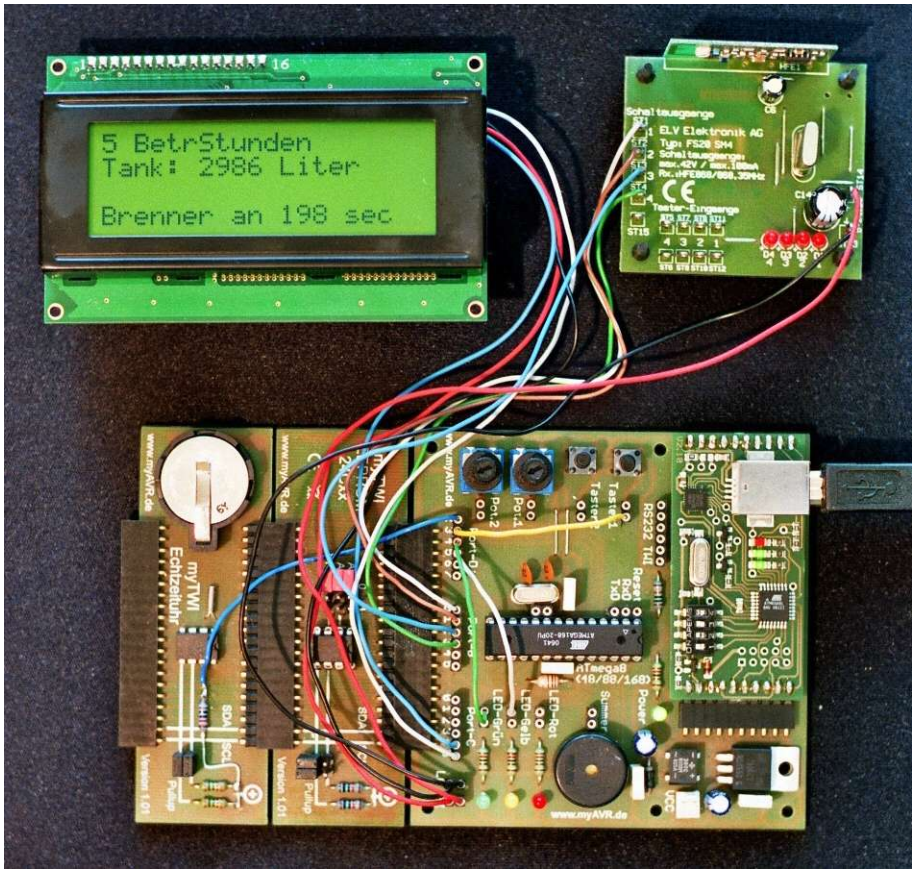
Klaus Bußmeyer

Heizölverbrauch, ein Mikroprozessor-Projekt von der Idee bis zur Fertigstellung



Die Erfassung und Auswertung des Verbrauchs eines Heizölbrenners mit einem Mikroprozessor ATmega168, programmiert in Bascom, unter Verwendung von I²C-Komponenten

Manuskript

Schenefeld, November 2009

BussmeyerMikro@aol.com

Inhalt

Vorwort

Bei dem vorliegenden Artikel handelt es sich um um die Zusammenfassung einer Dokumentation, die bei der Einarbeitung des Verfassers in das Thema „AVR-Mikroprozessoren“ entstanden ist. Dabei war der konkrete Anwendungs-Hintergrund zunächst eigentlich nebensächlich. Es wurde schließlich die Anwendung „Erfassung und Auswertung des Heizölverbrauchs“ gewählt, weil diese technisch für jedermann leicht überschaubar ist. Schließlich handelt es sich lediglich um die Erfassung der Ein- und Ausschaltzeiten eines Ölbrenners. Hierfür bedarf es auf der Eingabeseite nur eines Kontaktes, dessen Schließzeiten im Mikroprozessor erfasst werden müssen. Auf der Ausgangsseite genügt ein Display, um aktuelle Daten wie Betriebsstunden und Tankstand anzuzeigen.

Andererseits bietet diese kleine Anwendung aber genügend Möglichkeiten, ein recht breites Spektrum der verschiedenen Komponenten eines Mikroprozessor-Systems mit einzubeziehen. So behandelt dieser Artikel neben der reinen Mikroprozessor-Programmierung auch die Praxis der Interrupt-Technik, die Kommunikation mit dem PC als Host, die Erfassung von Datum und Uhrzeit, die Speicherung von Erfassungsdaten auf nicht-volatilem Speicher sowie die Darstellung von Ergebnissen auf einem LCD mit Hilfe von TWI-Komponenten, sowie die Anbindung von Funk-Komponenten an das System.

In der Praxis der Mikroprozessor-Anwendungen, insbesondere bei der Erfassung von Messwerten aus verschiedenen Quellen, wird man in vielen Fällen diese Daten zu einer zentralen Stelle übertragen müssen, um sie dort auszuwerten. Dies setzt einerseits voraus, dass eine Möglichkeit der Datenübertragung zum PC auch im Mikroprozessor besteht, und andererseits, dass auf dem PC ein Auswerteprogramm zur Verfügung steht. In diesem Artikel wird deshalb auf eine der möglichen Verbindungen zwischen Mikroprozessor und PC eingegangen, und das ist auf der Mikroprozessor-Seite der so genannte Bascom-Terminal-Emulator. Man kann mit dessen Log-Funktion auf recht bequeme Weise eine Import-Datei für „MS Excel“ erzeugen. Damit können auf dem Mikroprozessor erfasste Daten dem „MS Excel“ zugeführt werden und dort u.U. auch unter Zuhilfenahme von „VBA für Excel“ ausgewertet werden.

Entsprechend den zu behandelnden Komponenten ist dieser Artikel in mehrere Kapitel aufgeteilt. Jedes Kapitel beinhaltet jeweils eine in Betrieb zu nehmende Komponente und ein dazu gehöriges Programmbeispiel. Dabei wird mit einfachen Beispielen mit niedrigem Schwierigkeitsgrad begonnen. Mit jeder weiteren Komponente steigert sich dann der Grad der Schwierigkeit bzw. der Komplexität. Zum Schluss werden dann die einzelnen Komponentenunterstützungen zusammengefasst, erweitert, und in einer Gesamtanwendung „Erfassung des Heizölverbrauchs“ vereinigt.

Dieser Artikel wendet sich vornehmlich an Lernende und Studierende, geschrieben mit der Absicht, gemachte Erfahrungen weiterzugeben. Eine kommerzielle Verwendung bedarf meiner Zustimmung.

Informationen für Interessierte

Dieser Artikel setzt einige Grundkenntnisse der Mikroprozessor-Technik sowie der Programmierung voraus. Da der Artikel auf der Anwendung von AVR-Mikroprozessoren basiert, wäre z.B. das Studium von Dipl.-Ing. Toralf Riedel und Dipl.-Ing. Päd. Alexander Huwaldt, „**myAVR Lehrbuch Mikrocontrollerprogrammierung**“ als Voraussetzung wünschenswert, siehe www.myAVR.de**.**

Da als Programmiersprache das Bascom verwendet wird, ist eine gewisse Grundkenntnis eines der Basic-Dialekte wünschenswert. Man kann sich aber anhand der vielen kleinen Programm-Beispiele ganz gut mit Hilfe dieses Artikels in dieses Thema einarbeiten. Eine systematische Einführung für den Anfänger bietet Barry de Graaff, „**Lehr- und Experimentierbaukasten (EDB)**“, siehe www.mcselec.com**.**

Naturgemäß enthält die Materie eine Reihe von Kniffeligkeiten und Fallstricken, und ich habe mich bemüht, dafür Lösungen zu finden und diese hier zu dokumentieren. Warum sollte jeder von uns das Rad nochmals von neuem erfinden? Ich kann aber nicht ausschließen, dass mir selbst hierbei noch Fehler unterlaufen sind, und ich freue mich daher über jeden Hinweis, falls jemand auf solche Fehler stößt.

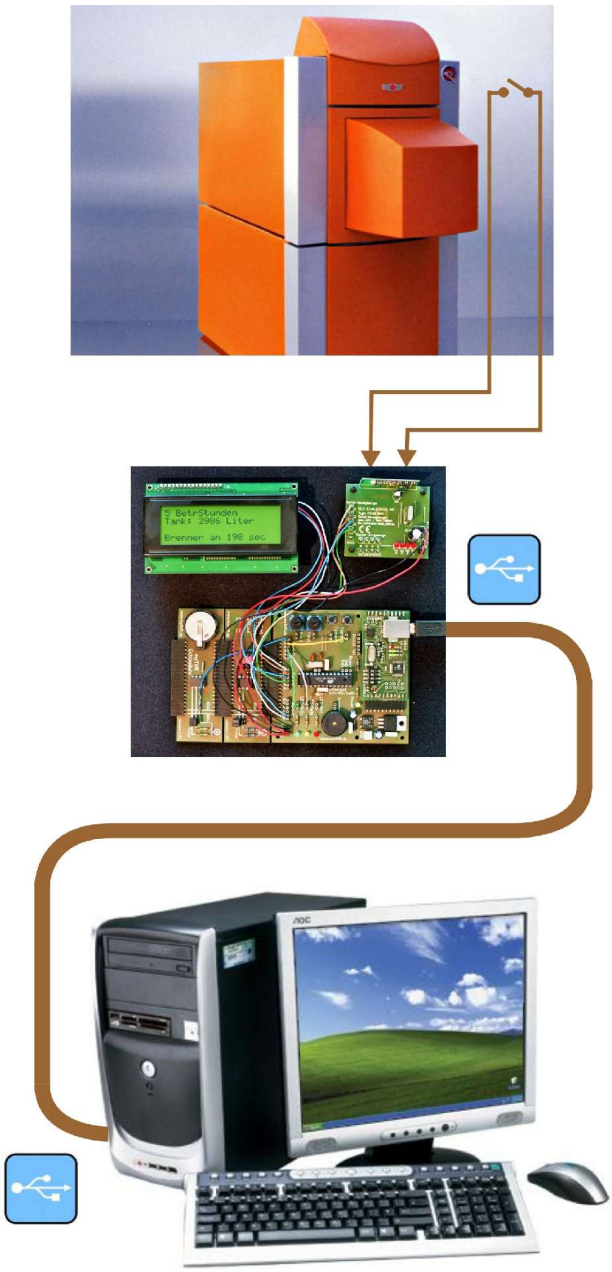
Was die im Artikel erwähnten zehn Beispielprogramme betrifft, so bin ich gern bereit, diese zur Verfügung zu stellen, allerdings gegen eine Schutzgebühr, deren Höhe ich derzeit noch nicht endgültig festgelegt habe. Das wird von der Frage abhängen, ob ich hierfür ein Gewerbe anmelden muss. Insofern kann ich zu diesem Zeitpunkt hierzu noch keine Aussage machen. Die Kosten für diese Programme werden ca. 20% der Kosten betragen, die Ihnen die die Anschaffung der Komponenten laut Beschaffungsliste entstehen. Falls Sie an den Programmen interessiert sind, wäre es hilfreich, wenn Sie mit mir Kontakt aufnehmen. Eine Kenntnis der Menge der Anfragen wäre hilfreich bei der Entscheidung, ob ein Gewerbe angemeldet werden muss. Wenden Sie sich bitte unter Angabe Ihres Namens und Ihrer Adresse an:

Klaus Bußmeyer
Friedrich-Ebert-Allee 46
22869 Schenefeld

e-mail: BussmeyerMikro@aol.com

Falls Sie an dem im Kapitel „Inbetriebnahme der Testumgebung“ erwähnten Programm „Programm_1_Erste_Inbetriebnahme“ interessiert sind, stelle ich dieses jedoch **kostenlos** als Quellcode auf Anfrage zur Verfügung. Sie können dieses Programm dann studieren und ausprobieren, die Testumgebung in Betrieb nehmen und dann entscheiden, ob Sie weitermachen wollen.

Die Grundidee



Zur Erfassung des Heizölverbrauchs wird der Einschaltzustand des Heizölbrenners dem Mikroprozessorsystem mitgeteilt, hier symbolisch dargestellt durch einen „potentialfreien Kontakt“.

Jede Schließzeit dieses Kontaktes wird im Mikroprozessorsystem nach Datum, Uhrzeit und Schließdauer des Kontaktes erfasst und gespeichert.

Die Schließzeiten werden im Mikroprozessor zu einer Gesamt-Betriebsstundenzahl aufsummiert. Über einen konstanten Umrechnungsfaktor wird der zugehörige Heizölverbrauch errechnet.

Hieraus wird schließlich der aktuelle Tankstand in Litern errechnet.

Der Ein/Ausschaltzustand, die aktuelle Betriebsstundenzahl und der aktuelle Tankstand werden kontinuierlich auf dem LCD dargestellt.

Von Zeit zu Zeit wird das Mikroprozessorsystem von der Heizunganlage gelöst und via USB an einen PC angeschlossen. Die gespeicherten Daten können nun auf den PC übertragen werden.

Hier können die Daten nun mit einem PC-Programm ausgewertet werden, z.B. mit MS Excel.

Ferner können in einem Dialog zwischen PC und Mikroprozessorsystem verschiedene Einstellungen vorgenommen werden, z.B. das Einstellen von Datum und Uhrzeit der Echtzeituhr, das Einstellen der Betriebsstunden auf einen Startwert und das Einstellen des Tankstandes auf einen Startwert.

Danach kann das Mikroprozessorsystem ohne weiteres wieder an die Heizungsanlage angeschlossen werden. Das System ist restartfest gegenüber Stromausfällen.

Beschaffungsliste

Software http://www.mcselec.com/
1 Stück BASCOM-AVR Compiler

myAVR-Komponenten http://shop.myavr.de/index.php?sp=shop/katlist.htm
1 Stück myAVR Board MK2 USB, bestückt
1 Stück myTWI Add-On EEPROM
1 Stück myTWI Add-On Echtzeituhr
1 Stück ATmega 168-20PU Mikroprozessor
1 Stück Quarz 20MHz

FS20-Komponenten http://www.elv.de/
1 Stück 4-Kanal-Schaltmodul FS20SM
1 Stück 2-/4-Kanal-Handsender FS20S4
1 Stück Klingelsignal-Erkennung FS20KSE **- oder -** (siehe Kapitel „Anschluss des Schaltmoduls FS20SM via Pin Change Interrupt“)
1 Stück 2-Kanal-Lichtsensor FS20LS

I²C-Display http://www.roboter-teile.de
1 Stück LCD-Modul LCD03

Sonstiges http://www1.conrad.de
1 Stück 24C16 I²C Serial EEPROM 16k
1 Stück Widerstand 10k
Schaltdraht ,isoliert, Innen-Ø 0,5 mm, verschiedene Farben, siehe Titelbild
1 Stück Mousepad, als Unterlage für das Board

Software-Installation der Testumgebung

Installation von Bascom-AVR

Für die Kompilation von Bascom-Programmen stellt MCS-Electronics zwei Versionen zur Verfügung;

- eine kostenfreie Demo-Version mit der Begrenzung von max. 4k Bascom Quell-Code,
- eine kostenpflichtige Voll-Version.

Da der in diesem Projekt vorgestellte Quell-Code 4k übersteigen wird, benötigen Sie die kostenpflichtige Voll-Version.

Die Installation der gelieferten CD ist selbsterklärend. Folgen Sie den vorgegebenen Installationsschritten bis das Produkt vollständig installiert ist.

Installation des USB-Gerätetreibers CP210x

Die Installation dieses Gerätetreibers erfolgt von der mit dem myAVR Board MK2 USB mitgelieferten CD. Zu beachten ist, dass die Installation in zwei Schritten erfolgt, was für Ungeübte verwirrend sein mag. Im Zweifel schauen Sie sich bitte die „Kurzdokumentation zur Installation des Gerätetreibers CP210x“ (PDF-Dokument) an. Sie findet sich auf der mitgelieferten CD auf ..\Fremd\myAVR-Board_USB-Treiber.

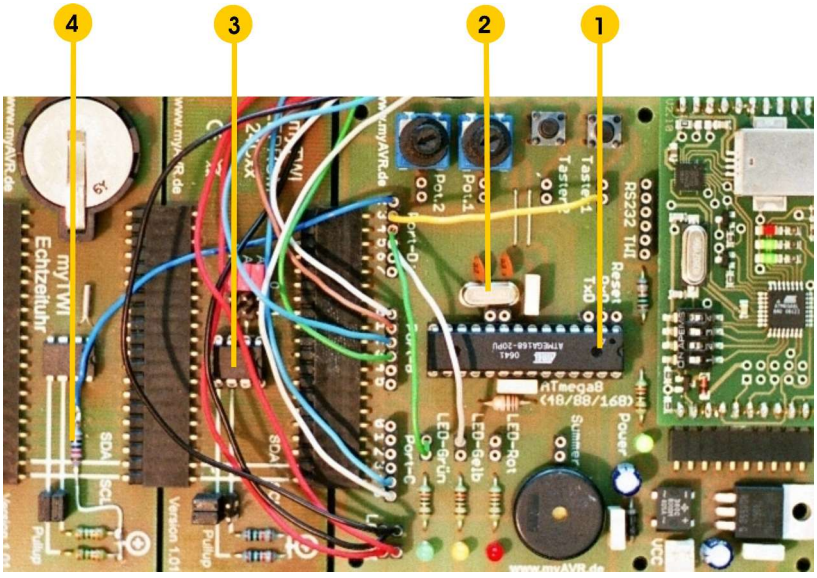
Nach erfolgter Installation sollten Sie in „Arbeitsplatz→ Systemsteuerung → System → Hardware → Gerätemanager“ unter „Anschlüsse COM und LPT“ die Eintragung „CP210x USB to UART Bridge Controller (COMx)“ finden. COMx ist hier die Nummer des COM-Anschlusses, die Ihnen die Installation automatisch zugewiesen hat. Es sei schon hier darauf verwiesen, dass die Nummer dieses COM-Anschlusses später bei der Inbetriebnahme des Bascom-Compilers mit den „Bascom-AVR IDE Optionen“ übereinstimmen muss.

Installation des myAVR ProgTool

Dieses Programm wird u.a. zum Brennen der so genannten „Fuses“ auf dem ATmega168 benötigt. Fuses sind letztlich Konfigurationseinstellungen, die dem Mikroprozessor bestimmte Betriebsweisen mitteilen. Das Tool „myAVR ProgTool“ findet man im Download-Bereich von www.myAVR.de unter http://shop.myavr.de/index.php?sp=download.sp.php. Installieren Sie dieses Tool auf Ihrem Rechner.

Installation der Hardware

Bauteile tauschen



Hinweis : Für jedes Board bzw. jede Karte gilt die Orientierung nach der Ausrichtung der Kartenbeschriftung. In diesem Sinne ist „oben“ die Kartenkante oberhalb der Kartenbeschriftung, in diesem Sinne ergibt sich „unten“, „links“ und „rechts“.

