bis PB3 verwendet werden könnten. Zeichnen wir uns ruhig einmal die gesamte Beschaltung für eine Ampelkreuzung auf.




*Ampelkreuzung*

Das Programm ist nicht irgendwo zu ändern, sondern in Write_IO.Klar, die Initialisierung muss angepasst werden und auch der Schrittzähler, aber die Zuweisung an die Ausgänge geschieht einzig und allein in Write_IO. So behält man auch den Überblick.

```
;*************************************************************
;*          Ausgabe an die IO-Ebene                  *
```

```
;*          Port C Bit 1-5 = Relais / LED                *
;*************************************************************
Write_IO:
   IN        Reg_B, PortC                ; gesamten Port C lesen
   ANDI      Reg_B, 11100001             ; Bit 1-4 löschen
   LDS       Reg_A, Ampel                ; Variable für Ampel-LED holen
   LSL       Reg_A                       ; einmal link schieben
   ANDI      Reg_A, 00111110             ; Bit 0 und Bit 5-7  ausmaskieren
   Or        Reg_B, Reg_A                ; Register a und B zusammenführen
   OUT       PortC, Reg_B                ; Port beschreiben
;*************************************************************
;       Zuweisung für zweite Ampelgruppe            *
;*************************************************************
    LDS      Reg_A, Ampel                ; Variable für Ampel-LED erneut holen
    SWAP     Reg_A                       ; Nibble tauschen
    ANDI     Reg_A, 0b00001111           ; oberen vier Bit löschen, unteren ausmaskieren
    In       Reg_B, PortB                ; akt. Status Port B laden
    ANDI     Reg_B, 0b11110000           ; PB0 bis PB3 löschen
    OR       Reg_B, Reg_A                ; Mit Register A verodern
    Out      PortB, Reg_B                ; und Port B ausgeben
RET
```

Mit dieser Übung haben wir einen Grundstein für einen Dezimal nach Siebensegment Decoder gelegt.

### *2.14.6 Ziffernanzeige (7 Segment)*

Eine Siebensegmentanzeige mit einem Controller anzusteuern ist in der Regel kein großes Problem. Zuerst muss man wissen, wie die Anschlüsse belegt sind. Es gibt Anzeigen mit gemeinsamer Kathode (GND) oder gemeinsamer Anode (VCC). Die Belegung der Pins findet man im Datenblatt. Liegt aber eine unbekannte Anzeige vor, ist es auch kein Problem, die Belegung herauszufinden. Dazu wird einfach irgendeinen Pin an GND gelegt und mit VCC über einen Widerstand von 330 Ohm (Bei VCC = 5 V) werden die anderen abgetastet. Entweder, es leuchten die Segmente auf, es leuchtet nur ein Segment oder passiert nichts dergleichen. (A) Bei letzterem einfach die Polarität tauschen und den Vorgang wiederholen.

Nun müsste zumindest ein Segment leuchten. (B). Man läßt einen Anschluss und tastet mit dem anderen weiter. Wenn kein weiteres Segment leuchtet hat man den Gemeinsamen erwischt. Er wird dann fest an das Potential gelegt. Am Bestten skizziert man die Pinbelegung gleich mit. Die anderen Segmente werden dann wieder mit Widerstand im Stromkreis abgetastet. Jetzt sollten alle Segmente einmal aufleuchten. So können wir nun ein Anschaltbild erstellen und die Pins den Segmenten a bis g zuordnen. Die Segmente werden von oben im Uhrzeigersinn durch benannt, Das letzte Segment g ist der Balken in der Mitte. Hier noch einmal den Vorgang in Bildern


*Anzeige Belegung finden*

Fall A: Überhaupt kein leuchtendes Segment – falsch gepolt

Fall B: Ein Segment leuchtet, aber beim Abtasten mit Widerstand leuchtet kein weiteres Segment ( Fall C ) - Der noch angeschlossene ist nicht der Gemeinsame

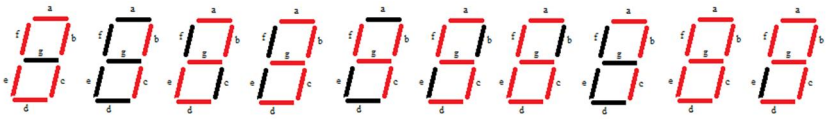
Fall D – Der Gemeinsame ist gefunden. Nun kontrollieren, ob VCC (gemeinsame Anode) oder GND (gemeinsame Kathode)

Solange der Gemeinsame fest an ein Potenzial angeschlossen wird, kann eine Anzeige vom Controller so beschaltet werden. Der Strom durch ein Segment ist in diesem Fall 5V/ 330 Ohm das entspricht etwa 15 mA. Das kann der Ausgang eines Controllers ohne Probleme leisten. Aber wir möchten doch mehr als eine Anzeige haben und da ist ein wenig mehr Aufwand zu betreiben. Schließlich werden für jede Anzeige 7 Segmentanschlüsse erforderlich und das wären bei 4 Anzeigen dann schon mal 28 IO-Pins des Controllers.

### *2.14.7 Zahl zu Siebensegment*

Doch soweit sind wir noch nicht. Erst einmal müssen wir uns Gedanken machen, wie Zahlenwerte zwischen 0 und 9 als Ziffern auf einer Anzeige dargestellt werden können. Dazu erst einmal wieder ein Byte, welches später in Write_IO seinen Inhalt als Bits an die Ausgänge bringt. Nennen wir es Matrix. Was muss dieses Byte leisten?

Dazu werfen wir einen Blick auf die Darstellung der Zahlen auf einem Siebensegmentdisplay an.



| Segment | x | g | f | e | d | c | b | a |
|---|---|---|---|---|---|---|---|---|
| Bit- Nr. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| Matrix 0 | Matrix 1 | Matrix 2 | Matrix 3 | Matrix 4 | Matrix 5 | Matrix 6 | Matrix 7 | Matrix 8 | Matrix 9 |
|---|---|---|---|---|---|---|---|---|---|
| 00111111 | 00000110 | 01011011 | 01001111 | 01100110 | 01101101 | 01111101 | 00000111 | 01111111 | 01101111 |

*Zahlenmatrix*

So wie es sich hier darstellt ist ein Array erforderlich, welches den Code von Matrix 0 bis Matrix 9 enthält. Deshalb deklarieren wir die Variablen Martix0 bis Matrix9 und initialisieren diese in einer Routine, die wir vor der Programmschleife aufrufen.

Um diese Matrix mit dem Code der Anzeige zu füllen, schreiben wir eine kleine Initialisierungsroutine. Dieses Vorgehen kennen wir bereits von anderen Initialisierungen.

```
;*****************************************************************************
;*     Initialisierung der Matrix für eine 7 Segmentanzeige        *
;*     Bit 0 = Segment a = PortB Bit 0                              *
;*     Bit 1 = Segment b = PortB Bit 1                              *
;*     Bit 2 = Segment c = PortB Bit 2                              *
;*     Bit 3 = Segment d = PortB Bit 3                              *
;*     Bit 4 = Segment e = PortB Bit 4                              *
;*     Bit 5 = Segment f = PortB Bit 5                              *
;*     Bit 6 = Segment g = PortD Bit 7                              *
;*****************************************************************************
Init_Matrix:
   LDI      Reg_A, 0b00111111        ;   Matrix 0
   STS      Matrix0 = Reg_A
   LDI      Reg_A, 0b00000110        ; Matrix 1
```

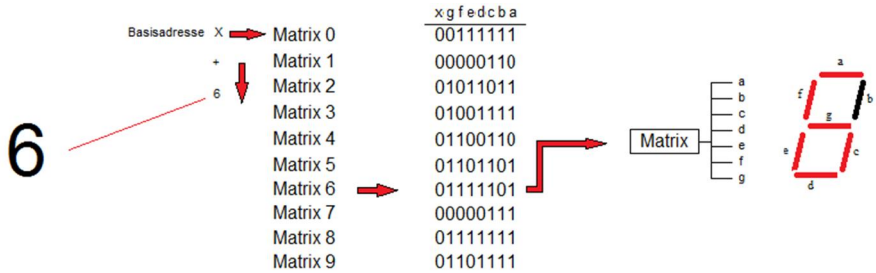
```
    STS     Matrix1= Reg_A
    LDI     Reg_A, 0b01011011      ; Matrix 2
    STS     Matrix2= Reg_A
    LDI     Reg_A, 0b01001111      ; Matrix 3
    STS     Matrix3= Reg_A
    LDI     Reg_A, 0b01100110      ; Matrix 4
    STS     Matrix4= Reg_A
    LDI     Reg_A, 0b01101101      ; Matrix 5
    STS     Matrix5= Reg_A
    LDI     Reg_A, 0b01111101      ; Matrix 6
    STS     Matrix6= Reg_A
    LDI     Reg_A, 0b00000111      ; Matrix 7
    STS     Matrix7= Reg_A
    LDI     Reg_A, 0b01111111      ; Matrix 8
    STS     Matrix8= Reg_A
    LDI     Reg_A, 0b01101111      ; Matrix 9
    STS     Matrix9= Reg_A
RET
```

Wie immer nach der Erstellung einer Initialisierung wird auch der Aufruf gleich platziert.

```
;*****************************************************************
;*                    Bereich Initialisierung                    *
;*****************************************************************
Start:
LDI   Reg_A, High(RamEnd)          ; erste Maßnahme Stack setzen
OUT   SPH, Reg_A
LDI   Reg_A, Low(RamEnd)
OUT   SPL, Reg_A
            ; weitere Initialisierungen
RCALL   Init_IO
RCALL   Init_USART                 ; serielle Schnittstelle parametrieren
RCALL   Init_Timer1                ; Initialisierung Timer
; Initialisieren Wertearray (Schrittkonstanten, Siebensegmentcode, etc.)
RCALL   Init_Var                   ;Initialisierung Variablen ( Eintragen Defaultwerte)
RCALL   Init_Ampelphasen           ;Initialisierung Variablen ( Eintragen Defaultwerte)
RCALL   Init_Matrix        ;Initialisierung Variablen ( Eintragen Defaultwerte)
```

Eine Zahl in einen Siebensegmentcode zu wandeln ist nun ganz einfach. Klar, dass der passende Code zur 0 in der Matrix0 steht. Habe ich eine 6 ist das die Speicherzelle Matrix0 +6. Dabei ist die Adresse von Matrix0 die

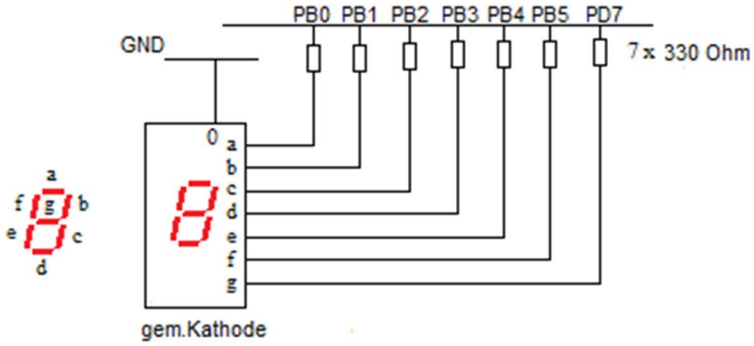
Basisadresse und die Zahl der Offset. Eine solche Tabellenstruktur haben wir bereits bei der Ampel eingesetzt. Hier ist es nicht anders. Die Skizze




verdeutlicht, wie aus der Zahl 6 die Ziffernanzeige 6 wird.

*Zahlen codieren*

Dazu ein paar Experimente. Zuerst der Schaltplan für den Aufbau der Hardware.




*Beschaltung einer Ziffernanzeige*

Damit wir hier nun etwas zu sehen bekommen, werden wir mit einem Taster den Wert hochzählen. Dazu brauchen wir eine Variable **Zaehler**. Im ersten Experiment wird der Taster direkt abgefragt in der Variable **New_In**, der schon den entprellten Eingang enthält.

```
Test_Anzeige_Cnt:
    LDS     Reg_A, New_In          ; Taster prüfen
    ANDI    Reg_A, 0b00000001      ;
    BREQ    End_Test_Anzeige
    LDS     Reg_A, Zaehler         ; Taster betätigt, Zählen durchführen
    INC     Reg_A
```

```
    CPI     Reg_A, 10              ; Wert < 10 ?
    BRLO Set_Counter
    CLR    Reg_A                   ; bei 0 anfangen
Set_Counter:
   STS    Zaehler, Reg_A          ; und speichern
    LDI    XL, Low(Matrix0)       ; Basisadresse Matrix
    LDI    XH, High(Matrix0)
    ADD   XL, Reg_A               ; Zahlenwert zuaddieren
    ADC   XH, Zero
    LD     Reg_A, X               ; Anzeigecode laden
    MOV  Ablage, Reg_A            ; Zwischenspeichern
    IN      Reg_B, PortB
    ANDI   Reg_B, 0b11000000      ; Port B ausmaskieren
    ANDI   Reg_A, 0b00111111
    OR     Reg_A, Reg_B           ; zusammenführen
    Out    PortB, Reg_A           ; und ausgeben
    In      Reg_B, PortD          ; Port D lesen
    MOV   Reg_A, Ablage           ; Matrix zurückholen
    ANDI   Reg_A, 0b01000000      ; letztes Bit ausmaskieren
    LSL    Reg_A                  ; nach links schieben
    ANDI   Reg_B, 0b01111111      ; Port D ausmaskieren
    OR     Reg_A, Reg_B           ; Register zusammenführen
    Out     PortD, Reg_A          ; und ausgeben
RET
```

Für diesen Test habe ich alles in eine Routine geschrieben. Das ist natürlich nicht nach der Vorgabe von EVA, aber es soll auch nur ein schneller Test sein.Die Subroutine wird in der Programmschleife direkt aufgerufen.

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO              ; Eingänge lesen
   RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit          ; Bearbeiten „V“
   RCALL    Blinker              ; Blinkerbit bilden
   RCALL     IO_Event_To_1       ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL     Taster_Event        ; Tasterereignisse bearbeiten
   RCALL     Test_Anzeige_Cnt    ; Ausgabe auf 7-Segmentanzeige
```

```
   RCALL    Chk_Time_Flag      ; Zeitereignisse bearbeiten
   RCALL    Chk_Receive        ; Datenempfang prüfen
   RCALL    Is_First           ; erstes empfangenes Byte prüfen und markieren
   RCALL    Is_Second          ; Datenempfang endgültig auswerten
   RCALL    Write_IO           ; und ausgeben
RJMP Loop
```

Der Zähler läuft sofort hoch, wenn wir den Taster betätigen, einzelne Zählung ist nicht möglich. Um eine einzelne Zählung bei jedem Tastendruck zu erhalten, müssen wir auf die Flankenbits in der Variablen Event_To_High oder Event_To_Low zurückgreifen. Um für weitere Änderungen gerüstet zu sein, verteilen wir einfach die Aufgaben.

Dazu bauen wir uns eine kleine Routine Count_Up, die nur den Zähler bei jedem Aufruf um eins erhöht und bei 10 von vorn beginnt. Damit ist sie zufrieden. Was andere mit dieser Information machen ist dieser Routine völlig egal.

```
;*****************************************************************
;*  Universeller Zählerbaustein für diverse                      *
;*  Anwendungen. Kann beliebig erweitert werden                  *.
;*****************************************************************
Count_Up:
  LDS    Reg_A, Zaehler
  Inc    Reg_A
  CPI    Reg_A, 10
  BRLO   Set_Zaehler
  CLR    Reg_A
Set_Zaehler:
  STS    Zaehler, Reg_A
RET
```

Dieser Baustein ist so durchsichtig, das auf eine Kommentierung verzichtet werden kann. Ein solches Konstrukt ist auch nach Wochen und Monaten durchsichtig. Allerdings nicht, warumm ein solch einfaches Modul erschaffen wurde und für diese Information ist der Kommentarbereich im Kopf der Subroutine ideal.

Die Subroutine Test_Anzeige_Cnt benennen wir in **Set_Anzeige** um und passen sie an. Natürlich muss auch in der Programmschleife der Aufruf in **Set_Anzeige** geändert werden.

```
Set_Anzeige:
```

```
    LDS    Reg_A, Zaehler          ; Taster betätigt, Zählen durchführen
    LDI    XL, Low(Matrix0)        ; Basisadresse Matrix
    LDI    XH, High(Matrix0)
    ADD    XL, Reg_A               ; Zahlenwert zuaddieren
    ADC    XH, Zero
    LD     Reg_A, X                ; Anzeigecode laden
    MOV    Ablage, Reg_A           ; Zwischenspeichern
    IN     Reg_B, PortB
    ANDI   Reg_B, 0b11000000       ; Port B ausmaskieren
    ANDI   Reg_A, 0b00111111
    OR     Reg_A, Reg_B            ; zusammenführen
    Out    PortB, Reg_A            ; und ausgeben
    In     Reg_B, PortD            ; Port D lesen
    MOV    Reg_A, Ablage           ; Matrix zurückholen
    ANDI   Reg_A, 0b01000000       ; letztes Bit ausmaskieren
    LSL    Reg_A                   ; nach links schieben
    ANDI   Reg_B, 0b01111111       ; Port D ausmaskieren
    OR     Reg_A, Reg_B            ; Register zusammenführen
    Out    PortD, Reg_A            ; und ausgeben
RET
```

Der Aufruf von **Count_Up** erfolgt in der Bearbeitung der Tasterevents. Dabei testen wir Bit 0. Ist es gesetzt, wird es quittiert und die Subroutine für diesen Job **Count_Up** aufgerufen.

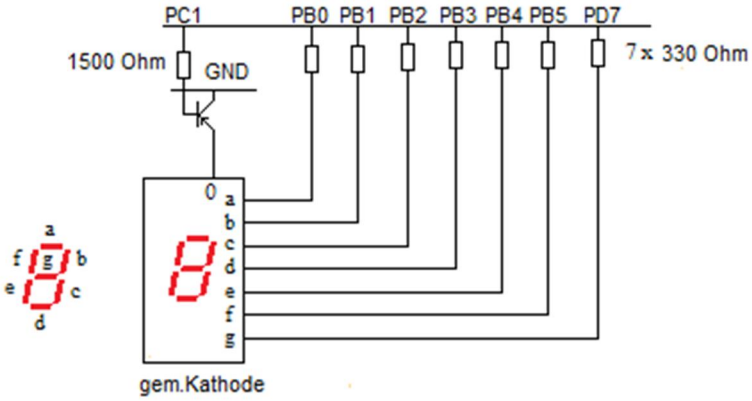
```
;*********************************************************************
;*                    Tastersignal prüfen                            *
;*********************************************************************
Taster_Event:
   LDS    Reg_A, Event_To_High    ; Ereignisbits laden
   ANDI   Reg_A, 0b00000001       ; Nur Bit 1 prüfen (Ergebnis ist nicht 0)
   BREQ   Chk_Taster_2H           ; Ergebnis 0, teste nächstes Bit
   LDS    Reg_A, Event_To_High    ; Ereignisbits laden
   ANDI   Reg_A, 0b11111110       ; Ereignisbit 0 löschen
   STS    Event_To_High, Reg_A    ; Und Ereignisse zurückschreiben
   RCALL  Count_Up                ; Zähler hochzählen. Eigene Subroutine
Chk_Taster_2H:
   ANDI   Reg_A, 0b00000010       ; Nur Bit 1 prüfen (Ergebnis ist nicht 0)
   BREQ   Chk_Taster_Low          ; Ergebnis 0, dann nächster Test
  ; etc.....
```

Nun haben wir eine komplette Aufgabenteilung. Dadurch wird es auch ziemlich einfach, die Sekunden zu zählen. Dazu brauchen wir nur das Zeitereignis Sekunde um dort den Aufruf Count_Up unterzubringen. In der Bearbeitung der Tasterevents entfernen wir den Aufruf.

Das Ergebnis ist ein fortlaufender Zählerzwischen 0 und 9. Hier haben wir nun Gelegenheit, auch den Zeitabstand zu prüfen. Passt die Zeit? Wenn die Kommunikation mit dem PC funktioniert, sollte die Zuordnung CPU-Frequenz passen. Jetzt kann nur noch ein Fehler im Teiler oder im Vergleichswert des Counters liegen, wenn der Zeittakt nicht stimmt. Diese Einstellungen finden wir in der Initialisierung des Timers.

Ist alles korrekt, sollte auch der Zähler im Sekundentakt laufen.

Erweitern wir nun dieses Experiment und setzen einen Transistor zwischen GND und der gemeinsamen Kathode ein. Schnell ist auch ein Schaltbild skizziert.




*Ziffernanzeige mit Transistor*

Die Basis des Transistors schalten wir über einen Widerstand von 4,7kOhm an den Ausgang PC1. Den müssen wir nun mit einer 0 programmieren, damit die Ziffer angezeigt wird. Führt der Ausgang eine 1, ist die Anzeige dunkel.

Auch das testen wir mit einem kleinen Programm. Dazu erfinden wir wieder eine kleine Subroutine, die in Abhängigkeit vom Zählerwert die Anzeige zu und abschaltet. Der Name Sel_Ziffer scheint mir für die Routine geeignet.

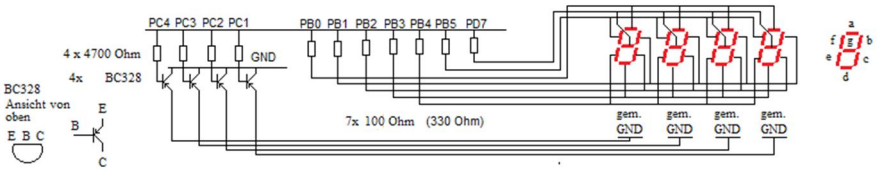
```
;*****************************************************************
;*    Ziffer auf Anzeige ein- und ausblenden                    *
;*****************************************************************
Sel_Ziffer:
   LDS   Reg_A,Zaehler          ; läuft unabhängig in anderer Routine
   CPI   Reg_A,4                ; abschalten zwischen 3
   BREQ  Ziffer_Off
   CPI   Reg_A,7                ; und 7
   BRNE  End_Sel_Ziffer
   In    Reg_A,PortC            ; Port erst einlesen
   COM   Reg_A                  ; Bits für 1-Logik drehen
   ORI   Reg_A,0b00000010       ; 1 schaltet ein
   COM   Reg_A                  ; und wieder drehen 0
   Out   PortC,Reg_A            ; und Ausgabe
   RJMP  End_Sel_Ziffer
Ziffer_Off:
   In    Reg_A,PortC            ; Port erst einlesen
   COM   Reg_A                  ; Bits für 1-Logik drehen
   ANDI  Reg_A,0b11111101       ; 0=Anzeige aus. hier steht spatter ein anderes Register
   COM   Reg_A                  ; und wieder drehen
   Out   PortC,Reg_A            ; und Ausgabe
End_Sel_Ziffer:
RET
;-----------------------------------------------------------
```

Die Anzeige bleibt bei den Zahlen 4,5 und 6 aus.

### *2.14.8 Anzeige multiplexen*

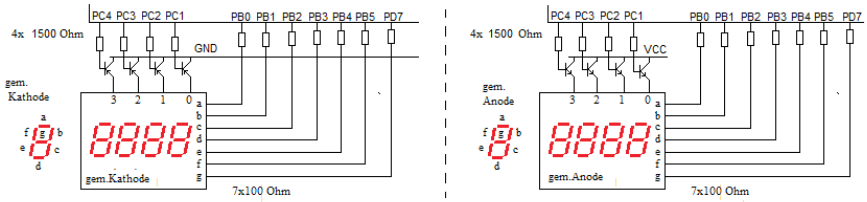
Zur Darstellung einer 4stelligen Zahl könnte man 4x eine solche Anzeige aufbauen. Das ist aber nicht notwendig, wenn die Anzeigen der Reihe nach mit dem Gemeinsamen angesteuert werden und dazu passend die Segmente. Dann braucht es nur einen Ausgang je Anzeige + 7 Ausgänge für die Segmente. Bei einer 4 stelligen Anzeige dann eben nur 11 IO-Pins des Controllers. jede zusätzliche Anzeige wäre dann lediglich ein Ausgang mehr. Diesen Betrieb nennt man Multiplexen. Man macht sich einfach die Trägheit unseres Gehirns zunutze, um eine ganze Zahl zu sehen statt einzelner Anzeigen, die durchgetaktet werden. Die Vorarbeit ist bereits geleistet.

Beschalten wir nun eine mehrstellige Siebensegmentanzeige. Manche müssen die Segmente separat verdrahtet bekommen, bei anderen sind auch die Segmente untereinander bereits verbunden. Gehen wir einmal davon aus, es sind einzelne Anzeigen, die jede für sich die Segmente a, b, c, d, e, f und g sowie den gemeinsamen GND herausgeführt haben. Hier schalten wir zuerst alle Segmente parallel, das heißt, alle Segmente a werden verbunden, alle b, alle c und so weiter. Dann werden die gemeinsamen mit einem Transistor auf die Portbits PC1, PC2, PC3 und PC4 geschaltet. Dazu wie immer ein Schaltplan.




*Schaltplan mehrstellige Anzeige*

Nun könnte zu Recht die Frage gestellt werden, darf man mehrere Dioden parallel an einen Widerstand hängen. Nein, aber das hier ist etwas anders. Durch die Transistoren werden die Anzeigen nicht alle auf einmal eingeschaltet, sondern der Reihe nach, so dass immer nur ein Segment, also eine LED angeschlossen ist. Er ist auch als Stromtreiber erforderlich, da ein Controllerausgang nicht den Strom von 7 Segmenten liefern kann und bei einer 8 sind alle Segmente zugeschaltet. Das würde bei einem Strom pro Segment die stolze Summe von rnd. 100 mA bedeuten. Ein Transistor hingegen ist mit ein paar µA zur Ansteuerung zufrieden und liefert die 100 mA für die Anzeige locker.. Den gesamten Strom der Anzeige leitet nun der Transistor. Sehen wir uns einmal diesen Abschnitt unserer Experimentierschaltung an.




*Anzeige im Multiplexbetrieb.*

Die Anzeigen brauchen etwas mehr Strom, damit sie hell genug werden. Man kann im Datenblatt nachlesen, wie hoch der Impulsstrom sein darf, aber man kann auch Pi mal Daumen ausprobieren, ob 20 mA oder 30 mA eine gut lesbare Anzeige liefern. Aber im Moment möchte ich die Anzeigen ja noch statisch ansteuern und da sind die 330 Ohm erforderlich, um die Anzeige nicht zu zerstören.

Kommen wir nun zum Programm. Es ist eine Routine erforderlich, die die vier Ziffern der Reihe nach ansteuert. Dazu wird eine Variable benötigt, in der ein Bit durchgeschoben und das dann am Port C ausgegeben wird. Definieren wir also eine Variable Multiplx und auch einen Zähler, denn es ist ja nur eine begrenzte Anzahl von Anzeigen aufgebaut.

```
Multiplx:    .Byte 1        ; Schieberegister
                            ; Bit 1 = Select Ziffer 1
                            ; Bit 2 = Select Ziffer 2
                            ; Bit 3 = Select Ziffer 3
                            ; Bit 4 = Select Ziffer 4
Ziffer_Cnt:  .Byte 1        ; Stellenzähler
```

Diese Variablen fügen wir nun unserem Programm hinzu und schreiben die Routine Sel_Ziffer um.