Folgende ICs bzw. Komponenten müssen ausgetauscht bzw. ergänzt werden:

1. Auf myAVR Board Atmega 8L rausnehmen, ATmega168 einsetzen (Markierung zeigt nach oben).
2. Auf myAVR Board 3,6 MHz-Quarz auslöten, 20 MHz-Quarz einlöten (Orientierung egal).
3. Auf myTWI Eeprom 24C02 durch 24C16 ersetzen (Markierung zeigt nach oben).
4. Auf myTWI Echtzeituhr Pullup-Widerstand 10k von SQW/OUT nach +5V einlöten (Widerstand zwischen zweitem Pin von oben rechts nach + , siehe Bild).

Die Verdrahtung der Komponenten

Stecken Sie die Komponenten wie auf dem Titelbild gezeigt zusammen und platzieren Sie sie auf dem Mousepad.

Sie müssen nun folgende Verdrahtungen vornehmen (orientieren Sie sich bitte am Titelbild):

1. Verbindung (gelb) zwischen Taster 1 und Port D3.
2. Verbindung (grün) zwischen LED-Grün und Port D4.
3. Verbindung (grau) zwischen LED-Gelb und Board Port D5.

4. Verbindung (blau) zwischen myTWI Echtzeituhr SQW/OUT und myAVR Board Port D2.
5. Verbindung (rot) zwischen LCD03 4-pol. Steckverbindung ganz oben (+5V) und HI auf myAVR Board.
6. Verbindung (schwarz) zwischen LCD03 4-pol. Steckverbindung ganz unten (0V) und LO auf myAVR Board.
7. Verbindung (blau) zwischen LCD03 4-pol. Steckverbindung zweite von oben (SDA) und myAVR Board Port C4.
8. Verbindung (grau) zwischen LCD03 4-pol. Steckverbindung dritte von obem (SCL) und myAVR Board Port C5.
9. Verbindung (rot) zwischen FS20 SM4 +5V und HI auf myAVR Board.
10. Verbindung (schwarz) zwischen FS20 SM4 0V und LO auf myAVR Board.
11. Verbindung (grau) zwischen FS20 SM4 Schaltausgang 1 und myAVR Board Port B0
12. Verbindung (braun) zwischen FS20 SM4 Schaltausgang 2 und myAVR Board Port B1
13. Verbindung (blau) zwischen FS20 SM4 Schaltausgang 3 und myAVR Board Port B2
14. Verbindung (grün) zwischen FS20 SM4 Schaltausgang 4 und myAVR Board Port B3

Jumpering

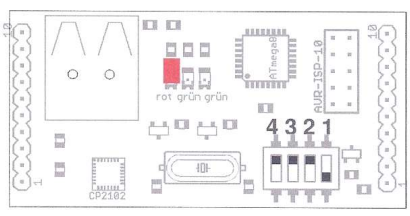
Folgende Jumper müssen entsprechend gesetzt werden:

1. Am Display LCD03 Jumper auf der Lötseite entfernen.
2. Auf myTWI Eeprom drei Adress-Jumper entfernen (für 24C16).
3. Auf myTWI Eeprom zwei Jumper für Pullup SDA, SCL entfernen (die Jumper an der myTWT Echtzeituhr bleiben drin).

Fusing

Vorab sei hier wegen schlechter Erfahrungen darauf hingewiesen, dass die „Fuse-Brennerei“ eine heikle Sache werden kann, wenn man nicht genau weiß, was das Ändern von Fuses genau bedeutet und welche Auswirkungen die Änderungen auf andere Parameter haben können. Aber das ist nun einmal die typische Situation eines Anfängers, dass einem die vielfältigen Abhängigkeiten und gegenseitigen Bedingtheiten noch nicht klar sind. Erschwerend kommt hinzu, dass aufgrund der Vielzahl der Kombinationsmöglichkeiten verschiedener Produkte kaum zuverlässige Literatur zu finden ist, die dieses komplexe Thema verständlich behandelt.

Deshalb sei hier der Rat gegeben, hier zunächst einmal „ausgetretene Pfade“ zu benutzen, bevor man eigene Wege geht. Der folgende Weg hat in der Praxis zum Erfolg geführt, analoges gilt auch für die korrespondierenden Parameter im Kapitel „Inbetriebnahme der Testumgebung“.



1. Schließen Sie das myAVR Board via USB an dem zugewiesenen Port an Ihren Rechner an.

2. Stellen Sie die DIP-Schalter des mySmartUSB MK2 Programmers auf den Modus „Programmierung“, wie im Bild dargestellt. (Schalter 4, 3, 2 oben, Schalter 1 unten). Es sollte jetzt die rote LED auf dem Programmer leuchten.

3. Starten Sie das **myAVR ProgTool**.
4. Wählen Sie unter „Hardware“ das myAVR Board MK2 USB.
5. Stellen Sie als Controller ATmega168 ein.
6. Stellen Sie bei „myAVR Board MK2 USB“ Ihren entsprechenden **COMx** ein, führen Sie den Test durch.
7. Stellen Sie auf „Brennen → Fuses brennen → Low“ ein.
8. Stellen Sie „Ext Full Swing Osc. PWRDWN/RESET 16k CK/14 CK + 65 ms“ **und** „Divide clock by 8 internally“ ein.
9. Klicken Sie auf „Jetzt schreiben“. Zum Schreiben sollten jetzt die LEDs auf dem USB-Programmer flackern.
10. Überprüfen Sie das Ergebnis mit „Hardware auslesen“.

Jetzt sollte Ihr ATmega168 den geforderten Umgebungsbedingungen angepasst sein.

Übrigens: Fuses vergisst man leicht. Falls Sie jemals einen neuen ATmega168 einsetzen, vergessen Sie bitte nicht, ihn entsprechend zu brennen.

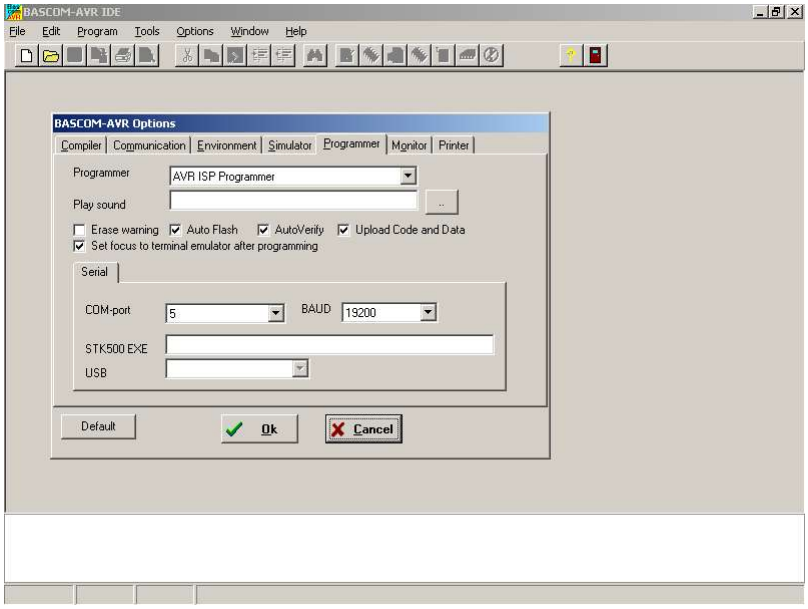
Inbetriebnahme der Testumgebung

Die Einstellung der Optionen für die IDE

Unter „Integrated Development Environment“ (IDE) wird die Gesamtheit aller Funktionen des Bascom-Compilers von MCS Electronics verstanden, die zum Editieren, Kompilieren, Brennen und Testen der selbst entwickelten Programme zur Verfügung stehen. Dazu zählt auch die Möglichkeit einer Simulation des Programmablaufs (ohne Mikroprozessor). Einzelheiten hierzu können z.B. dem Buch Marius Meissner, **„BASCOM-AVR IDE – Entwicklungsumgebung“** entnommen werden. Siehe http://www.marius-meissner.de/DE/Buch/IDE-BASCOM.htm.

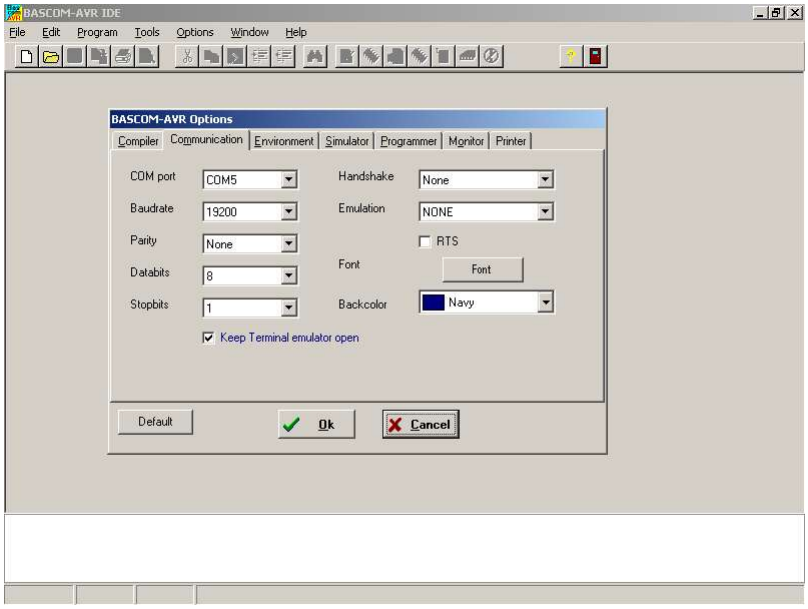
Für die hier zu diskutierende Inbetriebnahme dieser Entwicklungsumgebung mit dem myAVR Board USB 2.0 und dem USB-Gerätetreiber CP210x beschränken wir uns hier auf die hierzu relevanten Dinge.

1. Die Festlegung des Programmer-Typs



Nachdem Sie den Bascom-AVR Compiler aufgerufen haben, wählen Sie „Options → Programmer“. Stellen Sie hier sicher, dass „AVR ISP Programmer“ ausgewählt ist und das unter „COM Port“ diejenige Zahl eingetragen ist, die Ihnen bei der Installation des des USB-Gerätetreibers CP210x zugewiesen worden ist. Als Übertragungsgeschwindigkeit tragen Sie bitte 19200 Baud ein.

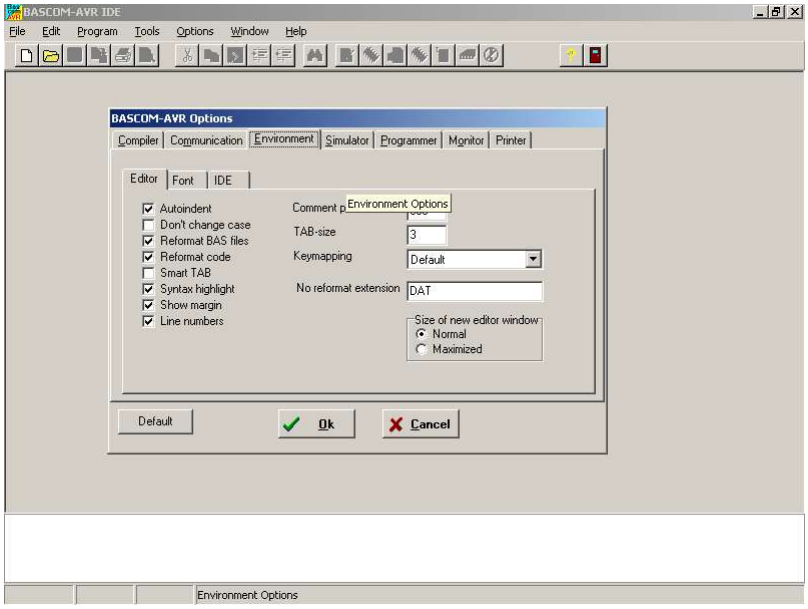
2. Die Festlegung der Communication-Parameter



Hier ist wie im vorigen Punkt der zugewiesene COM Port einzutragen. Die restlichen Parameter sind wie in der Abbildung einzutragen.

Damit ist die notwendige Anpassung des IDE bereits erledigt. Sie können die restlichen, hier nicht erwähnten Optionen nochmals durchblättern und, falls abweichend, die Baud-Eintragungen auf 19200 setzen.

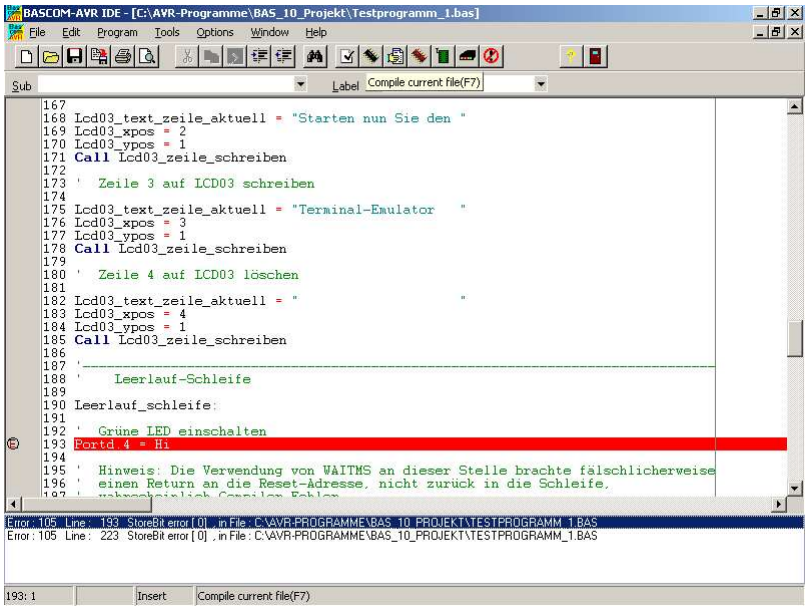
3. Die Festlegung der Environment-Parameter



Bei den Editor-Parametern wird hier „Line numbers“ empfohlen. Zwar ist die fortlaufende Zeilennummerierung im Quellprogramm letztlich Geschmackssache, sie kann aber beim Auffinden von Fehlern, die der Compiler ausgewiesen hat, hilfreich sein.

Das Kompilieren des Testprogramms

Programm: Programm_1_Erste_Inbetriebnahme
Zum Kompilieren öffnen Sie das Programm nun über „File → Open“. Obwohl die Einzelheiten des Quellcodes in den weiteren Kapiteln noch diskutiert werden, sollten Sie sich vielleicht an dieser Stelle schon etwas damit beschäftigen, dies vor allem auch deshalb, weil dieser Quellcode gewissermaßen die Keimzelle der gesamten weiteren Programmierung darstellt. Alle Erweiterungen, die zu diesem Projekt gehören, werden in diesen Quellcode eingebaut werden.



Wenn Sie nun auf „Compile current file (F7)“ drücken, sollte das hier dargestellte Bild erscheinen. Wie Sie an den Fußzeilen sehen, enthält das Programm zwei Fehler. Diese Fehler sind selbstverständlich absichtlich eingebaut, um Ihnen einen kleinen Einblick zu geben, wie man Fehler auffindet und korrigiert.

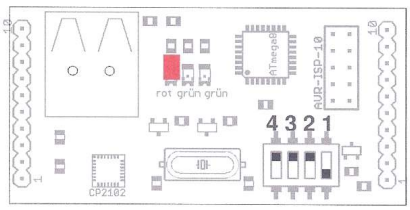
Zunächst ist der Fehler „StoreBit Error“ zugegebenerweise nicht sehr aussagekräftig.

Nun, mit dieser Problematik werden Sie zurande kommen müssen, und es geht hier um das Sammeln von Erfahrungen. Wenn Sie einen Doppelklick auf eine der Fehlerzeilen machen, wird die fehlerhafte Zeile rot unterlegt dargestellt. In diesem Falle handelt es sich offenbar um einen Ausgabebefehl auf Portd.4, und zwar in beiden Fehlerfällen.

Was könnte hier falsch sein ? Nun, ich will Sie hier nicht zu lange auf die Folter spannen: der Ausdruck „Hi“ ist kein Bascom-Schlüsselwort und muss deswegen vor seinem Gebrauch im Programm mit der Anweisung Const definiert werden. Solche Fehler unterlaufen einem Programmierer häufig mal, wenn er während der Programmierung eine neue Konstante oder Variable einführt und vergisst, diese im Definitionsteil des Programms zu deklarieren.

Wenn Sie nun in den Definitionsteil des Programms gehen, dann werden Sie in der Zeile 103 diese fehlende Konstante finden, die aber hier künstlich zur Erzeugung dieses Fehlers „auskommentiert“ worden ist. Entfernen Sie einfach das Kommentarzeichen, und das Programm wird jetzt fehlerfrei kompiliert werden.

Das Brennen des Testprogramms



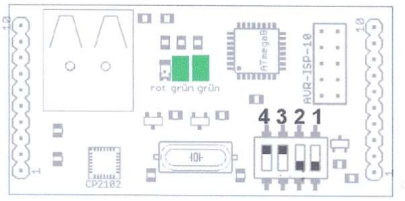
Bevor Sie mit dem Brennen beginnen, bringen Sie zunächst die DIP-Schalter auf dem Programmer in die im Bild gezeigte Stellung. Die rote LED auf dem Programmer sollte jetzt brennen. Starten Sie nun den Bascom-AVR Compiler und laden Sie das zu brennende „Programm_1_Erste_Inbetriebnahme“ mit „File → Open“. Klicken Sie nun auf „Run Programmer (F4) → Program“. Der Brennvorgang sollte jetzt starten. Falls die Nachricht „Com Port x not available“ erscheint, sind entweder die DIP-Schalter auf dem Programmer nicht richtig eingestellt, das USB-Kabel nicht richtig eingesteckt oder die „Options“ für den Programmer nicht richtig gesetzt. Prüfen Sie aber in diesem Falle zunächst über „Arbeitsplatz → Systemsteuerung → System → Hardware → Gerätemanager“, ob unter der Rubrik „Anschlüsse COM und LPT“ der CP210x USB to UART Bridge Controller (COM x) aktiviert ist. Der dort genannte COM Port x sollte mit den Programmer-

Optionen des Bascom-Compilers übereinstimmen. Wenn hier Übereinstimmung herrscht, sollte die Verbindung hergestellt sein. Im Zweifel ziehen Sie nochmals den USB-Stecker ab und stecken Sie ihn wieder auf. Die Verbindung sollte dann erneut aktiviert werden. An den Fortschrittsbalken auf dem Monitor und dem Flackern der grünen LED auf dem Programmer können Sie den Fortschritt des Brennens erkennen.

Nach dem erfolgreichen Brennen wird der Mikroprozessor automatisch gestartet.

Die Inbetriebnahme des Testprogramms

Hinweis: Bitte schließen Sie bei dieser Inbetriebnahme keine externe Stromversorgung an das Board. In dieser Betriebsweise wird das Board über das USB-Kabel mit Strom versorgt.



Zur Inbetriebnahme des Testprogramms ist es erforderlich, dass die Datenverbindung zum Terminal-Emulator aktiviert wird. Hierzu ist der DIP-Schalter 4 zunächst nach unten zu schieben, was die Stromversorgung via USB abschaltet. Dann ist der DIP-Schalter 2 ebenfalls nach unten zu bewegen. Der Programmer steht nun auf „Datenverkehr“. Nun können Sie den DIP-Schalter 4 wie im Bild gezeigt wieder nach oben bewegen, und der Mikroprozessor ist wieder eingeschaltet.

Auf dem LCD in Zeile 1 bis Zeile 3 sollte jetzt eine Start-Nachricht zu sehen sein und die grüne LED auf dem Board sollte im Sekundentakt blinken. Hiermit wäre der Start des Testprogramms jetzt erfolgreich vollzogen.

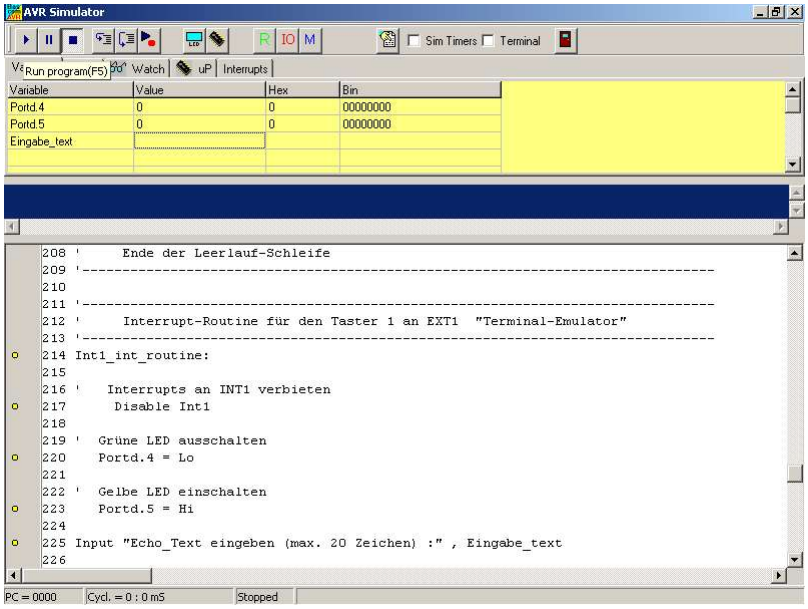
Sie können jetzt mit Hilfe des Terminal-Emulators einen Echo-Test durchführen. Falls der Terminal-Emulator noch nicht automatisch gestartet worden ist, starten Sie ihn jetzt mit „Run Terminal Emulator“. Das marineblaue Emulator-Fenster sollte jetzt auf dem Monitor zu sehen sein. Drücken Sie jetzt kurz auf den Taster 1 auf dem Board. Auf dem Terminal-Emulator sollte jetzt der Text „Echo_Text eingeben (max. 20 Zeichen):“ zu sehen sein. Ferner sollte die grüne LED auf dem Board erloschen sein und die gelbe LED auf Dauerlicht stehen. Das Programm wartet nun auf eine Eingabe am Terminal-Emulator. Nehmen Sie nun eine entsprechende Eingabe vor und drücken Sie auf die Eingabetaste.

Nun sollte die grüne LED auf dem Board wieder blinken und in Zeile 4 des LCD sollte der eingegebene Text als Echo zu sehen sein.

Wie Sie sehen, ist Letzteres leider nicht der Fall. Nun, an dieser Stelle ist bewusst ein Fehler in das „Programm_1_Erste_Inbetriebnahme“ eingebaut worden, der beim Kompilieren nicht erkannt werden kann, sondern der sich erst zur Laufzeit des Programms bemerkbar macht. Dies bietet die Gelegenheit, Sie mit dem Simulator ein wenig vertraut zu machen.

Mit Hilfe des Simulators kann man den Programmablauf simulieren, ohne dass der Mikroprozessor selbst die Programminstruktionen ausführt. Ein wesentliches Element ist hierbei die Einzelschritt-Ausführung. Wenn Sie wiederholt das Feld „Step into code (F8)“ anklicken, können Sie auf dem Monitor im Quellcode verfolgen, welche Instruktionen das Programm ausführt und ob es den gewünschten Weg geht.

Natürlich kann die „Einzelschritt-Tipperei“ in größeren Programmen recht zeitaufwändig werden. Aus diesem Grunde ist alternativ die Möglichkeit gegeben, das Programm bis zu einem bestimmten Punkt, dem „Breakpoint“ durchlaufen zu lassen. Am Breakpoint hält das Programm dann an und man kann ab jetzt mit Einzelschritten sich an die Fehlerstelle „herantasten“.



Wir wollen dies im vorliegenden Fehlerfalle einmal praktizieren. Starten Sie hierzu den Simulator mit „Simulate program (F2)“. Es erscheint auf dem Monitor ein Fenster wie auf dem Bild dargestellt. Falls das, was Sie auf dem Monitor sehen, nicht ganz dem dargestellten Bild entspricht, verschieben Sie bitte die drei Teilfenster so, dass Sie genug Platz zum Eingeben von drei Variablen haben und der Inhalt des Terminal-Emulators auf ca. 4 Zeilen begrenzt ist. So haben Sie ein Maximum an Platz, um möglichst viel vom Quellcode sehen zu können.

Geben Sie nun, wie im Bild dargestellt, die Namen von drei Variablen ein, jeweils mit Eingabetaste abgeschlossen. Klicken Sie nun auf „Run program (F5)“.

Nach einer kurzen Zeit werden Sie sehen, dass der momentane Programmzeiger zwischen zwei „Wait“-Instruktionen hin- und her springt. Die Kommentare in der Quellcode-Umgebung zeigen, dass sich das Programm offenbar in einer Warteschleife befindet und in dieser Warteschleife die grüne LED an- und ausgeschaltet wird. Aber bitte beachten Sie: das Board ist jetzt **nicht** in Betrieb !

Klicken Sie jetzt auf „Pauze (F8)“ und danach auf „Step into code (F8)“. Sie können jetzt sehen, wie das Programm in der Warteschleife herum läuft und sich der Zustand der Variablen Portd.4 fortlaufend ändert. Das Programm wird diese Schleife nur verlassen, wenn ein Interrupt auftritt. Dieser Interrupt ist normalerweise mit dem Taster 1 auf dem Board assoziert. Man kann ihn im Simulator künstlich erzeugen, wenn man den Tabulator „Interrupts“ aufruft und dort INT1 anklickt. Tun Sie dies bitte.

Sie sehen jetzt, dass das Programm (im Einzelschritt befindlich) momentan auf Zeile 1 des Programms springt. Dort liegt (in Wirklichkeit) der Verzweigungsvektor für die Interrupts. Nach ein paar weiteren Klicks auf „Step into code (F8)“ sehen Sie, dass die grüne LED ausgeschaltet wird und die gelbe LED eingeschaltet wird. Es sei an dieser Stelle jedoch nochmals daran erinnert, dass das Board jetzt nicht aktiv ist; Sie können den Zustand der LEDs nur erkennen, indem Sie sich die Variablen PortD.4 und PortD.5 auf dem Simulator ansehen. Als nächstes folgt dann im Quellcode eine „Input“-Anweisung. Einen Klick weiter sehen Sie dann, dass im marineblauen Feld des Terminal-Emulators der Text zur Eingabe-Aufforderung erscheint. Geben Sie nun (im Terminal-Emulator) einen beliebigen Text ein, abgeschlossen

mit der Eingabetaste. Wenn Sie sich die Variablen ansehen, werden Sie den eingegebenen Wert erkennen.

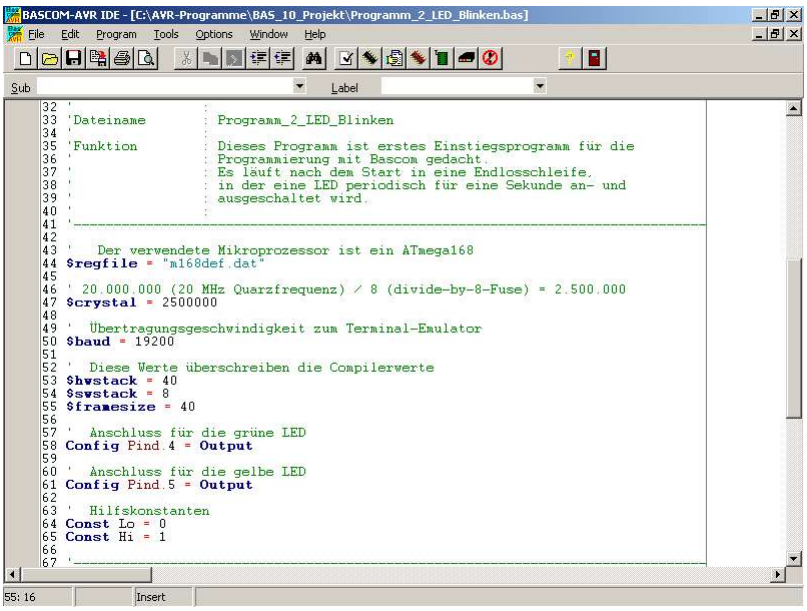
Der nächste Klick auf „Step into code (F8)“ bringt Sie nun auf eine „Return“-Anweisung. Hier wird offenbar die Interrupt-Routine verlassen und in die Warteschleife zurückgekehrt. Aber wo ist der notwendige Code für die Ausgabe des Echos auf dem LCD ? Nun, wenn sie sich die Anweisungen **nach** der „Return“-Anweisung ansehen, werden Sie sie dort finden. Offenbar wird dieser Teil des Quellcodes nie ausgeführt. Wie könnte der Fehler entstanden sein ? Ganz einfach: der Programmierer hatte zunächst in seinem Entwurf die Ausgabe auf das LCD noch nicht implementiert und deswegen an dieser Stelle die „Return“-Anweisung gesetzt. Später hat er den Code für die Ausgabe auf LCD eingefügt, jedoch das „zu frühe Return“ vergessen.

„Kommentieren“ Sie die zu frühe „Return“-Anweisung heraus, kompilieren und brennen Sie das Programm nun nochmals neu, und Sie werden sehen, dass das Echo nun auf dem LCD erscheint. Hiermit ist die erfolgreiche Inbetriebnahme des Testprogramms abgeschlossen.

Das erste eigene Bascom-Programm: Eine LED „erblinkt“ das Licht der Welt

Programm: **Programm_2_LED_Blinken**

In den folgenden Kapiteln soll die Programmentwicklung für das Projekt „Heizölverbrauch“ Schritt für Schritt dargestellt werden. Hierbei ist vorgesehen, dass der jeweils nächste Schritt auf dem vorigen aufbaut, so dass am Ende das komplette Programm steht. Die einzelnen Schritte bestehen im wesentlichen darin, dass jeweils eine weitere Programmiertechnik oder jeweils ein neuer Chip behandelt wird. Wir beginnen mit dem allereinfachsten Programm, einer Art „HalloWelt“-Programm, in dem wir eine an das Board angeschlossene LED periodisch zum Aufleuchten bringen. Typisch für dieses Programm und alle folgenden Programme ist dessen Zweiteilung in einen so genannten Deklarationsteil und den darauf folgenden eigentlichen Programmteil. Die allermeisten Definitionen im Deklarationsteil müssen zwingend dem Programmteil vorausgehen.



Insofern ist es gute Programmierpraxis, grundsätzlich alle Definitionen voranzusetzen.

Die erste ausführbare Programm-Instruktion ist dann damit automatisch der Einsprung in das Programm nach Power On/Reset. Das ist im hier dargestellten Falle die Instruktion „Portd.4 = Hi“. Es folgt eine kurze, beispielhafte Beschreibung der wesentlichen Definitionen. Näheres hierzu siehe bitte die „Help“-Funktion im Bascom-Compiler.

$regfile = „m168def.dat“

Mit dieser Angabe wird festgelegt, für welchen Mikroprozessor dieses Programm geschrieben wird. Mit dieser Angabe werden automatisch eine Reihe von Namen für Register, Ports und Adressen festgelegt, die im folgenden dann nicht extra vom Programmierer definiert werden müssen. Ein Beispiel ist die im folgenden verwendete Instruktion **Portd.4 = Hi**. Es ist gute Praxis, diese Definition zu verwenden, da es sehr unterschiedliche Atmel-Mikroprozessoren gibt, und nicht alle haben denselben Umfang an Registern, Ports und Adressen.

$crystal = 2500000

Der Name „Crystal“ ist in diesem Zusammenhang ein wenig irreführend, weil hiermit suggeriert wird, dass die Frequenz des verwendeten Quarzes auf dem Board gemeint sein könnte. Gemeint ist aber die Instruktions-Ausführungsgeschwindigkeit bzw. Taktung des Mikroprozessors, und die kann wegen einer „Fuse“ unterschiedlich von der Quarzfrequenz sein. Das Thema kann hier nicht erschöpfend behandelt werden und es muss auf die „Description“ für

den betreffenden Mikroprozessor verwiesen werden. Siehe http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf. Im vorliegenden Falle wird ein 20-MHz-Quarz verwendet, und die „divide-by-eight-Fuse“ ist gesetzt. Das führt zu einer Taktfrequenz von 2.500.000 Hz. Die Angabe ist für den Compiler wichtig, weil er z.B. hieraus errechnen kann, wie viele „NOP“-Instruktionen er ausführen muss, um eine Wartezeit von einer Sekunde („Wait 1“) zu erzeugen. Siehe Programmteil.

$hwstack, $swstack, $framesize

Siehe die „Help“-Funktion des Bascom-Compilers. Ich möchte hierzu aber zunächst eingestehen, dass ich die genaue Bedeutung dieser Anweisungen noch nicht voll verstanden habe, das gilt insbesondere für $framesize. Die „Help“-Funktion beschreibt diese Anweisungen m.E. sehr unzureichend. Das größte Manko hierbei ist, dass nicht klar ist, was passiert, wenn bei diesen Anweisungen versehentlich zu kleine Werte angegeben werden. Ich habe in meiner Programmierpraxis einige Male Laufzeitfehler ganz merkwürdiger Art erlebt. Ich habe dann versuchsweise diese Werte auf diejenigen Werte heraufgesetzt, wie sie hier angegeben sind und die Fehler waren plötzlich wie durch ein Wunder weg. Allerdings bin ich dem Problem nie ganz auf den Grund gegangen. Aber seitdem empfehle ich, die hier ausprobierten Werte zu verwenden.

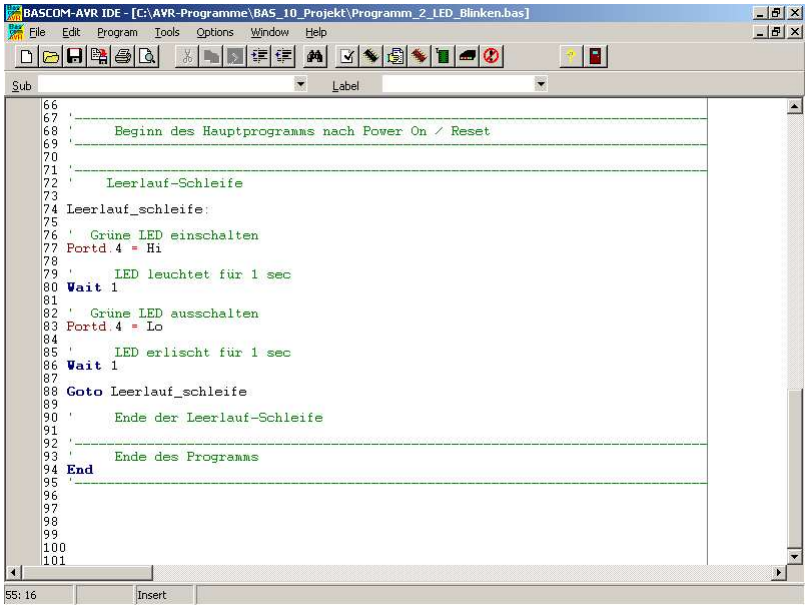
Config Pind.4 = Output

Bekanntlich sind die Ein- / Ausgabe-Pins des Mikroprozessors in Gruppen von acht Pins zu einem Port zusammengefasst. Die Hardware lässt es zu, die acht Pins eines Ports auch innerhalb eines Ports sowohl zur Eingabe als auch zur Ausgabe, also gemischt, zu verwenden. Vor der Programmausführung muss dies allerdings dem Compiler mitgeteilt werden. Die obige Anweisung ist ein Beispiel dafür, dass der Pin 4 des Ports D als Ausgabe-Pin verwendet werden soll.

Const Lo = 0

Mit dieser Anweisung kann eine Konstante definiert werden, d.h. die Konstante bekommt einen Namen, mit dem sie im Programm angesprochen werden kann. Da Namen einen wesentlich größeren Aussagewert als bloße Zahlen haben, wird die Lesbarkeit und damit das bessere Verständnis des Programms wesentlich erhöht.

Wir kommen nun zum Programmablauf, also dem ausführbaren Teil des Programm-Codes.



Portd.4 = Hi

Wie Sie auf dem Board-Photo sehen können, ist an Port D Pin 4 die grüne LED angeschlossen (grüne Drahtverbindung). Der Ausdruck „Portd.4“ ist ein Bascom-Schlüsselwort, welches dem Compiler über die „$regfile“-Eintragung als solches bekannt ist. Diese Anweisung schreibt also eine „1“ auf diesen Pin, schaltet somit die grüne LED ein.

Wait 1

Diese Anweisung lässt den Mikroprozessor für eine Sekunde in einer Leerlaufschleife verharren, das ist eine Anzahl von „NOP“- Operationen, deren Ausführung insgesamt eine Sekunde dauert. Um die Anzahl dieser „NOP“-Operationen berechnen zu können, muss der Compiler die Taktgeschwindigkeit des Mikroprozessors kennen. Diese hängt vom verwendeten Quarz oder der Frequenz des internen Oszillators und den gesetzten „Fuses“ ab. Die verwendete Taktfrequenz wurde vorher im Definitionsteil mit „$crystal“ definiert.

Der Rest des Programm-Codes ist nun selbsterklärend: Es ist hier eine Endlosschleife programmiert, in der die grüne LED jeweils für eine Sekunde an- und dann ausgeschaltet wird.

Goto Leerlauf_schleife

Hierbei handelt es sich um eine so genannte Sprung-Anweisung. Das Programm verlässt an dieser Stelle seinen linearen Ablauf und springt zu der Zeile, die vorher mit dem Label „Leerlauf_schleife“ versehen worden ist.

So, damit ist das erste Programm erfolgreich zum Laufen gebracht worden. Zur weiteren Einübung können Sie dieses Programm nun nach Lust und Laune mit den hier behandelten Elementen erweitern bzw. modifizieren. Sie können z.B. eine weitere LED zum Leuchten bringen, die Blinkzeiten verändern oder die hier dargestellte „Goto“-Schleife durch eine „Do“-Schleife ersetzen (siehe „Help“). Je nach dem Stand Ihrer Vorkenntnisse ist es sehr wichtig, dass Sie diese Basiskenntnisse festigen und vertiefen, denn wir werden mit dem Fortschreiten in den weiteren Kapiteln immer größere Schritte tun.

Programmunterbrechung auf Tastendruck: Externer Interrupt

Programm: **Programm_3_Externer_Interrupt**
In unserem vorigem Programmierbeispiel hatten wir einen ganz einfachen Anwendungsfall kennen gelernt. Es handelte sich hier um einen Vorgang, der sich periodisch wiederholt, das Aufblinken einer LED. Wir hatten gesehen, dass dies programmtechnisch in einer Endlosschleife realisiert wird, in der der Mikroprozessor wie ein Hamster im Hamsterrad im Kreis herum läuft. Der Mikroprozessor verbringt seine Zeit in diesem Beispiel weit über 99% mit Nichtstun. Er verfügt also über weit mehr Rechenkapazität als ihm hier abverlangt wird.