```
;*****************************************************************
;*    Ziffer auf Anzeige ein- und ausblenden                     *
;*****************************************************************
Sel_Ziffer:
  LDS   Reg_A,Ziffer_Cnt          ; Ziffernzähler laden
  LDS   Reg_B,Multiplx            ; Schieberregister laden
  LSL   Reg_B                     ;; Schieberegister links schieben
```

```
  INC   Reg_A                        ; Ziffernzähler hochzählen
  CPI   Reg_A,4                      ; vierstellige Anzeige Grenzwert erreicht ?
  BRLO  Next_Ziffer                  ; Wenn nicht, dann nächste Ziffer selektieren
  CLR   Reg_A                        ; sonst von vorn beginnen
  LDI   Reg_B, 0b00000010
Next_Ziffer:
  STS   Ziffer_Cnt, Reg_A            ; Ziffernzähler abspeichern
  STS   Multiplx, Reg_B              ; Schiebereister abpeichern
  In    Reg_C,PortC                  ; Port erst einlesen
  COM   Reg_A                        ; Bits drehen, Ziffer ist 0 aktiv
  ANDI  Reg_A,0b11100001             ; alle Portbits aus
  ANDI  Reg_B,0b00011110             ; ungültige Bits ausmaskieren
  OR    Reg_A,Reg_B                  ; mit Schieberegister verodern
  COM   Reg_A                        ; wieder drehen
  Out   PortC,Reg_A                  ; und Ausgabe
RET
;-----------------------------------------------------------
```

Wenn wir nun das Programm starten werden wir alle vier Ziffern gleichzeitig sehen, so als wären sie parallel geschaltet. Aber dem ist nicht so. Wir haben den Aufruf von dieser Routine noch in der Programmschleife. Durch den schellen Wechsel erscheint die komplette Anzeige und da wir nur eine Zahl aufschalten, könnte man das falsch interpretieren. Das dem so nicht ist, lässt sich leicht testen. Nehmen wir den Aufruf der Routine Set_Ziffer aus der Hauptschleife heraus und fügen sie im Zeitereignis Sekunde oder 100mSek. ein. Nun sehen wir, wie die Zahl durch die Ziffern durchgeschaltet wird.

```
;*********************** Zeitereignis 100 mSek *******************
;*        Zeitroutinen im Zeitraster 100 mSek                    *
;*************************************************************************
Event_100msek:                       ; Zeitevent bearbeiten
  LDS       Reg_A, Time_Flag
  ANDI      Reg_A,0b11111011         ; Zeitflag quittieren
  STS       Time_Flag, Reg_A
  RCALL     Sel_Ziffer               ; langsames Multiplexen
RET
;-----------------------------------------------------------------------
```

Die Vorbereitung für eine Anzeige im Multiplexbetrieb ist erledigt. Sind die Testergebnisse klar, kann der Aufruf auch im Zeitereignis 1mSek aufgerufen werden. Dann ergibt sich eine durchgehende 4 stellige Zahl.

Versuchen wir nun Zahlen auf der Anzeige sichtbar zu machen. Erweitern wir dazu die Variablen Counter0 und Counter1 mit Counter2 und Counter3 und ändern die Routine Count_Up, um diese Variablen im Zehntelsekundentakt hochzuzählen.

```
;*************************************************************
;*                    Zähler aktivieren                    *
;*************************************************************
Count_Up:
  LDS    Reg_A, Counter0          ; Ziffer 1 Zahl * 10 ^ 0
  LDS    Reg_B, Counter1          ; Ziffer 2 Zahl * 10 ^ 1
  LDS    Reg_C, Counter2          ; Ziffer 3 Zahl * 10 ^ 2
  LDS    Reg_D, Counter3          ; Ziffer 4 Zahl * 10 ^ 3
  Inc    Reg_A
  CPI    Reg_A, 10
  BRLO   Set_Zaehler
  CLR    Reg_A                    ; Übertrag nächste Dekade
  Inc    Reg_B
  CPI    Reg_B, 10
  BRLO   Set_Zaehler
  CLR    Reg_B                    ; Übertrag nächste Dekade
  Inc    Reg_C
  CPI    Reg_C, 10
  BRLO   Set_Zaehler
  CLR    Reg_C                    ; Übertrag nächste Dekade
  Inc    Reg_D
  CPI    Reg_D, 10
  BRLO   Set_Zaehler
  CLR    Reg_D
Set_Zaehler:
  STS    Counter0, Reg_A
  STS    Counter0, Reg_B
  STS    Counter0, Reg_C
  STS    Counter0, Reg_D
  RCALL  Convert_Matrix           ; Zahlen in 7 Segmentcode  umsetzen
RET
;-------------------------------------------------------------.
```

Der Aufruf Convert_Matrix ist erforderlich, um die Zahlen in den 7 Segmentcode umzusetzen und in den Ausgabepuffer zu legen. Dies ist bei jeder Änderung der Zahl erforderlich und vergrößert natürlich auch die Zykluszeit. Schließlich erfolgt der Aufruf alle 10 mSek.

```
;*************************************************************************
;*  Zahlen in konvertieren und in Matrixpuffer ablegen     *
;*************************************************************************
Convert_Matrix:
  CLR     Reg_A                  ; Ziffernzähler
  LDI     XL, Low(Counter0)      ; Basisadresse Quelle
  LDI     XH, High(Counter0)
  LDI     YL, Low(Code_Puffer)   ; Basisadresse Matrixpuffer (Ziel)
  LDI     YH, High(Code_Puffer)
Loop_Convert:
  Ld      Reg_B, X+
  LDI     ZL, Low(Matrix0)       ; Basisadresse Matrix
  LDI     ZH, High(Matrix0)
  ADD     ZL, Reg_B              ; Zahlenwert zuaddieren
  ADC     ZH, Zero
  LD      Reg_B, Z               ; Anzeigecode laden
  ST      Y+, Reg_B              ; Zwischenspeichern
  INC     Reg_A
  CPI     Reg_A, 4
  BRLO    Loop_Convert           ; Matrixpuffer für Ausgabe setzen
RET
;-----------------------------------------------------------------
```

Bis hierher ist es Sache der Routine Count_Up. Die generierten Zahlen in den Puffer für die Ausgabe zu schreiben. Dazu wird die Variable Code_Puffer umdeklariert.

```
Code_Puffer:     .Byte 4
```

Die Zahlen dort abholen und zur Anzeige zu bringen ist wiederum Sache des Multiplexers, dessen Subroutine Sel_Ziffer wir nun ändern werden.

Bisher war es Aufgabe des Multiplexers, eine Ziffer nach der anderen im Wechsel einzuschalten. Nun muss er sich auch darum kümmern, das auch die richtige Zahl erscheint. Die bereits erstellte Routine Set_Anzeige wird dort integriert und muss in der Hauptschleife entfernt werden.

```
;*********************************************************************
;*    Ziffer auf Anzeige ein- und ausblenden              *
;*    Anzeigen mit gemeinsamer Kathode                    *
;*********************************************************************
Sel_Ziffer:                                                    ;3
   LDS     Reg_A,Ziffer_Cnt       ; Stellenzähler laden        2
```

```
  LDS    Reg_B,Multiplx        ; Schieberegister laden                2
  LSL    Reg_B                 ; Select-Bit schieben                  1
  INC    Reg_A                 ; Stellenzähler hochsetzen             1
  CPI    Reg_A,4               ; Grenze prüfen                        1
  BRLO   Next_Ziffer           ;                                      2
  CLR    Reg_A                 ; evtl. von vorn anfangen              1
  LDI    Reg_B, 0b00000010     ;                                      1
Next_Ziffer:
  STS    Ziffer_Cnt, Reg_A     ; Stellenzähler abspeichern            2
  STS    Multiplx, Reg_B       ; Schieberegister abspeichern          2
  In     Reg_C,PortC           ; Port erst  einlesen                  1
  COM    Reg_C                 ; Bits drehen, Ziffer ist 0 aktiv      1
  ANDI   Reg_C,0b11100001      ; hier steht später ein anderes Register 1
  COM    Reg_C                 ; Bits drehen, Ziffer ist 0 aktiv      1
  Out    PortC,Reg_C           ; alle Ziffern aus                     1
  LDI    XL, Low(Code_Puffer)  ; Matrixpuffer Basisadresse laden      1
  LDI    XH, High(Code_Puffer) ;                                      1
  ADD    XL, Reg_A             ; Stellenzähler dazu addieren          1
  ADC    XH, Zero              ; Übertrag übernehmen (16 Bit Addition) 1
  LD     Reg_D, X              ; Matrix aktuelle Zahl holen           2
  MOV    Ablage, Reg_D         ; Zwischenspeichern                    1
  IN     Reg_E, PortB          ;                                      1
  ANDI   Reg_E, 0b11000000     ; Port B ausmaskieren                  1
  ANDI   Reg_D, 0b00111111     ; Bit 6 und 7 ausblenden               1
  OR     Reg_D, Reg_E          ; zusammenführen                       1
  Out    PortB, Reg_D          ; und ausgeben                         1
  In     Reg_E, PortD          ; Port D lesen                         1
  MOV    Reg_D, Ablage         ; Matrix zurückholen                   1
  ANDI   Reg_D, 0b01000000     ; letztes Bit (Bit 6) ausmaskieren     1
  LSL    Reg_D                 ; nach links schieben                  1
  ANDI   Reg_E, 0b01111111     ; Port D Bit 7 ausmaskieren            1
  OR     Reg_D, Reg_E          ; Register zusammenführen              1
  Out    PortD, Reg_D          ; und ausgeben                         1
  COM    Reg_C                 ; Stelle selektieren                   1
  ANDI   Reg_B,0b00011110      ;                                      1
  OR     Reg_C,Reg_B           ; aktuelle Ziffer einschalten          1
  COM    Reg_C                 ; für Ausgabe wieder drehen            1
  Out    PortC,Reg_C           ; und Ausgabe                          1
RET                            ;                                      4
;-------------------------------   insges.51 Taktzyklen ---------------------------------
```

Wie sich zeigt, benutzt diese Routine eine große Anzahl der Register. Der Grund ist ziemlich einfach: Möglichst schneller Code. Immerhin wird

diese Routine im Raster einer mSek. aufgerufen, und da ist dann Zeit schon ein wichtiges Kriterium. Ich habe einmal die benötigten Taktzyklen hinter die Befehle geschrieben, um den Zeitaufwand für diese Routine zu ermitteln. Diese Angabe findet man im Datenblatt des verwendeten Controllers. Bei einem Systemtakt von 8 MHz benötigt ein Takt die Zeit 0,000000125 Sek. oder 125 nSek. Mit 51 Takten belegt diese Routine die Zeit von 6,375 µSek. also weniger als 1% einer mSek. Damit liegen wir auf der sicheren Seite. Natürlich läßt der Atmega8 auch eine Taktfrequenz von 16 MHz zu und das halbiert die Durchlaufzeit.

Nun noch ein paar Worte zum Ablauf. Grundsätzlich ist die Anzeige erst einmal komplett abzuschalten, bevor eine neue Zahlenmatrix auf einen Ausgang gelegt wird. Der Grund ist das Nachleuchten der Ziffern. Lässt man die Abschaltung weg, leuchten alle Segmente ein wenig nach. Das ist störend. Daher wird auch der Port mit der Ziffernselektion zweimal zugewiesen. Dazwischen wird der Wert der neuen Ziffer von Anzeigepuffer auf die Anzeige geschaltet.

Übrigends, die Matrix läßt sich auch im Codesegment mit der Compileranweisung .db anlegen. Dazu wird am Ende des Programms einfach wieder eine Adressmarke gesetzt, in unserem Fall benutzen wir

```
Matrix0:    .db 0b00111111    ;  Matrix 0
Matrix1:    .db 0b00000110    ;  Matrix 1
Matrix2:    .db 0b01011011    ;  Matrix 2
Matrix3     .db 0b01001111    ;  Matrix 3
Matrix4:    .db 0b01100110    ;  Matrix 4
Matrix5:    .db 0b01101101    ; Matrix 5
Matrix6:    .db 0b01111101    ; Matrix 6
Matrix7:    .db  0b00000111    ; Matrix 7
Matrix8:    .db  0b01111111    ; Matrix 8
Matrix9:    .db 0b01101111    ; Matrix 9
```

Setzt man das Adressregister X auf Matrix0 und addiert die anzuzeigende Zahl, sollte die Anzeigematrix wie bei der Adressierung im Variablenbereich für die Zahl adressiert sein.

(ungeprüfte Option)

### *2.14.9 Anzeigen mit gemeinsamer Anode*

Die bisherige Auslegung des Programms bezog sich auf eine Anzeige mit gemeinsamer Kathode. Mit ein paar wenigen Änderungen ist aber auch eine Anzeige mit gemeinsamer Anode verwendbar.

```
;*****************************************************************
;*    Ziffer auf Anzeige ein- und ausblenden              *
;*    Anzeigen mit gemeinsamer Anode                      *
;*****************************************************************
Sel_Ziffer:                          ; Aufruf Subroutine                          3 Takte
   LDS     Reg_A,Ziffer_Cnt          ; Stellenzähler laden                        2
   LDS     Reg_B,Multiplx            ; Schieberegister laden                      2
   LSL     Reg_B                     ; Select-Bit schieben                        1
   INC     Reg_A                     ; Stellenzähler hochsetzen                   1
   CPI     Reg_A,4                   ; Grenze prüfen                              1
   BRLO    Next_Ziffer               ;                                            2
   CLR     Reg_A                     ; evtl. von vorn anfangen                    1
   LDI     Reg_B, 0b00000010         ;                                            1
Next_Ziffer:
   STS     Ziffer_Cnt, Reg_A         ; Stellenzähler abspeichern                  2
   STS     Multiplx, Reg_B           ; Schieberegister abspeichern                2
   In      Reg_C,PortC               ; Port erst  einlesen                        1
   COM     Reg_C                     ; Bits drehen, Ziffer ist 0 aktiv            1
   ANDI    Reg_C,0b11100001          ; hier steht später ein anderes Register     1
   COM     Reg_C                     ; Bits drehen, Ziffer ist 0 aktiv            1
   Out     PortC,Reg_C               ; alle Ziffern aus                           1
   LDI     XL, Low(Code_Puffer)      ; Matrixpuffer Basisadresse laden            1
   LDI     XH, High(Code_Puffer)     ;                                            1
   ADD     XL, Reg_A                 ; Stellenzähler dazu addieren                1
   ADC     XH, Zero                  ; Übertrag übernehmen (16 Bit Addition)      1
   LD      Reg_D, X                  ; Matrix aktuelle Zahl holen                 2
   MOV     Ablage, Reg_D             ; Zwischenspeichern                          1
   IN      Reg_E, PortB              ;                                            1
   COM     Reg_E                     ; auf 1 Logik anpassen  (Änderung)           1
   ANDI    Reg_E, 0b11000000         ; Port B ausmaskieren                        1
   ANDI    Reg_D, 0b00111111         ; Bit 6 und 7 ausblenden                     1
   OR      Reg_D, Reg_E              ; zusammenführen                             1
   COM     Reg_E                     ; und wieder zurückdrehen  (Änderung)        1
   Out     PortB, Reg_D              ; und ausgeben                               1
   In      Reg_E, PortD              ; Port D lesen                               1
   COM     Reg_E                     ; auf 1 Logik anpassen  (Änderung)           1
   MOV     Reg_D, Ablage             ; Matrix zurückholen                         1
```

```
  ANDI    Reg_D, 0b01000000     ; letztes Bit (Bit 6) ausmaskieren          1
  LSL     Reg_D                 ; nach links schieben                       1
  ANDI    Reg_E, 0b01111111     ; Port D Bit 7 ausmaskieren                 1
  OR      Reg_D, Reg_E          ; Register zusammenführen                   1
  COM     Reg_D                 ; und wieder zurückdrehen  (Änderung)       1
  Out     PortD, Reg_D          ; und ausgeben                              1
  COM     Reg_C                 ; Stelle selektieren                        1
  ANDI    Reg_B,0b00011110      ;                                           1
  OR      Reg_C,Reg_B           ; aktuelle Ziffer einschalten               1
  COM     Reg_C                 ; für Ausgabe wieder drehen                 1
  Out     PortC,Reg_C           ; und Ausgabe                               1
RET                             ;                                           4
;-------------------------------   insges.51 Taktzyklen ---------------------------------
```

Was auch immer wir im Programm tun, immer beziehen wir uns auf den logischen Pegel 1 und das entspricht Ein. Es macht uns die Arbeit leichter und die zwei Takte für das Zweimalige umdrehen der Bitlage belastet den Zyklus kaum.

.

### 2.14.10 *Anzeige mit verschiedenen Werten beschalten*

In der Erfassung externer Signale haben wir einen Taster mit Mode bezeichnet. Wir werden nun das Ereignis **Event_To_High** dazu benutzen, eine Variable Anz_Mode hochzuzählen und bei Erreichen eines Maximalwertes wieder von 0 beginnen. Natürlich muss diese Variable auch deklariert werden.

```
Anz_Mode:        .Byte 1     ; Int8  Zähler für verschiedene Programmmodi
```

**Event_To_High** haben wir bereits in der Flankenaufbereitung ausgiebig besprochen. Bit 0 ist für die Umschaltung Mode vorgesehen.

```
Set_Mode:
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b00000001
    BREQ    End_Set_Mode            ; Kein Eventbit, dann Ende
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b11111110       ; Eventbit löschen
    STS     Event_To_High, Reg_A    ; und zurückschreiben
    LDS     Reg_A, Anz_Mode
    INC     Reg_A
    STS     Anz_Mode, Reg_A
    CPI     Reg_A, 4                ; Grenzwert ist 3
    BRLO    End_Set_Mode
    CLR     Reg_A                   ; Beginnt wieder mit Mode 0
    STS     Anz_Mode, Reg_A
End_Set_Mode:
RET
```

Die Variable wird nun Werte bis maximal 3 zur Verfügung stellen. In der Hauptschleife fügen wir einfach **RCALL Set_Mode** ein. Ist kein Eventbit gesetzt, wird auch keine weitere Arbeit erledigt.

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO             ; Eingänge lesen
```

```
  RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
  RCALL    IO_Debounce_T        ; Entprellen mit Timer
  RCALL    Set_LED_Bit          ; Bearbeiten „V“
  RCALL    IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
  RCALL    Taster_Event         ; Tasterereignisse bearbeiten
  RCALL    IO_Event_Flanke      ; Eingänge beide Flanken erfassen
  RCALL    Set_Mode             ; Anzeige umschalten
  RCALL    Chk_Time_Flag        ; Zeitereignisse bearbeiten
  RCALL    Chk_Receive          ; Datenempfang prüfen
  RCALL    Is_First             ; erstes empfangenesByte prüfen und markieren
  RCALL    Is_Second            ; Datenempfang endgültig auswerten
  RCALL    Write_IO             ; und ausgeben
RJMP Loop
```

Über den Inhalt der Variable **Anz_Mode** können wir später entscheiden, welche Werte zur Anzeige geschaltet werden sollen. Damit wir uns auch den Wert von Anz_Mode ansehen können, ist eine Kopie der Variablen in Open_Eye zu filtern und ein neues Projekt anzulegen.

## 2.15 Bau einer Uhr

Wir haben nun gelernt eine 4 stellige Anzeige aufzubauen. Das ist ein Anlass, diese Anzeige mal etwas Sinnvolles anzeigen zu lassen. Eine Uhr könnte man damit aufbauen, oder einen Wecker, eine Stoppuhr. Aber halt, nicht alles auf einmal. So wie wir uns bei der Ampel vorgearbeitet haben, sollten wir es hier auch angehen, Schritt für Schritt. Wenn ein Teil eines Programms fertig ist und funktioniert, ist es oft nicht schwer, weitere Funktionen einzubauen. Beginnen wir erst einmal mit der Uhr. Um Sekunden zu zählen führen wir zusätzlich zu Minuten Einer, Minuten Zehner, Stunden Einer und Stunden Zehner noch eine Variable Sekunden ein.

```
Sekunden:  .Byte 1
E_Min:     .Byte 1
Z_Min:     .Byte 1
E_Std:     .Byte 1
Z_Std:     .Byte 1
```

Das Programm ist gar nicht so schwer. Wir haben ein Ereignis in der Timer ISR, das uns jede Sekunde ein Bit setzt. Mit diesem Ereignis haben wir ja auch schon die Ampel gesteuert. Jetzt wollen wir ein Uhrenprogramm hinzufügen. Dann brauchen wir diese Sekunden dort eigentlich nur noch zählen und daraus Minuten und Stunden ableiten.

```
Prg_Uhr:
   LDS     Reg_A, Sekunden        ; *** Sekunden  ***
   INC     Reg_A
   STS     Sekunden, Reg_A
   CPI     Reg_A, 60
   BRLO    End_Uhr
   CLR     Reg_A                    ; *** Minuten Einer***
   STS     Sekunden, Reg_A
   LDS     Reg_A, E_Min
   INC     Reg_A
   STS     E_Min, Reg_A
   CPI     Reg_A, 10
```

```
   BRLO    End_Uhr
   CLR      Reg_A                    ; *** Minuten Zehner***
   STS      E_Min, Reg_A
   LDS      Reg_A, Z_Min
   INC     Reg_A
   STS     Z_Min, Reg_A
   CPI     Reg_A, 6
   BRLO    End_Uhr
   CLR     Reg_A                    ; *** Stunden Einer  ***
   STS     Z_Min, Reg_A
   LDS     Reg_A, E_Std
   INC     Reg_A
   STS     E_Std, Reg_A
   LDS     Reg_B, Z_Std
   CPI     Z_Std, 2                 ; *** unter 20 Stunden ***
   BRLO    Chk_10
   CPI     Reg_A,4                  ; *** 20 Stundenbereich ***
   BRLO    End_Uhr
   CLR     Reg_A
   CLR    Reg_B
   STS     E_Std, Reg_A
   STD    Z_Std, Reg_B
   RJMP    End_Uhr
CHK_10:
   CPI     Reg_A, 10                ; *** unter 20 Stunden ***
   BRLO    End_Uhr
   CLR     Reg_A
   STS     E_Std, Reg_A
   INC     Reg_B
   STS     Z_Std, Reg_B
End_Uhr:
RET
```

Nun, das ist schon ein ziemlich langes Programm, aber von der Zeit bekommen wir noch nichts zu sehen. Dazu muss die Zeit noch in den Zahlenpuffer übertragen werden. Die Routine  Show_Uhr prüft, ob der Anzeigemode die Uhrzeit ist und überträgt gegebenenfalls mit gleichzeitiger Konvertierung die Daten in den Anzeigepuffer. Da wir nun von verschiedenen Wertepuffern die Daten zur Anzeige bringen, kann die Basisadresse der Quelldaten nicht mehr in der Subroutine Convert_Matrix erfasst werden. Dies muss im aufrufenden Programmteil geschehen. So ist gesichert, das die richtigen Werte zur Anzeige gelangen.

```
Show_Uhr:
```

```
    LDS   Reg_A, Anz_Mode
    CPI   Reg_A, 0                  ; Mode 0 aktiv- Uhrzeit anzeigen
    BRNE End_Show_Uhr              ; nein, dann Ende
    LDI XL, Low(E_Min)              ; Basisadresse Zeit setzen
    LDI  XH, High(E_Min)
    RCALL Convert_Matrix           ; und Werte in den Ausgabepuffer übertragen
End_Show_Uhr:
RET
```

Die Routine **Prg_Uhr** wird im Ereignisbearbeitung der Sekunde untergebracht. Ebenfalls reicht es, den Aufruf für Show_Uhr dort mit einzubinden, denn es ändert sich ja nur einmal in der Minute ein Wert. Eine Minute aber ist zu lang, denn die Datenaktualisierung könnte bei Wechsel des Programmmode auch bis zu einer Minute dauern.

```
Event_1Sek::
;------ Bereich der Zeiterreignisbearbeitung –1 Sek.---------
;-------------------------- Zeitflag quittieren -----------------------
     LDS       Reg_A, Time_Flags              ; Variable mit Zeitflags holen
     ANDI      Reg_A, 0b10111111              ; Flag für 1 Sek löschen
     STS       Time_Flags, Reg_A              ; Variable zurückschreiben
; Zeitjob
    RCALL   Prg:Uhr                          ; Aufruf der Uhr
    RCALL     Show_Uhr                       ; Uhr anzeigen ?
RET
```

Für einen Test allerdings setzen wir den Aufruf der beiden Routinen in das Zeitereignis 10mSek. Damit läuft die Zeit so schnell, das wir uns die Bearbeitung im Programmmode 0 ansehen können und testen, das in den anderen Modi die Uhr nicht sichtbar aktualisiert wird. Schalten wir durch bis Programmmode 0 sehen wir, das die Uhr zwischenzeitlich weitergelaufen ist.

### *2.15.1 Die Uhr stellen*

Nun wird auch die Uhrzeit angezeigt, aber mit 99% Sicherheit nicht korrekt. Also muss eine Routine zum Stellen der Uhr her.

Wenn Mode 0 die aktuelle Uhrzeit ist, dann ist Mode 1 die Anzeige der Stellwerte. Machen wir es uns erst einmal einfach und deklarieren vier Variablen zur Aufnahme der Einstellwerte.

```
E_Min_Stell:   .Byte 1
Z_Min_Stell:   .Byte 1
E_Std_Stell:   .Byte 1
Z_Std_Stell:   .Byte 1
```

Die Routine zum Aufruf aus dem Hauptprogramm, die diese Werte zur Ansicht bringen soll ist auch mit dem bisherigen Wissen kein Problem. Also müssen wir erst einmal die Routine schreiben, die die Stellzeit zur Anzeige bringt.

```
Stell_Uhr:
    LDS   Reg_A, Anz_Mode
   CPI   Reg_A, 1                    ; Mode 1 aktiv-Stellzeit Uhrt anzeigen
   BRNE End_Stell_Uhr               ; nein, dann ende
   LDI XL, Low(E_Min_Stell)         ; Basisadresse Zeit
   LDI  XH, High(E_Min_Stell)
  RCALL Convert_Matrix              ; In den Matrixpuffer übertragen
RET
```

Diese Routine rufen wir einfach in jedem Zyklus auf, da die Änderung auch schneller erfolgen kann und kein festes Zeitraster hat..

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO              ; Eingänge lesen
   RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit          ; Bearbeiten „V“
    RCALL    IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
    RCALL    Taster_Event         ; Tasterereignisse bearbeiten
    RCALL    Set_Mode             ; Anzeige umschalten
    RCALL    Chk_Time_Flag        ; Zeitereignisse bearbeiten
    RCALL    Show_Uhr             ; Uhr anzeigen ?
```

```
   RCALL    Stell_Uhr          ; Stellzeit  Uhr anzeigen ?
   RCALL    Chk_Receive        ; Datenempfang prüfen
   RCALL    Is_First           ; erstes empfangenesByte prüfen und markieren
   RCALL    Is_Second          ; Datenempfang endgültig auswerten
   RCALL    Write_IO           ; und ausgeben
RJMP Loop
```

Ja gut soweit, aber nun sehen wir nur 0 oder wirres Zeug, Schön wär ja, wenn die aktuelle Zeit beim Umschalten erst einmal übernommen wird. Gut, das bekommen wir hin. Am besten, immer wenn der Programmmode von 0 nach 1 wechselt, dann übernehmen wir die aktuelle Zeit in die Stellzeit. Dazu erst die Routine zur Übergabe der Zeit.

```
Get_Stell_Zeit:
   LDI    XL, Low(E_Min)          ; Basisadresse Zeit
   LDI    XH, High(E_Min)
   LDI    YL, Low(E_Min_Stell)    ; Basisadresse Wertepuffer
   LDI    YH, High(E_Min_Stell)
   LDI    Reg_B, 4
Loop_Stell:
   LD     Reg_A, X+               ; hole Zeitwert und erhöhe Adresse
   ST     Y+, Reg_A               ; Speichere Zeitwert und erhöhe Adresse
   DEC    Reg_B
   BRNE   Loop_Stell              ; wenn Reg_B 0 ist, ist Zerobit gesetzt
RET
```

Nun die Änderung in der Subroutine Set_Mode

```
Set_Mode:
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b00000001
    BREQ    End_Set_Mode           ; Kein Eventbit, dann Ende
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b11111110      ; Eventbit löschen
    STS     Event_To_High, Reg_A   ; und zurückschreiben
    LDS     Reg_A, Anz_Mode
    INC     Reg_A
    STS     Anz_Mode, Reg_A
    CPI     Reg_A, 1               ; Wechsel ist von 0 nach 1
    BRNE    Chk_Max                ; Überspringe nächsten Befehl
    RCALL   Get_Stell_Zeit         ; kopiere aktuelle Zeit in den Zeitpuffer stellen
```

```
    LDS     Reg_A, Anz_Mode          ; Register A wieder mit Programm- Mode laden
    Chk_Max:                         ; Prüfe Grenzwert
    CPI     Reg_A, 4                 ; Grenzwert ist 3
    BRLO    End_Set_Mode
    CLR     Reg_A                            ; Beginnt wieder mit Mode 0
    STS     Anz_Mode, Reg_A
End_Set_Mode:
RET
```

Wie ihr seht, ist kein Problem, aber nun müssen wir auch die Stellzeit ändern. Und dafür haben wir auch wieder Ereignisbits in **Event_To_High**, die dafür zu nutzen sind.

Bit 1 ist nach dem Schaltplan für das Schalten von Ziffer zu Ziffer vorgesehen und Bit 2 und 3 für auf und ab. Also brauchen wir nun erst einmal einen Stellenzähler.