Wir wollen in diesem zweiten Beispiel diese gewaltige freie Rechenkapazität wenigstens etwas mehr nutzen, indem wir ihn wenigstens eine weitere Aufgabe erledigen lassen. Diese weitere Aufgabe besteht im folgenden: Der Mikroprozessor soll, während er mit dem periodischen Blinken lassen der LED beschäftigt ist, zusätzlich überwachen, ob der Taster 1 auf dem Board gedrückt worden ist. Immer dann, wenn der Taster 1 gedrückt worden ist, soll der Mikroprozessor dafür sorgen, dass die eine (grüne) LED ausgeschaltet wird und die andere (gelbe) LED eingeschaltet wird und umgekehrt.

Grundsätzlich kann der Programmierer zur Lösung dieses Problems zwei verschiedene Techniken anwenden. Die erste nahe liegende Technik wäre es, in der Endlosschleife den Schaltzustand des betreffenden Pins an dem betreffenden Port abzufragen, an dem der Taster 1 angeschlossen ist. In unserem Fall ist das Pin 3 an Port D. Wenn also Pin 3 an Port D „Hi“ ist, müsste ein Programmzweig angesteuert werden, in dem die bisher blinkende LED ausgeschaltet wird und die andere LED eingeschaltet wird. Die Technik dieses periodischen Abfragens nennt man übrigens auch „Polling“.

Die Architektur der Mikroprozessortechnik stellt dem Programmierer allerdings noch eine weitere Methode zur Verfügung, und das ist die „Interrupt“-Technik. Wir entscheiden uns für diese Technik und wollen sie hier näher kennen lernen.

Die „Interrupt“-Technik ist ein integraler Bestandteil aller Mikroprozessoren und wird als Bestandteil der Hardware zur Verfügung gestellt. Seine Funktionsweise ist so, dass wenn dem Mikroprozessor mitgeteilt worden ist, dass ein bestimmter Pin an einem Port als „Interrupt“ definiert worden ist, die Hardware des Mikroprozessors den Signalzustand an diesem Pin automatisch überwacht und im Falle eines „Interrupts“ dieses Faktum der Logik des Mikroprozessors meldet. Dies geschieht bis dahin ohne Zutun des Programmierers. Wenn nun die Logik des Mikroprozessors dieses „Interrupt“-Ereignis erkennt, dann unterbricht sie den bisherigen sequentiellen Programmablauf und der Prozessor verzweigt an eine bestimmte, vorher festgelegte Stelle im Hauptspeicher, in der die Ansprungadresse für die (vom Programmierer zu programmierende) „Interrupt“-Routine abgelegt ist.

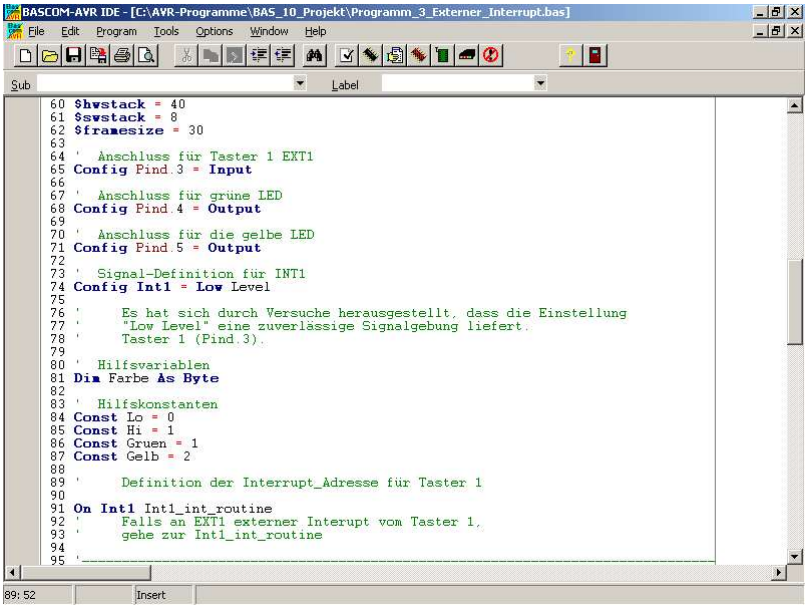
Nachdem in der „Interrupt“-Routine diejenige Arbeit erledigt worden ist, die der Programmierer für dieses Ereignis vorgesehen hat, wird mit einer speziellen Instruktion (RETI) das Ende dieser Routine erklärt, und damit kehrt der Prozessor (ohne Zutun des Programmierers) an diejenige Stelle im Programm zurück, an der es unterbrochen worden war.

Die Verwendung einer „Interrupt“-Routine erfordert also zweierlei:

- Im Deklarationsteil des Programms die Definition, welcher Pin an welchem Port der „Interrupt“-Pin sein soll (der ATmega168 hat zwei solcher Pins, INT1 und INT2), und die Angabe, wohin der Prozessor verzweigen soll, wenn ein „Interrupt“ auftritt, und

- an der eben definierten Adresse im Programm eine „Interrupt“- Routine, in der erledigt wird, was im Falle eines „Interrupts“ getan werden soll. Diese Routine muss mit einem „Return from Interrupt“ abgeschlossen werden. Im Falle von Bascom heißt diese Anweisung „Return“.

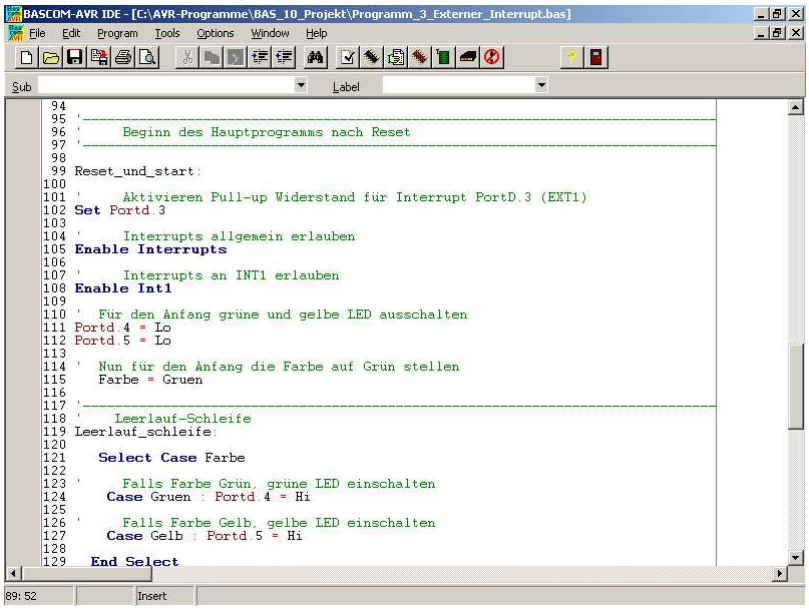
Wir wollen uns dies im Beispielprogramm ansehen.



Config Pind.3 = Input
Hier wird deklariert, dass der Pin 3 von Port D als Interrupt-Pin benutzt werden soll. Beim ATmega168 wird dieser Pin auch INT1 genannt. Er ist speziell für externe Interrupts vorgesehen (siehe die „Description“ für den ATmega168, es ist der Pin 5 des Chips).

Config Int1 = Low Level
Hier wird dem Compiler (und dem Prozessor) zusätzlich mitgeteilt, ob der

der Interrupt mit steigender, fallender Flanke oder bei „Lo“ ausgelöst werden soll. Wir haben uns für letzteres entschieden.



On Int1 Int1_int_routine
Hier wird, bezogen auf die vorige Definition, die Ansprung-Adresse im Programm bekannt gegeben, an der die „Interrupt“-Routine beginnt. Im Ausführungsteil des Programms sind gleich zu Beginn folgende Anweisungen notwendig:

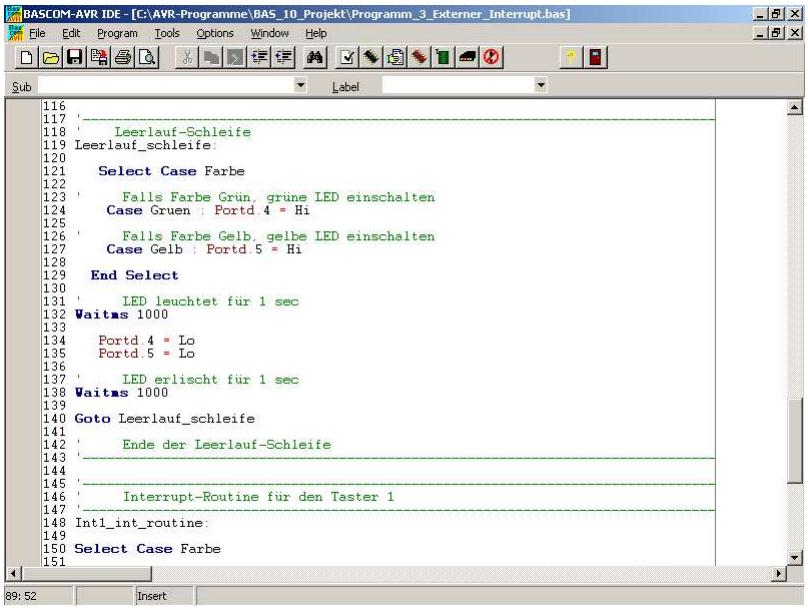
Set Portd.3
Um den Taster1 wie auf dem Board vorgesehen, unmittelbar an den Pin 3 anschließen zu können, muss mit dieser Instruktion der hardwaremäßig vorgesehene Pull-up-Widerstand aktiviert werden (Open-Collector-Ausgang)

Enable Interrupts

Nach einem „Power ON“ sind zunächst alle Interrupts des ATmega168 nicht zugelassen. Mit dieser Anweisung werden grundsätzlich Interrupts hardwaremäßig zugelassen

Enable Int1

Mit dieser Anweisung werden speziell für den PIN „INT1“ Interrupts zugelassen. Ohne diese beiden Anweisungen wird der Taster 1 also keinen Interrupt auslösen.

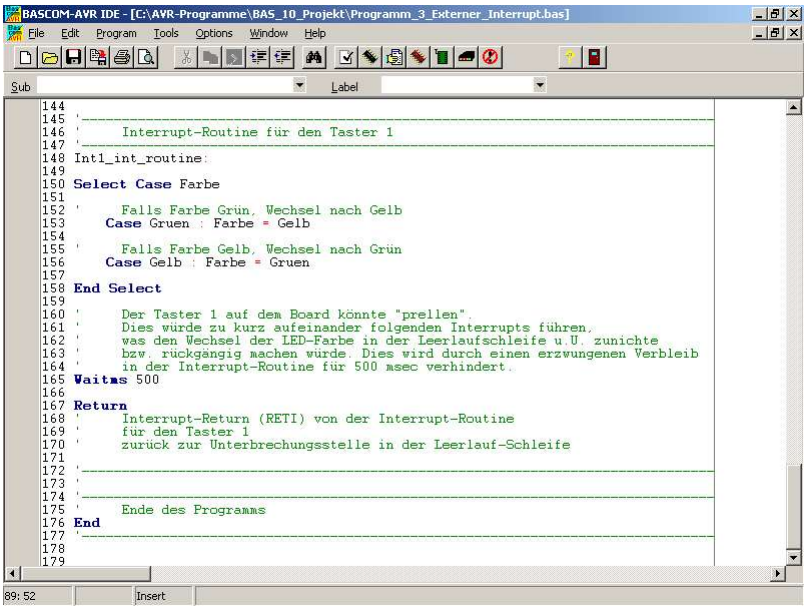


Nach diesen Anweisungen läuft das Programm nun in die Leerlaufschleife und es bleibt dem Leser überlassen, die leicht veränderte Logik zur alternativen Ansteuerung der LEDs selbst zu analysieren.

Wenn nun auf dem Board der Taster 1 kurz gedrückt wird und der Mikroprezessor dieses Signal erkannt hat, wird die in der Deklaration angegebene „Interrupt“-Routine angesteuert.

Für diesen kleinen Anwendungsfall ist ist die Logik recht einfach. Mit der „Case“-Anweisung wird die Farbe der derzeit blinkenden LED abgefragt und auf die jeweilige andere Farbe gewechselt.

Am Schluss der Routine folgt dann das bereits beschriebene „Return“, welches die „Interrupt-Routine“ beendet und eine Rückkehr des Prozessors an die Unterbrechungsstelle in der Leerlaufschleife veranlasst.



Waitms 500

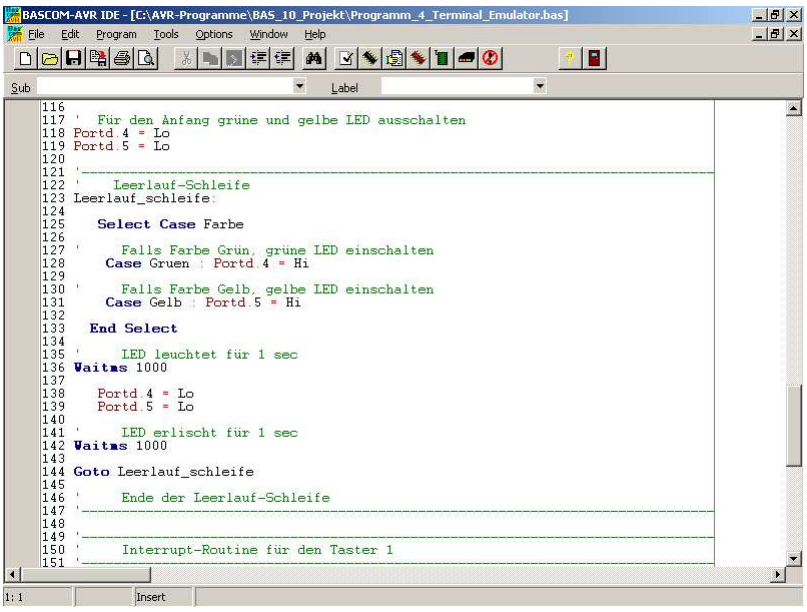
Verbliebe es noch, diese zunächst vielleicht unverständliche Anweisung (an dieser Stelle) zu diskutieren. Es mag für den Laien überraschend klingen, aber der Mikroprozessor verbringt in dieser „Interrupt“-Routine so wenig Zeit, dass er in Millisekunden wieder zurück in der Leerlaufschleife ist. Ein „Prellen“ des Tasters 1 kann also leicht zu mehreren Interrupts führen.

Mit dieser Anweisung wird die Prellzeit überbrückt.

Dialog mit dem PC via Bascom-Terminal Emulator

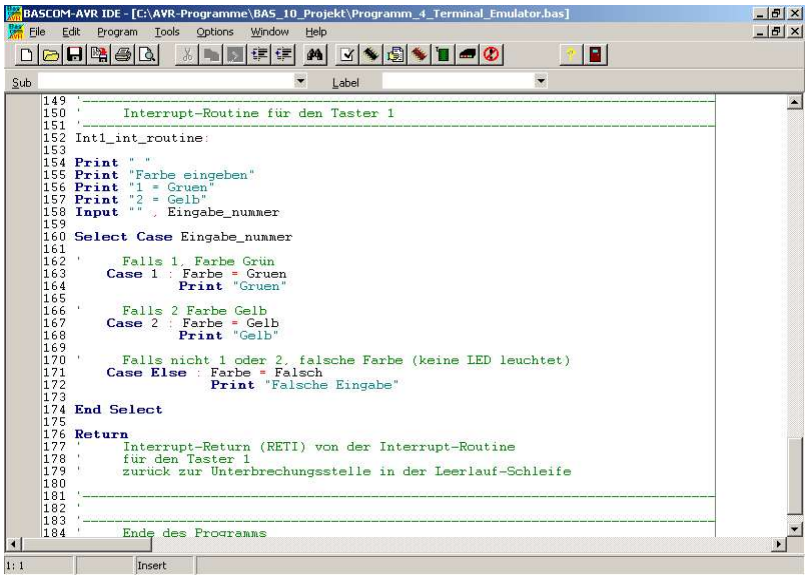
Programm: **Programm_4_Terminal_Emulator**
Beim Bascom-Terminal Emulator dient kurz gesagt die Tastatur und der Monitor des PC als Kommunikationsmedium mit dem Mikroprozessor. Mit Hilfe dieser Einrichtung ist es also möglich, Daten, die aus dem Mikroprozessor stammen, auf dem Bildschirm des PC darzustellen, und Eingaben, die von der Tastatur des PC stammen, zum Mikroprozessor zu übertragen.

Dieses Geschehen ist wohlgemerkt beim echten Betrieb des Mikroprozessors möglich, nicht etwa nur bei einer Simulation. Die Vorteile dieser Einrichtung liegen auf der Hand, gestatten sie es doch, unmittelbar zur Laufzeit des Mikroprozessors z.B. den momentanen Wert von Variablen des Mikroprozessor-Programms (durch einfaches „Ausdrucken“) auf dem PC darzustellen. Auf dem umgekehrten Wege ist es ebenso einfach möglich, Variablen im Mikroprozessor-Programm jederzeit vom PC her zu verändern. Und selbstverständlich kann man dieses Verfahren sowohl vorübergehend beim Testen als auch später für Steuerungszwecke einzusetzen und damit das Manko umgehen, dass Mikroprozessoren ja meistens gerade nicht sehr üppig mit Ein-/Ausgabegeräten ausgestattet sind.



Für die Kommunikation sind zwei Anweisungen vorgesehen, die „Print“-Anweisung für die Ausgabe von Informationen des Mikroprozessors auf dem Monitor, und die „Input“- Anweisung für die Annahme von Informationen von der Tastatur.

Das Programmbeispiel orientiert sich stark am vorigen Beispiel, indem es ebenfalls abwechselnd zwei LEDs angesteuert. In der Endlosschleife wird wie bisher je nach gewählter Farbe die entsprechende LED zum Blinken gebracht.



In der „Interrupt“-Routine wird dann mit mehreren „Print“- Anweisungen ein Auswahlmenü auf dem Monitor ausgegeben. Dann wird mit „Input“ eine Eingabe von der Tastatur erwartet.
Die Eingabe bestimmt dann die Farbauswahl („Case“-Anweisung).

Es sei an dieser Stelle darauf hingewiesen, dass die hier gewählte Programmiertechnik eigentlich eine „Todsünde“ ist, denn hier wird in einer „Interrupt“-Routine eine manuelle Eingabe abgewartet, und die hängt natürlich von der menschlichen Reaktion ab. Da der Mikroprozssor gewissermaßen in einer Warteposition „hängt“, während er sich in einer „Interrupt“-Routine befindet, steht er für andere Arbeiten z.B. im Hauptprogramm, aber normalerweise auch nicht für andere Interrupts, zur Verfügung. Von „Echtzeit“-Verarbeitung kann hier also wohl nicht mehr die Rede sein. Nun, im Falle dieses Anwendungsbeispiels können wir uns diese „Sünde“ leisten, da der gesamte weitere Anwendungsablauf von der Eingabe via „Input“ abhängig ist. Für den Mikroprozessor ist „sonst“ nichts zu tun.

Eine ganz andere Situation hätten wir jedoch beispielsweise, wenn wir auf einer weiteren Interrupt-Leitung sagen wir einen Feuermelder angeschlossen hätten. Dann würde, wenn der „Mann am PC“ die Eingabe verschläft, der Feuermelder „nicht durchkommen“. Denn es ist so, dass der Mikroprozessor „normalerweise“ keinen anderen Interrupt durchlässt, bevor nicht die „RETI“-Instruktion erreicht worden ist. Es gibt zwar grundsätzlich beim ATmega168 eine Abweichung vom „normalerweise“, und das wäre die Zulassung von „nested interrupts“ (also ein Interrupt kann den anderen momentan unterbrechen), aber das soll hier nicht diskutiert sein, weil es den gesteckten Rahmen sprengen würde. Wer will, kann dies in der „Description“ für den ATmega168 nachlesen.

Eine Lösung, die leidige Wartezeit auf eine Eingabe in einer „Interrupt“-Routine zu vermeiden, ist die Möglichkeit, in der „Interrupt“-Routine ein Ereignis-“Flag“ zu setzen, das dann in der Hauptschleife des Programms abgefragt und abgearbeitet werden kann. Auf diese Weise kann die „Interrupt“-Routine auf schnellstem Wege verlassen werden und der Mikroprozessor für andere Interrupts frei werden.

Programmierung des I2C/Serial Display LCD03

Programm: **Programm_5_I2C_Display**
Mit dem „I2C/Serial Display LCD03“ der englischen Firma Devantech Ltd., oder auch Robot Electronics genannt, wollen wir nun ein 20*4-zeiliges Display an unseren Mikroprozessor anschließen. Wie man erst beim genauen Hinsehen auf den Namen „I2C/Serial“ erkennt, bietet das LCD03 zwei verschiedene (serielle) Schnittstellen an, eine EIA-232-Schnittstelle und eine I2C-Schnittstelle. Dies jedenfalls ist mit dem Namen gemeint. Da beide Schnittstellen an dieselben Pins des LCD03 angeschlossen werden, muss dem LCD03 mitgeteilt werden, welche Schnittstelle gewählt wird. Dies geschieht mit einem Jumper auf dem LCD03. Wir entscheiden uns für die I2C-Schnittstelle, weil dies heute die Standard-Schnittstelle für Kommunikationen innerhalb eines Gerätes ist. Diese Schnittstelle kommt mit einer Versorgung von +5V aus, was für kleinere Geräte ein großer Vorteil ist. Für diesen Fall muss der Jumper, wie bereits erwähnt, gezogen werden. Für die weitere Beschreibung sollten Sie jetzt die „Technical Documentation“ des LCD03 parat haben.
Sie finden sie unter http://www.robot-electronics.co.uk/htm/Lcd03tech.htm .

Um die I2C-Schnittstelle in AVR-Bascom programmieren zu können, brauchen Sie keine detaillierten Kenntnisse von dessen Architektur. Es ist jedoch ratsam, dass Sie zumindest ein paar Grundlagen über diese Schnittstelle kennen. Sie finden diese unter
http://de.wikipedia.org/wiki/I%C2%B2C .

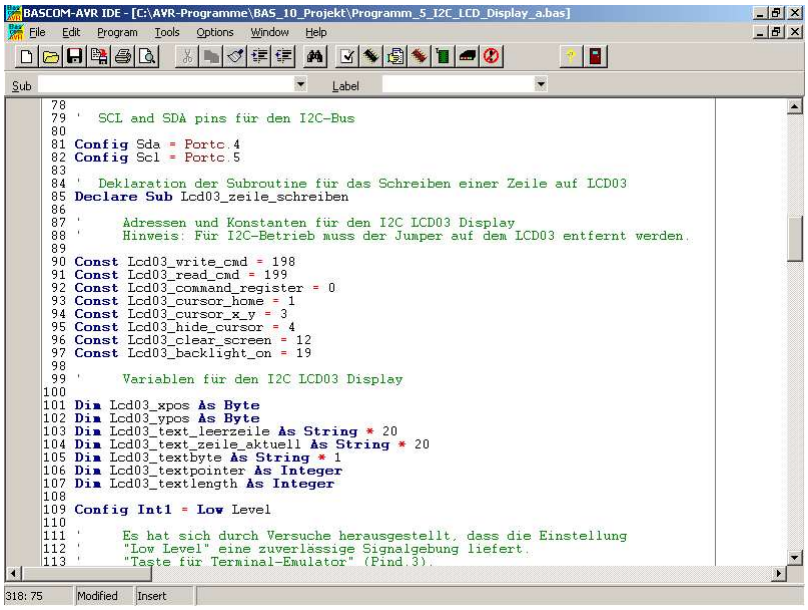
Wie bereits erwähnt, bietet das LCD03 eine Anzeige von vier Zeilen mit jeweils 20 Zeichen. Einmal zum Display geschickt, werden die Zeichen dort gespeichert und so lange angezeigt, bis sie von neuen Zeichen überschrieben werden. Das bedeutet, dass wenn man z.B. in Zeile 1 einen neuen Inhalt schreiben will, man nur diesen Zeileninhalt senden muss, die restlichen Zeilen bleiben erhalten. Dass man das Display zeilenweise anspricht, ist aber eine sehr häufige Anwendungsform. Eine entsprechende Softwareunterstützung ist daher wünschenswert.

Entsprechend dieser Forderung ist die hier vorgestellte Softwareunterstützung daher auch aufgebaut. Es ist eine Subroutine konzipiert worden, die drei Parameter als Eingabe erfordert: die Zeilennummer und die Spaltennummer, ab der geschrieben werden soll, und ein Text von maximal 20 Zeichen. Das LCD03 kommt diesem Wunsch insofern entgegen, als dass es ein Kommando anbietet, welches die Zeilen- und Spaltennummer adressiert (Kommando 3_{10} = Set Cursor, line, column). Es sei nur am Rande gesagt, dass bei dieser Art der zeilenweisen Adressierung die Spaltennummer typischerweise immer „1“ sein wird.

Es sollte an dieser Stelle noch erwähnt werden, dass es noch einen weiteren Grund gibt, die Ausgabe auf das LCD03 zeilenweise zu organisieren. Je nach eingestellter Taktfrequenz des Mikroprozessors ergibt sich nach einer Formel (siehe ATmega168 Description) eine Taktfrequenz auf dem I2C-Bus von mindestens ca. 100 KHz. Das ist schneller als das LCD03 Daten empfangen kann. Aus diesem Grunde werden die Empfangsdaten im LCD03 gepuffert. Der Empfangspuffer ist nur 64 Bytes groß. Das bedeutet, dass wenn mehr als 64 Bytes gesendet werden, die überschüssigen Bytes u.U. verloren gehen. Die hier vorgestellte Subroutine überträgt pro Aufruf 20 Zeichen für eine Leerzeile und dann unmittelbar danach bis zu 20 Zeichen Text. Damit läge sie auf der sicheren Seite. Es ist natürlich möglich, dass diese Subroutine

vier mal (für 4 Zeilen) unmittelbar hintereinander aufgerufen wird. Damit könnte der „Buffer Overflow“ theoretisch doch auftreten. Nun, in der Praxis ist dies bei der gewählten Mikroprozessor-Taktfrequenz nicht aufgetreten. Falls dies bei höheren Taktfrequenzen auftreten sollte (Symptom falsche Zeileninhalte), sollte am Ende der Subroutine eine „Waitms xx“ eingefügt werden.

Es folgt eine Beschreibung der Definitionen und Deklarationen für das LCD03.

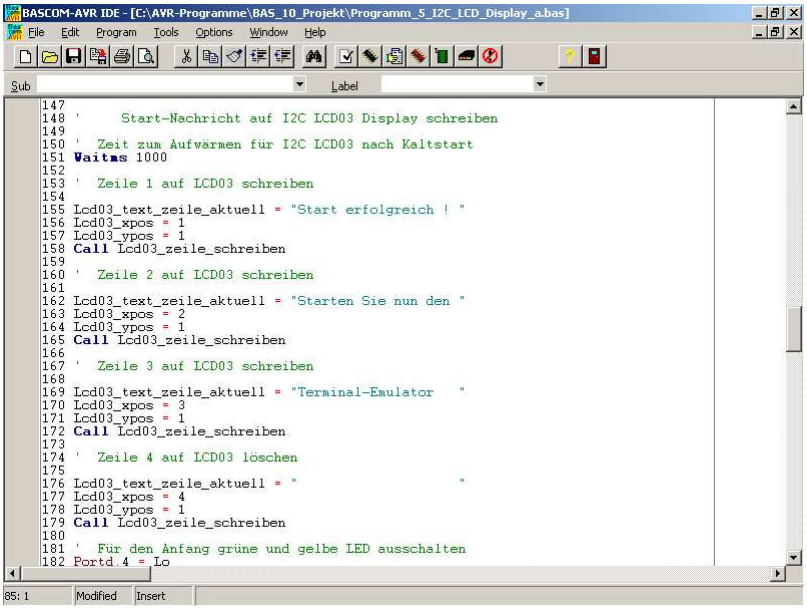


Config Sda = Portc.4
Config Scl = Portc.5
Beim ATmega168 sind die Busleitungen SDA und SCL hardwaremäßig auf Port C Pin 4 bzw. Port C Pin 5 festgelegt (es besteht hier keine freie Wahl). Dies wird hier dem Compiler mitgeteilt.

Declare Sub Lcd03_zeile_schreiben
Die noch vorzustellende Subroutine mit dem Namen „Lcd03_zeile_schreiben“ wird an dieser Stelle deklariert. Damit kann sie später im Ausführungsteil jederzeit mit einer „Call“-Anweisung aufgerufen werden.

Es folgt die Definition der Konstanten für das LCD03. Was die Werte betrifft, siehe hierzu die „Technical Documentation“ des LCD03. Was die Definitionen für die Variablen betrifft, so werden diese erst klarer werden, wenn der Inhalt der Subroutine „Lcd03_zeile_schreiben“ diskutiert werden wird.

Für den Fall, dass unmittelbar nach „Power On/Reset“ eine „Good Morning“-Nachricht auf das LCD03 geschrieben werden soll, ist zu beachten, dass das LCD03 eine gewisse Aufwärmzeit braucht. Dies ist hier durch **Waitms 1000** verwirklicht.



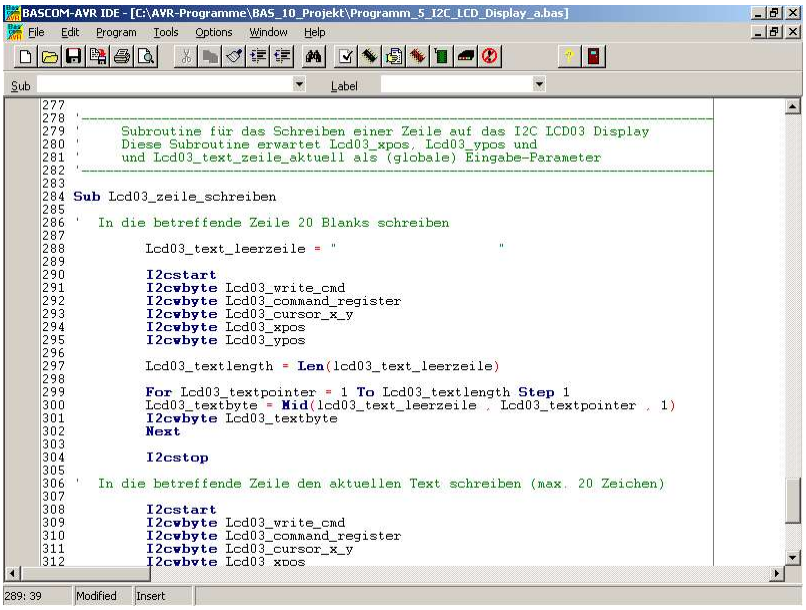
Es folgt dann im Ausführungsteil nach „Power On/Reset“ das Senden einer „Good Morning“-Nachricht auf die 4 Zeilen des LCD03.

Lcd03_text_zeile_aktuell
Diese Variable wird mit dem aktuellen Text für die jeweilige Zeile gefüllt.

Lcd03_xpos = 1
Lcd03_ypos = 1
Diese Variablen werden mit der aktuellen gewünschten Position gefüllt.

Call Lcd03_zeile_schreiben
Mit dieser Anweisung wird die noch zu diskutierende Subroutine „Call Lcd03_zeile_schreiben“ aufgerufen. Wenn die Subroutine ihr Ende erreicht hat (bei „Return“), kehrt der Programmablauf auf die nächste Anweisung nach dem „Call“ zurück.

Es folgt die Vorstellung der Subroutine „Lcd03_zeile_schreiben“. Der Ablauf besteht aus zwei Schreib-Sequenzen zum LCD03, jeweils beginnend mit „I2cstart“ und endend mit „I2cstop“. Die erste Sequenz schreibt 20 „Blanks“ in die jeweilige Zeile, löscht sie also, und die zweite Sequenz füllt die jeweilige Zeile mit dem gewünschten Text.



I2cstart
Als „Master“ setzt der Mikroprozessor hiermit das „Start“-Signal auf die „SCL“-Leitung.

I2cwbyte Lcd03_write_cmd
Das erste gesendete Byte enthält die Adresse des LCD03 und das Schreib-Kommando. Das LCD03 hat immer die Adresse $C6_{16}$ oder 198_{10} . Es ist also nur ein LCD03 am I2C-Bus möglich.

I2cwbyte Lcd03_command_register
Dieses Byte adressiert das „Command Register 0“. Dieses Byte hat keine direkte Wirkung und wird laut „Technical Documentation“ ignoriert. Man sollte es aber nicht weglassen, denn das führt merkwürdigerweise zu Fehlern.

I2cwbyte Lcd03_cursor_x_y
Mit diesem Byte wird das Kommando „3“ (Set Cursor, line, column) übertragen.

I2cwbyte Lcd03_xpos
I2cwbyte Lcd03_ypos
Diese beiden folgenden Bytes enthalten (gewissermaßen als Parameter für das vorige Kommando) die Zeilen- bzw. Spaltenadresse.

Lcd03_textlength = Len(lcd03_text_leerzeile)
For Lcd03_textpointer = 1 To Lcd03_textlength Step 1
Lcd03_textbyte = Mid(lcd03_text_leerzeile , Lcd03_textpointer , 1)
I2cwbyte Lcd03_textbyte
Next
In dieser „For“- „Next“-Schleife werden je nach Textlänge bis zu 20 Zeichen Text zum LCD03 gesendet.

I2cstop
Hiermit setzt der Mikroprozessor das „Stop“-Zeichen auf die „SCL“-Leitung. Die erste Schreib-Sequenz ist abgeschlossen.

Die zweite Schreib-Sequenz hat denselben Ablauf wie die erste, nur, dass jetzt der auszugebende Text gesendet wird. Da der Cursor bei einer reinen Ausgabe keine Funktion hat, wird er mit einem entsprechenden Kommando am Ende der zweiten Schreib-Sequenz unsichtbar gemacht. Ferner wird, ebenfalls mit einem weiteren Kommando, das Hintergrund-Licht eingeschaltet.

Da dieser Wechsel zwischen Daten und Kommandos ohne Umschalten so funktioniert, erhebt sich die Frage, wie das LCD03 diesen Unterschied erkennt. Die Erklärung kann der „Technical Documentation“ entnommen werden: Zeichen von 0_{10} bis 27_{10} werden als Kommandos interpretiert, Zeichen von 32_{10} bis 255_{10} werden als Daten interpretiert.

Was den Gesamtablauf des Programms „Programm_5_I2C_Display“ betrifft, so möge es der Leser selbst analysieren: Nach „Power On“ wird eine dreizeilige Nachricht auf das LED03 geschrieben. Dann kann man den Terminal-Emulator starten, indem der „Taster 1“ auf dem Board kurz gedrückt wird. Dort erscheint ein kleines Menü, das eine Farbauswahl vorgibt. Je nach Wahl der Farbe wird die jeweilige LED zum Blinken gebracht bzw. die Farbwahl auf dem LED03 angezeigt.

Eine Uhr, die nicht stehen bleibt: Programmierung der I2C-Echtzeituhr DS1307

Programm: **Programm_6_I2C_Echtzeituhr**
Mit diesem Programm soll nun ein weiterer Chip in Betrieb genommen werden, und das ist die Echtzeituhr „DS1307“ von Dallas MAXIM.

Sie sollten jetzt auch das „Dallas MAXIM DS1307 I2C Real Time Clock Datasheet“ parat haben. Es findet sich unter http://www.sparkfun.com/datasheets/Components/DS1307.pdf.

Diese Echtzeituhr ist speziell für den Betrieb an einem I2C-Bus ausgelegt und kann somit mit anderen Komponenten an diesem Bus kommunizieren, z.B. mit dem ATmega168. Die Uhr wird durch einen externen Quarz gesteuert. Bei entsprechendem Betrieb liefert die Uhr beim Auslesen ihrer Register die Sekunde, die Minute, die Stunde sowie den Tag, den Monat und das Jahr., sowie den Tag der Woche. Sie unterstützt das Stundenformat von 12-hour als auch 24-hour. Ferner unterstützt die Uhr automatisch die unterschiedlich langen Monate und sie berücksichtigt auch Schaltjahre bis zum Jahr 2100.

Eine Besonderheit dieser Uhr ist, dass sie in einer Schaltung über zwei Stromversorgungen betrieben werden kann. Zum einen ist das die übliche +5V-Versorgung. Wenn die Uhr nur mit dieser Versorgungsoption betrieben wird, funktioniert die Uhr nur so lange korrekt wie die Stromversorgung steht. Im Falle einer Stromabschaltung gehen die Einstellungen der Uhr verloren und müssen beim Wiederanlauf neu initiiert werden.

Die Uhr kann aber zusätzlich über einen speziellen Anschluss mit einer 3V-Spannung versorgt werden. Dies gestattet die Versorgung mit einer 3V-Lithium-Knopfzelle. Die Schaltungslogik erkennt einen etwaigen Stromausfall an der +5V-Versorgung sofort und schaltet automatisch auf die 3V-Versorgung um. Das ergibt in der Praxis mit einer Knopfzelle eine Ausfallsicherheit von mehr als zehn Jahren.