```
Pos_Cnt:   .Byte 1
```

Die Zählung funktioniert genau wie Mode, deshalb kann **Set_Mode** kopiert und angepasst werden. Die neue Routine nenne ich **Sel_Ziffer**. Die Funktion darf aber nur ausgeführt werden, wenn auch Mode 1, also Uhrzeit stellen aktiv ist.

```
Select_Ziffer:
    LDS     Reg_A, Anz_Mode
    CPI     Reg_A, 1                 ; Mode Prüfen
    BRNE    End_Select_Ziffer
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b00000010        ; Taster Ziffer auswählen
    BRLO    End_Select_Ziffer                ; Kein Eventbit, dann Ende
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b11111101        ; Eventbit löschen
    STS     Event_To_High, Reg_A     ; und zurückschreiben
    LDS     Reg_A, Pos_Cnt
    INC     Reg_A
    STS     Pos_Cnt, Reg_A
    CPI     Reg_A, 4                 ; Grenzwert ist 3
    BRLO    End_Select_Ziffer
    CLR     Reg_A                    ; Beginnt wieder mit Mode 0
    STS     Pos_Cnt, Reg_A
End_Select_Ziffer:
RET
```

Um die einzelnen Ziffern zu setzen schreiben wir eine Subroutine. Mittlerweile sollten wir ja schon ein wenig Übung haben und gut damit zurecht kommen..

```
;*****************************************************
;*                    Zahl einstellen                *
;*****************************************************
Set_Zahl_Up:
    LDS     Reg_A, Anz_Mode             ; Mode Uhrzeit stellen?
    CPI     Reg_A, 1
    BREQ    End_Zahl_Up                 ; wenn nicht, dann Ende
    LDS     Reg_A, Event_To_high        ; Ereignisbit laden
    ANDI    Reg_A, 0b00000100           ; Nur Bit 2 prüfen
    BREQ    End_Zahl_Up                 ; wenn Ergebnis 0, dann Ende
    LDS     Reg_A, Event_To_high        ; Ereignisbit laden
    ANDI    Reg_A, 0b11111011           ; Nur Bit 2 auf 0 setzen
    STS     Event_To_high, Reg_A        ; und Eventbit löschen
    LDI     XL, Low(E_Min_Stell)        ; Basisadresse der Stellzeit
    LDI     XH, High(E_Min_Stell)
    LDS     Reg_A, Pos_Cnt              ; aktuellen Stellenzeiger laden
    ADD     XL, Reg_A                   ; zur Basisadresse addieren
    CLR     Reg_A
    ADC     XH, Reg_A                   ; 16 Bit Addition
    LD      Reg_A, X                    ; adressierte Zahl laden
    INC     Reg_A                       ; erhöhen
    ST      X, Reg_A                    ; und speichern
    CPI     Reg_A, 10                   ; prüffen auf Überlauf
    BRLO    End_Zahl_Up
    CLR     Reg_A                       ; evtl. rücksetzen
    ST      X, Reg_A                    ; und speichern
End_Zahl_Up:
RET
```

In dieser Routine setzen wir bei jedem Tastendruck mit Taster 2 im Mode 1 den Wert der Stellzeit eins hoch, der durch den Stellenzeiger adressiert ist. Der Grenzwert ist hier 9. Danach beginnen wir mit dem Zähler wieder bei 0. Mehr nicht. Außer, das wir das Event des Tasters löschen. Aber das versteht sich ja von selbst.

Den Stellwert nach unten zählen ist ähnlich.

```
;*****************************************************
```

```
;*                    Zahl einstellen                    *
;*****************************************************
Set_Zahl_Down:
    LDS     Reg_A, Anz_Mode              ; Mode Uhrzeit stellen?
    CPI     Reg_A, 1
    BREQ    End_Zahl_Down                ; wenn nicht, dann Ende
    LDS     Reg_A, Event_To_high         ; Ereignisbit laden
    ANDI    Reg_A, 0b00001000            ; Nur Bit 3 prüfen
    BREQ    End_Zahl_Down                ; wenn Ergebnis 0, dann Ende
    LDS     Reg_A, Event_To_high         ; Ereignisbit laden
    ANDI    Reg_A, 0b11110111            ; Nur Bit 3 auf 0 setzen
    STS     Event_To_high, Reg_A         ; und Eventbit löschen
    LDI     XL, Low(E_Min_Stell)         ; Basisadresse der Stellzeit
    LDI     XH, High(E_Min_Stell)
    LDS     Reg_A, Pos_Cnt               ; aktuellen Stellenzeiger laden
    ADD     XL, Reg_A                    ; zur Basisadresse addieren
    CLR     Reg_A
    ADC     XH, Reg_A                    ; 16 Bit Addition
    LD      Reg_A, X                     ; adressierte Zahl laden
    DEC     Reg_A                        ; erhöhen
    ST      X, Reg_A                     ; und speichern
    CPI     Reg_A, 0                     ; prüffen auf Überlauf
    BRGE    End_Zahl_Down
    LDI     Reg_A, 9                     ; evtl. auf 9 setzen
    ST      X, Reg_A                     ; und speichern
End_Zahl_Down:
RET
```

Auch diese drei Subroutinen tragen wir einfach in die Main_Loop ein.

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO              ; Eingänge lesen
   RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit          ; Bearbeiten „V“
   RCALL    IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL    Set_Mode             ; Anzeige umschalten
   RCALL    Chk_Time_Flag        ; Zeitereignisse bearbeiten
   RCALL    Stell_Uhr            ; Stellzeit  Uhr anzeigen ?
   RCALL    Sel_Ziffer           ; Ziffer weiterschalten
   RCALL    Set_Zahl_Up          ; Zahl hochzählen
   RCALL    Set_Zahl_Down        ; Zahl runterzählen
```

```
    RCALL    Chk_Receive        ; Datenempfang prüfen
    RCALL    Is_First           ; erstes empfangenesByte prüfen und markieren
    RCALL    Is_Second          ; Datenempfang endgültig auswerten
    RCALL    Write_IO           ; und ausgeben
RJMP Loop
```

Bleibt noch die Übernahme der eingestellten Zeit. Das erledigen wir mit dem Wechsel des Mode von 1 nach 2.

```
Set_Mode:
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b00000001
    BREQ    End_Set_Mode          ; Kein Eventbit, dann Ende
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b11111110     ; Eventbit löschen
    STS     Event_To_High, Reg_A  ; und zurückschreiben
    LDS     Reg_A, Anz_Mode
    INC     Reg_A
    STS     Anz_Mode, Reg_A
    CPI     Reg_A, 1              ; Wechsel ist von 0 nach 1
    BRNE    Chk_Set_Time          ; Prüfe,, ob Zeit zu setzen ist
    RCALL   Get_Stell_Zeit        ; kopiere aktuelle Zeit in den Zeitpuffer stellen
    RJMP    End_Set_Mode          ; hier fertig
Chk_Set_Time:
    CPI     Reg_A, 2
    BRNE    Chk_Max               ; Grenzwertprüfung
    RCALL   Set_Stell_Zeit        ; kopiere gestellte Zeit in die Uhr
    RJMP    End_Set_Mode          ; her fertig
Chk_Max:                          ; Prüfe Grenzwert
    CPI     Reg_A, 4              ; Grenzwert ist 3
    BRLO    End_Set_Mode
    CLR     Reg_A                 ; Beginnt wieder mit Mode 0
    STS     Anz_Mode, Reg_A
End_Set_Mode:
RET
```

Und dazu selbstverständlich auch die Routine Set_Stell_Zeit.

```
Set_Stell_Zeit:
    LDI     XL, Low(E_Min)        ; Basisadresse Zeit
    LDI     XH, High(E_Min)
```

```
   LDI   YL, Low(E_Min_Stell)        ; Basisadresse Wertepuffer
   LDI   YH, High(E_Min_Stell)
   LDI   Reg_B, 4
Loop_Set_Time:
   LD      Reg_A,Y+                  ; hole Zeitwert und erhöhe Adresse
   ST      X+, Reg_A                 ; Speichere Zeitwert und erhöhe Adresse
   DEC   Reg_B
   BRNE  Loop_Set_Time               ; wenn Reg_B 0 ist, ist Zerobit gesetzt
RET
```

Es ist dieselbe Routine wie Get_Stellzeit, nur die beiden Adressregister sind getauscht. Die Uhr sollte nun funktionieren und sich stellen lassen. Um die Ganggenauigkeit zu überwachen, setzen wir den Aufruf der Uhr wieder in das Zeitereignis Sekunde und lassen sie einfach laufen.

### *2.15.1.1 Blinkende Ziffer*

Nun wäre es schön, zu erkennen, welche Zahl überhaupt zur Zeit selektiert ist. Die Idee, diese Anzeige lassen wir blinken. Wie das Funktionieren soll? Ganz einfach mit einer Maskierung. Dazu nehmen wir zwei Variablen. Eine für die Maske und eine für den Blinktakt.

```
Blink_Maske:   .Byte 1
Blinker:       .Byte 1
```

Der Blinker und die Blink_Maske werden nun in der Initialisierung mit 255 oder binär mit 11111111 besetzt.

```
LDI     Reg_A, 0b11111111
STS     Blink_Maske, Reg_A
STS     Blinker, Reg_A
```

Im Zeitevent 1 Sekunde werden nun die Bits in der Variablen Blinker alle mit dem Befehl COM invertiert. Somit wechseln diese Bits im Sekundentakt ihre Lage.

```
;*********************** Zeitereignis  mSek ***********************
;*        Zeitroutinen im Zeitraster Sekunde                    *
;*************************************************************************
Event_sekunde:                  ; Zeitevent bearbeiten
  LDS    Reg_A, Time_Flag
  ANDI   Reg_A,0b11110111        ; Zeitflag quittieren
  STS    Time_Flag, Reg_A
 ;************ Zeitjobs ************
   LDS   Reg_A, Blinker
   COM   REG_A
   STS   Blinker, Reg_A
   RCALL Prg_Uhr
   RCALL Show_Uhr
End_Event_Sekunde:
RET
;-----------------------------------------------------------------
```

Nun muss nur noch ein Vergleich zwischen dem Zähler des Multiplexer und dem Zähler für die Selektion zur Korrktur stattfinden. Da ich die

Routine des Multiplexers nicht noch mit dieser Aufgabe versehen möchte, baue ich das als Aufruf für eine Subroutine in die Programmschleife ein.

```
RCALL    Chk_Blinker
```

Und dazu natürlich die Subroutine

```
Chk_Blinker:
    LDS      Reg_A, Anz_Mode       ; Anzeigemodus
    CPI      Reg_A, 1              ; Uhr stellen
    BRNE     Set_Mask_1            ; Wenn nicht Mode 1 dann Maaske auf 1
    LDS      Reg_A, Ziffer_Cnt     ; Stellenzähler Multiplexer
    LDS      Reg_B, Pos_Cnt        ; Positionszähler Ziffer
    CP       Reg_A, Reg_B
    BRNE     Set_Mask_1            ; Wenn nicht gleich,
    LDS      Reg_A, Blinker
    RJMP     End_Chk_Blinker       ; Maskenbits = Blinkerbits
 Set_Mask_1::
    LDI      Reg_A, 0b11111111     ;  alle Maskenbits auf 1
 End_Chk_Blinker:
    STS      Blink_Maske, Reg_A
RET
```

Und dazu der Aufruf in der MainLoop:

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO              ; Eingänge lesen
   RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit          ; Bearbeiten „V“
   RCALL    IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL    Set_Mode             ; Anzeige umschalten
   RCALL    Chk_Blinker          ; Blinkerbit bilden
   RCALL    Chk_Time_Flag        ; Zeitereignisse bearbeiten
   RCALL    Show_Uhr             ; Uhr anzeigen ?
   RCALL    Stell_Uhr            ; Stellzeit  Uhr anzeigen ?
   RCALL    Sel_Ziffer           ; Ziffer weiterschalten
   RCALL    Set_Zahl_Up          ; Zahl hochzählen
   RCALL    Set_Zahl_Down        ; Zahl runterzählen
   RCALL    Chk_Receive          ; Datenempfang prüfen
   RCALL    Is_First             ; erstes empfangenesByte prüfen und markieren
```

```
  RCALL    Is_Second          ; Datenempfang endgültig auswerten
  RCALL    Write_IO           ; und ausgeben
RJMP Loop
```

Nun muss diese Aufbereitung zum Blinken der Anzeige mit in den Multiplexer eingebaut werden. Dazu setze ich eine zusätzliche Und-Verknüpfung mit dem Maskenbyte **Blink_Mask**. Es ist entweder komplett 1 oder abhängig vom Blinker.

```
;*************************************************************************
;*    Ziffer auf Anzeige ein- und ausblenden (Multiplexer)  *
;*************************************************************************
Sel_Ziffer:
  LDS     Reg_A,Ziffer_Cnt          ; Stellenzähler laden
  LDS     Reg_B,Multiplx            ; Schieberegister laden
  In      Reg_C,PortC               ; Port erst  einlesen
  COM     Reg_C                     ; Bits drehen, Ziffer ist 0 aktiv
  ANDI    Reg_C,0b11100001          ; hier steht später ein anderes Register
  COM     Reg_C                     ; Bits drehen, Ziffer ist 0 aktiv
  Out     PortC,Reg_C               ; alle Ziffern aus
  LDI     XL, Low(Code_Puffer)      ; Matrixpuffer Basisadresse laden
  LDI     XH, High(Code_Puffer)
  ADD     XL, Reg_A                 ; Stellenzähler dazu addieren
  ADC     XH, Zero                  ; Übertrag übernehmen (16 Bit Addition)
  LD      Reg_D, X                  ; Matrix aktuelle Zahl holen
 ;********************** Einbau der Blinkmaske *************************
 LD       Reg_E. Blink_Maske        ;Einbau Blinkmaske
  AND     Reg_D, Reg_E
  MOV     Ablage, Reg_D             ; Zwischenspeichern
  IN      Reg_E, PortB
  ANDI    Reg_E, 0b11000000         ; Port B ausmaskieren
  ANDI    Reg_D, 0b00111111         ; Bit 6 und 7 ausblenden
  OR      Reg_D, Reg_E              ; zusammenführen
  Out     PortB, Reg_D              ; und ausgeben
  In      Reg_E, PortD              ; Port D lesen
  MOV     Reg_D, Ablage             ; Matrix zurückholen
  ANDI    Reg_D,  0b01000000        ; letztes Bit (Bit 6) ausmaskieren
  LSL     Reg_D                     ; nach links schieben
```

```
  ANDI     Reg_E, 0b01111111          ; Port D Bit 7 ausmaskieren
  OR       Reg_D, Reg_E               ; Register zusammenführen
  Out      PortD, Reg_D               ; und ausgeben
  COM      Reg_C                      ; Stelle selektieren
  ANDI     Reg_B,0b00011110
  OR       Reg_C,Reg_B                ; aktuelle Ziffer einschalten
  COM      Reg_C                      ; für Ausgabe wieder drehen
  Out      PortC,Reg_C                ; und Ausgabe
;********************** Anpassung Ziffernzähler **************************
  LDS    Reg_A,Ziffer_Cnt             ; Stellenzähler laden
  LDS    Reg_B,Multiplx               ; Schieberegister laden
  LSL    Reg_B                        ; Select-Bit schieben
  INC    Reg_A                        ; Stellenzähler hochsetzen
  CPI    Reg_A,4                      ; Grenze prüfen
  BRLO  Next_Ziffer
  CLR    Reg_A                        ; evtl. von vorn anfangen
  LDI    Reg_B, 0b00000010
Next_Ziffer:
  STS    Ziffer_Cnt, Reg_A            ; Stellenzähler abspeichern
  STS    Multiplx, Reg_B              ; Schieberegister abspeichern
RET
;-----------------------------------------------------------
```

Die Routine musste auch etwas umgebaut werden, da die falsche Ziffer geblinkt hat. So darf der Ziffernzähler erst zum Schluss weiter geschaltet werden. Deshalb ist der obere Teil nach unten an den Schluss kopiert. Lediglich die beiden Ladebefehle für Ziffern_Cnt und Multiplx müssen zusätzlich am Anfang stehen bleiben. Dadurch und durch den Blinker bekommt diese Routine 7 Takte mehr für die Ausführung.

### *2.15.2 Alarmzeit Weckfunktion*

Braucht man eine Alarmzeit, so soll diese während des Einstellvorganges ja auch angezeigt werden. Definieren wir diese Variablen auch gleich mit.

```
E_Min_Alarm: .Byte 1
Z_Min_Alarm: .Byte 1
E_Std_Alarm: .Byte 1
Z_Std_Alarm: .Byte 1
```

Hier schalten wir den Mode 2 ein.

Dieser Vorgang ist dem Stellen der Uhr sehr ähnlich. Auch hier funktioniert Taster 2 um die Ziffer auszuwählen und Taster 2 und 4 setzen den Wert hoch oder runter. Mode 2 schickt die Werte aus den dazu verwendeten Variablen an den Wertepuffer zur Ausgabe. Zuerst einmal die Routine, die diese Werte in den Anzeigepuffer kopiert.

```
;*************************************************************
;*                    Alarmzeit anzeigen                     *
;*************************************************************
Show_Alarmzeit:
   LDS   Reg_A, Anz_Mode
   CPI   Reg_A, 2                          ; Mode 2 aktiv-Stellzeit Uhrt anzeigen
   BRNE End_Show_Alarmzeit                 ; nein, dann ende
   LDI  XL, Low(E_Min_Alarm)               ; Basisadresse Zeit
   LDI  XH, High(E_Min_Alarm)
   RCALL  Convert_Matrix                   ; Weckzeit zur Ausgabe leiten
End_Show_Alarmzeit:
RET
;-------------------------------------------------------------
```

Der Aufruf dieser Routine, ja ihr habt es erraten, geschieht natürlich ebenfalls in der Main_Loop. Es ist ja nur die Anzeige, der Stellvorgang muss aber auf die Variablen E_Min_Alarm usw. erfolgen. Diese sind im zum Vergleich wichtig.

Nun ist noch das Stellen der Ziffern erforderlich. Klar, man könnte die bereits erstellten Routinen erweitern, aber das macht den Code unleserlich. Besser ist eine Kopie der beiden Setzroutinen mit einer entsprechenden Anpassung auf Mode und Basisadresse.

```
;*****************************************************
;*                    Zahl einstellen                *
;*****************************************************
Set_Alarm_Up:
    LDS     Reg_A, Anz_Mode              ; Mode Wecker stellen?
    CPI     Reg_A, 2
    BRNE    End_Alarm_Up                 ; wenn nicht, dann Ende
    LDS     Reg_A, Event_To_high         ; Ereignisbit laden
    ANDI    Reg_A, 0b00000100            ; Nur Bit 2 prüfen
    BREQ    End_Alarm_Up                 ; wenn Ergebnis 0, dann Ende
    LDS     Reg_A, Event_To_high         ; Ereignisbit laden
    ANDI    Reg_A, 0b11111011            ; Nur Bit 2 auf 0 setzen
    STS     Event_To_high, Reg_A         ; und Eventbit löschen
    LDI     XL, Low(E_Min_Alarm)         ; Basisadresse der Stellzeit
    LDI     XH, High(E_Min_Alarm)
    LDS     Reg_A, Pos_Cnt               ; aktuellen Stellenzeiger laden
    ADD     XL, Reg_A                    ; zur Basisadresse addieren
    CLR     Reg_A
    ADC     XH, Reg_A                    ; 16 Bit Addition
    LD      Reg_A, X                     ; adressierte Zahl laden
    INC     Reg_A                        ; erhöhen
    CPI     Reg_A, 10                    ; prüffen auf Überlauf
    BRLO    End_Alarm_Up
    CLR     Reg_A                        ; evtl. rücksetzen
End_Alarm_Up:
    ST      X, Reg_A                     ; und speichern
RET
```

Auf einen Stellvorgang nach unten verzichte ich, weil ich den Taster 4 brauche, um den Alarm ein- oder abzuschalten. Doch wie signalisiere ich das? Auch hier hilft das Blinken der Anzeige. Ist der Alarm ein, lass ich alle Ziffern blinken, ist der Alarm aus, dann blinkt nur die aktuelle Ziffer, die grad gestellt werden kann. Gute Idee, nun noch umsetzen. Erst einmal brauchen wir ein Bit Alarm ein oder aus. Dafür opfern wir eine Variable und bennenen sie Prg_Ctrl. Hier können wir uns dann solche Programm-Status-Bits ablegen.

```
Prg_Ctrl:     .Byte 1    ; Byte Programmstatusbits
                         ; Bit 0  = Alarm ein = 1, aus = 0
                         ; Bit 1  = Alarm hat ausgelöst = 1
```

Natürlich ist noch Platz für weitere Bits, aber im Moment reicht die Beschreibung. Zuerst die kleine Routine Alarm_on_Off, geschaltet mit Taster 4.

```
;*****************************************************
;*              Alarm ein oder aus                   *
;*****************************************************
Alarm_On_Off:
    LDS     Reg_A, Anz_Mode             ; Mode Wecker stellen?
    CPI     Reg_A, 2
    BRNE    End_Alarm_onoff             ; wenn nicht, dann Ende
    LDS     Reg_A, Event_To_high        ; Ereignisbit laden
    ANDI    Reg_A, 0b00001000           ; Nur Bit 3 prüfen
    BREQ    End_Alarm_onoff             ; wenn Ergebnis 0, dann Ende
    LDS     Reg_A, Event_To_high        ; Ereignisbit laden
    ANDI    Reg_A, 0b11110111           ; Nur Bit 3 auf 0 setzen
    STS     Event_To_high, Reg_A        ; und Eventbit löschen
    LDS     Reg_A, Prg_Ctrl             ; Statusbits Programm
    LDI     Reg_B, 0b00000001           ; Statusbit 0 selektieren
    EOR     Reg_A, Reg_B                ; Bit 0 invertieren
    ANDI    Reg_A, 0b11111101           ; evtl. Wecker aus
    STS     Prg_Ctrl, Reg_A             ; und zurückschreiben
End_Alarm_onoff:
RET
```

Diese beiden Routinen **Set_Alarm_Up** und **Alarm_On_Off** werden noch der Programmschleife hinzugefügt. Jetzt müssen die Funktion noch in Chk_Blinker eingebunden werden. Dazu betrachten wir die Subroutine und überlegen, was zu ändern ist. Zur Zeit wird nur Mode 1 abgefragt. Ok, dann erweitern wir auf Mode 2, denn auch hier möchte ich die Stelle durch blinken markieren, die gerade gestellt wird.

```
Chk_Blinker:
    LDS     Reg_A, Anz_Mode     ; Anzeigemodus
    CPI     Reg_A, 1            ; Uhr stellen
    BREQ    Set_Blink_Mode2     ; wenn ja, dann Blinkmode setzen
    LDS     Reg_A, Anz_Mode     ; Anzeigemodus
    CPI     Reg_A, 2            ; Uhr stellen
    BRNE    Set_Mask_1          ; Wenn nicht Mode 2 dann Maske auf 1
    LDS     Reg_A, Prg_Ctrl     ; Wenn Bit gesetzt, dann nicht Stelle abfragent
    ANDI Reg_A, 0b00000001
    BRNE    Set_BlinkMaske
Set_Blink_Mode2:                ; selektierte Stelle= Ausgabestelle
    LDS     Reg_A, Ziffer_Cnt   ; Stellenzähler Multiplexer
    LDS     Reg_B, Pos_Cnt      ; Positionszähler Ziffer
    CP      Reg_A, Reg_B
```

```
    BRNE   Set_Mask_1              ; Wenn nicht gleich,
Set_BlinkMaske:
    LDS      Reg_A, Blinker
    RJMP    End_Chk_Blinker        ; Maskenbits = Blinkerbits
 Set_Mask_1:
    LDI       Reg_A, 0b11111111        ;  alle Maskenbits auf 1
 End_Chk_Blinker:
    STS      Blink_Maske, Reg_A
RET
```

Obwohl gar nicht soviel geändert wurde wird euch das Ergebnis überraschen. Aber nun ist ja noch die Auslösung des Alarms an der Reihe. Dies wird wieder eine eigenständige Subroutine, die in der Main_Loop aufgerufen wird.

```
Chk_Alarm:
    LDS      Reg_A, Prg_Ctrl         ; Programmstatusbits laden
    ANDI     Reg_A, 0b00000001   ; Alarm eingeschaltet?
    BREQ    End_Chk_Alarm         ; wenn nicht, dann Ende
    LDS      Reg_A, E_Min            ; Minuten 1
    LDS      Reg_B, E_Min_Alarm  ; Alarmzeit Minuten 1
    CP       Reg_A, Reg_B
    BRNE    Reset_Zeitbereich       ; wenn nicht gleich, dann Ende
    LDS      Reg_A, Z_Min            ; Minuten 1
    LDS      Reg_B, Z_Min_Alarm     ; Alarmzeit Minuten 1
    CP       Reg_A, Reg_B
    BRNE    End_Chk_Alarm         ; wenn nicht gleich, dann Ende
    LDS      Reg_A, E_Std            ; Minuten 1
    LDS      Reg_B, E_Std_Alarm     ; Alarmzeit Minuten 1
    CP       Reg_A, Reg_B
    BRNE     End_Chk_Alarm         ; wenn nicht gleich, dann Ende
    LDS      Reg_A, Z_Std            ; Minuten 1
    LDS      Reg_B, Z_Std_Alarm     ; Alarmzeit Minuten 1
    CP       Reg_A, Reg_B
    BRNE    End_Chk_Alarm         ; wenn nicht gleich, dann Ende
    LDS      Reg_A, Prg_Ctrl         ; Programmstatusbits laden
    ANDI     Reg_A, 0b00000100    ;  Zeitbereichsbit prüfen
    BRNE     End_Chk_Alarm         ; bereits gesetzt, keine neue Aktivierung
    LDS      Reg_A, Prg_Ctrl         ; Programmstatusbits laden
    ORI      Reg_A, 0b00000110    ; Zeitbereich und  Alarm aktivieren
    STS      Prg_Ctrl, Reg_A
Reset_Zeitbereich:
    LDS      Reg_A, Prg_Ctrl         ; Programmstatusbits laden
    ANDI     Reg_A, 0b11111011    ; Zeitbereich und  Alarm aktivieren
```

```
    STS       Prg_Ctrl, Reg_A
End_Chk_Alarm :
   LDS      Reg_A,Out_Ctrl          ; und dann  direkt auf die Ausgabe leiten
   LDS      Reg_B, Prg_Ctrl
   ANDI     Reg_A, 0b11011111       ; Relaisausgang PC5 auf 0
   ANDI     Reg_B,0b00000010        ; nur Alarmbit lassen
   SWAP     Reg_B                   ; Nibbletausch
   OR       Reg_A, Reg_B            ; Register zusammenführen
   STS      Out_Ctrl, Reg_A         ; Alarmbit steuert das Relais
RET
```

Diesen Alarm hören wir nun bis zum Weltuntergang. Ah ja, toll. Aber wie beende ich nun diesen nervigen Alarm? Auch diese Frage hat mich erst einmal gehindert, das Relais direkt einzuschalten. Unser Wecker soll ja Einschalten und nach Quittierung abschalten. Die zu vergleichende Zeit ist aber 1 Minute gültig. Wird das Alarmbit dann nicht gleich wieder gesetzt? Klar, wenn nicht darauf geachtet wird, dass eben nur eine Flanke ausgewertet wird. Mit einer Flankenauswertung haben wir schon Taster behandelt. Damit diese Flanke erkannt wird, ist nicht nur ein Alarmbit notwendig, sondern auch ein weiteres Bit, welches anzeigt, das die Uhr im Fenster der eingestellten Alarmzeit ist. Und nur, wenn beide Bits 0 sind, wird das Signalbit für den Alarm gesetzt. Das kann natürlich nur einmal im Alarmfenster passieren. Ist der Vergleich im Minutenbereich nicht mehr gegeben, ist das Alarmfenster verlassen und das Bit kann gelöscht werden. Vielleicht sollte auch erst eine LED statt eines Relais angeschlossen werden, um den Alarm zu testen.

Ergänzen wir auch gleich im Kommentarfeld der Variablen Prg_Ctrl die Verwendung des Bits.