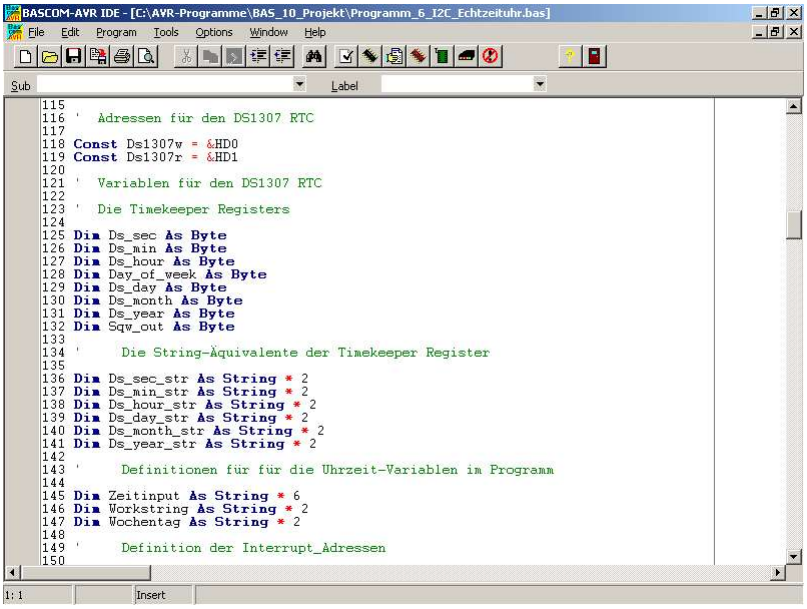
Um die nun folgenden Bascom-Deklarationen und Definitionen besser verstehen zu können, wollen wir uns die so genannten „Timerkeeper Registers“ des DS1307 näher ansehen. Wenn es um Kommunikation über den I2C-Bus mit dem DS1307 geht, wird der Inhalt dieser Register im Falle des Auslesens dem Master (ATmega168) zur Verfügung gestellt. Der Master (ATmega168) kann diese Register aber auch jederzeit beschreiben. Damit wird das Einstellen der Uhr auf neue Werte vollzogen.

Wie Sie sehen, sind die „Timekeeper Register“ byte-weise organisiert, und zwar so, dass pro Byte jeweils in den ersten vier Bit die Zehnerstelle eines Wertes abgelegt ist, und in den zweiten vier Bit die Einerstelle. Man nennt diese Darstellungsform auch „Packed BCD“ , eine Darstellungsform, die heute immer weniger zur Anwendung kommt und daher manchem Anfänger unbekannt sein mag. Als Programmierer muss man sich diesen Umstand klarmachen, da es sonst leicht zu Fehlern kommt. Wir sind es gewohnt, dass eine Zahl z.B. als „Byte“ deklariert, die Zahlen von 0 bis 255 aufnehmen kann. Wollen wir diese Zahl z.B. auf einem Display ausgeben, dann wandeln wir den Wert mit **Str(Zahl)** um und das Problem ist erledigt. Das

würde in diesem Fall der „Timekeeper Register“ zu falschen Werten führen, weil in dem Byte ja zwei Zahlen enthalten sind, eine Zehner- und eine Einerstelle. Es muss also zunächst einmal dafür gesorgt werden, dass der Wert, sagen wir z.B. der Sekundenwert, in einen ein Byte langen Integer-Wert umgewandelt wird. Hierfür ist Bascom bestens gerüstet, weil es speziell für diesen Zweck eine Anweisung zur Verfügung stellt, und die lautet **Makedec(Zahl**). Erst nachdem wir diese Anweisung auf den gepackten BCD-Wert angewendet haben, können wir in gewohnter Weise mit diesem Wert weiterarbeiten.

Umgekehrt ist es ähnlich: wenn wir einen Uhrzeitwert zum „Timerkeeper Register“ schicken wollen, werden wir diesen zunächst als „String“ von einem Eingabemedium lesen und diesen Wert mit **Val(Zahl)** in einen Integer von einem Byte Länge verwandeln. Dieser Wert wird dann in einem zweiten Schritt mit der Anweisung **Makebcd(Zahl)** in das erforderliche Format umgewandelt.

So, nachdem ich auf diese Besonderheit beim DS1307 für den Anfänger hoffentlich ausreichend und für den Experten sicherlich langweilig, eingegangen bin, brauche ich auf die entsprechenden Stellen im Code nicht mehr besonders einzugehen. Der Rest ist simple Handarbeit.



Zeilen 125 - 132
Diese Definitionen beinhalten die „Timekeeper Register“ in ihrer Reihenfolge von Adresse 00_{16} bis 07_{16}. Im Sinne der I2C-Kommunikation sind dies die benutzten Variablen, wenn der Mikroprozessor in die „Timekeeper Register“ hineinschreibt bzw. wenn diese vom Mikroprozessor ausgelesen werden.

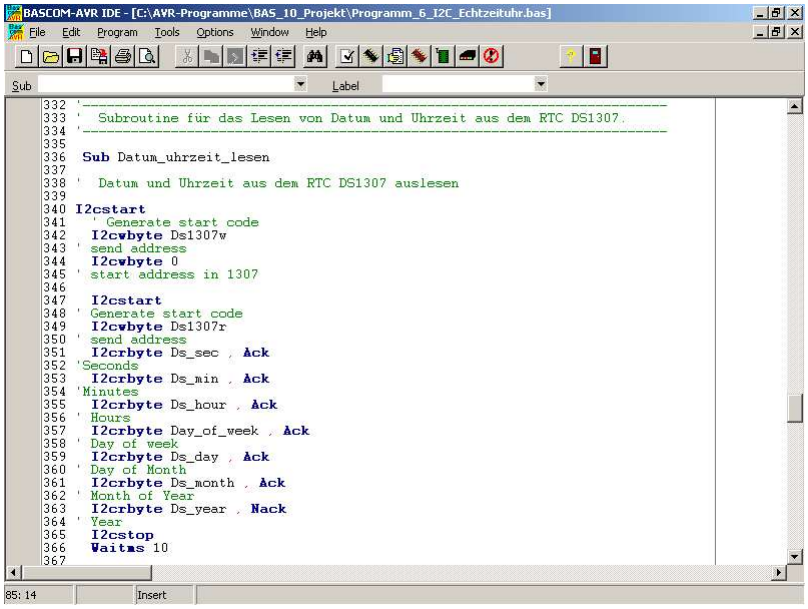
Zeilen 136 - 141
Diese Definitionen beinhalten die „String“-Äquavalente der obigen Werte, wenn diese benötigt werden.

Zeilen 145 – 147
Diese weiteren Definitionen werden bei der Programmierung des DS1307 benötiogt.

Zu den Definitionen für die Echtzeituhr DS1307 zählt ebenfalls noch eine Subroutine mit dem Namen „Datum_uhrzeit_lesen“, siehe den Original-Code.

Subroutine „Datum_uhrzeit_lesen“
Wir wollen die Beschreibung der Programmierunterstützung mit dieser Subroutine beginnen. Das Hauptprogramm ruft diese Subroutine immer dann auf, wenn es Datum und Uhrzeit aus dem DS1307 benötigt. Da das Hauptprogramm sonst keine eigene Uhrenfortschreibung vornimmt, kann dies sehr häufig vorkommen. Nach dem Aufruf legt die Subroutine die aktuellen Werte in den Variablen gemäß Zeilen 125 -132 nieder, zusätzlich als „String“-Äquivalente in

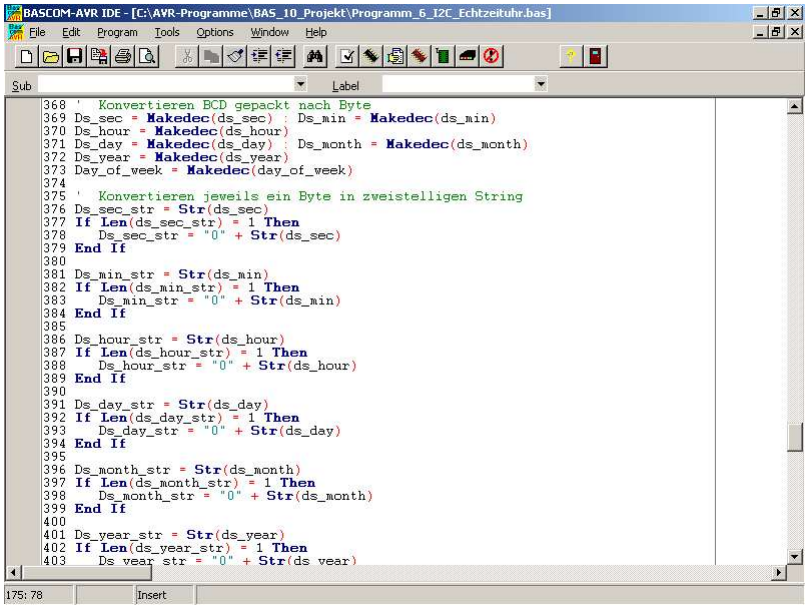
den Variablen gemäß den Zeilen 136 – 141. Die Subroutine kann in zwei Teilen beschrieben werden, dem Kommunikationsteil und dem Umwandlungsteil.



Zum Start der Kommunikation setzt der ATmega168 zunächst die Start-Bedingung auf die SCL-Leitung. Dann wird als erstes Byte die Adresse zum Schreiben des DS1307 gesendet. Diese muss $D0_{16}$ lauten. Da dies ein Schreibbefehl ist, folgt eine Adresse aus dem „Timekeeper Register“, und zwar die Adresse 00. Nun will aber der ATmega168 Daten aus dem DS1307 herauslesen. Er tut dies, indem er einen neuen Start-Befehl

auf die SCL-Leitung setzt, dieses mal gefolgt von einer Adressierung zum Lesen ($D1_{16.}$) Der nächste gesendete Befehl ist dann ein Lese-Befehl, und der DS1307 überträgt den Inhalt von „Timekeeper Register 0“ zum ATmega168. Dieser sendet ein „ACK“ zur Bestätigung. Der nächste Lese-Befehl wird dann aus dem „Timekeeper Register 1“ bedient, und so fort bis zu dem Zeitpunkt wo der ATmega168 „NACK“ sendet zum Zeichen dafür, dass er alle Daten bekommen hat.

Man beachte bitte, dass die Reihenfolge der Daten vom Programmierer nicht steuerbar ist. Wissend, welche Daten zu erwarten sind, muss er dafür sorgen, dass die Daten in den richtigen Variablen abgelegt werden. Die Inkrementierung der „Timekeeper Register“ wird automatisch vom DS1307 vorgenommen. Mit dieser Sequenz sind alle Bestandteile von Datum und Uhrzeit auf dem ATmega168 angekommen.



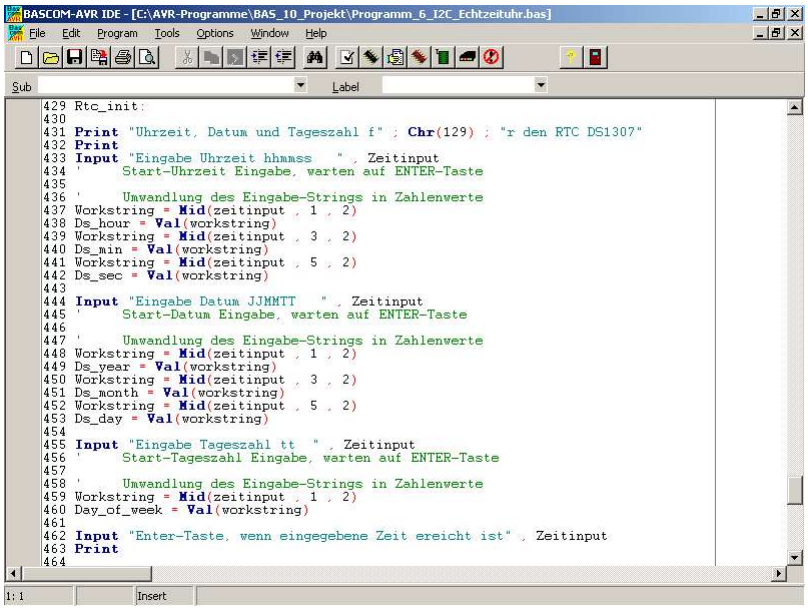
Um die gesendeten Zeichen, die ja im gepackten BCD-Format vorliegen, z.B. auf einem Display sichtbar machen zu können, müssen in einem weiteren Schritt entsprechend nach „String“ konvertiert werden. Das geschieht in zwei Schritten. Zuerst wird mit **Makedec (Zahl)** in einen Integer verwandelt, dann mit **Str(Zahl)** in der Format „String“ konvertiert. Es wird darauf geachtet, dass jeder Teilstring

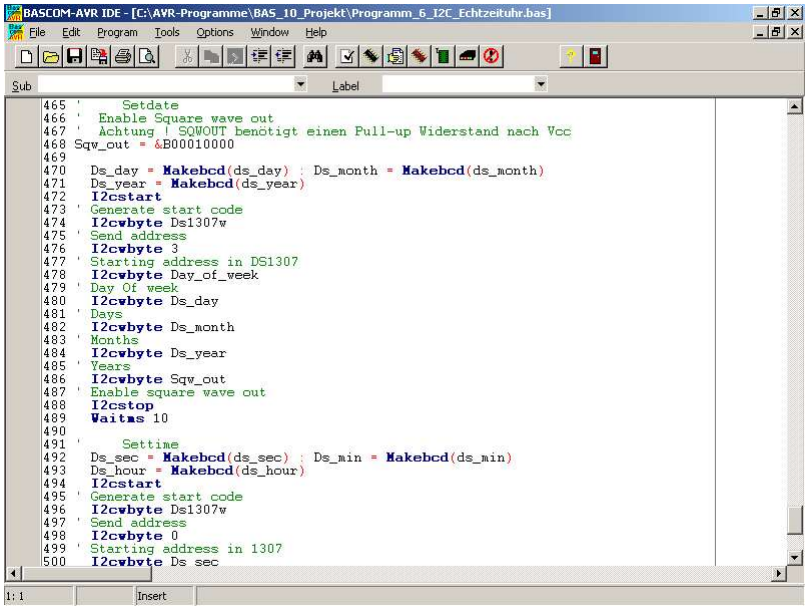
immer aus zwei Bytes (evtl. mit führender Null) besteht.

Routine für das Einstellen von Datum und Uhrzeit

Nachdem Sie nun die „Timekeeper Registers“ des DS1307 kennengelernt haben und wissen, dass das Auslesen von Datum und Uhrzeit nichts weiteres ist als den Inhalt dieser Register zum ATmega168 zu übertragen, werden Sie sich leicht vorstellen können, das das Setzen der Uhr ganz ähnlich ist, nur, dass die „Timekeeper Register“ jetzt vom ATmega168 beschrieben werden müssen. Hierbei ist natürlich wiederum zu beachten, dass das spezielle gepackte BCD-Format sichergestellt sein muss, eine kleine Fleißarbeit für den Programmierer.

Als Quelle für die Datum- und Uhrzeiteingabe dient uns ein Terminal-Emulator-Dialog. Er erfolgt, wie hier dargestellt, in drei Schritten, zuerst mit der Eingabe von Stunde, Minute, Sekunde (hhmmss) als sechsstelligem String (hhmmss). Dann folgt in einem zweiten Eingabeschritt die Eingabe des Jahres, des Monats und des Tages (JJMMTT), ebenfalls als sechstelligem String. Schließlich erfolgt die Eingabe der Tageszahl als zweistelliger String. Die Uhr läuft dann los, wenn mit einer weiteren Eingabe die ENTER-Taste verlangt wird.





Der Leser sollte mittlerweile genug Erfahrung gesammelt haben, um nachvollziehen zu können, auf welchem Wege die Informationen von der Eingabe her an die richtigen Stellen in den „Timekeeper Registern“ gelangen.

Beachten Sie bitte, dass die einzelnen Werte jeweils vor dem Senden zum DS1307 mit der Anweisung **Makebcd(Zahl)** in das gepackte BCD-Format umgewandelt werden.

Anderes als beim Lesen sind wir beim Schreiben nicht an eine bestimmte Reihenfolge der „Timekeeper Register“ gebunden. Trotzdem muss der richtige Ort der Information natürlich sichergestellt sein.

Damit sollte die Echtzeituhr DS1307 hinreichend erklärt sein.

Und selbstverständlich werden Datum und Uhrzeit in diesem Programm auf dem LCD03 in Zeile 4 dargestellt.

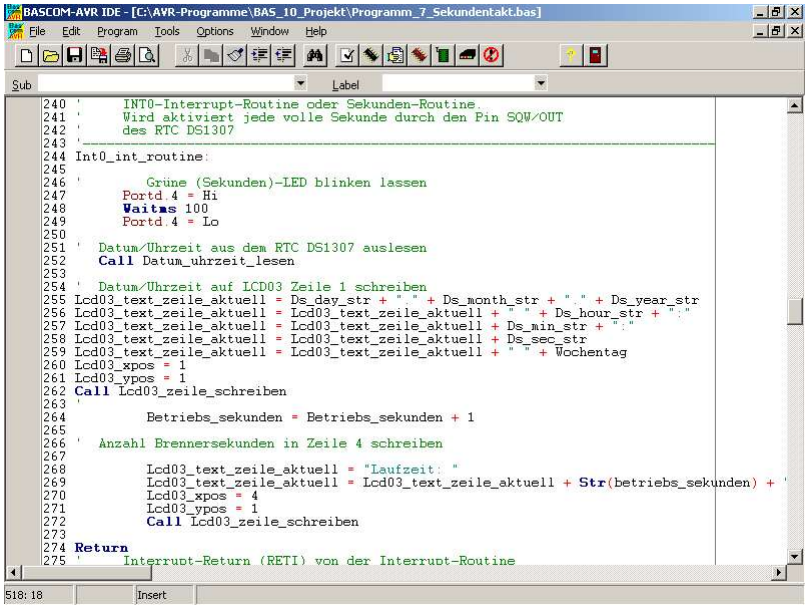
Ein Sekunden-Interrupt, abgeleitet vom DS1307

Programm: **Programm_7_Sekundentakt**
Im vorigen Programm hatten wir die Echtzeituhr DS1307 in Betrieb genommen. Diese Uhr, die ja, wenn sie einmal vom Programmierer gestartet ist, ohne unserer Zutun von selbst läuft, muss allerdings periodisch ausgelesen und zur Anzeige gebracht werden. Dies geschah im vorigen Programm in der Hauptschleife, und zwar jeweils nach Wartezeiten von einer Sekunde.

Hier soll nun eine alternative Methode vorgestellt werden. Diese Methode ist der „Sekunden-Interrupt“. Die Grundidee hierbei ist es, mittels irgend einer Zeitbasis dafür zu sorgen, dass im Mikropozessor ein sekundlicher Interrupt ausgelöst wird. Dieser sekundliche Takt wird dann vom Programm benutzt, um periodisch sich wiederholende Dinge zu tun, wie z.B. das Anzeigen von Ergebnissen auf einem Display, das Abfragen von Status-Zuständen, das Update von Zählern, usw. Häufig wird als Zeitbasis in diesem Falle ein Timer verwendet, der jeweils nach einer Sekunde einen Interrupt auslöst. Dies erfordert dann oft jeweils einen entsprechenden Abgleich mit der verwendeten Quarzfrequenz und die richtige Einstellung der so genannten „Prescaler-“Werte.

Die hier verwendete Methode ist dagegen sehr einfach, wird uns doch gewissermaßen ein 1-Sekundentakt vom DS1307 „frei Haus“ geliefert. Der DS1307 hat nämlich einen speziellen Pin, an dem der Sekundentakt hardwaremäßig abgegriffen werden kann, und das ist der Pin 7 SWQ/OUT. Mit einem Pullup-Widerstand abgeschlossen (extern!) liefert er ein TTL-kompatibles Signal, das direkt mit einem Input-Pin des ATmega168 verbunden werden kann. Wir wählen in unserem Fall den noch freien Pin INT0.