```
Prg_Ctrl:     .Byte 1    ; Byte Programmstatusbits
                         ; Bit 0  = Alarm ein = 1, aus = 0
                         ; Bit 1  = Alarm hat ausgelöst = 1
                         ; Bit 2  = Zeitfenster Alarm
```

Eine andere Maßnahme ist auch denkbar und so hab ich die Zuweisung an den Ausgang erst einmal auskommentiert. Wird der auskommentierte

Teil aktiviert, steuert das Bit 1 aus Prg_Ctrl das Bit 5 von Out_Ctrl. Nun zum Löschvorgang. Vielleicht Taster 4? Er schaltet den Wecker scharf, also kann er ihn auch abschalten. Deshalb wird vor der Zuweisung an Prg_Ctrl in der Routine Alarm_On_Off Bit 1 auf jeden Fall zurückgesetzt. Bleibt noch die normale Uhr, denn im Mode Uhr stellen und Stoppuhr ist der taster 4 anderweitig belegt.

```
;*****************************************************
;*                    Wecker  aus                    *
;*****************************************************
Wecker_Aus:
    LDS     Reg_A, Anz_Mode                 ; Mode Wecker stellen?
    CPI     Reg_A, 0
    BREQ    End_Wecker_Aus                  ; wenn nicht, dann Ende
    LDS     Reg_A, Event_To_high            ; Ereignisbit laden
    ANDI    Reg_A, 0b00001000               ; Nur Bit 3 prüfen
    BREQ    End_Wecker_Aus                  ; wenn Ergebnis 0, dann Ende
    LDS     Reg_A, Event_To_high            ; Ereignisbit laden
    ANDI    Reg_A, 0b11110111               ; Nur Bit 3 auf 0 setzen
    STS     Event_To_high, Reg_A            ; und Eventbit löschen
    LDS     Reg_B, Prg_Ctrl                 ; Statusbits Programm
    ANDI    Reg_B, 0b11111101               ; Weckerbit löschen
    STS     Prg_Ctrl, Reg_B                 ; und zurückschreiben
End_Wecker_Aus:
RET
```

In beiden Fällen bleibt das Bereichsbit außen vor. Dieses wird ausschließlich nur außerhalb des Zeitfensters gelöscht und im Zeitfenster zusammen mit dem Alarm gesetzt. Bleibt noch in der Ziffernauswahl zum Einstellen des Alarms Mode 2 mit einzubinden. So wird diese Routine nun um die Abfrage Mode 2 erweitert.

```
Selec_Ziffer:
    LDS     Reg_A, Anz_Mode
    CPI     Reg_A, 1                        ; Mode 1  Prüfen
    BREQ    Set_Select
    CPI     Reg_A, 2                        ; Mode 2 Prüfen
    BREQ    Set_Select
    RJMP    End_Select_Ziffer               ; nicht Mode 1 oder 2
Set_Select:
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b00000010               ; Taster Ziffer auswählen
    BREQ    End_Select_Ziffer               ; Kein Eventbit, dann Ende
    LDS     Reg_A, Event_To_High
```

```
    ANDI    Reg_A, 0b11111101          ; Eventbit löschen
    STS     Event_To_High, Reg_A       ; und zurückschreiben
    LDS     Reg_A, Pos_Cnt
    INC     Reg_A
    STS     Pos_Cnt, Reg_A
    CPI     Reg_A, 4                   ; Grenzwert ist 3
    BRLO    End_Select_Ziffer
    CLR     Reg_A                      ; Beginnt wieder mit Mode 0
    STS     Pos_Cnt, Reg_A
End_Select_Ziffer:
RET
```

Und selbstverständlich gehören die Aufrufe zur Alarmzeit auch in die Programmschleife

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO            ; Eingänge lesen
   RCAll    IO_Debounce        ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit        ; Bearbeiten „V“
   RCALL    IO_Event_To_1      ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL    Set_Mode           ; Anzeige umschalten
   RCALL    Chk_Blinker        ; Blinkerbit bilden
   RCALL    Chk_Time_Flag      ; Zeitereignisse bearbeiten
   RCALL    Show_Uhr           ; Uhr anzeigen ?
   RCALL    Stell_Uhr          ; Stellzeit  Uhr anzeigen ?
   RCALL    Sel_Ziffer         ; Ziffer weiterschalten
   RCALL    Set_Zahl_Up        ; Zahl hochzählen
   RCALL    Set_Zahl_Down      ; Zahl runterzählen
   RCALL    Set_Alarm          ; Alarmzeit stellen
   RCALL    Alarm_On_Off       ; Alarm scharfschalten
   RCALL    Wecker_Aus         ; Wecker ausschalten
   RCALL    Chk_Receive        ; Datenempfang prüfen
   RCALL    Is_First           ; erstes empfangenesByte prüfen und markieren
   RCALL    Is_Second          ; Datenempfang endgültig auswerten
   RCALL    Write_IO           ; und ausgeben
RJMP Loop
```

Nun ist auch der Wecker mit all seinen Funktionen komplett.

### *2.15.3 Stoppuhr*

Eine Stoppuhr ist auch immer wieder ein nützlicher Gegenstand. Und mit unserer Anzeige sind wir in der Lage, Zeiten bis 9999 mSekunden zu erfassen oder anders 9,999 Sekunden. Natürlich ist auch ein Verzicht auf die Genauigkeit der Zehntelsekunde möglich und mit einer hundertstel Sekunde ist der Zeitbereich bereits bei 99 Sekunden und 99 Hundertstel. Wie ist vorzugehen? Eigentlich kein Problem. Die Umschaltung von Mode 3 auf Mode 4 setzt die Zähler zurück. Ein weiteren Rücksetzen im Mode Stoppuhr übernimmt Taster 1. Taster 3 übernimmt die Start-Stop-Funktion. Aber zuerst einmal die erforderlichen Variablen, die natürlich auch im Mode 4 auf den Anzeigepuffer geschaltet werden.

```
Stop_mSek:          .Byte 1
Stop_mSek_10:       .Byte 1
Stop_mSek_100:      .Byte 1
Stop_mSek_1000:     .Byte 1
```

Beginnen wir mit dem Rücksetzen der Stoppuhr.

```
Reset_Stoppzeit:
    LDS     Reg_A, Anz_Mode
    CPI     Reg_A, 3                    ; Mode 3  Stoppuhr
    BRNE    End_Reset_Stoppzeit
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b00000010           ; Taster 1 wird Reset
    BREQ    End_Reset_Stoppzeit         ; Kein Eventbit, dann Ende
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b11111101           ; Eventbit löschen
    STS     Event_To_High, Reg_A        ; und zurückschreiben
    STS     Stop_mSek,Zero              ; und in die Zeitzähler eintragen
    STS     Stop_mSek_10. Zero
    STS     Stop_mSek_100, Zero
    STS     Stop_mSek_1000, Zero
    LDS     Reg_A, Prg_Ctrl             ; Aktivierungsbit in Prg_Ctrl
    ANDI    Reg_A, 0b11101111           ; löschen
   STS      Prg_Ctrl, Reg_A
End_Reset_Stoppzeit :
RET
```

Nun steht hier etwas von Aktivierungsbit. Das ist ein Bit in Prg_Ctrl, das den Zeitzähler freigibt. Schreiben wir in der Deklaration von der Variablen Prg_Ctrl einen weiteren Kommentar.

```
Prg_Ctrl:      .Byte 1    ; Byte Programmstatusbits
                          ; Bit 0  =  Alarm ein = 1, aus = 0
                          ; Bit 1  =  Alarm hat ausgelöst = 1
                          ; Bit 2  = Zeitfenster Alarm
                          ; Bit 3 nicht benutzt
                          ; Bit 4  = Stopuhr läuft = 1
```

Damit bei der Bedienung nicht versehentlich Reset betätigt wird, legen wir das Starten und Stoppen auf Taster 4. Eine ähnliche Funktion haben wir mit dem Bit 0 bei unserer Alarmzeit. Deshalb können wir diesmal auch mit wenigen Änderungen eine Kopie von Alarm_On_Of verwenden

```
;*****************************************************
;*              Alarm ein oder aus                   *
;*****************************************************
Stopzeit_On_Off:
    LDS     Reg_A, Anz_Mode              ; Mode Wecker stellen?
    CPI     Reg_A, 3
    BRNE    End_Stopzeit_onoff           ; wenn nicht, dann Ende
    LDS     Reg_A, Event_To_high         ; Ereignisbit laden
    ANDI    Reg_A, 0b00001000            ; Nur Bit 3 prüfen
    BREQ    End_Stopzeit_onoff           ; wenn Ergebnis 0, dann Ende
    LDS     Reg_A, Event_To_high         ; Ereignisbit laden
    ANDI    Reg_A, 0b11110111            ; Nur Bit 3 auf 0 setzen
    STS     Event_To_high, Reg_A         ; und Eventbit löschen
    LDS     Reg_A, Prg_Ctrl              ; Statusbits Programm
    LDI     Reg_B, 0b00010000            ; Statusbit 0 selektieren
    EOR     Reg_A, Reg_B                 ; Bit 0 invertieren
    STS     Prg_Ctrl, Reg_A              ; und zurückschreiben
End_Stopzeit_onoff :
RET
```

Natürlich soll auch die Stoppzeit angezeigt werden. Also müssen wir die Werte in den Ausgabepuffer übertragen.

```
;*****************************************************
;*            Anzeige der Stoppuhr                   *
;*****************************************************
Set_Stoppuhr:
    LDS     Reg_A, Anz_Mode              ; Mode Stoppuhr?
    CPI     Reg_A, 3
```

```
    BRNE   End_Stoppuhr                    ; wenn nicht, dann Ende
    LDI    XL, Low(Stop_mSek)                ; Basisadresse der StoppZeit
    LDI    XH, High(Stop_mSek)
    RCALL  Convert_Matrix                  ; Stoppuhr zur Ausgabe leiten
End_Stoppuhr:
RET
```

Diese drei Routinen werden nun in der Main_Loop untergebracht.

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL   Read_IO              ; Eingänge lesen
   RCAll   IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL   Set_LED_Bit          ; Bearbeiten „V“
   RCALL   IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL   Set_Mode             ; Anzeige umschalten
   RCALL   Chk_Blinker          ; Blinkerbit bilden
   RCALL   Chk_Time_Flag        ; Zeitereignisse bearbeiten
   RCALL   Show_Uhr             ; Uhr anzeigen ?
   RCALL   Stell_Uhr            ; Stellzeit  Uhr anzeigen ?
   RCALL   Sel_Ziffer           ; Ziffer weiterschalten
   RCALL   Set_Zahl_Up          ; Zahl hochzählen
   RCALL   Set_Zahl_Down        ; Zahl runterzählen
   RCALL   Set_Alarm            ; Alarmzeit stellen
   RCALL   Alarm_On_Off         ; Alarm scharfschalten
   RCALL   Wecker_Aus           ; Wecker ausschalten
   RCALL   Stopzeit_On_Off      ; Stoppuhr Start – Stopp
   RCALL   Reset_Stoppzeit
   RCALL   Show_Stoppuhr        ; Anzeige Stoppuhr
   RCALL   Chk_Receive          ; Datenempfang prüfen
   RCALL   Is_First             ; erstes empfangenesByte prüfen und markieren
   RCALL   Is_Second            ; Datenempfang endgültig auswerten
   RCALL   Write_IO             ; und ausgeben
RJMP Loop
```

Nun muss die Stoppuhr nur noch laufen, wenn das Bit 4 in Prg_Ctrl gesetzt ist.

```
;*****************************************************
;*                Stoppuhr  aktiv                *
```

```
;*****************************************************
Stopp_Zeit:
    LDS     Reg_A, Anz_Mode             ; Mode Wecker stellen?
    CPI     Reg_A, 3
    BRNE    End_Stopp_Zeit              ; wenn nicht, dann Ende
    LDS     Reg_A, Prg_Ctrl             ; Statusbits Programm
    ANDI    Reg_A, 0b00010000           ; Statusbit 4 prüfen
    BREQ    End_Stopp_Zeit              ; wenn nicht, dann Ende
    LDS     Reg_A, Stop_mSek            ; Ab hier in zehnerschriten zählen
    INC     Reg_A
    STS     Stop_mSek , Reg_A
    CPI     Reg_A, 10
    BRLO    End_Stopp_Zeit
    CLR     Reg_A
    STS     Stop_mSek, Reg_A                ;
    LDS     Reg_A,  Stop_mSek_10
    INC     Reg_A
    STS     Stop_mSek_10, Reg_A
    CPI     Reg_A, 10
    BRLO    End_Stopp_Zeit
    CLR     Reg_A
    STS     Stop_mSek_10, Reg_A
    LDS     Reg_A,   Stop_mSek_100
    INC     Reg_A
    STS     Stop_mSek_100, Reg_A
    CPI     Reg_A, 10
    BRLO    End_Stopp_Zeit
    CLR     Reg_A
    STS     Stop_mSek_100, Reg_A
    LDS     Reg_A, Stop_mSek_1000
    INC     Reg_A
    CPI     Reg_A, 10
    BREQ    End_Stopp_Zeit              ; höchste Zahl 9999
    STS     Stop_mSek_1000, Reg_A
End_Stopp_Zeit:
RET
```

Den Aufruf der Stoppuhr bringen wir im Zeitereignis Event_1mSek unter. Ist ein größerer Messbereich mit einer geringeren Auflösung erwünscht, dann kommt dieser Aufruf halt einfach in Event_10mSek.

```
Event_1msek:
```

```
;------ Bereich der Zeiterreignisbearbeitung –1 mSek---------
    LDS      Reg_B, Debounce_Cnt
    CPI      Reg_B, 0      ; Vergleich ist wichtig, da eine Ladeanweisung kein Zerobit setzt.
    BREQ     Quit_Event_1mSek                ; Wenn zähler auf 0, keine weitere Aktion
    Dec      Reg_B
    STS      Debounce_Cnt, Reg_B
    BRNE     Quit_Event_1mSek
    LDS      Reg_A, Akt_In
    STS      New_In, Reg_A
 ; Zähler hat bis 0 gezählt. Der neugelesene Wert ist gültig und wird übernommen
;--------------------- Stopuhr aufrufen ------------------------
    RCALL    Stopp_Zeit
;--------------------- Zeitflag quittieren ------------------------
Quit_Event_1mSek:
    LDS      Reg_A, Time_Flags                ; Variable mit Zeitflags holen
    ANDI     Reg_A, 0b11111110                ; Flag für 1 mSek löschen
    STS      Time_Flags, Reg_A                ; Variable zurückschreiben
End_Event_1mSek:
RET
```

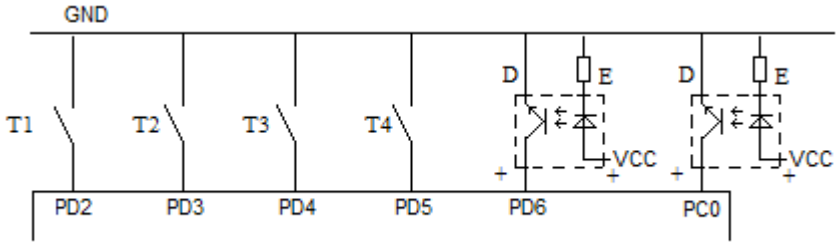
Verlassen wir nun das Kapitel mit der Überschrift Uhr und wenden uns anderen Aufgaben zu.

## 2.16 Rundenzähler

Möchten wir bspw. bei einer Slotcar-Rennbahn Runden zählen, gut, 2 x 2 Anzeigen, jeweils von 0 bis 99 Runden. Dazu führen wir einen weiteren Mode ein. Einfach in der Routine Set_Mode den Grenzwert von 4 auf 5 setzen. #Anzeige mit verschiedenen Werten beschalten

```
Lap_Cnt_A_0:        .Byte 1
Lap_Cnt_A_1:        .Byte 1

Lap_Cnt_B_0:         .Byte 1
Lap_Cnt_B_1:        . Byte 1
```

Die Belegung der Taster ist immer noch Taster 1 Umschaltung Mode. Taster 2 nehmen wir, um die Zählerwerte zurück zu setzen. Damit könnten wir Gabellichtschranken statt dem Taster 3 und 4 anschließen. Nun möchte ich aber alle Funktionalitäten erhalten und es ist ja auch nur eine zweite Gabellichtschranke erforderlich. Im Schaltplan war ja bereits eine Gabellichtschranke verbaut. Dazu hier ein kleiner Ausschnitt.




*Schaltplan Eingänge*

Nehmen wir für die zweite Gabellichtschranke erst einmal PC0. Das Potentiometer kommt später. Den Portpin hatten wir auch auf Eingang geschaltet und auch den Pull_Up Widerstand aktiviert.

Bei der Erfassung der Signale durch die Gabellichtschranken benutzen wir wieder Flanken. Die Bits 0 bis 4 sind ja bereits vergeben, aber das macht nichts. In der Routine Read_IO ist PD6 bereits erfasst, ist also nur noch PC0 einzulesen und der Variablen New_In hinzu zufügen. Der Kommentarbereich der Variablen wird entsprechend ergänzt. Zuerst aber ein Blick auf Read_IO.

```
;---------- Lesen aller Eingänge und Signalanpassung ------------
;*************************************************
;*                          Port D                          *
;*        Bit 0 und 1    serielle Verbindung                *
;*        Bit 2 bis 5    Eingänge Taster                    *
;*        Bit 6          Eingang erste Gabellichtschranke   *
;*        Bit 7          Ausgang Anzeige                    *
;*                          Port C                          *
; *       Bit 0          Eingang zweite Gabellichtschranke  *
;*************************************************
Read_IO:                        ; Eingänge einlesen
    In      Reg_A, PInD         ; Port D lesen
    COM     Reg_A               ; Bits drehen
    ANDI    Reg_A, 0b01111100   ; Nur Eingänge übernehmen
    LSR     Reg_A               ; nach rechts schieben (00111110)
    LSR     Reg_A               ; nach rechts schieben (00011111)
; hier fügen wir nun die Abfrage von Portpin PC0 hinzu. Dafür nehmen wir das Register B
    In      Reg_B, PInC         ; Port B lesen
    COM     Reg_B               ; Bits drehen
    ANDI    Reg_B, 0b00000001   ; Nur Bit 0 übernehmen
    SWAP    Reg_B               ; Bit 0 nach Bit 4 durch Nibbletausch
    LSL     Reg_B               ; Bit auf Stelle 5 schieben
    OR      Reg_A, Reg_B        ; Und Eingang hinzufügen
; Der Eingang PC0 ist jetzt in den Events auf Bit 5 abgelegt.
    STS     Akt_In, Reg_A
RET
```

Da wir bei der Flankenerkennung das gesamte Byte behandelt haben und auch in der Entprellroutine, brauchen wir dort nun keine Hand anlegen. Mit dieser Änderung ist alles komplett erledigt und auch das Ereignis der Gabellichtschranken ist in einem Eventbit abgelegt.

### *2.16.1 Anzeige Rundenzähler*

Wollen wir nun die Anzeige als Rundenzähler verwenden, sind die Zählerwerte Lap_Cnt in den Anzeigepuffer zu übertragen

```
;*****************************************************
;*          Anzeige Rundenzähler              *
;*****************************************************
Show_Runden:
    LDS     Reg_A, Anz_Mode                 ; Mode Rundenzähler?
    CPI     Reg_A, 4
    BRNE    End_Lap_Cnt                     ; wenn nicht, dann Ende
    LDI     XL, Low(Lap_Cnt_A_1)            ; Basisadresse der Rundenzähler
    LDI     XH, High(Lap_Cnt_A_1)
    RCALL   Convert_Mattrix
End_Lap_Cnt:
RET
```

So, das war die Anzeige. Nun werdet ihr fragen, ich hab doch zwei Rundenzähler mit jeweils zwei Stellen. Muss ich die nicht separat betrachten? Nein, denn wir beschreiben einen Puffer. Natürlich, die Zähler sind getrennt. Eine große Ähnlichkeit mit dem Aufbau hat die Soutine Set_Mode. Dort setzen wir mit einem Tasterevent den Wert von Mode hoch. Hier ist es kein Taster, hier sind es jetzt die Gabellichtschranken, aber ansonsten funktioniert das genauso.

```
Lap_Cnt_A:
    LDS     Reg_A,  Anz_Mode                ; Mode Rundenzähler?
    CPI     Reg_A, 4
    BRNE    End_Lap_Cnt_A
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b0010000
    BREQ    End_Lap_Cnt_A                   ; Kein Eventbit, dann Ende
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b11011111               ; Eventbit löschen
    STS     Event_To_High, Reg_A            ; und zurückschreiben
    LDS     Reg_A, Lap_Cnt_A_0              ; Rundenzähler A erste Stelle
    INC     Reg_A                           ; erhöhen
    STS     Lap_Cnt_A_0, Reg_A              ; Abspeichern
    CPI     Reg_A, 10                       ; Wert kleiner 10 ok
    BRLO    End_Lap_Cnt_A
    CLR     Reg_A                           ; Übertrag in nächste Stelle
```

```
    STS     Lap_Cnt_A_0, Reg_A          ; Einer Stelle =0 abspeichern
    LDS     Reg_A, Lap_Cnt_A_1          ; Zehnerstelle holen
    INC     Reg_A                       ; und erehöhen
    CPI     Reg_A, 10                   ; 10 erreicht,
    BREQ    End_Lap_Cnt_A               ; dann nicht mehr abspeichern
    STS     Lap_Cnt_A_1, Reg_A          ; höchster Wert 99
End_Lap_Cnt_A:
RET
```

Die zweite Spur ist genauso einfach.

```
Lap_Cnt_B:
    LDS     Reg_A,  Anz_Mode            ; Mode Rundenzähler?
    CPI     Reg_A, 4
    BRNE    End_Lap_Cnt_B
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b0100000
    BREQ    End_Lap_Cnt_B               ; Kein Eventbit, dann Ende
    LDS     Reg_A, Event_To_High
    ANDI    Reg_A, 0b10111111           ; Eventbit löschen
    STS     Event_To_High, Reg_A        ; und zurückschreiben
    LDS     Reg_A, Lap_Cnt_B_0          ; Rundenzähler A erste Stelle
    INC     Reg_A                       ; erhöhen
    STS     Lap_Cnt_B_0, Reg_A          ; Abspeichern
    CPI     Reg_A, 10                   ; Wert kleiner 10 ok
    BRLO    End_Lap_Cnt_B
    CLR     Reg_A                       ; Übertrag in nächste Stelle
    STS     Lap_Cnt_B_0, Reg_A          ; Einer Stelle =0 abspeichern
    LDS     Reg_A, Lap_Cnt_B_1          ; Zehnerstelle holen
    INC     Reg_A                       ; und erehöhen
    CPI     Reg_A, 10                   ; 10 erreicht,
    BREQ    End_Lap_Cnt_B               ; dann nicht mehr abspeichern
    STS     Lap_Cnt_B_1, Reg_A          ; höchster Wert 99
End_Lap_Cnt_B:
RET
```

Nun noch Reset für die Anzeige. Dazu einfach wieder ein Tastaturereignis nehmen und anpassen. Wenn wir Taster 4 nehmen, können wir die Routine Wecker aus anpassen.

```
;*****************************************************
;*        Rundenuzähler  rücksetzen          *
;*****************************************************
```

```
Reset_Lap_Cnt:
    LDS     Reg_A, Anz_Mode              ; Mode Rundenzähler?
    CPI     Reg_A, 4
    BRNE    End_Reset_Cnt            ; wenn nicht, dann Ende
    LDS     Reg_A, Event_To_high         ; Ereignisbit laden
    ANDI    Reg_A, 0b00001000            ; Nur Bit 3 prüfen
    BREQ    End_Reset_Cnt            ; wenn Ergebnis 0, dann Ende
    LDS     Reg_A, Event_To_high         ; Ereignisbit laden
    ANDI    Reg_A, 0b11110111            ; Nur Bit 3 auf 0 setzen
    STS     Event_To_high, Reg_A         ; und Eventbit löschen
    CLR     Reg_A                        ;  Register A auf 0
    STS     Lap_Cnt_A_0                  ;  alle Werte auf 0
    STS     Lap_Cnt_A_1
    STS     Lap_Cnt_B_0
    STS     Lap_Cnt_B_1
End_Reset_Cnt  :
RET
```

Nun basteln wir diese vier Subroutinen wieder in die Main_Loop

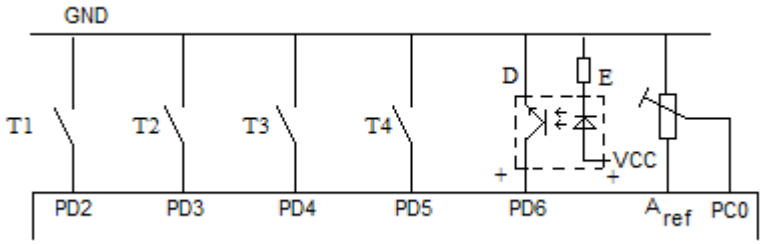
```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL    Read_IO              ; Eingänge lesen
   RCAll    IO_Debounce          ; Eingänge entprellen, gültig in Variablen New_In
   RCALL    Set_LED_Bit          ; Bearbeiten „V“
   RCALL    IO_Event_To_1        ; Ereignisbits bilden. Ergebnis in Event_To_High
   RCALL    Set_Mode             ; Anzeige umschalten
   RCALL    Chk_Blinker          ; Blinkerbit bilden
   RCALL    Chk_Time_Flag        ; Zeitereignisse bearbeiten
   RCALL    Show_Uhr             ; Uhr anzeigen ?
   RCALL    Stell_Uhr            ; Stellzeit  Uhr anzeigen ?
   RCALL    Reset_Lap_Cnt        ; Rundenzähler löschen
   RCALL    Show_Runden          ; Rundenzähler anzeigen?
   RCALL    Lap_Cnt_A            ; Runden zählen Slot A
   RCALL    Lap_Cnt_B            ; Runden zählen Slot B
   RCALL    Sel_Ziffer           ; Ziffer weiterschalten
   RCALL    Set_Zahl_Up          ; Zahl hochzählen
   RCALL    Set_Zahl_Down        ; Zahl runterzählen
   RCALL    Set_Alarm            ; Alarmzeit stellen
   RCALL    Alarm_On_Off         ; Alarm scharfschalten
```

```
    RCALL    Wecker_Aus          ; Wecker ausschalten
    RCALL    Stopzeit_On_Off     ; Stoppuhr Start - Stopp
    RCALL    Set_Stoppuhr        ; Anzeige Stoppuhr
    RCALL    Chk_Receive         ; Datenempfang prüfen
    RCALL    Is_First            ; erstes empfangenesByte prüfen und markieren
    RCALL    Is_Second           ; Datenempfang endgültig auswerten
    RCALL    Write_IO            ; und ausgeben
RJMP Loop
```

Nun verabschieden wir uns erst einmal von Uhren und Rundenzählern.

## 2.17 Einen Analogen Eingang einlesen

Es gibt auch Anwendungen, die nicht nur binäre, sondern auch analoge Eingänge auswerten sollen. Dazu müssen wir den Eingang PC 0 wieder mit der Beschaltung vom Ausgangsplan belegen.




*Schaltplan Analogeingang*

ARef ist eine stabile Spannung, die vom Controller ausgegeben wird. Statt ARef ist auch VCC möglich, nur VCC kann von den genormten 5 V abweichen. Das verfälscht den Wert und möglicherweise bleibt die Spannung bei Lastwechsel nicht Stabil. Auch das würde zur Ungenauigkeit beitragen.

### 2.17.1 Analogeingang initialisieren

Der erste wichtige Schritt ist die Initialisierung von PC5 zu ändern. Wir haben in Init_IO dem Portbit intern den Pull_Up Widerstand zugeschaltet. Der wird jetzt entfernt.