Die Inbetriebnahme dieser Zeitbasis ist denkbar einfach. Hierfür ist das „Timekeeper Register“ mit der Adresse 07_{16} zuständig. Ein Schreiben des Inhaltes **&B00010000** in das Register 07_{16} setzt die Zeitbasis in Betrieb.



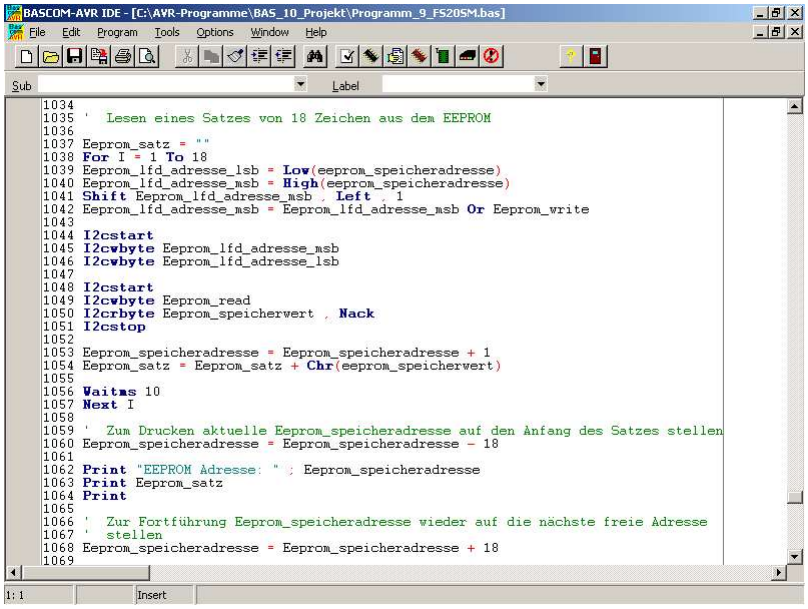
Wie man im nebenstehenden Beispiel einer Sekunden-Routine sehen kann, wird dort einmal die LED periodisch zum Blinken gebracht, es wird alle Sekunde die aktuelle Uhrzeit aus dem DS1307 ausgelesen und auf dem LCD03 zur Anzeige gebracht, und es wird ein Betriebs-Sekundenzähler hochgezählt, dessen Inhalt ebenfalls sekundlich auf dem LCD03 angezeigt wird.

Ein Speicher, der nicht vergisst: Programmierung des I2C-Eeprom AT24Cx

Programm: **Programm_8_I2C_Eeprom**
Mit der Inbetriebnahme des AT24C16, eines I2C-Eeprom, wird nach dem LCD03 und der Echtzeituhr DS1307 der Reigen der Chips am I2C-Bus des ATmega168 vorerst abgeschlossen. Wie auch schon bei der Echtzeituhr, liegt auch hier der klare Vorteil dieses Chips in seiner Restart-Festigkeit, d.h. auf einmal in diesem Speicher abgelegte Werte kann wieder zugegriffen werden, auch wenn die Schaltung einen Stromausfall gehabt hat.

Das Lesen vom Eeprom bzw. das Schreiben zum Eeprom ist an und für sich eine einfache Sache, indem ähnlich wie schon beim DS1307 der Chip mit einer Einheitenadresse adressiert wird und mit einem weiteren Byte die gewünschte Byte-Adresse angegeben wird. Es gibt aber beim Umgang mit dieser Einheiten- und Byte-Adresse eine Spitzfindigkeit, die sehr leicht bei der Programmierung zu Fehlern führen kann. Haben Sie hierzu bitte die „Description“ http://www.atmel.com/dyn/resources/prod_documents/doc0180.pdf parat.

Wie man der „Description“ entnehmen kann, verfügt der Eeprom über eine Einheitenadresse und eine dort so genannte „Word“-Adresse. Mit der „Word“-Adresse (von 8 Bit) kann man 255 Bytes adressieren. Bei einem AT24C16 haben wir es jedoch mit 2048 Bytes zu tun. Hierfür wird, wie man sich leicht überzeugen kann, ein Adressraum von 11 Bit benötigt. Die fehlenden 3 Bit sind laut „Description“ mit in der Einheitenadresse enthalten. Leider ist es nun so, dass in der Einheitenadresse in dem LSB noch „Read/Write“ verschlüsselt ist. Man kann also Einheiten- und „Word“-Adresse leider nicht als einen „Integer“, bestehend aus 2 Bytes, auffassen und diesen Wert beim Hochzählen der fortlaufenden Speicheradressen einfach inkrementieren.



Das Beispiel hier gilt erst für das nächste Programm, aber es sei hier schon besprochen, weil es in diese Thematik gehört. Beim blockweisen Schreiben bzw. Lesen von Eeprom-Informationen, wird die fortlaufende Eeprom-Adresse selbst im Eeprom verwaltet und bei Bedarf von dort geholt bzw. auch upgedatet. Da die „Word“-Adresse ja immer stimmt, brauchen wie als Programmierer nur die Einheitenadresse zu „editieren“.

Das geschieht mit **Shift Eeprom_lfd_adresse_msb , Left , 1**

Anschluss des Schaltmoduls FS20SM via Pin Change Interrupt

Programm: **Programm_9_FS20SM**
Mit dem Anschluss des Schaltmoduls „FS20SM“ an das Board kommen wir unserem Ziel ein entscheidendes Stück näher, den Verbrauch eines Heizölbrenners zu erfassen und auszuwerten. In der Sammlung der Schaltungskomponenten, die wir an das Board angeschlossen haben, fehlte ja bisher noch diejenige Komponente, die es gestattet, die Ein-und Ausschaltereignisse des Heizölbrenners zu erfassen. Hierfür ist nun das Schaltmodul „FS20SM“ vorgesehen.

Das Schaltmodul „FS20SM“ der Firma ELV A.G. gehört zu einer Serie von Haustechnik-Geräten, auch „FS20-System“ genannt, die aus verschiedenen Sendern und Empfängern bestehen, welche im 868 MHz-Band miteinander kommunizieren können. Das Schaltmodul „FS20SM“ hat in diesem System eine Empfängerfunktion. Es kann von beliebigen Sendern des FS20-Systems auf vier separaten Kanälen Signale empfangen. Ausgangsseitig können diese vier Kanäle direkt an die Ports eines Mikroprozessors angeschlossen werden (Open Collector, TTL-kompatibel), was das Schaltmodul zu einem idealen Kandidaten für unsere Schaltung macht.

Natürlich muss die Frage erlaubt sein, was denn die Gründe dafür sind, warum zur Signalübertragung von der Ölheizung zum Mikroprozessor eine Funkstrecke eingesetzt wird. Einen zwingenden Grund gibt es dafür natürlich nicht, aber es sollten doch zwei Vorteile genannt werden, die diese Lösung empfehlenswert machen:

1. Der Aspekt der Sicherheit
   Durch die Funkstrecke ist absolut sichergestellt, dass es keine galvanische Verbindung zwischen der Elektrik bzw. Elektronik der Heizung und dem Mikroprozessor gibt, und dies wird nicht durch ein elektronisches Bauteil wie einen Optokoppler sichergestellt, dessen Funktionsweise nicht jedermann einsichtig ist, sondern durch eine klare Separation der Anordnung. Jeder Heizungsmonteur, der eine Anlage warten soll, wird ein Gefühl des Missbehagens haben, wenn er sieht, wenn ein dubioses Kabel aus der Heizung zu einem selbst gebastelten Bausatz geführt ist.

2. Der Komfort-Aspekt
   Durch die entfallende Verkabelung zwischen der Heizung und dem Mikroprozessor kann man den Aufstellungsort des Mikroprozessors in relativ großen Grenzen frei wählen. Zwar wird in geschlossenen Räumen, zumal wenn die Räume durch Betonwände bzw. -decken getrennt sind, die Reichweite des „FS20-Systems“ schnell erreicht, aber innerhalb des Heizungskellers sollte der Aufstellungsort des Mikroprozessors beliebig sein können. Und durch den Einsatz eines so genannten „Repeaters“ kann man u.U. eine Erhöhung der Reichweite bis in ein anderes Stockwerk des Hauses erreichen. Hier ist allerdings „Ausprobieren“ angesagt, allgemeine Aussagen lassen sich hier nicht treffen.

Zum Empfänger der passende Sender

Natürlich muss nun auch besprochen werden, wie denn ein Ein/Ausschaltsignal am Heizölbrenner abgenommen werden kann, und mit welchem Sender dieses Signal dann an den Empfänger gesendet werden sollte.

Innerhalb des „FS20-Systems“ sind hier zwei Möglichkeiten zu nennen:

1. Die „FS20-Klingelsignal-Erkennung“ FS20KSE der Firma ELV AG
   Dieser Baustein ist ursprünglich dafür entwickelt worden, den Signaldraht einer Haustürklingel anzuzapfen, das Klingelsignal (die Klingelspannung) in einen Funkbefehl umzuwandeln und in einem Empfänger des „FS20-Systems“ zur Verfügung zu stellen. Die Versorgungsspannung für eine Haus-Klingelanlage wird normalerweise durch einen Kleinsignaltrafo geliefert, der, auf der Primärseite am Netz angeschlossen, auf der Sekundärseite eine Kleinspannung von 12V~ liefert. Diese Spannung wird über den Klingelknopf zur Klingel geschaltet. Im Falle einer Beschaltung mit der „FS20-Klingelsignal-Erkennung“ liegt diese parallel zur Klingel, d.h. wenn die Klingel ihre Versorgungsspannung erhält, erhält die „FS20-Klingelsignal-Erkennung“ ebenfalls ihre Versorgungsspannung.

   Wenn man die „FS20-Klingelsignal-Erkennung“ nun im Zusammenhang mit einem Heizölbrenner anwenden will, wird in dieser Umgebung normalerweise keine geschaltete 12V~-Versorgung zur Verfügung stehen. Bei der geschalteten Ölbrenner-Versorgung handelt es sich um 230V~. Es muss daher parallel zum Ölbrenner ein Trenntrafo geschaltet werden, der primärseitig an der geschalteten Brenner-Versorgungsspannung liegt und sekundärseitig mit seinem Ausgang von 12V~ die „FS20-Klingelsignal-Erkennung“ versorgt.

   Es ist daher ein Eingriff in das Heizungssystem erforderlich, den man nur von einem Fachmann vornehmen lassen sollte. Wenn Sie nicht autorisierter Fachmann sind, rate ich von einem persönlichen Eingriff dringend ab !

   Ihr Heizungsmonteur kann Ihnen ein Kabelende aus dem Heizungs-Schaltkasten herauslegen, das die geschaltete Brennerspannung führt und an dem sich eine Schuko-Muffe befindet. Hier kann dann der Trenntrafo mit der „FS20-Klingelsignal-Erkennung“ bequem angesteckt werden.

2. Der „FS20-Mini-Lichtsensor“ FS20LS der Firma ELV AG
   Dieser kaum streichholzschachtelgroße Sensor reagiert auf Lichteinfall und sendet sowohl beim Einschalten als auch beim Ausschalten von Licht ein Schaltsignal im Sinne des „FS20-Systems“. Die Lichtempfindlichkeit dieses Sensors ist recht groß und kann in weiten Grenzen justiert werden. So reicht z.B. das Aufleuchten bzw. Erlöschen einer Leuchtdiode oder Glimmlampe, um ein Funksignal auszulösen. Wenn also in Ihrem Heizungssystem an irgendeiner Stelle, z.B. in der Steuerung, ein solches Lichtsignal vorhanden ist, wenn der Ölbrenner läuft, dann wäre dieses kleine Gerät ein idealer Kandidat, da es keines Eingriffs in das Heizungssystem bedarf. Natürlich muss dies im praktischen Fall ausprobiert werden, und es bedarf der entsprechenden Justage und der Fernhaltung von Streulicht. Eine große Hilfe ist hierbei eine LED auf dem Sensor, die aufblinkt, wenn der Sensor ein Funksignal sendet.

Unabhängig davon, für welche Sender-Lösung Sie sich entscheiden, ist es für die Praxis der Schaltungsentwicklung eine große Hilfe, zusätzlich einen Hand-Sender aus der FS20-Serie als Testsender einzusetzen. Hierfür kommt in erster Linie der Schlüsselbund-Sender „FS20S4“ in

Betracht. Ähnlich wie das Schaltmodul „FS20SM“ besitzt dieser Sender vier Kanäle, so dass alle vier Kanäle getestet werden können.

Die Inbetriebnahme von Sender und Empfänger
Haben Sie hierzu bitte die entsprechenden mitgelieferten Betriebsanleitungen von Sender und Empfänger bereit.

Die folgende Kurzbeschreibung soll keineswegs die Betriebsanleitung ersetzen, sie soll vielmehr für den Erst-Einsteiger ein kurzer Überblick über die Philosophie“ des FS20-Systems sein. Hierzu muss man sich zunächst vor Augen führen, dass in diesem Haus-System mehrere Sender und Empfänger in Betrieb sein können, die sich gegenseitig nicht stören dürfen. Das gilt auch über die Hausgrenzen hinaus, also auch für andere Betreiber von FS20-Systemen, z.B. in Nachbarhäusern. Um diese Situation bewältigen zu können, werden im FS20-System verschlüsselte Signale verwendet. Die Verschlüsselung wird primär in den Sendern vorgenommen und ist Obliegenheit des jeweiligen Betreibers. Durch Einstellen gewisser Codes an den Sendern muss also gewährleistet sein, dass sich die Geräte in der Nachbarschaft nicht gegenseitig stören. Das ist grundsätzlich keine leichte Aufgabe, wenn man nicht weiß, wie der Nachbar seine Geräte codiert hat. Der Ausweg aus diesem Dilemma heißt „ausprobieren“. Jedenfalls ist das FS20-System als solches nicht in der Lage, Störungen zu erkennen. Aber die Vielzahl der Codierungsmöglichkeiten ist in der Praxis so hoch, dass eine Störfreiheit erreicht werden kann. Eine grundsätzliche Regel bei der Einstellung der Codierung lautet daher, dass man einfache Codierungen, wie z.B. nur „Einsen“ etc. vermeiden sollte.

Um dem Anwender eine leichtere Vergabe der Codierung zu ermöglichen, sieht das FS20-System zunächst den achtstelligen „Hauscode“ vor. Das Konzept sieht vor, dass alle Sender und Empfänger, die ein Benutzer in seinem Haus betreibt, alle denselben „Hauscode“ benutzen sollen. Dies muss natürlich durch Einstellung an jedem Sender durchgeführt werden.

Weiter geht es in der Strukturierung der Adressen mit den „Adressgruppen“ und „Einzeladressen“. Diese Kombination ist vierstellig. Die Aufteilung in diese beiden Bestandteile bietet die Möglichkeit, einzelne Sender im Haussystem zu unterscheiden und einzelne Kanäle in den Sendern zu adressieren. Für den Schlüsselbund-Sender „FS20S4“ bedeutet das, dass jedem der vier Kanäle eine separate Unteradresse und damit eine separate Ansprechbarkeit zugeordnet werden kann. Führen Sie nun bitte die Codierung des Senders gemäß der Betriebsanleitung durch. Und noch ein Rat: Notieren Sie bitte Ihre Codierungen.

Es stellt sich nun die Frage, wie ein Empfangskanal eines Empfängers auf eine Taste des Sendern abgestimmt werden kann. Dies ist sehr einfach:

1. Man nimmt den betreffenden Empfänger in Betrieb und stellt den gewünschten Kanal nach der Betriebsanleitung auf „Hören“ (eine LED blinkt),

2. man drückt auf diejenige Taste des Senders, die ein Signal an diesem Kanal auslösen soll (die LED verlischt).

Mit dem Verlöschen der LED ist der Vorgang abgeschlossen. Vielleicht noch ein Hinweis zum Schluss: Der auf „Hören“ eingestellte Empfänger verarbeitet das **erste** Sender-Signal, das er „hört“. Insofern ist sicherzustellen, dass andere Sender in der Nähe in dieser Zeit kein Signal senden.

Der Pin-Change-Interrupt des ATmega168

Haben Sie für dieses Kapitel bitte wiederum die „Description“ für den ATmega168 bereit. Siehe http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf , obwohl hier leider gesagt werden muss, dass die Beschreibung für den „Pin-Change-Interrupt“ alles andere als eine Bettlektüre ist. Sage mir keiner mehr etwas Schlechtes über das Juristen-Deutsch, dieses Techniker-Englisch hier stellt das locker in den Schatten. Dieses Manko setzt sich übrigens fort im „Bascom-Help“, was den „Pin-Change-Interrupt“ mit keinem Wort erwähnt. Man ist hier völlig auf sich selbst gestellt. Ich habe neuere Veröffentlichungen hierüber nicht studiert, bessere Darstellungen wären hier wünschenswert. Aus diesem Grunde möchte ich hier den Versuch machen, eine kleine Beschreibung zu liefern, die halbwegs dem normalen Menschenverstand entspricht.

Schauen Sie sich hierzu bitte zunächst einmal in der „Description“ auf Seite 2 die „Pin Configurations“, z.B. für das PDIP-Gehäuse genau an. Auf den ersten Blick ist alles ganz einfach: man kann leicht erkennen, welche Port-Pins welchem Gehäuse-Pin zugeordnet sind. Was jedoch vom Anfänger leicht übersehen wird, ist eine ganz fundamentale Eigenschaft der ATmega-Mikroprozessoren, dass nämlich die verschiedenen Pins für ganz verschiedene Funktionen verwendet werden können. Diese verschiedenen Funktionen werden im allgemeinen durch das Laden spezieller Register ermöglicht, was natürlich in der Obliegenheit des Programmierers liegt. Die Namen der alternativ möglichen Funktionen eines Pins sind in den „Pin-Configurations“ in Klammern dargestellt. So sehen Sie z.B., dass der Gehäuse-Pin 17, auf dem normalerweise Port B Pin 3 liegt (PB3), alternativ als „PCINT3“ fungieren kann. Der Name steht für „Pin-Change-Interrupt“. Diese Kenntnis der Nomenklatur dieser Begriffe wird von den Verfassern stillschweigend vorausgesetzt, wenn es um die Beschreibung des „Pin-Change-Interrupt“ in der „Description“ ab Seite 69 geht. Dies vorweg.

Nun wird sich der unbefangene Leser zunächst fragen, was das ganze Trara mit dem „Pin-Change-Interrupt“ denn eigentlich soll. Wir haben ja bereits zwei externe Interrupts des ATmega168 kennen gelernt, nämlich den „INT0“ auf Port D Pin 2 bzw. Gehäuse-Pin 4 und „INT1“ auf Port D Pin 3 bzw. Gehäuse-Pin 5, und die auch erfolgreich implementiert. Wenn es um weitere Interrupts gehen soll, sollte sich diese Systematik doch einfach auf weitere Pins erweitern. Dem ist aber nicht so. Die Begründung hierfür muss ich Ihnen schuldig bleiben. Ich kann hier nur vermuten, dass dieser Bruch mit der Kontinuität bzw. der Diskontinuität der Entwicklungsgeschichte dieser Mikroprozessoren zu tun hat. Für eine erfolgreiche Anwendung ist es aber unumgänglich zu wissen, dass es zwei verschiedene Sorten von Interrupts beim ATmega168 gibt und dass diese unterschiedlich programmiert werden.