```
Init_IO:
    ……
    IN      Reg_A, PortC
    ANDI    Reg_A, 0B11111110              ; Pull_Up Widerstand aus
    OUT     Port_C, Reg_A
   ……..
RET
```

Nun erfolgt ein weiterer Aufruf einer Initialisierung vor der Programmschleife.

```
;*****************************************************************
;*          Initialisieren Analog- Digitalwandler        *
;*          Kanal 0   linksbündig ( 1 Byte)              *
;*          Vorteiler 128                                *
;*          internre Referenzspannung 2,56 V             *
;*   die erste  Zuschaltung erfolgt im Programm          *
;*****************************************************************
Init_ADC:
   LDI      Reg_A, (1<<REFS1)|(1<<REFS0)|(1<<ADLAR)
   OUT    ADMUX, REG_A
   LDI      Reg_A, (1<<ADEN)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
; ADC enabled, ADC Interrupt Enabled, Prescaler 128
  OUT    ADCSRA, Reg_A
RET
```

Was haben wir hier alles eingestellt und wieso braucht der AD-Wandler einen Interrupt? Hier ist es wichtig einen Blick in das Datenblatt zu werfen.

Da ist erst einmal die Festlegung der Referenzspannung auf 2,56 V. Dies wird mit den beiden Bits REFS1 und REFS0 festgelegt. Tabelle 74 im Datenblatt zeigt die Zusammenhänge.

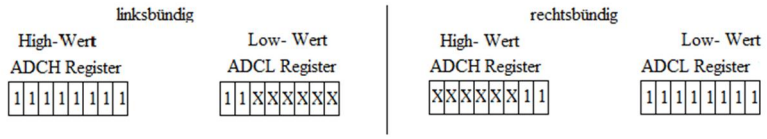
ADLAR ist ein interessantes Bit. Wie wir wissen, ist der ATMEGA8/16 ein Controller, der in der Regel mit 8 Bit Registern arbeitet. Nun hat der ADC

aber eine Auflösung von 10 Bit, was zu einem wesentlich höheren Programmieraufwand führt. Nicht immer ist es aber erforderlich, diese 10 Bit-Auflösung auch zu benutzen, denn es bedeutet bei 2,56V nur eine Ungenauigkeit von 2,56/1024 *4

Betrachten wir einmal die Werte:

Eine Auflösung von 10 Bit bedeutet, das 2,56 V in 1024 Schritten aufgelöst wird. Das entspricht einer Spannung von 0,0025 V. Werden nun die unteren 2 Bits nicht bewertet, so habe ich eine Auflösung von 2,56/ 256 und damit eine Schrittspannung von 0,01 V. Wenn eine solche Schrittspannung ausreicht, warum dann mit 10 Bit arbeiten, 8 tun es dann auch und es läßt sich prima einfach mit den normalen Registern arbeiten. ADLAR sagt nun folgendes aus: rechts- oder linksbündige Auflösung.

Werfen wir einmal einen Blick auf das Register, in dem der gewandelte Wert abgelegt ist.




*ADC Register*

In der rechtsbündigen Darstellung ist der Wert in den beiden Registern wertemäßig richtig abgelegt. Bit 0 hat eine Auflösung von 0,0025 V. Aber eine einfache Verarbeitung in einem einzigen Arbeitsregister ist nur durch eine 16 Bit Arithmetik durchzuführen. Alles zweimal nach rechts schieben oder teilen durch 4. Beide Operationen gehen über 16 Bit.

Einfacher ist es, die linksbündige Wandlung einzuschalten. Nun stimmt zwar der Gesamtwert nicht zur Auflösung von 0,0025V, aber wenn man nur den High –Wert betrachtet, ist es ein vollständiges Byte mit einer Bewertung Bit 0 von 0,01 V. Die beiden Bits in ADCL interessieren dann nicht mehr. Über das Bit ADLAR wird diese Ausgabe gesteuert.

Das Register ADMUX beinhaltet aber noch vier weitere interessante Bits. MUX0 bis MUX3. In Tabelle 75 wird die Zuordnung der Eingänge PC0 bis PC5 (Kanäle) eingegeben. Da wir nichts geändert haben, liegt die Kanalwahl bei PC0.

### *2.17.2 Analogwert lesen Warten oder Interrupt*

Bevor wir tiefer in den ADC einsteigen deklarieren wir zuerst einmal zwei Variablen, die später den gelesenen Analogwert aufnehmen.

```
Analog_Low:   .Byte1          ; Int8
Analog_High:  .Byte 1         ; Int8
```

Nun kommen wir zur Parametrierung vom ADC Control und Statusregister ADCSRA und beginnen mit ADEN. ADC Enable gibt den Wandler frei.

ADSC startet eine Wandlung. Dazu ein paar Worte. Eine Wandlung ist nicht einfach mal so 2V =11001000. Eine Wandlung läuft in mehreren Schritten ab. Dafür hat der AD-Wandler einen Takt, der vom CPU-Takt abgeleitet wird. Ist eine Wandlung fertig, zeigt der Controller dies in einem weiteren Statusbit an und das Programm kann auf den Wert zugreifen. Das funktioniert so nach dem Motto, welche Spannung liegt am Port an? Und wenn du es weißt, dann melde dich. Bis dahin warte ich oder gehe meiner Arbeit nach. Das ist nur eine Frage, wie die Antwort des Wandlers programmiert wird. Warte ich, reicht ein Bit –bin fertig. Mach ich mit meiner Arbeit weiter, braucht es einen Interrupt, also etwaas, was mir auf die Schulter klopft, ich soll mal nach den Werten schaun. Wir wählen den Weg Interrupt, weil ich Programme hasse, die auf irgend etwas warten müssen. Somit ist klar, wenn ein Wandelvorgang gestartet wird, muss auch ein Interrupt programmiert sein, der anspricht, wenn das Ergebnis vorliegt. Somit kommen wir zum Bit ADIE Interrupt Enabled. Ist dieses Bit gesetzt wird nach fertiger Wandlung der Interrupt ADCCaddr in der Interupt Vector Table aufgerufen. Eine nachfolgende ISR erledigt dann die Verarbeitung.

Der andere Weg ist den ADC zu starten und auf das Ergebnis zu warten. Dabei ist das Bit ADSC behilflich, welches nur während der Wandlung gesetzt ist und mit 0 anzeigt, das die Wandlung beendet ist.

```
Read_ADC:
   SBI     ADCSRA,  ADSC            ; Start der Wandlung
Chk_Ergebnis:
  SBIC     ADCSRA,  ADSC            ; einen Befehl überspringen, wenn 0
  RJMP     Chk_Ergebnis
  In       Reg_A, ADCL              ; zuerst das LowByte
  In       Reg_A, ADCH              ; dann HighByte. LowByte hat keine Bedeutung
  STS      Analogwert, Reg_A        ; Ablegen in einer Variablen
RET
```

Bei dieser Vorgehensweise ist z. B. eine Wandlung aus einem Zeitereignis aufzurufen, da die Wandlung hier gezielt gestartet wird.

Das Ergebnis lässt sich mit Open_Eye schon ganz gut nachvollziehen. Wird der Wert des Potis geändert, erhält man auf Analog_High einen Wert zwischen 0 und 255. Der Wert in Analog_Low ist nicht von Bedeutung.

Wenden wir uns nun dem Interrupt des ADC zu. Wird mit das Ergebnis in einer ISR gelesen, dann ist die Warteschleife überflüssig und die nächste Wandlung kann sofort eingeleitet werden.

```
ISR_Read_ADC:
   In      Ablage_SREG, SREG
   Push    Reg_A                    ; ISR, daher Register retten
   In      Reg_A, ADCL              ; zuerst das LowByte
   STS     Analog_Low, Reg_A        ; Ablegen in einer Variablen
   In      Reg_A, ADCH              ; dann HighByte. LowByte hat keine Bedeutung
   STS     Analog_High, Reg_A       ; Ablegen in einer Variablen
   SBI     ADCSRA,  ADSC            ; Start der Wandlung
   POP     Reg_A
   OUT     SREG, Ablage_SREG
RETI
```

Dieser Aufruf muss dann in der IVT eingetragen werden. Der Befehl RETI wird auskommentiert.

```
.org ADCCaddr  RJMP ISR_Read_ADC     ; ADC Interrupt Vector Address
   ;RETI
```

Vor und Nachteile dieser beiden Varianten sollen nicht verschwiegen werden. Zuerst Read_ADC.

Vorteil: Hier erfolgt der Aufruf gezielt vom Programm. Wenn ich den Analogwert brauche, hole ich ihn ab. Damit kann ich ein Zeitereignis beauftragen, z.B. im Sekundentakt.

Nachteil: Die Zykluszeit verlängert sich um die Wandlerzeit, da in der Routine auf das Ergebnis gewartet wird.

Vorteil einer Interruptbehandlung ist klar. Der Leseauftrag wird in der Initialisierung nach SEI-Befehl gegeben. Danach wird die Arbeit

fortgesetzt. Soll sich doch der Wandler melden, wenn er einen Wert ermittelt hat.

Nachteil: Verwendete Register müssen auf dem Stack zwischengespeichert werden. Das erfordert auch ein paar Taktzyklen, aber nicht soviel Zeit, wie die Warteschleife.

Wenn nach der Erfassung sofort wieder ein Leseauftrag vergeben wird, erhöht sich auch die Zykluszeit, diesmal aber durch die Vielzahl von Interrupts, die der AD-Wandler auslöst. Abhilfe schafft der Aufruf nach wie vor aus dem Programm heraus und die Auswertung in der ISR ohne einen Folgeauftrag. Dazu einfach in der ISR den Befehl

```
SBI     ADCSRA,  ADSC                    ; Start der Wandlung
```

auskommentieren oder entfernen.

### *2.17.3 Anzeige Analogwert*

Was aber fangen wir mit dem Wert an, der nun in den Variablen Analog_Low und Analog_High abgelegt ist. Nun, zuerst einmal ist Analog_Low nur dann von Interesse, wenn die gesamte 10 Bit Auflösung eingestellt ist. Dann besteht die Möglichkeit, den Wert über Open_Eye zu einer 16Bit integer zusammenzuführen und auszulesen. Für eine Verarbeitung im Controller reicht die linksbündige Wandlung und der Wert in Analog_High.

In erster Linie können wir mit dem Poti verschiedene Werte einstellen und über Open_Eye sichtbar machen. Schlecht wär eine Ausgabe auf unserem Display aber auch nicht. Und es wär schön, nicht den Binärwert, sondern den Spannungswert zu sehen.

Dazu folgendes: Bei einer Referenzspannung von 2,56 V und der 8Bit Auflösung entspricht die Anzeige der Voltzahl. Ähem, nicht ganz, denn die Variable kann nur von 0 bis 255, die Spannung aber von 0 bis 2,56 V. Diesen kleinen Fehler von 0,01 V vernachlässige ich aber einmal, da meine Anzeige sowieso keine Realzahl darstellt und die Mathematik dahingehend nicht unmöglich, aber nicht einfach zu durchschauen ist.

Wenn wir nun einen weiteren Wert der Anzeige zuführen möchten, brauchen wir auch einen weiteren Speicher für die Wertaufbereitung und einen weiteren Mode. Der Mode ist kein Problem, einfach den Grenzwert hochsetzen.

Für den Wertebereich definieren wir erst einmal vier Variablen nach gewohnter Art..

```
Analog0:        .Byte 1
Analog1:        .Byte 1
Analog2:        .Byte 1
Analog3:        .Byte 1
```

Diese brauchen nicht zurückgesetzt werden, da sie aus der Variable Analogwert berechnet werden. Dafür schreiben wir wieder eine separate Routine.

```
Analog_Out:                     ; Ausgabe Analogeingang auf Display
    LDS     Reg_A, Anz_Mode     ; Ausgabe in Mode 5
```

```
    CPI     Reg_A, 5
    BRNE    End_Analog_Out
    LDS     Reg_A, Analog_High          ; Register A  mit Analogwert
    CLR     Reg_B                       ; Stelle 1
    CLR     Reg_C                       ; Stelle 2
    CLR     Reg_D                       ; Stelle 3
Loop_Set_Analog:
    CPI     Reg_A, 10                   ; Wenn Wert <10 dann fertig
    BRLO    Set_Wert_Analog
    Sub     Reg_A, 10
    INC     Reg_B                       ; Teilerwert durch 10
    CPI     Reg_B,10
    BRLO    Loop_Set_Analog
    CLR     Reg_B
    INC     Reg_C                       ; Teilwert durch 100
    RJMP    Loop_Set_Analog
Set_Wert_Analog:
    STS     Analog0, Reg_A              ; Einer
    STS     Analog1. Reg_B              ; Zehner
    STS     Analog2, Reg_C              ; hunderter
    STS     Analog3, Reg_D              ; tausender (0)
    LDI     XL, Low(Analog0)            ; Basisadresse Analogwert laden
    LDI     XH, High(Analog0)
    RCALL   Convert_Matrix
End_Analog_Out:
RET
```

Doch wo rufen wir nun den Baustein Analog_Out auf? Natürlich geht das in der Hauptschleife, aber vermutlich wird dann sehr oft der gleiche Wert geteilt, weil der ADC gar nicht so schnell ist. Das belastet unsere Zykluszeit unnötig. Grad haben wir uns entschlossen, das der ADC sich melden soll, wenn er fertig ist und nun soll wieder unnötig Programm bearbeitet werden? Denken wir mal ein paar Seiten zurück. Dort sprach ich von Jobflags. Diese lassen sich hier genau so gut verwenden. Einfach in der ADC_ISR ein Jobflag setzen und schon ist es möglich, mithilfe des Jobflags die Teilung dann nur einmal durchlaufen zu lassen. Erst der nächste Interrupt setzt das Jobflag erneut. Jobflags haben wir auch schon in der Variablen Prg_Ctrl.

Mal sehen, ob da was frei ist. Bit 3 zeigt noch eine Lücke. Schließen wir diese erst einmal

```
Prg_Ctrl:     .Byte 1    ; Byte Programmstatusbits
                          ; Bit 0  =  Alarm ein = 1, aus = 0
                          ; Bit 1  =  Alarm hat ausgelöst = 1
                          ; Bit 2  = Zeitfenster Alarm
                          ; Bit 3  =  neuen Analogwert gelesen
                          ; Bit 4  = Stopuhr läuft = 1
```

Solange nun Bit 3 nicht gesetzt ist, braucht auch keine neue Wertezuweisung berechnet werden und an die Ausgabe erfolgen. Fügen wir nun diese Prüfung hinter die Mode-Abfrage.

```
Analog_Out:                          ; Ausgabe Analogeingang auf Display
    LDS     Reg_A, Anz_Mode          ; Ausgabe in Mode 5
    CPI     Reg_A, 5
    BRNE    End_Analog_Out
    LDS     Reg_A, Prg_Ctrl          ; Änderungsbit aus ISR prüfen
    ANDI    Reg_A, 0b00001000
    BREQ    End_Analog_Out           ; kein neuer Wert, dann Ende
    LDS     Reg_A, Prg_Ctrl
    ANDI    Reg_A, 0b11110111                ; Änderungsbit quittieren
    STS     Prg_Ctrl, Reg _A         ; und wieder abspeichern
    .......
```

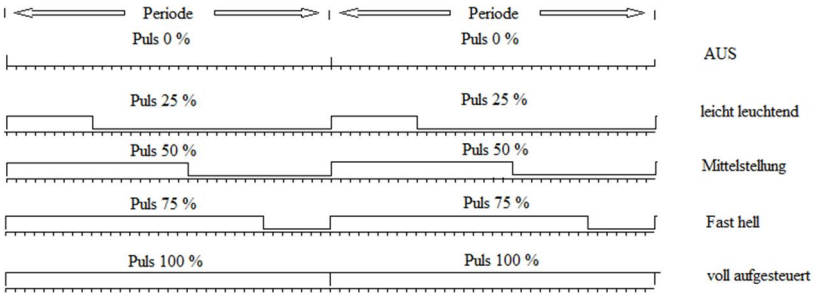
Nun muss das Bit in der ISR noch gesetzt werden. Allerdings setzt die Oder-Verknüpfung die Statusbits im Statusregister und deshalb muss es hier auch noch gesichert werden.

```
ISR_Read_ADC:
   Push   Reg_A                      ; ISR, daher Register retten
   In     Reg_A, SReg                ; Statusregister zusätzlich sichern
   Push   Reg_A
   In     Reg_A, ADCL                ; zuerst das LowByte
   In     Reg_A, ADCH                ; dann HighByte. LowByte hat keine Bedeutung
   STS    Analogwert, Reg_A          ; Ablegen in einer Variablen
   ;SBI   ADCSRA,  ADSC              ; Start der Wandlung
  ;neuen Wert über Zeitereignis anfordern
   LDS    Reg_A, Prg_Ctrl            ; Controlbits Programm  laden
   ORI    Reg_A, 0b00001000          ; Signal neuer Analogwert
   STS    Prg_Ctrl, Reg_A            ; und eintragen
   POP    Reg_A                      ; Statusregister wieder herstellen
   Out    Sreg, Reg_A
   POP    Reg_A                      ; Register a wieder herstellen
RETI
```

Das spart viel Zykluszeit.

## 2.18 PWM Mit dem Potentiometer dimmen

Die LED ist noch angeschlossen, starten wir nun ein weiteres Experiment. Die Pulsweitenmodulation wird gern zur Regelung von verschiedensten Anwendungen benutzt. Hier kann nur das Prinzip und die Grundlage anhand einer LED erklärt werden. Zuerst einmal die Erklärung mit einer Skizze.




*PWM-Signal*

Wir sehen hier fünf Impulsdiagramme von jeweils zwei Perioden. Die Zeiteinheit ist hier für die Periode erst einmal nicht von Bedeutung. Das erste Diagramm oben zeigt zwar den Abschnitt, aber keinen Impuls. Damit ist auch klar, da ist nichts messbar. Wenn da etwas angeschlossen ist, dann ist es AUS.

Das Diagramm darunter zeigt einen Impuls von 25 % der Periode. Hier wird man schon etwas messen können und wenn eine LED angeschlossen ist, wird sie auch schwachleuchten.

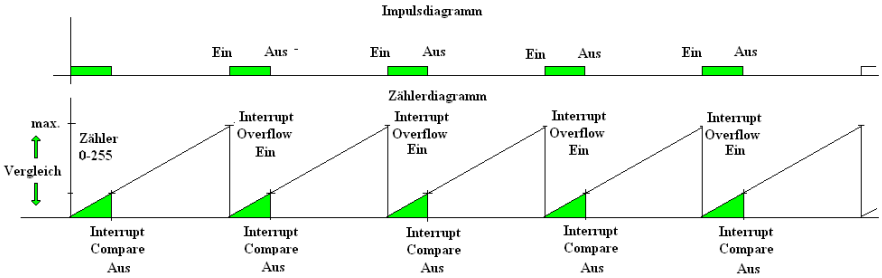
Bei 50 %, das ist ein symetrisches Rechtecksignal, ist schon etwas mehr zu sehen. Schließlich ist die Vollaussteuerung bei 100 % erreicht. Um ein solches Signal zu bilden brauchen wir einen Timer. Der Atmega8 hat drei, Timer1 haben wir bereits im Einsatz und das ist auch der einzige, der eine 16 Bit Struktur hat. Die Timer 0 und 2 sind nur mit einem Register bestückt und 8 Bit breit. Doch das ist hier nicht von Bedeutung, ein 8 Bit Timer reicht für unser Vorhaben durchaus.

### *2.18.1 PWM mit Timer2*

Timer2 hat hat einen Fast PWM Modus. Das ist genau die richtige Wahl. Dabei werden zwei Interrupts des Timers verwendet. Einmal der Interrupt Compare und einmal Overflow. Eine zeitgenaue Teilung ist nicht erforderlich und so reicht es, den Vorteiler auf 128 einzustellen um bei 16 MHz eine Grundfrequenz von ca. 488 Hz zu bekommen. Auch hier ist es notwendig, die ISR so schnell wie möglich ablaufen zu lassen und deshalb wird der Ausgang ohne Eventbit direkt gesetzt oder gelöscht.

Und das funktioniert folgendermaßen. Der Counter im Timer2 läuft von 0 bis 255 und löst dort einen Interrupt Überlauf aus und beginnt von vorn. Dabei setzen wir den Ausgang. Irgendwann erreicht er den Wert, der im Vergleichsregister eingetragen ist und löst den Interrupt bei Vergleich (Compare) aus. Hier löschen wir den Ausgang.

Dazu eine Skizze :




*PWM-Ablaufdiagramm*

Zuerst die Initialisierung. Hier hilft uns das Datenblatt des Controllers.

```
;*****************************************************************
;*   Timer 2 eingesetzt für Pulsweitenmodulation     *
;*      Vorteiler einstellen auf 128                  *
;*      Interrupt bei Überlauf                        *
;*      WGM20 und WGM 22 Fast PWM Mode               *
;*      CS22 und CS20  = Teiler 128                   *
;*      OCR2 Compare-Register Timer 2                 *
;*****************************************************************
Init_Timer2:
    ldi   Reg_A, (1<<WGM20) | (1<<WGM21) | (1<<CS22) | (1<<CS20)
  out   TCCR2, Reg_A
  LDI   Reg_A,1
  Out   OCR2, Reg_A
RET
;-----------------------------------------------------------------
```

Für die Interruptbits setzen wir eine eigene Subroutine zur Initialisierung. Die Timer teilen sich alle dieses Register und der Timer1 hat dort auch schon ein Bit belegt.

Diese Zuweisung entfernen wir nun in der Initialisierung und setzen für alle Timer die Bits in einer eigenen Initialisierungsroutine.

```
;******************* Timer 1 Parametrierung ************************
;*   Timer 1 ist die Zeitbasis für alle zeitabhängigen          *
;*   Ereignisse. Er erzeugt einen Interrupt je mSek In          *
;*   den Zeitregistern weden diese mSek. aufaddiert und         *
;*   die Zeitbasis gebildet. Ist vielleicht mal für eine        *
;*   andere Anwendung von Interesse. Aber die Zeitbasis         *
;*   ist abhängig vom Takt. In diesem Fall muß die              *
;*   Taktfrequenz 12 MHz sein.                                  *
;*****************************************************************
Init_Timer1:                          ; Einstellung 12 MHz
  LDI    Reg_A, high( 2000 - 1 )      ; Set Compare Value für eine msek
  OUT    OCR1AH, Reg_A                ; bei 16 MHz 2000-1 eintragen
  LDI    Reg_A, low( 2000 - 1 )
  OUT    OCR1AL, Reg_A
;++++++++++++++++++++++++++++++++++++++++++++
;*      Vorteiler CS12 CS11 CS10                    *
;*                 0     0     0      No Timer      *
;*                 0     0     1      Teiler 1      *
;*                 0     1     0      Teiler 8      *
;*                 0     1     1      Teiler 64     *
;*                 1     0     0      Teiler 256    *
;*                 1     0     1      Teiler 1024   *
;*                 1     1     0      Zähler 1-0    *
;*                 1     1     1      Zähler 0-1    *
;*************************************************************
; CTC Modus einschalten Vorteiler auf 8
  LDI    Reg_A, (1<<WGM12) | (1<<CS11)   ; Bit WGM 12 ist CTC Modus ein
  OUT    TCCR1B, Reg_A                   ; Bit CS11 ist Teiler 8 vorgeschaltet
  LDI    Reg_A, (1<<OCIE1A)              ; Interrupt CTC setzen
  OUT    TIMSK, Reg_A
RET
```

Dafür schreiben wir eine neue Subroutine.

```
;*****************************************************************
;*      OCIE2: Interrupt bei Timer2 Compare          *
; *     TOIE2: Interrupt bei Timer2 Overflow         *
;*      OCIE1A: Interrupt bei Timer1 Compare         *
;*****************************************************************
Init_Timer_Interrupt:
  LDI     Reg_A, (1<<OCIE2) | (1<<TOIE2) | (1<<OCIE1A)
  out    TIMSK, Reg_A
RET
;-----------------------------------------------------------------
```

Nun sind noch die zugehörigen ISRs erforderlich. In einer ISR wird der Ausgang PC5 direkt gesetzt ( LED ist aus), in der anderen gelöscht. (LED ist an). Den Zugriff auf den Port C entfernen wir aus der Routine Write_IO. Hier den Weg über die Variable Out_Ctrl zu gehen benötigt zu viel Zeit. Der Aufbau der beiden ISR vom Timer2 sind schnell geschrieben. Dabei ist eigentlich nur wichtig, benutztes Register und SREG sichern. Port lesen, Bit setzen oder löschen und Port schreiben. Keine großartige Sache.

```
;*****************************************************************
;*      Interrupt bei Vergleich  mit Analogwert       *
;*****************************************************************
ISR_Timer2_Cmp:
  In       Ablage_SReg, SREG   ; Statusregister sichern
  SBI   PortC, 5                ; PortC bit 5 Setzen ( LED aus )
  OUT   SREG, Ablage_SReg    ; Register A wieder herstellen
RETI
;-----------------------------------------------------------------
```

```
;*************************************************************
;*              Interrupt bei Überlauf                       *
;*************************************************************
ISR_Timer2_Ovf:
  In        Ablage_SReg, SREG   ; Statusregister sichern
  CBI   PortC, 5                ; PortC bit 5 löschen ( LED an )
  OUT     SREG, Ablage_SReg  ; Register A wieder herstellen
RETI
;-------------------------------------------------------------
```

Nun bleibt noch der Eintrag dieser beiden ISR von Timer2 in der IVT und das RETI dort zu entfernen oder auszukommentieren.

```
.org  OC2addr        RJMP ISR_Timer2_Cmp   ; Output Compare2 Interrupt Vector Address
   ;RETI
.org  OVF2addr    RJMP ISR_Timer2_Ovf   ; Overflow2 Interrupt Vector Address
   ;RETI
```

Dem Test steht jetzt nichts mehr im Weg. Mit dem Potentiometer soll die Helligkeit der LED eingestellt werden. Aber halt, da fehlt ja noch der Eintrag des Analogwertes in das Vergleichsregister von Timer2. Das ist in der Subroutine ISR_Read_ADC schnell mit der Zuweisung des Analogwertes an das Vergleichsregister erledigt.

```
;*****************************************************************
;*                ISR  ADC Analogwert lesen              *
;*****************************************************************
ISR_Read_ADC:
  IN      Ablage_SREG, SREG       ; SREG sichern
  Push    Reg_A                   ; ISR, daher Register retten
  In      Reg_A, ADCL             ; zuerst das LowByte
  STS     Analog_Low, Reg_A
  In      Reg_A, ADCH             ; dann HighByte. LowByte hat keine Bedeutung
  STS     Analog_High, Reg_A      ; Ablegen in einer Variablen
;*********** hier erfolgt die Zuweisung *****************
  Out     OCR2, Reg_A
  SBI     ADCSRA,  ADSC           ; Start der Wandlung
  ; ****für schnelle Reaktion nächste Wandlung hier aufrufen ****
  LDS     Reg_A, Prg_Ctrl         ; Controlbits Programm  laden
  ORI     Reg_A, 0b00001000       ; Signal neuer Analogwert
  STS     Prg_Ctrl, Reg_A         ; und eintragen

  POP     Reg_A
  OUT     SREG, Ablage_SREG       ; SREG zurückholen
RETI
```

Für den Versuch nehmen wir den zeitgesteuerten Aufruf Read_ADC aus dem Timer_Event und setzen in der ISR das Bit für die nächste Wandlung. Um den ADC aber einmal anzustoßen, wird Read_ADC in der Initialisierungsroutine aufgerufen. Die LED läßt sich nun schön gleichmäßig mit dem Potentiometer dimmen. Dennoch könnte die häufige Wandlung, bedingt durch einen neuen Anstoß in der ISR den Controller zu stark beschäftigen und es reicht doch völlig aus, den neuen Wert zum Herunterdimmen alle 10 mSek. anzuforden. Schließlich soll der Controller ja auch für die anderen Aufgaben noch Zeit haben und der Timer2 löst auch bei 16 MHz und dem Teiler 128 ca. alle 2 mSek. einen Overflow- und einen Compare- Interrupt aus. Das ist zwar viel Zeit, aber dennoch soll ja noch die Anzeige funktionieren und was wir sonst noch programmieren wollen. Selbst wenn ein neuer Analogwert nur alle 1/10 Sek. abgerufen wird, stört das den Helligkeitsregler nicht. Ist ein Motor mit einer entsprechenden Schaltung angeschlossen, braucht es schon schnelle Reaktionen, doch mit 10 mSek. sind wir auf der sicheren Seite.

So streichen wir den Befehl `SBI ADCSRA, ADSC ; Start der Wandlung`

Und schreiben eine eigene kleine Routine für den Start zur Wandlung eines Analogwertes.

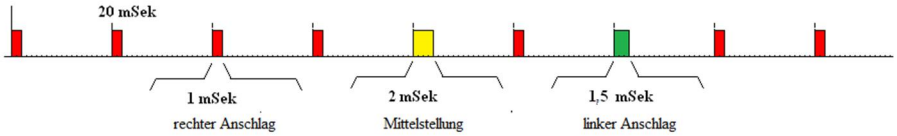
```
;*************************************************************
;*                    Analogwert einlesen                    *
;*************************************************************
Read_ADC:
   SBI    ADCSRA, ADSC                    ; Start der Wandlung
RET
```

Den Aufruf dieser Routine können wir nun mit `RCALL Read_ADC` in einem beliebigen Zeitevent einbauen.

## 2.19 Ein Modellbauservo steuern

Sicherlich nicht uninteressant ist die Möglichkeit, mit einem Mikrocontroller ein Servo in eine bestimmte Stellung zu fahren oder ihn mit einem Analogwert auszulenken. Schließlich haben wir ja immer noch das Poti angeschlossen. Zugegeben, es ist nicht ganz so einfach, wie es sich anhört, aber schließlich haben wir ja schon eine Menge Erfahrung und so wird die Programmierung mutig angegangen.

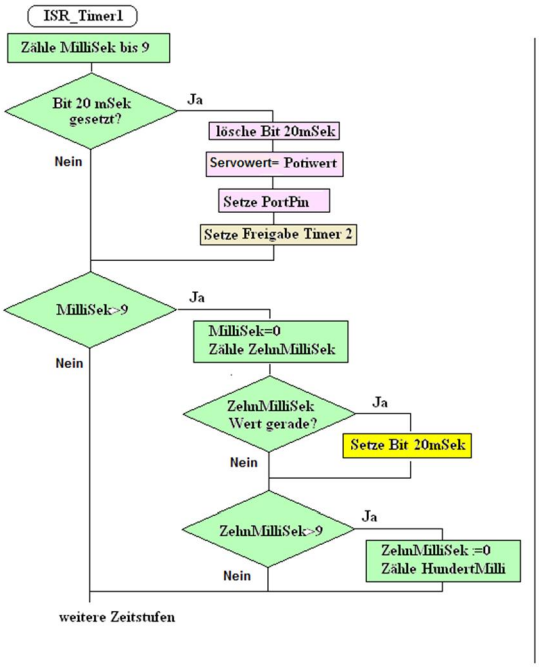
Zuerst einmal überlegen wir, wie ein Servo funktioniert. Das Internet ist voll von Beschreibungen, so dass ich mich hier nur auf das Grundprinzip einlasse. Ein Impuls, der alle 20 mSek. für eine Millisekunde ansteht, lenkt das Servo an einen Anschlag. Wird dieser Impuls auf zwei Millisekunden verlängert, fährt das Servo an den anderen Anschlag. So kann definiert werden, dass ein Impuls von 1,5 mSek. ein Servo in die Mittelstellung bringt und von dort durch Verlängern oder Verkürzen des Impulses eine beliebige Auslenkung statfinden kann.

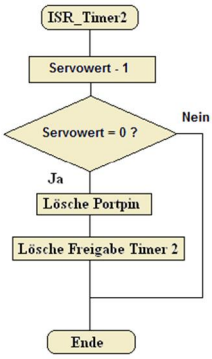



*Pulsmodulation Servo*

So, nun reden wir von Zeiten im µSek. Bereich. Schafft das der Atmega mit 16 MHz? Und ist auch für das restliche Programm dann noch Zeit?

Zugegeben, hier kommt es auf jeden Befehl an. Aber wir programmieren Assembler und in dieser Sprache ist nachvollziehbar, welcher Zeitaufwand in dem Programm zur Ausführung benötigt wird. Klar ist auch, das ohne den Interrupt von Timer2 nix geht, denn nur der hat noch die Compare-Funktion außer dem Timer 1 und den brauchen wir für eine ziemlich genaue Zeitbasis. Bei der Servoansteuerung allerdings dürfen wir ein klein wenig schludern, das Bruchteile der Auslenkung sowieso nicht aufgelöst werden können. Eine Skizze dieser Aufgabe schadet erst einmal nicht und schafft ein wenig Übersicht.




*ISR Timer 1 und Timer 2*

Fassen wir kurz zusammen:

Den Grundtakt von 20 mSek. löst der Timer1 auf. Das ist nicht einmal besonders schwer. Wir haben in der Timer_ISR Zeittakte von 1mSek, 10 mSek, 100 mSek und einer Sekunde. Dazu zählen wir Register je Dekade von 0 bis 9. Betrachten wir dazu das Bitmuster fällt etwas auf:

0 = 0 0 0 0 0 0 0 0
1 = 0 0 0 0 0 0 0 1
2 = 0 0 0 0 0 0 1 0
3 = 0 0 0 0 0 0 1 1
4 = 0 0 0 0 0 1 0 0
5 = 0 0 0 0 0 1 0 1
6 = 0 0 0 0 0 1 1 0
7 = 0 0 0 0 0 1 1 1
8 = 0 0 0 0 1 0 0 0
9 = 0 0 0 0 1 0 0 1

*Bitmuster Zahlenreihe*

Bit 0 wechselt bei jeder geraden und ungeraden Zahl. Somit ist eine 1 bei 1, 3, 5, 7 und 9 und das bedeutet, diese 1 entsteht alle 20 mSek., wenn ich die Dekade 10 mSek. zähle. Werfen wir nun einen Blick in die ISR, wo wir die 10 mSek. zählen.

```
;************************* 10 msek *************************
   CLR      MilliSek
   ORI      Reg_A, 0b00000010          ; Zeitflag 10 mSek. setzen
   INC      Zehn_Milli
   CP       Zehn_Milli, zehn
   BRLO     End_ISR_Timer1
```

Das Bit 0 von Register Zehn_Milli läßt sich nun auf 1 oder 0 auswerten. Dafür gibt es einen Befehl zum überspringen des nächsten Befehles in Abhängigkeit vom Bitstatus. SBRC wenn Bit 0 und SBRS wenn Bit gesetzt ist. Mit einem direkten Sprung wird dann in Folge ein Bereich bis zu einer Marke ausgelassen, wenn das Bit 0 nicht gesetzt ist. In diesen Bereich setzen wir im Time_Flag ein freies Bit, z.B. Bit 7 und den Ausgang PC5. Diese Erweiterung der Befehle fügen wir vor dem Vergleich auf 10 ein.

```
;************************* 10 msek *************************
   CLR      MilliSek
   ORI      Reg_A, 0b00000010          ; Zeitflag 10 mSek. setzen
   INC      Zehn_Milli
   SBRC     ZehnMilli,0
   RJMP     Chk_Zehn
   ORI      Reg_A, ßb10000000
   SBI      PortC, 5
Chk_Zehn:
   CP       Zehn_Milli, zehn
   BRLO     End_ISR_Timer1
```

Nun haben wir ein Zeitflag, das nach 20 mSek. erneut gesetzt wird, ebenso einen Ausgang. Nun besagt die Beschreibung für das Servosignal, das dieses zwischen 1 und 2 mSek. anstehen soll. Die eine mSek. sind leicht hinzubekommen. Dazu werten wir einfach in der vorherigen Zählung der mSek das Zeitflag entsprechend Bit 7 aus. Wenn gesetzt, dann lösche es und setz den Ausgang zurück. Soweit so gut. Wer

einen Oszilloskopen besitzt, kann dieses Signal betrachten. Jeder andere muss meinen Worten glauben, denn selbst eine LED an PC5 wird durchgehend leuchten. Trotzdem haben wir eine Möglichkeit, das zu testen. Legen wir diese Befehlssequenz doch einfach mal in den Bereich der Sekundenzählung. Dazu wird alle 200 mSek. das Setzen des Zeitflags und Portpin PC5 durchgeführt und die Rückstellung im Sekundenbereich.

Hier ein Auszug und die Erweiterung dazu:

```
;************************ 100 msek ************************
   CLR      Zehn_Milli
   ORI      Reg_A, 0b00000100              ; Zeitflag 100 mSek setzen
   Inc      Zehntel
   SBRC     Zehntel, 0                     ; wenn Bit 0 Sprungbefehl auslassen
   RJMP     Chk_Zehn
   ORI      Reg_A, ßb10000000              ; Zeitflag setzen
   SBI      PortC, 5                       ; Portbit setzen
Chk_Zehn:
   CP       Zehntel, zehn
   BRLO     End_ISR_Timer1
;************************ 1 Sek ************************
   ORI      Reg_A, 0b00001000              ; Zeitflag 1 Sek setzen
   CLR      Zehntel
   ANDI     Reg_A,0b01111111               ; Zeitflag llöschen
   CBI      PortC, 5                       ; Portbit löschen
End_ISR_Timer1:
   STS      Time_Flag, Reg_A
   POP      Reg_A
   OUT      SREG, Ablage_SReg
RETI
```

Bei dieser Programmierung muss selbstverständlich der Programmteil im Bereich 10 mSek. auskommentiert werden, falls er schon eingetragen ist. Das Ergebnis  ist ein deutliches flackern der LED. Entfernen wir nun diese Einträge wieder und wenden uns der eigentlichen Aufgabe zu.

Dazu wird das Programm im Bereich 10 mSek wieder aktiviert oder eingefügt.

```
;************************* 10 msek *************************
   CLR      MilliSek
   ORI      Reg_A, 0b00000010          ; Zeitflag 10 mSek. setzen
   INC      Zehn_Milli
   SBRC     ZehnMilli,0
   RJMP     Chk_Zehn
   ORI      Reg_A, 0b10000000
   SBI      PortC, 5
Chk_Zehn:
   CP       Zehn_Milli, zehn
   BRLO     End_ISR_Timer1
```

Der nächste Schritt ist also im Bereich mSek, also davor erforderlich. Wenn wir jetzt im Abschnitt 1 mSek zählen das Bit 7 in Reg_A (Time_Flag) abfragen, ist die erste mSek verstrichen, Löschen wir jetzt den Ausgang PC5 wird sich das Servo an den linken Anschlag legen. Nun gilt es das Zeitfenster etwas zu erweitern, aber das ist nicht mit dem Timer1 durchzuführen. Was nun?

### *2.19.1 Zwei Timer für einen Job*

Im Kapitel PWM haben wir den Timer2 bereits erfolgreich eingesetzt. Vielleicht ist auch hier die Möglichkeit gegeben, ihn einzusetzen. Dazu muss aber die Betriebsart etwas umgestellt werden. Wir brauchen nur noch den Compare-Interrupt und das mit Rücksetzen des Zählregisters. Also die Betriebsart CTC. Der Vorteiler wird erst einmal mit 256 eingestellt. Das ergibt einen Interrupt bei Vergleichswert 2 alle 32 µSek. Dieser Wert mit 32 multipliziert ergibt ca. 1 mSek. teilt man nun 1000, das sind die µSek, die zur Aussteuerung des Servos benötigt werden, erhält man einen Wert, der dicht bei 32 liegt, genau 31,25. Setzen wir also ein Zählregister auf 31 und lassen es herabzählen, können wir das Erreichen der 0 leicht erkennen. Das wären 32 Schritte. Nehmen wir noch einen Schritt weg, damit das Servo nicht hart an den Anschlag fährt und so erhalten wir die Zahl 30, die wir nun in ein Zählregister eintragen. Das Zählregister benennen wir Servo und dafür benutzen wir eines der Register unterhalb der 16, zum Beispiel Register 11, das wir vorher Count_L genannt haben, diese Bezeichnung aber nie benutzten.

Die Nutzung eines Zählregisters und nicht einer Variablen liegt einzig in der schnelleren Verarbeitung. Hier habe ich eine Interruptroutine, die alle 32 µSek, aufgerufen wird. Das passiert zwar nur während einer nax. Zeit von einer mSek und dann erst wieder nach 20 mSek, aber dennoch soll unser Programm zwischenzeitlich auch noch ein wenig weitermachen.

Die Initialisierung von Timer 2 wird auf die Betriebsart CTC mit einem Vorteiler von 256 eingestellt, der Interrupt aber noch nicht freigegeben. Das Compareregister bekommt den Wert 2, um möglichst wenige nterrupts zu bekommen.

Das Servo arbeitet mit 30 Schritten völlig ausreichend genau und eine höhere Auflösung ginge sehr stark in die Zykluszeit ein.

```
;*****************************************************************
;*   Timer 2 eingesetzt für Pulsweitenmodulation     *
;*       Vorteiler einstellen auf 128                 *
;*       Interrupt bei Überlauf                       *
;*       WGM20 und WGM 22 Fast PWM Mode              *
;*       CS22 und CS21  = Teiler 256                  *
;*       OCR2 Compare-Register Timer 2               *
;*****************************************************************
Init_Timer2:
  ldi   Reg_A, (1<<WGM21) | (1<<CS22) | (1<<CS21)      ;  CTC-Mode, Teier 256
  out   TCCR2, Reg_A
  LDI   Reg_A, 2
  Out   OCR2, Reg_A                                    ; Comparewert auf 1
RET
;-----------------------------------------------------------------
```

Bevor wir uns mit der Programmierung im Abschnitt mSek. befassen, sollten wir uns darüber klar sein, was wir vom Interrupt des Timer2 erwarten. Viel darf es nicht sein, denn schließlich darf die ISR nur ein paar µSek. der Zyluszeit abzwacken. Den Grund hatte ich ja schon gesagt. Das ist nicht grad viel, aber keine Sorge. Die paar Aufgaben erledigt die ISR schon in einem akzeptablen Zeitbereich.

Das sind Herunterzählen des Registers Servo und bei erreichen eines 0 Wertes das Löschen des Portbits sowie die Abschaltung der eigenen ISR.. Schreiben wir also erst einmal eine neue ISR für Timer2

```
;*************************************************************
;*      Interrupt bei Vergleich  mit Analogwert       *
;*       Aufruf alle 48 µSek                          *
;*************************************************************
ISR_Timer2_Cmp_S:
  Push  Reg_A                    ; Reg_A wird benötigte und muss gesichert werden
  In      Ablage_SReg, SREG      ; Statusregister sichern
  DEC   Servo                    ; Register für Analogwert max. Wert 30 = ca. 1, 4 mSek
  BRNE End_ISR_T2                ; wenn nicht 0, dann Ende
  CBI     PortC, 5               ; PortC bit 5 löschen
  IN      Reg_A, TIMSK           ; Interrupt Controlregister nach Reg_A
  CBR  Reg_A, 0b01111111         ; Interrupt Timer 2 abschalten ( Bit 7 löschen)
  OUT   TIMSK, Reg_A             ; und eintragen
End_ISR_T2:
  OUT   SREG, Ablage_SReg        ; Register A wieder herstellen
  POP    Reg_A
RETI
;-------------------------------------------------------------
```

So wie diese Routine aufgebaut ist, benötigt sie ca. 20 Takte oder in Zeit bei 16 MHz 1,25 µSek. Dies kann mit den Informationen aus dem Datenblatt im Abschnitt Instruction Set Summary leicht berechnet werden, da dort die Anzahl der Takte für jeden Befehl angegeben sind. Wir liegen also gut im grünen Bereich.

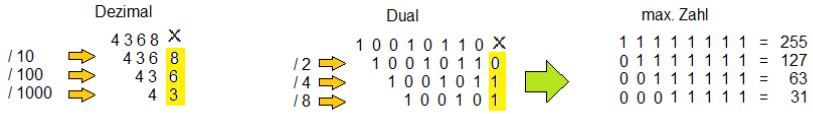
Den Namen habe ich einfach mit _S für Servo ergänzt. So braucht auch nur in der IVT der Aufruf angepasst werden. Der Befehl RJMP ISR_Timer2_OVL wird ausgeblendet und dafür RETI wieder aktiviert.

Nun lautet die Überschrift zwei Timer für einen Job. Ja, denn Timer1 ist derjenige, der Timer2 einschaltet oder besser gesagt, startet. Dazu wird im Bereich mSek. abgefragt, ob das 20mSek. Bit im Reg_A mit den Time_Flags gesetzt ist. Wird das erkannt, kopieren wir den Analogwert in das Register Servo und setzen das Interrupt Freigabebit von Timer2.

Zur besseren Übersicht stelle ich noch einmal beide beteiligten Abschnitte von Timer1 vor.

```
   LDS        Reg_A, Time_Flag
;************************* 1 msek*************************
   ORI        Reg_A, 0b00000001          ; Zeitflag 1 mSek. setzen
   INC        MilliSek                   ; Millisekunde erhöhen
   SBRS       Reg_A, 7                   ; Bit 7 in Reg_ A =1, dann übernächster Befehl
   RJMP       CHK_Ten_Milli              ; gehe zur Prüfung Überlauf
   LDS        Reg_B,  Analog_High        ; gelesener Analogwert (Poti)
   Mov        Servo, Reg_B               ;  nach Register Servo
   ANDI       Reg_A,0b01111111           ;  Zeitflag 20 mSek löschen
   IN         Reg_B, TIMSK               ; Timer Interrupt Contro Register laden
   ORI        Reg_B,0b10000000           ; Interrupt Timer 2 einschalten (Bit 7)
   OUT        TIMSK, Reg_B               ; und eintragen
Chk_Ten_Milli:
   CP         MilliSek, Zehn
   BRLO       End_ISR_Timer1
;************************* 10 msek *************************
   CLR        MilliSek
   ORI        Reg_A, 0b00000010          ; Zeitflag 10 mSek. setzen
   INC        Zehn_Milli                 ;  nächste dekade erhöhen
   SBRC       ZehnMilli,0                ;  Bit 0 gerade oder ungerade ?
   RJMP       Chk_Zehn                   ;  Zahl gerade
   ORI        Reg_A, 0b10000000          ;  Zeitflag 20 mSek. setzen
   SBI        PortC, 5                   ;  Ausgang Port C Bit 5 setzen
Chk_Zehn:
```

Der Rest läuft dann schon von allein. Aber der Wert von Analog_High könnte viel zu hoch laufen. Das Ergebnis ist ein Servo, was in einem kleinen Potibereich über den gesamten Stellweg fährt. Nun habe ich




schon angedeutet, das ein Wert in Servo von 30 ideal wäre. Bei dieser Zahl braucht es nun keine besondere Mathematik, um den Analogwert in diesen Bereich aufzulösen. Dazu wieder mal ein Ausflug in unsere Welt der dezimalen Zahl. Wenn wir mal eine vierstelige Zahl einfach nach rechts schieben und dabei die rechte Zahlhinter ein Komma schieben, kommt dieses Verschieben einer Teilung von 10 gleich. Verschieben wir eine Binärzahl nach rechts ist das auch eine Teilung, allerdings nur durch 2, da eine Binärzahl 2 in der Stelle nicht existiert und erst in der nächst höheren bewertet wird, genau wie die 10 auch. Dafür habe ich mal eine Skizze entworfen, die auch gleich etwas deutlich macht.

### *Teilen durch schieben*

Lassen wir nach einem Schieben die letzte Zahl jeweils wegfallen, ergeben sich immer die Ganzahligen Anteile einer Teilung. Im Dezimalsystem fallen die Einerstellen weg. Im Dualsystem auch, aber der Wert ist auch nur max. 1. Nehmen wir einmal die Zahl 255, das ist der größte Wert in einem Byte und schieben diesen Wert nach rechts. Wir erhalten 127, denn vorn wird das Byte mit 0 aufgefüllt. Ein weiteres Schieben bringt dann aus max.255 nur noch 63 und wenn es noch einmal durchgeführt wird bleibt von ehemals 255 nur noch 31 übrig.

Das erlaubt den Schluss, das aus Zahlen von 0 bis 255 nun Zahlen von 0 bis 31 werden. Passen wir nun die Leseroutine des Analogeinganges entsprechend an.

```
ISR_Read_ADC:
   In      Ablage_SREG, SREG
   Push    Reg_A                    ; ISR, daher Register retten
   In      Reg_A, ADCL              ; zuerst das LowByte
   STS     Analog_Low, Reg_A        ; Ablegen in einer Variablen
   In      Reg_A, ADCH              ; dann HighByte. LowByte hat keine Bedeutung
   LSR     Reg_A                    ;  max. 127
   LSR     Reg_A                    ; max-63
   LSR     Reg_A                    ; max.31
   CPI     Reg_A, 30                ; rechter  Anschlag (Grenzwert max.)
   BRLO    Chk_2
   LDI     Reg_A 30                 ; maximalen Grenzwert eintragen
Chk_2:
   CPI     Reg_A, 2                 ; linker Anschlag (Grenzwert min.)
   BRGE    Set_Analog
   LDI     Reg_A, 2                 ; minimalen Grenzwert eintragen
Set_Analog:
   STS     Analog_High, Reg_A       ; Ablegen in einer Variablen
   POP     Reg_A
   OUT     SREG, Ablage_SREG
RETI
```

Durch die drei LSR-Befehle wird der Potiwert nur noch vo 0 bis 31 bewertet. Damit fahren wir aber den Servo sowohl in die untere als auch in die obere Begrenzung. Mit einer einfachen Programmierung wird nun noch eine Begrenzung für den unteren und eine Begrenzung für den oberen Analogwert eingebaut und fertig. Nach wie vor sollte der Analogwert in der Siebensegmentanzeige aufzuschalten sein und so ist auch diese Programmierung leicht zu überprüfen..

Berechnen wir einmal die Laufzeit der ISR von Timer 2 denn diese generiert alle 30 µSek. einen Interrupt. Schaft sie in diesem Zeitfenster ihre Arbeit und bleibt noch für den restliche Programmbearbeitung  Zeit?

```
;*************************************************************
;*      Interrupt  alle 32 µSek                              *
;*     Vorteiler 256                                         *
;*     OCR Register 2 (CTC)                                 *
;*************************************************************
ISR_Timer2_Cmp_S:                ;  Für Aufruf und Rücksprung ca. 8 Takte
  Push  Reg_A                    ; 2 Takte
  In      Ablage_SReg, SREG      ; 1 Takt
  DEC   Servo                     ; 1 Takt
  BRNE End_ISR_T2                ; 1/ 2 Takte
  CBI     PortC, 5               ; 1 Takt
  IN      Reg_A, TIMSK           ; 1 Takt
  CBR  Reg_A, 0b01111111          ; 1 Takt
  OUT   TIMSK, Reg_A             ; 1 Takt
  OUT   SREG, Ablage_SReg       ; 1 Takt
  POP    Reg_A                   ; 2 Takte
RETI
;--------------------- max. ca. 20 Takte --------------------
```

Aufruf und Rücksprung einer ISR ca. 7 Takte
Push und POP jeweils 2 Takte
7 Befehle mit jeweils einem Takt
BRNE mit max. 2 Takten

Zusammen ca.20 Takte und somit eine Laufzeit von 20 * 62,5 nSek bei 16 MHz. Mit einer Laufzeit von ca. 1,25 µSek. können wir leben.

## 2.20 Verteiler für Programmgruppen

In manchen Fällen ist abhängig von einer Zahl der Aufruf verschiedener Subroutinen erforderlich. In unserem Fall trifft das auf die Anzeige und dem gerade aktiven Programmmode zu.

Mode 0: Anzeige Uhr
Mode 1:Anzeige Stellzeit
Mode 2: Anzeige Alarmzeit
Mode 3: Anzeige Stoppuhr
Mode 4: Anzeige Rundenzähler
Mode 5: Anzeige Analogwert

In der Programmschleife Main_Loop rufen wir alle Unterprograme auf und prüfen in der Routine den Programmmode, um die richtige auszuwählen und die anderen abzuweisen. Warum kann man nicht eine Art Case-Anweisung anwenden? So würde nur die aktive Subroutine aufgerufen, was einer kleinen Zykluszeit entgegen kommt.

Die Antwort ist ein Verteiler. Der Atmega8 hat da eine ganz elegante Lösung für eine solche Aufgabe.

Wir wissen, das die Doppelregister X, Y und Z auf Adressen von Variablen eingerichtet werden können, um so auf Datenfelder zugreifen zu können. Da Variablen auch nur Adressmarken sind, stellt sich die Frage, ob nicht auch die Namen von Subroutinen, schließlich sind das ja auch nur Adressmarken, in so ein Doppelregister geladen werden können. Klar geht das. Allerdings gilt hier zu beachten, das ein Befehl immer aus einem Wort, besteht und so aufeinanderfolgende Adressmarken immer den Abstand von 2 Adressen haben. Nun zeigt ein Blick in das Datenblatt in die Rubrik Instruction Set Summary bei den Sprungbefehlen den Befehl IJMP und ICALL sowie die Erklärung dazu. Das Z-Register liefert dem Prorammcounter (PC) die Zieladresse. Die Information Indirect Jump to (Z). Aber was nutzt es uns?

Angenommen, wir schreiben 5 Sprungbefehle hintereinander und setzen ein Adresslabel vor den ersten Sprung. Nehmen wir weiter einmal an, diese Anzahl der Sprungbefehle fangen bei Adresse 48 an.

Dann steht im Speicher

```
48:  Sprung1
49:  Sprung2
50:  Sprung3
51:  Sprung4
52:  Sprung5
53:  Sprung6
```

Mit folgenden Befehlen können wir nun die erste Adresse in das Z-Register laden

```
LDI    ZL, Low(48)
LDI    ZH, High(48)
```

Ersetzen wir die Zahl 48, kommen wir in bekannte Gefilde.

```
  LDI    ZL, Low(Sprungtabelle)
  LDI    ZH, High(Sprungtabelle)
Sprungtabelle:
  RJMP     Prg_1
  RJMP     Prg_2
  RJMP     Prg_3
  RJMP     Prg_4
  RJMP     Prg_5
  RJMP     Prg_6
```

Weiterhin wissen wir, das unser Programm den Modus in der Variablen Anz_Mode mit einem Wert zwischen 0 und 5 hält. Diese Variable laden wir in das Register A und addieren den Wert zum Z Register.

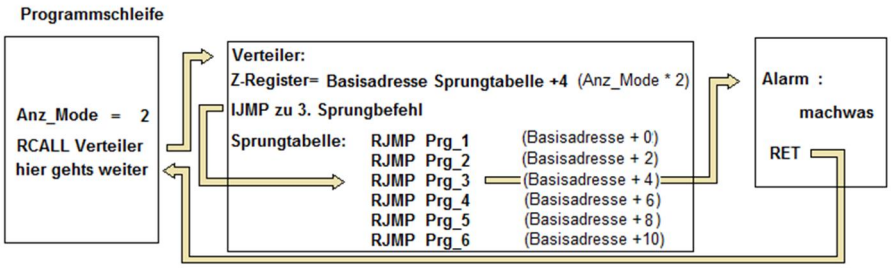
```
ADD    ZL, Reg_A
ADC    ZH, Zero
```

Somit zeigt die Adresse im Z Register auf den Sprung zu einer Subroutine, die den gewünschten Programmteil ausführt. Zusammenhängend ist die Befehlsfolge verständlicher:

```
Verteiler:
    LDI     ZL, Low(Sprungtabelle)          ; Adresse Anfang Sprungtabelle
    LDI     ZH, High(Sprungtabelle)
    LDS     Reg_A, Anz_Mode                 ; aktuellen Programmmode laden
   CPI      Reg_A, 6                        ; max. Programmteile abfragen
   BRLO     Set_Adresse                     ; kleiner, dann alles ok
   LDI      Reg_A, 0                        ; wenn Wert überschritten, reage 0 ein
Set_Adresse::
   ADD      ZL, Reg_A                       ;  und zur Anfangsadresse addieren
   ADC      ZH, Zero                        ; Übertrag in höherwertige Adresse
   IJMP                                     ; Sprung zur Adresse im Z Register
Sprungtabelle:                              ; Sprünge zu den Programmteilen
    RJMP    Uhr
    RJMP    Stellen
    RJMP    Alarm
    RJMP    Stoppuhr
    RJMP    Runden
    RJMP    Analog
RET
```

Das RET am Ende wäre nicht erforderlich, dient aber hier dem sichtbaren Ende der Routine. Ja, aber diese Routine wird doch mit RCALL aufgerufen, da braucht es doch den RET. Nun ja, sicher, aber nicht hier. Diese Tabelle funktioniert ähnlich der Interrupt Vector Table. Dort sind auch nur RJMP-Befehle zu den Interrupt Service Routinen. Der RETI erfolgt am Ende der ISR.

Hier erfolgt der RET am Ende der Routinen Uhr, Stellen etc. Die Skizze macht den Ablauf deutlich.




*Ablauf über Sprungtabelle*

Im Verteiler werden die Gruppenroutinen nicht mit RCALL aufgerufen, wie wir es vom Aufruf der Unterprogramme gewohnt sind, sondern mit einem RJMP, einem Sprung ohne Rückkehr. Allerdings müssen wir sicherstellen, dass der berechnete Wert im Z-Register innerhalb dieser Tabelle bleibt. Darum wird noch ein Grenzwert abgefragt, bevor der Registerinhalt verdoppelt wird.

Kommen wir nun zu den Gruppenroutinen. Diese enthalten eine Sammlung von Aufrufen der Unterprogramme, die in diesem Programmmodus aufgerufen werden müssen.

```
;*****************************************************************************
;*          Und die zugehörigen Gruppenroutinen                  *
;*****************************************************************************
;********** Aufrufe zur  Anzeige und Steuerung der Uhr *************
 Uhr::
  RCALL   Show_Uhr                  ; Uhr anzeigen
  RCALL   Wecker_Aus
RET
;-----------------------------------------------------------------
;************ Aufruf der Routinen zum Stellen der Uhr ***************
Stellen:
  RCALL   Stell_Uhr                   ; Stellzeit  Uhr anzeigen
  RCALL   Select_Ziffer
  RCALL   Chk_Blinker
  RCALL   Set_Zahl_Up               ; Tasterbedienung Uhrzeit stellen
  RCALL   Set_Zahl_Down
RET
;-----------------------------------------------------------------
;************ Aufrufe der Routinen für die Alarmfunktion ************
```

```
Alarm:
  RCALL   Show_Alarmzeit          ; Alarmzeit anzeigen
  RCALL   Set_Alarm_Up            ; Tasterbedienung Alarmzeit stellen
  RCALL   Select_Ziffer
  RCALL   Chk_Blinker
  RCALL   Alarm_On_Off
RET
;-----------------------------------------------------------------
;*********** Aufrufe der Routinen für die Stoppuhrfunktion **********
Stoppuhr:
  RCALL   Reset_Stopzeit          ; Tasterbedienung Stoppuhr
  RCALL   Stopzeit_on_Off
  RCALL   Show_Stopuhr            ; Stoppuhr anzeigen
RET
;-----------------------------------------------------------------
;************ Aufrufe der Routinen für die Rundenzählung ************
Runden:
  RCALL   Show_Runden             ; Anzeige Rundenzähler
  RCALL   Lap_Cnt_A
  RCALL   Lap_Cnt_B
  RCALL   Reset_Lap_Cnt           ; Tasterbedienung Rundenzähler
RET
;-----------------------------------------------------------------
;********* Aufrufe der Routinen für die Analogwerterfassung *********
Analog::
   RCALL   Analog_Out             ; Anzeige Analogwert
RET
;-----------------------------------------------------------------
```

Nun ist es nicht mehr erforderlich, in den Subroutinen den aktuellen Programmmodus abzufragen. Diese Prüfung wird nun entfernt und auch der Aufruf dieser Subroutinen aus der Programmschleife. Lediglich bei Chk_Blinker muss die Abfrage nach dem Programmmodus bleiben, da er im Alarm und Uhrzeit stellen unterschiedlich arbeitet.

Die Schleife unseres Programms ist nun wieder ganz schön geschrumpft und gut übersichtlich.

```
;*****************************************************
;* Schleife Hauptprogramm *
;*****************************************************
Loop:
   RCALL   Read_IO              ; Eingänge lesen
   RCALL   IO_Debounce_T        ; Entprellen mit Timer
   RCALL   IO_Event_Flanke      ; Beide Flanken erfassen
   RCALL   Chk_Time_Flags       ; Bearbeiten von Zeitereignissen
   RCALL   IO_Event_Flanke      ; Beide Flanken erfassen
   RCALL   Chk_Time_Flags       ; Bearbeiten von Zeitereignissen
   RCALL   Set_Mode             ; Anzeige umschalten
   RCALL   Verteiler            ; Sprungtabelle
   RCALL   Chk_Alarm            ; muss in jedem Mode laufen
   RCALL   Chk_Receive          ; Datenempfang prüfen
   RCALL   Is_First             ; erstes empfangenesByte prüfen und markieren
   RCALL   Is_Second            ; Datenempfang endgültig auswerten
   RCALL   Write_IO             ; und ausgeben
RJMP Loop
```

Aus der Programmschleife wird der Verteiler mit Call aufgerufen. Das zugehörige RET steht am Ende der Subroutinen der Programmgruppen.

## 2.21 EEPROM

Nun ist es leider die Eigenschaft des DSEG, das die darin enthaltenen Werte bei einem Spannungsausfall keineswegs ihre Einstellwerte beibehalten. Wenn die Variablen nicht in einer Initialisierung auf einen definierten Wert gesetzt werden, ist der Inhalt rein zufällig. Wie aber kann ich zum Beispiel den eingestellten Alarmwert so ablegen, das er bei einem Spannungsausfall erhalten bleibt ?

Die Lösung ist das ESEG, ein EEprom.

Dort können Werte hinterlegt werden, die wie auch das Programm, bei einem Spannungsausfall erhalten bleiben. Allerdings ist dieser Speicher nur begrenzt beschreibbar. Das Datenblatt spricht von 100000 Schreibzyklen. Nun, um eine eingestellte Alarmzeit zu sichern, wird dieser Wert niemals erreichbar. Aber Programmvariablen, die ständig überschrieben werden, dürfen hier nicht angelegt sein. Mehrere tausend Male würde ein Schreibzyklus pro Sekunde durchgeführt und so ist schon nach kurzer Zeit ein zerstörter EEProm das Ergebnis. Auch der Programmspeicher, das CSEG (Codesegment) ist nicht unendlich beschreibbar. Hier redet der Hersteller von nur 10000 Schreibzugriffen. Zum Experimentieren und Beschreiben sicherlich ein niemals erreichter Wert.

Einzig und allein unbegrenzt beschreibbar ist eben nur das DSEG (Datensegment), allerdings für den Preis, das alle Werte bei einem Spannungsausfall verloren sind.

Aber ist das denn wirklich so schlimm? In der Regel nicht, da die Variablen im DSEG sowieso ständig von Programm in Abhängigkeit von der Peripherie neu berechnet werden und so sind lediglich ein paar Werte wichtig. Wie diese nun in den EEProm geschrieben werden können, will ich mit dem nächsten Programmteil beschreiben.

### 2.21.1 *Datenablage im EEPROM*

Die Schreib- und Lesezugriffe auf die EEProm Speicherzellen sind nicht mit den Zugriffen auf den Speicher im DSEG zu vergleichen. Ähnlich der seriellen Schnittstelle werden die Daten unter einer eigenen Adressierung über ein Datenregister gehandelt. Man könnte sagen, das ein EEProm wie ein eigenständiger Controller sich selbt verwaltet und lediglich über eine Schnittstelle die Information bekommt, was und wohin. Die Basis ist auch im EEProm eine Adressmarke auf einen Speicherbereich. Da wir nur diese Marke brauchen, aber 4 Werte belegen müssen, wird im ESEG die Marke angelegt und dort über eine Compilerdirektive mit .db die vier Speicherzellen nicht nur reserviert, sondern auch gleich beschrieben.

```
.ESEG
Alarm_Set:    .DB    0,0,0,0
```

Die Adresse der Marke Alarm kann nun wie gewohnt einem Doppelregister zugewiesen werden.

```
LDI   ZL, Low(Alarm_Set)
LDI   ZH, High(Alarm_Set)
```

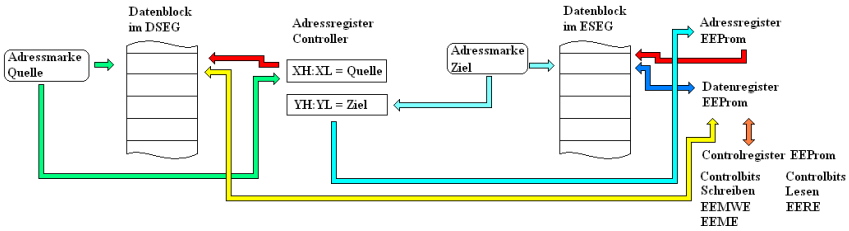
Um dorthin die eingestellte Weckzeit zu schreiben, brauchen wir auch die Adresse der Quelle.

```
LDI   XL, Low(E_Min_Alarm)
LDI  XH, High(E-Min_Alarm)
```

Wie geht es nun weiter? So wie bisher gewohnt mit Quelle laden und in Ziel speichern nicht. Die Quelle in ein Register laden, ok, das geht auch weiterhin so.

```
LD  Reg_A, X
```

Damit steht die Einerstelle vom Minutenwert der Alarmzeit im Register A. Nun soll dieser Wert in das erste Byte im ESEG-Speicher. Dazu muss der EEProm die Aufgabe bekommen, diesen Wert aus einer Übergabeschnittstelle zu übernehmen und selbstständig eintragen. Also, ein Teil der Schnittstelle ist das EEPROM-Adressregister, ein zweiter das EEProm-Datenregister. Ein Funktionsschema soll bei den Übungen ein wenig helfen, den Vorgang zu verstehen.




*Schreibvorgang EEPROM*

Das Adressregister bekommt den Wert des Doppelregisters Z, das die Zieladresse führt.

```
OUT     EEARH, ZH        ; Mit Z-Register die EEProm - Speicheradresse setzen
OUT     EEARL, ZL        ; erst ZH, dann ZL immer darauf achten
```

Das Datenregister bekommt den Wert von Register A

```
OUT     EEDR, Reg_A      ; Datenregister beschreiben
```

Nun soll der Wert im EEPRom eingebrannt werden und das ist ein Vorgang, der physikalisch etwas anders verläuft, wie eine normale Speicherzelle zu beschreiben. Deshalb ist es auch scheinbar so kompliziert. Zuerst wird das Statusregister gesichert und alle Interrupts mit der globalen Wegname der Interruptfreigabe gesperrt. Dann wird mit einem Controlbit die Schreibrichtung vorgegeben und mit einem weiteren Bit das Beschreiben der Speicherzelle im EEProm veranlasst.

```
IN      Ablage_SReg, SReg     ; Statusregister in Register laden
CLI                           ; Global Interrupts sperren. Keine Interrupts
SBI     EECR, EEMWE           ; Datenrichtung Schreiben vorbereiten
SBI     EECR, EEWE            ; und ausführen
OUT     SReg, Ablage_SReg     ; Statusregister zurückholen
```

Mit der Rückübertragung des gesicherten SREG ist auch der Interrupt wieder freigegeben, da das globale Freigabebit im Statusregister steht und noch aktiv ist.

Nun fehlt noch die Prüfung, ob ein vorhergehender Schreib- oder Lesevorgang abgeschlossen ist, denn auch wie beim USART ist der Controller schneller wie die Schnittstelle und muss hier etwas warten.

Außerdem übertragen wir doch vier Byte und deshalb programmieren wir gleich eine Schleife. Klar, das es noch ein Zählregister geben muss, um eine Abbruchbedingung setzen zu können.

```
;****************** Eintrag in EEprom schreiben *********************
;*        X ist das Aressregister für den EEProm                    *
;*        Das Register Work_A dient der Datenübergabe               *
;*****************************************************************************
Set_EEProm:
    CLR     Reg_B                       ; Datenzähler
    LDI     XL, Low(E_Min_Alarm)        ; Quelladresse
    LDI     XH, High(E_Min_Alarm)
    LDI     ZL, Low(Alarm_Set)          ; Zieladresse
    LDI     ZH, High(Alarm_Set)
Write_EProm:
    SBIC    EECR, EEWE                  ; Zugriffsbit noch gesetzt
    RJMP    Write_EProm                 ; dann warten
    OUT     EEARH, ZH                   ; Mit Z-Register EEProm  adressieren
    OUT     EEARL, ZL                   ; erst ZH, dann ZL immer darauf achten
    LD      Reg_A, X+                   ; Wert aus Adresse X laden und x erhöhen
    OUT     EEDR, Reg_A                 ; Datenregister beschreiben
    IN      Ablage_SReg, SReg           ; Statusregister sichern
    CLI                                 ; Global Interrupts sperren , keine Interrupts
    SBI     EECR, EEMWE                 ;  Schreiben vorbereiten
    SBI     EECR, EEWE                  ; und ausführen
    OUT     SReg, Ablage_SReg           ; Statusregister zurückholen, Interrupts freigeben
    ADIW    Z, 1                        ; Zieladresse erhöhen
    INC     Reg_B                       ;  Datenzähler  erhöhen
    CPI     Reg_B, 4                    ; Grenzwert erreicht ?
    BRLO    Write_EPRom                 ; nächstes Datum
RET
```

Nun wollen wir bei einem Neustart auch den Alarm in den SRAM eintragen, Dazu müssen die Speicherzellen des EEProm gelesen werden. Bis auf das Abschalten der Interrupts und das Setzen des Schreibbits sowie die Datenrichtung der Kopierroutine ist der Vorgang gleich.

```
;************************ EEPRom Daten lesen **************************
; *   X ist das Aressregister für den EEProm                          *
;*    Das Register Work_A dient der Datenübergabe                    *
;*****************************************************************************
   Get_EEProm:
    CLR       Reg_B                        ; Datenzähler
    LDI       ZL, Low(E_Min_Alarm)         ; Zieladresse
    LDI       ZH, High(E_Min_Alarm)
    LDI       XL, Low(Alarm_Set)           ; Quelladresse
    LDI       XH, High(Alarm_Set)
Read_EEProm:
    SBIC      EECR, EEWE        ; Zugriffsbit noch gesetzt
    RJMP      Read_EEProm       ; dann warten
    OUT       EEARH, XH         ; Mit Z-Register die EEProm - Speicheradresse setzen
    OUT       EEARL, XL         ; erst ZH, dann ZL immer darauf achten
    SBI       EECR, EERE        ; mit setzen von Bit EERE in Datenregister übetragen
    IN        Reg_A, EEDR       ; Register mit Inhalt vom EPRom-Datenregister  laden
    ST        +Z, Reg_A
    ADIW      X, 1
    INC       Reg_B
    CPI       Reg_B, 4          ; Grenzwert erreicht ?
    BRLO      Read_EEProm
RET
```

Nun müssen diese beiden Routinen noch aufgerufen werden. Die Stellen im Programm sollten klar sein. Die Routine Get_EEProm, also EEPRom lesen, wird in einer der Initialisierungsroutinen aufgerufen, z. B: Init_Variable.

Die Routine Set_EEProm wird In Set_Mode bei Verlassen des Stellmodus der Alarmzeit eingetragen und das ist der Wechsel von Mode 2 nach Mode 3.

**Warnung:**

**Wenn dieser Teil in das Programm übernommen wird, muß sichergestellt sein, dass es keinen Aufruf von Set_EEProm ohne Tasterbetätigung gibt. Die Quittierung des Taster-Events ist unbedingt sicherzustellen! Wird der EEProm unkontrolliert ständig beschrieben, ist der Controller innerhalb kürzester Zeit zerstört!**

Obwohl das Beschreiben des EEProm etwas gewöhnungsbedürftig scheint, sollte doch deutlich sein, wie einfach das Programm in der Programmschleife gehalten ist. Keine große und unübersichtliche Auflistung von Befehlen, sondern klare Schritte.

## 2.22 Signale mit Interrupt erfassen

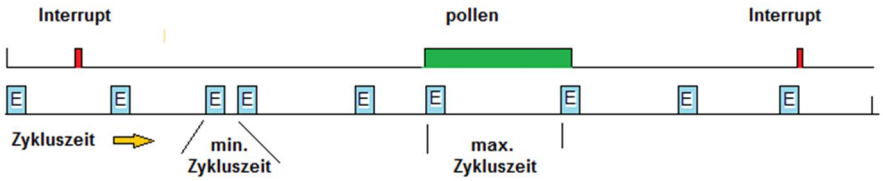
Bisher haben wir Eingänge nur einmal am Programmzyklus eingelesen und das ist auch fast immer richtig, aber es gibt auch hier keine Regel ohne Ausnahme. Der Hersteller der Controller hat dies vermutlich geahnt, als er die Controller gebaut und dem Atmega8 gleich zwei interruptfähige Eingänge verpasst hat. Beantworten wir aber zuerst einmal die Frage: „Wann ist eine Signalerfassung per Interrupt erforderlich?“

Im Normalfall, wenn Taster oder mechanische Kontakte auf einem Eingang liegen, ist in der Regel die Schaltzeit länger als die Zykluszeit. Hier reicht es völlig aus, diese Signale zu Pollen, das heißt im Programmzyklus einmal zu lesen. Da Kontakte auch prellen, dieses Thema habe ich bereits angesprochen, macht es auch wenig Sinn, dieses Prellen auch noch mit einem Interrupt zu erfassen. Wozu braucht man dann eigentlich Eingänge, die einen Interrupt auslösen. Nun, dafür gibt es mehrere Gründe. So kann man einen Controller in einen "Schlafmodus" legen. Dabei arbeitet er sein Programm nicht mehr ab und das spart Energie. Aber, um aus diesem Zustand wieder belebt zu werden, braucht es einen Wecker, und das ist dann unter Anderem auch der Interrupt von einem externen Signal an einem interruptfähigen Eingang.

Es gibt aber noch einen anderen Grund. Ich habe die Verwendung eines Controllers aufgezeigt, um bei einer Slotcar- Rennbahn die Runden zu zählen und Zeiten zu nehmen. Nun, ein Rundensignal wird aller Wahrscheinlichkeit vermutlich erst nach mehreren Sekunden auftreten, aber, und da liegt das Problem, die Signallänge ist entscheidend.

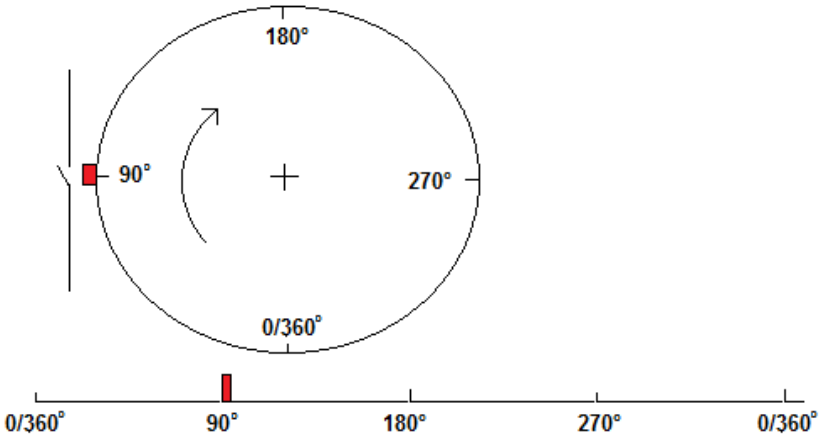
Angenommen, das Fahrzeug erreicht eine Geschwindigkeit von 30 kmh. Der Schleifer, der in der Spur geführt wird, ist ca. 15 mm, das sind 0,015 m. Die Geschwindigkeit ist 30000 m / std = 8,333 m/ sek. Auf die 15 mm Schleiferlänge bezogen ist unser Signal mal gerade 1,8 mSek. lang. Solange die Zykluszeit unter 1,8 mSek liegt, kann man diesen Eingang pollen. Aber wer sagt denn, das die max. Geschwindigkeit bei 30 km/h liegt? Vielleicht geht es auch um echte Sportflitzer, und die bringen es zum Teil bis zu 50 km/ h und mehr.

Denkbar auch ein Fahrrad, welches in den Speichen einen Magneten und am Rahmen einen Reedkontakt besitzt. Der Reedkontakt reagiert auf das Magnetfeld des Magneten und schließt wenn er in den Einflussbereich kommt. Auch hier ist das Signal nur kurz, die Periode aber lang.




*Entscheidungshilfe für Interrupt*

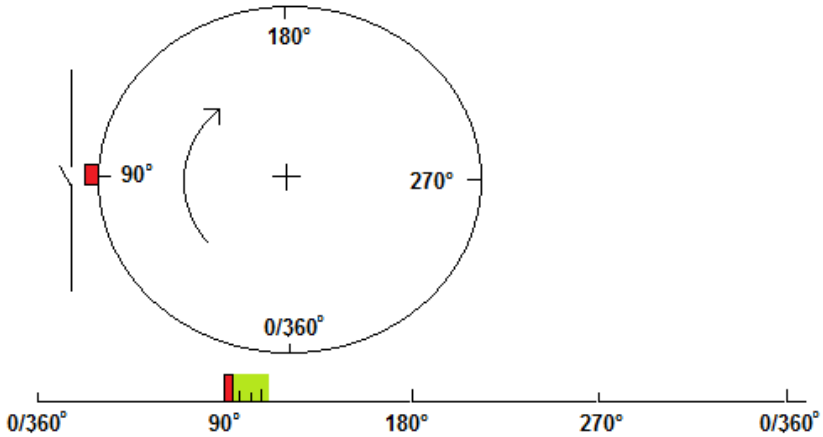
Die Skizze zeigt deutlich, ob ein Signal für einen Interrupt bestimmt oder gepollt werden kann. Das E steht hier für Einlesen der Eingänge. Der Rest der Zykluszeit ist V für Verarbeiten und A für Ausgabe. Ist das Signal länger als die längste Zykluszeit, dann ist sicher gestellt, das es einmal gelesen wird und nicht übersehen werden kann. Anders die kurzen Signalspitzen. Da ist es zwar möglich, das sie gerade beim Lesen der Eingänge auftreten, aber eben nicht sicher. Der eingefärbte Bereich markiert die Stelle, wo im Prinzip die Eingänge gelesen werden. Das linke rote Signal wird nicht erkannt. Sicher gelesen wird nur das lange grüne Signal. Da es länger als der längste Programmzyklus ansteht. Nehmen wir für ein weiteres Beispiel ein Rad. Auch hier macht die Skizze den Signalverlauf deutlich.




*Imuls Umdrehung*

Da ist der sichere Weg ein Interrupt. Das Signal muss ja nur detektiert werden. Danach ist alle Zeit der Welt, auf dieses Ereignis zu reagieren.

Sind in Interrupteingänge mit einem kontaktbehafteten Signal belegt, ist auch hier Entprellen angesagt. Allerdings macht es keinen Sinn, ein paar Millisekunden zu warten, bis das Signal stabil ist. Hier gehen wir einen anderen Weg. Sobald der Signalpegel erkannt ist, wird für eine kleine Zeitspanne dieser Zustand eingefroren. Da wir von kurzen Impulsen reden, verlängern wir diese sozusagen.




*Impuls verlängert*

Nachdem uns nun der Grund für IO-Interrupts klar ist, wenden wir uns einem praktischen Experiment zu. Wie immer müssen die Eingänge in einer Initialisierung eingerichtet werden und da ist der Blick in das Datenblatt unverzichtbar.

### *2.22.1 Interrupt einrichten*

In der Rubrik External Interrupts werden die beiden Interrupt INT0 und INT1 aufgeführt. Ein Blick auf die Pinbeschaltung liefert PD2 und PD3. Also gibt es spezielle Eingänge. Gut, das wir unsere Eingänge bereits in Port D liegen haben. So brauchen wir die Hardware nicht umbasteln.

Gut, aber es schadet nicht, noch weitere Informationen dem Datenblatt zu entnehmen. Da gibt es in der Rubrik I/O Ports einen Abschnitt Alternate Function of PortD. In diesem Abschnitt ist dann auch beschrieben, wie aus einem normalen Eingang ein Interrupteingang wird. Das Port D2 und Port D3 auf Eingang gestellt sein muss, ist klar und auch bereits eingestellt. Pullup sind eingeschaltet. Das bleibt so.

Die nächste Information erhalten wir im Abschnitt External Interrupts. Hier sind die Controll- und Steuerregister mit den Bits beschrieben, die das Verhalten von PD2 und PD3 beeinflussen. Da ist zum Beispiel das Auslöseverhalten steigendes oder fallendes Signal. Kklar,, das wir fallendes Signal wählen, denn wenn ein Taster betätigt wird, wechselt der Eingang von 1 auf 0. Im Register MCUCR werden nun die Bits ISC11 und ISC01 für einen Interrupt bei fallender Flanke gesetzt.

```
LDI    Reg_A, (1<<ISC11)| (1<<ISC01)
MOV  MCUCR, Reg_A
```

Im nächsten Schritt wird im GICR Register das Interruptflag für die Eingänge gesetzt. Tritt nun ein Ereignis ein, welches ein Signal von 1 nach 0 auslöst, wird die Stelle in der IVT adressiert, die für diesen Interrupt vorgesehen ist. Da beide Eingänge mit einem Interrupt eingerichtet sind, werden auch beide Interruptfreigabe-Bits gesetzt.

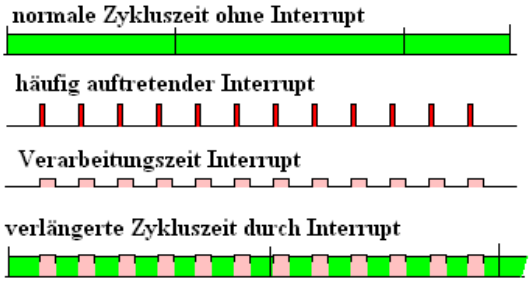
```
LDI Reg_A, (1<<Int1)|(1<<Int0)
MOV  GICR. Reg_A
```

Diese vier Befehle packen wir in eine eigene Initialisierung, welche dann vor der Programmschleife aufgerufen werden.

```
Init_Int_IO:
   LDI     Reg_A, (1<<ISC11) | (1<<ISC01)  ; Int0 und Int1 bei fallender Flanke
   MOV     MCUCR, Reg_A
   LDI     Reg_A, (1<<Int1) | (1<<Int0)    ; Aktivierung Interrupt für Int0 und Int1
   OUT     GICR, Reg_A
RET
```

### 2.22.2 Die Interruptservice Routine für IO

Nun ist eine Bearbeitung von diesem Ereignis erforderlich und das ist wie wir bereits kennen in einer ISR, die aus der IVT aufgerufen wird, zu erledigen. Hier gibt es wieder etwas zu beachten. Ist es möglich, das dieser Interrupt in einem Programmzyklus mehrfach auftritt, muss die ISR das Ereignis bearbeiten. Zum Beispiel einen Wert zählen. Eine solche Aufgabe ist aber unter Umständen kritisch und es bedarf genauester Prüfung. Vielleicht ist auch sogar eine externe Signalaufbereitung erforderlich. Dazu mal ein Signalschema, welches dieses Problem




verdeutlicht.

*Verlängerte Zykluszeit durch Interrupt*

In der obersten Zeile haben wir eine normale Zykluszeit. Fügen wir diesem Programm eine Interruptroutine zu, die in regelmäßigem Zeitintervall auslöst. Die Zeile zwei zeigt das Zeitraster und Zeile drei die Bearbeitungszeit der zugehörigen ISR. In der untersten Zeile ist nun zu erkennen, dass sich der normale Programmzyklus wesentlich verlängert hat und in einem Zyklus der Interrupt 6 mal bearbeitet wird. Wäre nur ein Interrupt ausgelöst worden, würde die Verlängerung kaum ins Gewicht fallen. Bei einem kurzen Impuls, der nach langer Pause auftritt, gibt es überhaupt kein Problem, das Signal mit einem Interrupt zu erfassen und im normalen Programm zu bearbeiten. Eine solche Aufgabe übernimmt ein Event- oder Jobflag, wie wir bereits von der Flanken- oder den Zeitflags her kennen. Haben wir aber eine schnelle Folge kurzer Impulse zu erwarten, kann das die Bearbeitung innerhalb des Programmzyklus stark verlangsamen.

Um die Wirkung der neuen ISR zu prüfen, lösen wir erst einmal die beiden Eingänge aus der Read_IO.

```
;********** Lesen aller Eingänge und Signalanpassung *************
;*    Port D                                                    *
;*      Bit 0 und 1    serielle Verbindung                      *
;*      Bit 2 bis 5    Eingänge Taster     2 und 3  nach ISR    *
;*      Bit 6          Eingang Gabellichtschranke               *
;*      Bit 7          Ausgang Anzeige                          *
;*****************************************************************************
Read_IO:                    ; Eingänge einlesen
  In     Reg_A, PInD                ; Port B lesen
  COM    Reg_A                      ; Bits drehen
 ; ANDI   Reg_A, 0b01111100          ; Ändern !
  ANDI   Reg_A, 0b01110000          ;Neu !  Nur Eingänge 4, 5 und 6 übernehmen
  LSR    Reg_A                      ; nach rechts schieben (00111110)
  LSR    Reg_A                      ; nach rechts schieben (00011111)
  STS    New_In, Reg_A
RET
;-----------------------------------------------------------------------------
```

Nun sind die Taster 3 und 4 in den Bits 0 und 1. Portpin D6 liegt im unteren Nibble auf Bit 2. Die Taster auf Port D2 und D3 sind hier nicht mehr eingebunden.

Diese beiden Eingänge werden nun in zwei separaten ISR erfasst.

```
ISR_In_D2:
    Push   Reg_A                        ; Register A sichern
    Push   Merker
    MOV    Ablage_SREG, SREG            ; Statusregister sichern
    LDI    Reg_A, 5                     ;  Entprellzeit laden
    STS    Wait_Time0, Reg_A            ; getrennte Prellzeiten für Int0 und Int1
    LDS    Reg_A, Event_To_High; gültige entprellte EIngänge laden
    ORI    REG_A, 0b00000100            ; Bit für Portpin (Taster 1 ) direkt setzen
    STS    Event_To_High, Reg_A                ; und eintragen
    LDI    Reg_A,  (1<<Int0)            ; Aktivierung Interrupt für Int0
    COM    Reg_A                        ; Register A invertieren Aktivierung ist 0
    IN     Merker, GICR
    AND    Merker, Reg_A
    OUT    GICR, Merker                 ; und Int0 deaktivieren
    OUT    SREG, Ablage_SReg
    POP    Merker
```

```
    POP     REG_A
RETI
```

Diesmal lesen wir gar nicht den Eingang, sondern weisen einfach der bereits existierenden Variablen Event_To_High das Bit für den entsprechenden Eingang mit Status 1 zu. Dann schalten wir den Interrupt für eine Zeit von 5 mSek. ab. Das bewirkt, das ein Prellen des Tasters keinen weiteren Interrupt auslöst und nach 5 mSek. sollte auch diesmal das Prellen beendet sein. Dann kommt der nächste Interrupt auch erst wieder, wenn der Taster erneut gedrückt wird. Die ISR von Int1 ist im Prinzip eine angepasste Kopie der ISR Int0 .

```
ISR_In_D3:
    Push   Reg_A                        ; Register A sichern
    Push   Merker
    MOV    Ablage_SREG, SREG            ; Statusregister sichern
    LDI    Reg_A, 5                     ;  Entprellzeit laden
    STS    Wait_Time1, Reg_A            ; getrennte Prellzeiten für Int0 und Int1
    LDI    Reg_A, Event_To_High; gültige entprellte EIngänge laden
    ORI    REG_A, 0b00001000            ; Bit für Portpin (Taster 2 ) direkt setzen
    STS    Event_To_High, Reg_A                 ; und eintragen
    LDI    Reg_A,  (1<<Int1)            ; Aktivierung Interrupt für Int1
    COM    Reg_A                        ; Register A invertieren Aktivierung ist 0
    IN     Merker, GICR
    AND    Merker, Reg_A
    OUT    GICR, Merker                 ; und Int1 deaktivieren
    OUT    SREG, Ablage_SReg
    POP    Merker
    POP    REG_A
RETI
```

Diesmal sollten wir auch zwei verschiedene Wartezeiten definieren. So ist die Zeitspanne bis zur Aktivierung des nächsten Interrupts genau festgelegt. Nimmt man nur eine Zeitvariable, ist die Wartezeit zwischen min. 5 mSek und max. 9 mSek. In unserem Experiment sicherlich ohne Bedeutung.

Die beiden ISR werden nun noch in der IVT eingetragen.

```
.org INT0addr      RJMP  ISR_In_D2     ; External Interrupt0 Vector Address  INT0
;RETI
.org INT1addr      RJMP  ISR_In_D3     ; External Interrupt1 Vector Address
```

```
;RETI
```

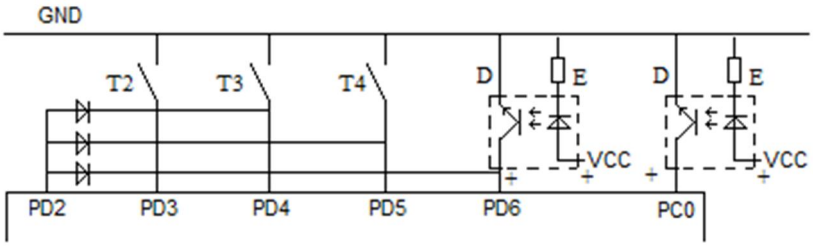
Soweit ist alles eingerichtet. Bleibt noch, die ISR nach Ablauf der Wartezeit wieder zu aktivieren. Das erledigen wir im Zeitereignis 1 mSek.

```
Event_1mSek::
;------ Bereich der Zeiterreignisbearbeitung –1 mSek---------
;--------------------------- Zeitflag quittieren -----------------------
    LDS     Reg_A, Time_Flags           ; Variable mit Zeitflags holen
    ANDI    Reg_A, 0b11111110           ; Flag für 1 mSek löschen
    STS     Time_Flags, Reg_A           ; Variable zurückschreiben
    LDS     Reg_A, Wait_Time0
    CPI     Reg_A, 0
    BREQ Chk_Wait1
    DEC     Reg_A
    STS     Wait_Time0, Reg_A
    BRNE Chk_Wait1
    LDI     Reg_A, (1<<Int0)            ; Aktivierung Interrupt für Int0
    IN      Merker, GICR
    OR      Merker, Reg_A
    OUT     GICR, Merker                ; und Int0 aktivieren

BRNE Chk_Wait1:
    LDS     Reg_A, Wait_Time1
    CPI     Reg_A, 0
    BREQ    ZeitJob_mSek
    DEC     Reg_A
    STS     Wait_Time1; Reg_A
    BRNE    ZeitJob_mSek
    LDI     Reg_A, (1<<Int1)            ; Aktivierung Interrupt für Int1
    IN      Merker, GICR
    OR      Merker, Reg_A
    OUT     GICR, Merker                ; und Int1 aktivieren
ZeitJob_mSek:

RET
```

### 2.22.3 Mehrere Eingänge mit Interrupt einlesen

Es soll ja Anwendungen geben, da müssen mehr als zwei Eingänge sofort erfasst werden. Der Controller hat aber nur D2 und D3 für eine solche Aufgabe vorgesehen. Mit einem kleinen Trick können wir auch Signale von anderen Eingängen mit einem Interrupt erfassen. Dazu wird das Signal der nicht interruptfähigen Eingänge mit einer Diode auch auf D2 oder D3 geschaltet. Dieser geht zwar für eine weitere Verwendbarkeit verloren, aber wenn man mit den Porteingängen auskommt, ist dies eine Möglichkeit.

Der Schaltplan ist einfach:




*Schaltplan Eingänge auf Interrupt*

Port D Bit 2 liefert den Int0. Sobald eines der Signalgeber durchschaltet und ein 0-Signal an einen Eingang liefert, wird dieses 0_Signal auch an PortD Bit 2 erkannt. In der ISR ist nun nur noch auszuwerten, welche Portbits zusätzlich ihren Status verändert haben. Dazu wird die ISR von D2 angepasst.

```
ISR_In_D2:
    Push   Reg_A                        ; Register A sichern
    Push   Merker
    MOV    Ablage_SREG, SREG            ; Statusregister sichern
    LDI    Reg_A, 5                     ;  Entprellzeit laden
    STS    Wait_Time0, Reg_A            ; getrennte Prellzeiten für Int0 und Int1
    LDS    Reg_A, Event_To_High; gültige entprellte EIngänge laden
    ORI    REG_A, 0b00000100            ; Bit für Portpin (Taster 1 ) direkt setzen
    STS    Event_To_High, Reg_A                ; und eintragen
    LDI    Reg_A,  (1<<Int0)            ; Aktivierung Interrupt für Int0
    COM    Reg_A                        ; Register A invertieren Aktivierung ist 0
    IN     Merker, GICR
    AND    Merker, Reg_A
    OUT    GICR, Merker                 ; und Int0 deaktivieren
    IN     Reg_A, PortD
```

```
    COM    Reg_A
    ANDI   Reg_A, 0b01110000        ; gültige Portpins maskieren
    LSR Reg_A                       ; nach rechts schieben (00111000)
    LSR Reg_A                       ; nach rechts schieben (00011100)
    LDS    Reg_B, In_To_High        ; Ereignisflags laden
    OR     Reg_A, Reg_B             ; Inhalt von Register A zumischen
    STS    In_To_High, Reg_A        ; Ereignisflags wieder ablegen
    OUT    SREG, Ablage_SReg
    POP    Merker
    POP    REG_A
RETI
```

Die Abwandlung liegt eigentlich nur in der sofortigen Übernahme der Eingänge D4-D6 in die Event-Variable nachdem die Bits an die richtige Stelle geschoben wurden. In der Read_IO müssen dann die Eingänge natürlich ausgeblendet werten.

```
Read_IO:                            ; Eingänge einlesen
    In      Reg_A, PInD             ; Port B lesen
    COM     Reg_A                   ; Bits drehen
    ANDI    Reg_A, 0b01111100       ; Nur Eingänge übernehmen
    ANDI    Reg_A, 0b00001000       ; D2, D4, D5 und D6 ausmaskieren
    LSR     Reg_A                   ; nach rechts schieben (00000100)
    LSR     Reg_A                   ; nach rechts schieben (00000010)
    STS     Akt _In, Reg_A
RET
```

Nun ist hier nur noch D3 aus Port D erhalten. Ist dieser Eingang auch einem interrupt zugeordnet, muss er ebenfalls in der Maskierung auf 0 gesetzt werden.

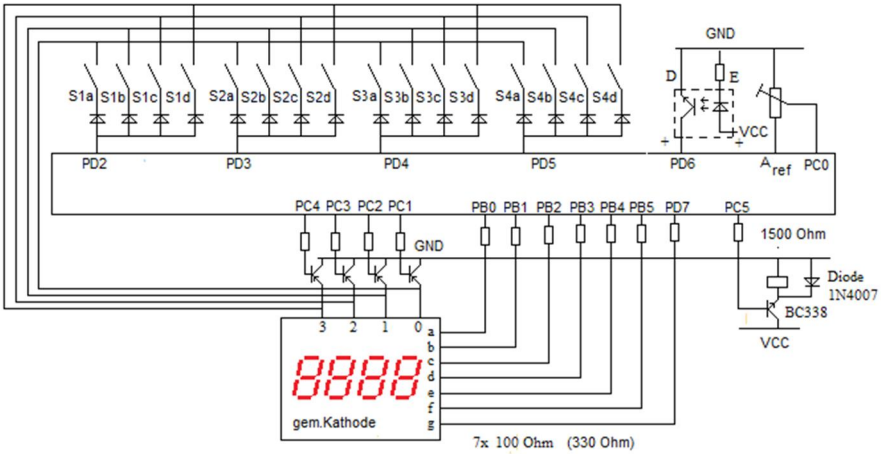
Noch wird die Verarbeitung im zyklischen Programm durchgeführt. Das reicht auch für kurze Signale, die eine längere Pausenzeit haben. So wie das Signal, welches vom Reedkontakt eines Rades von einem Fahrrad generiert wird. Hier ist nur das Erkennen des Signales wichtig. Für die Verarbeitung bleibt genug Zeit im zyklischen Programm.

Natürlich lassen sich so auch Signale von anderen Porteingängen erfassen.

### 2.22.4 Tastaturmatrix

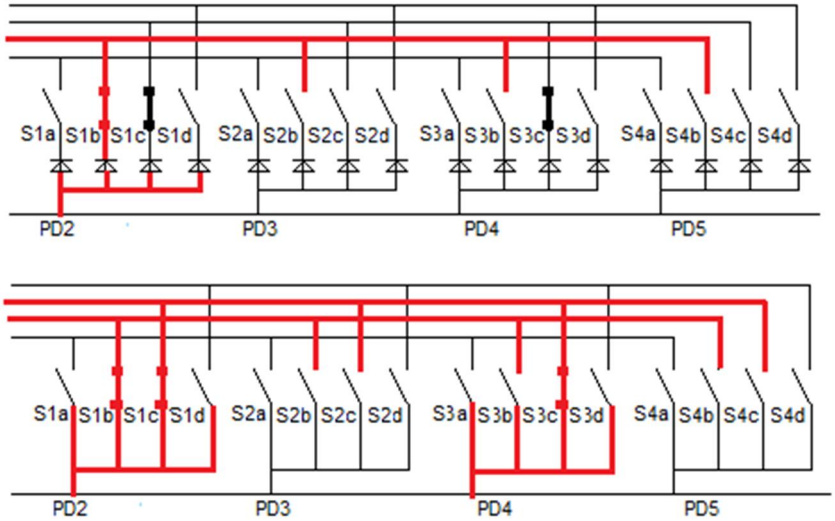
Manchmal ist es erforderlich, weitere Eingänge zu bekommen. Auch das ist ohne zusätzliche Hardware durchaus möglich und Softwaretechnisch nicht grad eine unlösbare Aufgabe.

Werfen wir doch einmal einen Blick auf den Kern der Schaltung und erweitern auf vier Tasterebenen




*Schaltplan Zielaufbau mit Tastermatrix*

Jedesmal, wenn eine Anzeigenstelle angesteuert wird, legen wir auf vier Taster einen geschalteten GND. Die Dioden verhindern, das falsche Eingänge eingekoppelt werden können, denn selbstverständlich können verschiedene Schalter in den Gruppen eingeschaltet sein. Die folgende Skizze zeigt die Zusammenhänge:




*Fehler Tastermatrix ohne Dioden*

Ein weiterer Fehler ist die zusätzliche Ansteuerung einer weiteren Anzeigestelle, denn die Kanalleitungen kommen von den Transistoren, die einen gemeinsamen GND einer Siebensegmentanzeige durchschalten. Mit den Dioden funktioniert es aber einwandfrei. Es werden zwar immer die gleichen Portpins eingelesen, aber der Zähler für die Stellenanzeige hat jedesmal einen anderen Wert. Diesen kann man zur indirekten Adressierung der Variablen nutzen, die das Abbild der Eingänge enthalten.

```
Read_IO:                          ; Eingänge einlesen
    In      Reg_A, PInD             ; Port B lesen
    COM     Reg_A                   ; Bits drehen
    ANDI    Reg_A, 0b01111100       ; Nur Eingänge übernehmen
    LSR     Reg_A                   ; nach rechts schieben (00011110)
    LSR     Reg_A                   ; nach rechts schieben (00001111)
    LD      Zl, Low(Akt_In_0)       ; Basisadresse Abbild Eingänge
    LD      Zl, High(Akt_In_0)      ; Basisadresse Abbild Eingänge
    LDS     Reg_B, Ziffer_Cnt       ; Stellenzähler Anzeige
    ADD     ZL, Reg_B               ; Zur Basisadresse addieren
    ADC     ZH, Zero                ; Register mit Wert 0 für Übertrag addieren
    ST      Z, Reg_A                ; Portwert auf adressierte Speicherstelle schreiben
RET
```

Eine so gewaltige Änderung war für die Leseroutine gar nicht erforderlich. In gleicher Weise werden nun die Entprellroutinen und Ereignisflags gesetzt. Dabei ist es erforderlich, im Variablenbereich immer die Variablen als Block einzurichten, damit über die Basisadresse indirekt adressiert

```
Akt_In_0: .Byte 1 ; Ersetzt Akt_In
Akt_In_1: .Byte 1
Akt_In_2: .Byte 1
Akt_In_3: .Byte 1

New_In_0: .Byte 1 ; Ersetzt New_In
New_In_1: .Byte 1
New_In_2: .Byte 1
New_In_3: .Byte 1

Old_In_0: .Byte 1 ; Ersetzt Old_In
Old_In_1: .Byte 1
Old_In_2: .Byte 1
Old_In_3: .Byte 1

In_To_High_0: .Byte 1 ; Ersetzt In_To_High
In_To_High_1: .Byte 1
In_To_High_2: .Byte 1
In_To_High_3: .Byte 1

In_To_Low_0: .Byte 1 ; Ersetzt In_To_Low
In_To_Low_1: .Byte 1
In_To_Low_2: .Byte 1
In_To_Low_3: .Byte 1

In_Debounce_0: .Byte 1 ; Ersetzt In_Debounce
In_Debounce_1: .Byte 1
In_Debounce_2: .Byte 1
In_Debounce_3: .Byte 1

Debounce_Cnt_0: .Byte 1 ; Ersetzt Debounce_Cnt
Debounce_Cnt_1: .Byte 1
Debounce_Cnt_2: .Byte 1
Debounce_Cnt_3: .Byte 1
```

Wenn die Variablen so gruppiert werden, ist die Anpassung überhaupt kein Problem

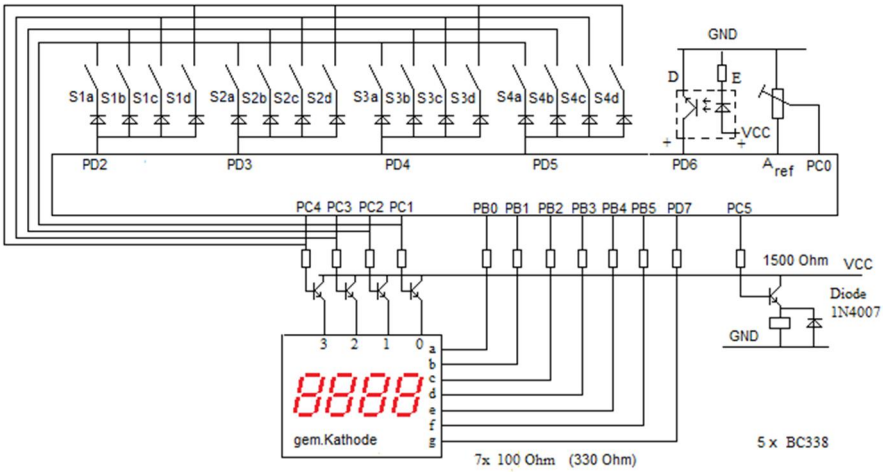
```
;*****************************************************
;*          Entprellen von Eingängen          *
;*****************************************************
IO_Debounce:
   LDI      ZL, Low(Akt_In_0)          ; Basisadresse Eingänge aktuell laden
   LDI      ZH, High(Akt_In_0)
   LDS      Reg_C, Ziffer_Cnt          ; Stellenzähler Anzeige
   ADD      ZL, Reg_C                  ; Zur Basisadresse addieren
   ADC      ZH, Zero                   ; Register mit Wert 0 für Übertrag addieren
   LD       Reg_A, Z                   ; adressierte Variable laden
   LDI      ZL, Low(In_Debounce_0)  ; Basisadresse Vergleichswert Eingänge
   LDI      ZH, High(In_Debounce_0)
   ADD      ZL, Reg_C                  ; Zur Basisadresse addieren
   ADC      ZH, Zero                   ; Register mit Wert 0 für Übertrag addieren
   LD       Reg_B, Z                   ; adressierte Variable laden
   EOR      Reg_B, Reg_A               ; Ergebnis in Register B, Register A nicht verändern
   BREQ     Chk_Debounce_Time          ; Wenn beide gleich, Prellzeit prüfen
   ST       Z, Reg_A,                  ; aktuellen Wert für nächsten Vergleich ablegen
   LDI      ZL, Low(DebounceCnt_0)   ; Basisadresse Prellzeit
   LDI      ZH, High(DebounceCnt_0)
   ADD      ZL, Reg_C                  ; Zur Basisadresse addieren
   ADC      ZH, Zero                   ; Register mit Wert 0 für Übertrag addieren
   LDI      Reg_B, 50                  ; Überwachungszeit neu setzen
   ST       Z, Reg_B                   ; Zählvariable an Adresse speichern
   RJMP     End_Debounce
   Chk_Debounce_Time:
   LDI      ZL, Low(DebounceCnt_0)       ; Basisadresse Prellzeit
   LDI      ZH, High(DebounceCnt_0)
   ADD      ZL, Reg_C                  ; Zur Basisadresse addieren
   ADC      ZH, Zero                   ; Register mit Wert 0 für Übertrag addieren
   LD       Reg_B, Z                   ; adressierte Variable laden
   CPI      Reg_B, 0                   ; Vergleich ist wichtig, Ladeanweisung setzt kein Zerobit.
   BREQ  End_Debounce                  ; Wenn zähler auf 0, keine weitere Aktion
   Dec      Reg_B
   ST       Z, Reg_B                   ; Zählvariable an Adresse speichern
   BRNE     End_Debounce
   LDI      ZL, Low(New_In_0)        ; Basisadresse Eingänge gültig laden
   LDI      ZH, High(New_In_0)
   ADD      ZL, Reg_C                  ; Zur Basisadresse addieren
   ADC      ZH, Zero                   ; Register mit Wert 0 für Übertrag addieren
   ST       Z, Reg_A                   ; Zähler hat bis 0 gezählt. Der neue Wert ist gültig
End_Debounce:
RET
```

So wie die Entprellroutine erweitert wurde, muss auch die Routine für die Ereignisflags angepasst werden.

```
;*********************************************
;*          Ereignisse Signalwechsel am Eingabe-Port.      *
;*********************************************
IO_Event_Flanke:
 LDI      ZL, Low(Old _In_0)          ; Basisadresse Eingänge alt laden
 LDI      ZH, High(Old _In_0)
 LDS      Reg_C, Ziffer_Cnt           ; Stellenzähler Anzeige
 ADD      ZL, Reg_C                   ; Zur Basisadresse addieren
 ADC      ZH, Zero                    ; Register mit Wert 0 für Übertrag addieren
 LD       Reg_A, Z                    ; adressierte Variable laden
 MOV Ablage, Reg_A                    ; alten Wert zwischenspeichern
  LDI     XL, Low(New _In_0)          ; Basisadresse Eingänge neu laden
  LDI     XH, High(New _In_0)
  ADD     XL, Reg_C                   ; Zur Basisadresse addieren
  ADC     XH, Zero                    ; Register mit Wert 0 für Übertrag addieren
  LD      Reg_B, X                    ; adressierte Variable laden
  EOR     Reg_A, Reg_B                ; EOR= Exclusiv-Oder Verknüpfung. 1 nur bei Unterschied
  AND     Reg_A, Reg_B                ; Unterschied in Reg.A, Reg_B alter Wert
  BREQ    End_Event_High              ; nur wenn ein Wechsel von 0 nach 1 erfolgt ist
  LDI     YL, Low(Event_To_High_0)    ; Basisadresse Flanke auf 1 laden
  LDI     YH, High(Event_To_High_0)
  ADD     YL, Reg_C                   ; Zur Basisadresse addieren
  ADC     YH, Zero                    ; Register mit Wert 0 für Übertrag addieren
  LD      Reg_D, Y                    ; adressierte Variable laden
  OR      Reg_A, Reg_D                ; neues Bit hinzufügen
  ST      Y, Reg_A                    ; und eintragen
End_Event_High :
  MOV     Reg_B, Ablage               ; Zwischenspeicher lesen
  LD      Reg_A, X                    ; adressiert neu gelesener Wert der Eingänge
  ST      Old_In, Z                   ; neuen Wert in Ablage „old_In“ kopieren,
; ist hier bereits erforderlich, da Reg_A überschrieben wird
 EOR      Reg_A, Reg_B     ; EOR= Exclusiv-Oder Verknüpfung. Eine 1 nur bei Unterschied
 AND      Reg_A, Reg_B                ; Unterschied in Reg.A, alter Wert in Reg_B
 BREQ End_Event_Flanke                ; nur wenn ein Wechsel von 0 nach 1 erfolgt ist
  LDI     YL, Low(Event_To_Low_0)     ; Basisadresse Flanke auf 1 laden
  LDI     YH, High(Event_To_Low_0)
  ADD     YL, Reg_C                   ; Zur Basisadresse addieren
  ADC     YH, Zero                    ; Register mit Wert 0 für Übertrag addieren
  LD      Reg_B, Y                    ; adressierte Variable laden
  OR      Reg_A, Reg_B                ; neues Bit hinzufügen
 ST       Y, Reg_A                    ; und eintragen
End_Event_Flanke:
```

RET

Allerdings gibt es auch ein paar funktionelle Abweichungen. Die Taster werden bei einer blinkenden Anzeige nicht mehr sofort reagieren, sondern müssen festgehalten werden. Der Grund: das “0”-Signal an die Anzeige wird für die Zeit “Anzeige aus” unterdrückt. Dieses Problem ließe sich durch eine weitere externe Schaltstufe mit Transistoren und einem zusätzlichen Ausgang lösen. Allerdings werden dann die Ausgänge für die Segmente invertiert und die Tasterkanäle direkt von den Ausgängen abgegriffen, um die gleiche Funktionalität zu behalten. Eine Abhilfe schafft eine Anzeige mit gemeinsamer Anode. Dafür werden alle Ausgänge für die Anzeige einfach invertiert und die Tasterreihen direkt an den Ausgang für die Gemeinsamen gelegt. Das 0 – Signal des Ausgangs steuert den Transistor auf und gleichzeitig wird auf die Tastergruppe ein GND geschaltet.




*Schaltplan Zielaufbau mit Tastermatrix und Anzeige mit gem. Anode*

## Abschlussbetrachtung

Sicherlich sind nicht alle Themen besprochen, nicht alle Fragen baeantwortet. Oftmals ist auch eine eigene Recherche im Internet erforderlich. Dennoch werden mit diesem Buch Grundsteine für die Arbeit mit PC und Mikrocontroller gelegt. Die vorgestellten Programmstrukturen sind Basis für eigene Projekte. Auch wenn ein altertümlicher Controller, eine längst veraltete Version der Entwicklungssoftware eingesetzt wurde, sind die beschriebenen Programme und Experimente mit geringen Änderungen auch auf andere, modernere Hard- und Software anzuwenden. Es war nicht Ziel dieses Buches, Visual Basic 2008 zu vermitteln, sondern den Einsatz und die Struktur der Software. So ist die Programmierung auch mit aktuelleren Ausgabeständen umzusetzen. Vielleicht ist die Bedienung etwas anders, möglicherweise sind auch andere Syntax zu beachten, aber die hier gewonnenen Erfahrungen geben das Gefühl zur Benutzung. Gleiches gilt für die Arbeit mit AVR Studio 4.x Sicher, zur Zeit ist AVR Studio x.0 aktuell, aber beim Erwerb durch den Leser vielleicht auch schon überholt. Auch Assembler als Prorammiersprache ist bei vielen verpönt. Ist doch der Programmieraufwand höher und die Portierbarkeit nach Ansicht der Experten nicht gegeben. Na ja, so ganz korrekt ist diese Aussage nicht. Aber es geht nicht um die Schulung einer Programmiersprache, sondern wie man Programme schreibt und einsetzt. Dieses Buch macht deutlich, das Visual Basic auf PC und Assembler auf Controllerebene doch einiges gemeinsam haben. Nicht im Syntax und im Befehlsumfang, aber in der Weise, wie ein Programm entwickelt und aufgebaut wird. Es ist unsere Aufgabe, Werkzeuge zu benutzen und nicht immer ist der Accuschrauber unbedingt erforderlich. Manchmal tut es auch ein simpler Schraubendreher, um mal einen Vergleich mit Hochsprache und Assembler aufzuzeigen. Das Ergebnis zählt.