Der „Pin-Change-Interrupt“, der jetzt behandelt werden soll, müsste eigentlich „Level-Change-Interrupt“ heißen, weil für ihn typisch ist, dass jede Änderung des Spannungspegels an dem betreffenden Pin, also von „Hi“ auf „Lo“ als auch von „Lo“ auf „Hi“ einen Interrupts erzeugt, was letztlich einen „Branch“ in die Vektortabelle des Mikroprozessors bedeutet. Das ist aber nicht der entscheidende Unterschied zu den beiden externen Interrupts „INT0“ und „INT1“, denn die kann man durchaus auch so programmieren. Entscheidend ist die unterschiedliche Programmierung dieser neuen Interrupts mit anderen Registern, und diese Tatsache muss man wohl oder übel hinnehmen. Da lohnt kein „Philosophieren“.

Weil das nun eben mal so ist, wenden wir uns der praktischen Frage zu, wie diese neuen Interrupt zu programmieren sind, und das läuft, wie bereits angedeutet, auf die Frage hinaus, welche Register wie geladen werden müssen, um z.B. aus dem Gehäuse-Pin 17 bzw. Port B Pin 3 einen „Pin-Change-Interrupt“ zu machen und welcher Interrupt-Vektor in diesem Fall angesteuert wird. Allerdings, - und das sei hier hervor gehoben -, müssen wir uns als Programmie-

rer darauf einstellen, dass wir bei jedem „Level-Change“ an dem betreffenden Pin einen Interrupt erwarten müssen, und das bedeutet für einen Impuls von „Hi“ auf „Lo“ und wieder zurück auf „Hi“ **zwei** Interrupts. Es gibt keine Möglichkeit, nur auf die steigende oder fallende Flanke des Impulses zu reagieren wie bei „INT0“ und „INT1“.

Ferner ist bei der Programmierung auch noch zu beachten, dass im Gegensatz zu „INT0“ und “INT1“ bei den „Pin-Change-Interrupts“ nicht mehr für jeden einzelnen Interrupt-Pin eine eigene Branch-Vektor-Adresse zur Verfügung steht. Die Branch-Vektor-Adressen beziehen sich auf einen ganzen Port, nicht auf einen einzelnen Pin. Das heißt für den Programmierer, dass er für den Fall, dass mehrere Pins an einem Port als „Pin-Change-Interrupt“ definiert sind, programmiertechnisch sicherstellen muss, welcher Pin an dem betreffenden Port nun gerade den Interrupt ausgelöst hat (durch Auslesen des Ports).

Pin-Change-Interrupt in Bascom , Definitionsteil
Nach all der tiefschürfenden Theorie entpuppt sich die praktische Programmierung als schiere Leichtigkeit. Wir stellen uns zur Aufgabe, Port B Pin 0 bis Pin 3 für das Schaltmodul „FS20SM“ als „Pin-Change-Interrupt“ zur Verfügung zu stellen.

```
Config Pinb.0 = Input
Config Pinb.1 = Input
Config Pinb.2 = Input
Config Pinb.3 = Input
```

Hiermit werden die betreffenden Pins als Input konfiguriert.

```
On Pcint0 Portb_int_routine
```

Hier wird für den Branch-Vektor PCINT0 die Adresse der (noch zu schreibenden Interrupt-Routine definiert.

```
Set Portb.0
Set Portb.1
Set Portb.2
Set Portb.3
```

Hiermit werden die Pull-up-Widerstände für die Pins aktiviert.

```
Pcmsk0 = &B00001111
```

Diese Instruktion lädt das „Pin Change Mask Register 0“ auf einen Wert, so dass die Pins PCINT0, PCINT1, PCINT2 und PCINT3 „enabled“ werden. Siehe „Description“ Seite 70.

```
Enable Pcint0
```

Die Instruktion „enabled“ Interrupts auf der Branch-Adresse PCINT0. Siehe „Description“ Seite 61.

Und das war es schon, was die Definitionen betrifft.

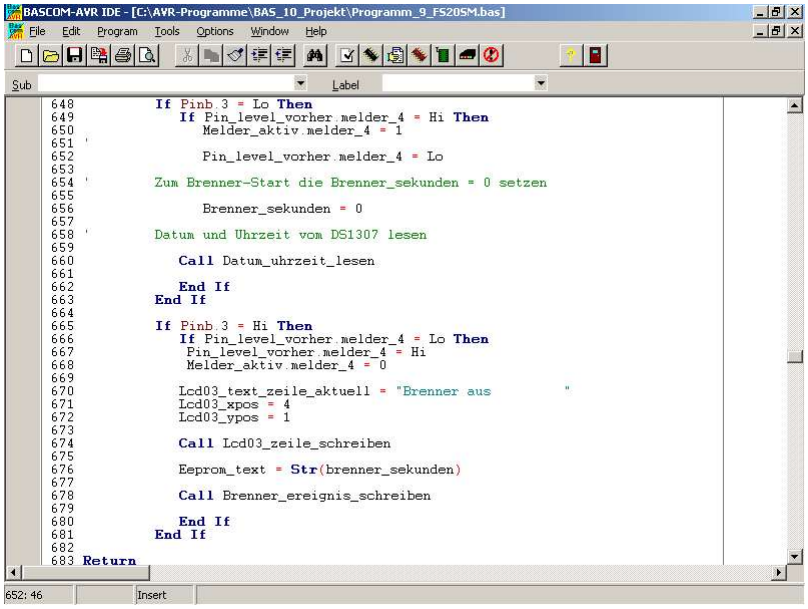
Pin-Change-Interrupt in Bascom , Interrupt-Routine
Zum besseren Verständnis des Aufbaus der Logik sei hier kurz rekapituliert, welche Situation der Programmierer hier behandeln muss. Wenn diese Routine angesteuert wird, hat sich ein „Level-Change“ an einem der Pins PCINT0, PCINT1, PCINT2 oder PCINT3 ereignet (es können auch mehrere gleichzeitig gewesen sein).

Hinweis: In Bezug auf unsere Schaltung wird es zwar nur **einen** Interrupt geben, nämlich den an PCINT3 (denn wir haben nur einen Heizölbrenner), aber wir wollen diese Routine gleich so gestalten, dass alle vier Kanäle des „FS20SM“ später benutzt werden können.

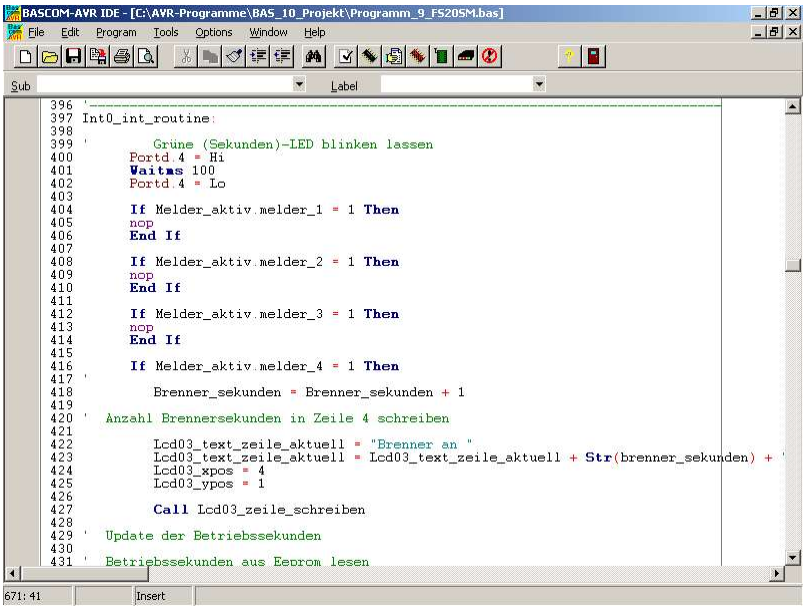
Da wir zunächst nicht wissen, welcher Pin den Interrupt ausgelöst hat, müssen wir dies erst einmal durch Auslesen der Pins feststellen.

Wichtig: Hierbei soll gelten, dass ein „Level-Change“ von „Hi“ nach „Lo“ einen **Einschaltvorgang** bedeutet, und ein „Level-Change“ von „Lo“ nach „Hi“ einen **Ausschaltvorgang.**

Neben dem aktuellen Level des Pins, den wir durch Auslesen ermitteln, müssen wir also auch den Level „vorher“ des Pins kennen. Da es ja mehrere Interrupts (nacheinander) geben kann, kann es passieren, dass z.B. ein gegebener Pin mehrfach auf „Lo“ ausgelesen werden kann, sich an diesem Pin aber keine Aktivität ereignet hat. Dieser „Vorher“-Level wird programmiertechnisch durch einen Schalter namens **Pin_level_vorher** (aufgeteilt in vier Bits), festgehalten und verwaltet. Mit Bezug auf die gesamte Programmstruktur kommen wir somit zu einem Konstrukt, in dem wir für jeden Pin feststellen können, ob es sich um einen Ein- oder Ausschaltvorgang handelt.



Die Abbildung zeigt den Teil der Interrupt-Routine, der den Pin 3 für den aktuellen Heizölbrenner-Interrupt behandelt. Hierbei ist die Variable **Melder_aktiv.melder_4** als Vehikel zu verstehen, mit dem dieses Interrupt-Ereignis auch außerhalb der Interrupt-Routine abgefragt werden kann. Im vorliegenden Fall wird dies in der Sekunden-Routine verwendet, damit man dort weiß, wann der Betriebs-Sekundenzähler erhöht werden muss. Der Rest sei dem Leser zum Selbststudium überlassen.



Sekunden-Routine

Hier wird das Zusammenspiel zwischen der Interrupt-Routine und der Sekunden-Routine mit Hilfe der Variablen **Melder_aktiv.melder_4** gezeigt.

Zusammengefasst kann man also festhalten, dass das Ereignis „Brenner ein“ zunächst in der Interrupt-Routine durch Analyse des Port-Zustandes erfasst wird, und dass dann das Hochzählen der Brenner-Sekunden in der Routine für den Sekunden-Interrupt vorgenommen wird. Entsprechend stoppt das Ereignis „Brenner aus“ dann wieder diesen Vorgang. Die vorliegende Struktur erlaubt dabei die gleichzeitige Behandlung von vier solchen Interrupt-Quellen. Mit Bezug auf unser praktisches Projekt „Erfassung der Einschaltzeiten eines Heizölbrenners“ könnte eine denkbare und nahe liegende Erweiterung die zusätzliche Erfassung der Einschaltzeiten der Kondensatpumpe sein (Brennwertkessel) bzw. die daraus errechenbare Menge des abgeführten Kondensats. Ich denke, das könnte manche unangenehme Wahrheit über den Wert von Brennwertkesseln an den Tag bringen, aber das ist nicht Gegenstand dieses Projektes.

Das Projekt „Heizölverbrauch“, die End-Version des Programms

Einsatz der „FS20-Klingelsignal-Erkennung“ FS20KSE

Programm: **Programm_10_Gesamt_KSE**
Dieses Programm ist die komplette End-Version unseres Projektes „Heizölverbrauch“, falls als es um den Einsatz der „FS20-Klingelsignal-Erkennung“ FS20KSE geht. Für den Einsatz des „FS20-Mini-Lichtsensor“ FS20LS siehe weiter unten.

Gegenüber dem Programm **Programm_9_FS20SM** wird hier von der sekundenweisen Erfassung und Fortschreibung der Einschaltzeiten eine stundenweise Fortschreibung der Einschaltwerte eingeführt, was in der Praxis vollkommen ausreicht. Die bereits bestehende Programmstruktur ist ansonsten die gleiche geblieben, d.h. mit Ausnahme von kleineren Anpassungen werden hier dieselben Routinen benutzt wie im Vorgänger-Programm.

Neu hinzugekommen ist in diesem Programm die Erfassung und Fortschreibung des aktuellen Tankstandes auf dem Eeprom. Hierzu ist es erforderlich, dass die erfasste Zahl der Betriebsstunden in Heizölverbrauch in Litern umgerechnet werden muss. Natürlich hängt der Verbrauch einerseits von der Zahl der Betriebsstunden ab und andererseits von den Eigenschaften des Heizölbrenners, hier besonderes von der verwendeten Brennerdüse. Es gilt nun, diesen Düsendurchsatz in Litern/Stunde zu ermitteln. Hierüber gibt im allgemeinen das Typenschild auf dem Heizölbrenner Auskunft. Hier ist der Brennerdurchsatz allerdings oft in kg/Stunde angegeben. Die Umrechnung erfolgt nach der Formel

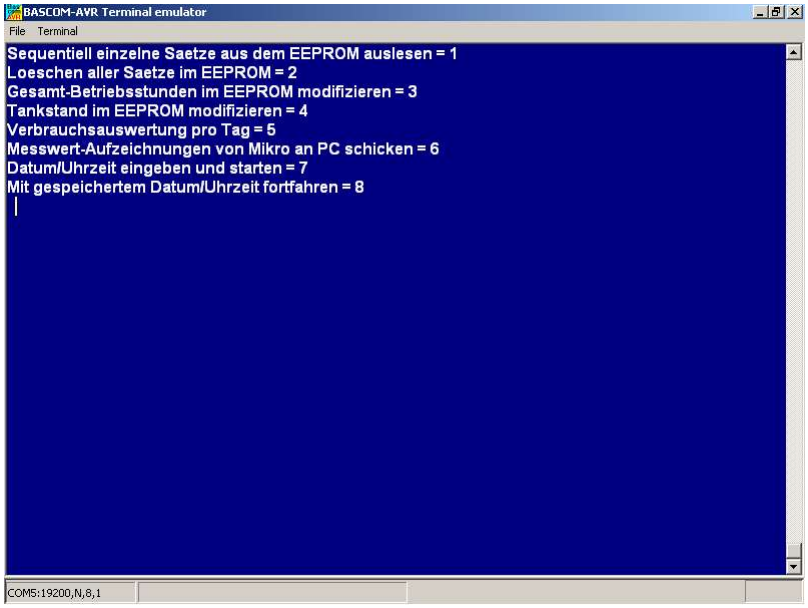
$$Durchsatz[cl/h] = 100 * Durchsatz[kg/h] / \rho_{Heizöl}[kg/l]$$

Hierbei ist $\rho_{Heizöl}$ = 0,86 [kg/l] die Dichte von Heizöl EL. Bei einem typischen Brennerdurchsatz von 1,7 kg/h für eine 17 kW-Heizungsanlage betrüge der Umrechnungsfaktor also 198 cl/h, das heißt, nach jeder abgelaufenen Brenner-Betriebsstunde wären 198 Zentiliter vom aktuellen Tankinhalt abzuziehen, und das ist genau das, was in dem Programm verwirklicht ist. Der Umrechnungsfaktor kann durch Verändern der Konstante

```
Const   Tank_liter_pro_stunde = 198  [cl/h]
```

angepasst werden. Zu den Gründen für die Wahl der Einheit „Zentiliter“ siehe den Kommentar im Quellcode.

Weitere Änderungen und Ergänzungen betreffen hauptsächlich das Auswahlmenü, das vom Terminal-Emulator angesteuert werden kann. Im wesentlichen sind hier hinzu gekommen die Optionen für die Modifikation der Anfangswerte der Gesamt-Betriebsstunden wie auch des aktuellen Tankinhaltes. Ferner sind hinzu gekommen die Möglichkeit, den Heizölverbrauch für einen vorgegebenen Tag aus den Daten des Eeprom zu ermitteln, sowie die Möglichkeit, alle im Eeprom gespeicherten Datensätze zum PC zu übertragen.



Diese weitere Option eröffnet uns die Möglichkeit, die Ein/Ausschaltzeiten z.B. in eine EXCEL-Tabelle zu übernehmen um sie dort entweder nur zu dokumentieren oder aber vielleicht auch, sie einer weiter führenden Verarbeitung zuzuführen, z.B. um sie mit separat erfassten Außentemperaturdaten in Korrelation zu bringen.

Beim diesem Programm ist weiterhin eine der endgültigen Aufgabenstellung entsprechende Modifikation der Anzeige auf dem I2C-Display vorgenommen worden.

In Zeile 1 werden jetzt die Gesamt-Betriebsstunden angezeigt. In Zeile 2 steht jetzt der aktuelle Tankinhalt in Litern, Zeile 3 bleibt leer, Zeile 4 zeigt den Betriebszustand des Brenners sowie die abgelaufenen Brenner-Sekunden pro Brennperiode an.

Einsatz des „FS20-Mini-Lichtsensor“ FS20LS

Programm: **Programm_10_Gesamt_LS**
Was die Leistungsmerkmale dieses Programms betrifft, so ist es mit dem eben beschriebenen „Programm_10_Gesamt_KSE“ identisch.

Leider ist es aber offenbar so, dass hinsichtlich der Zusammenarbeit mit dem Schaltmodul FS20SM der Sender FS20KSE und der Sender FS20LS nicht identisch sind. Während der Sender FS20KSE in der Lage ist, den Ein- wie auch den Ausschaltbefehl über ein- und denselben Kanal zu übertragen (genau wie der Handsender FS20S4), benötigt der Sensor des Senders FS20LS für den Einschaltbefehl einen Kanal und für den Ausschaltbefehl den zweiten Kanal (siehe die Bedienungsanleitung des FS209LS, Seite 7, die *-Bemerkung). Dies bedeutet natürlich, dass am Empfänger FS20SM ebenfalls zwei Kanäle eingerichtet werden müssen. Dies bedingt allerdings eine Programm-Modifikation gegenüber dem Programm „Programm_10_Gesamt_KSE“.

Konkret gesprochen werden in diesem „Programm_10_Gesamt_LS“ am Empfänger FS20SM die Kanäle 1 und 2 statt des Kanals 4 benutzt. Der Einschaltbefehl wird somit auf Port B Pin 0 übertragen und der Ausschaltbefehl auf Port B Pin 1. Dies bedingt eine entsprechende Änderung in der Programmlogik, die in diesem Programm verwirklicht ist.

Im Zusammenhang mit der Umstellung des Ein- bzw. Ausschaltbefehls auf zwei separate Kanäle muss man sich zusätzlich klar machen, dass ein „Level Change“ an einem gegebenen Pin nun eine etwas andere Bedeutung hat. Betrachten wir z.B. den Einschaltvorgang, der nun auf Port B Pin 0 liegt. Ein „Level Change“ von „Hi“ nach „Lo“ bedeutet zwar immer noch „Einschaltvorgang“, aber ein „Level Change“ von „Hi“ nach „Lo“ an demselben Pin bedeutet nicht mehr „Ausschaltvorgang“. Vielmehr bedeutet ein „Level Change“ von „Hi“ nach „Lo“ an Port B Pin 1 jetzt „Ausschaltvorgang“.

Natürlich ist dies in der Programm-Modifikation entsprechend berücksichtigt, aber es stellt sich jetzt die Frage, **wann** der Sender das Einschaltsignal wieder zurück nimmt bzw. wann der „Level Change“ von „Lo“ nach „Hi“ stattfindet. Nun, dies ist am Sender FS20LS Gott sei Dank einstellbar (siehe die Bedienungsanleitung, Seite 17). Ich empfehle die kürzest mögliche Einschaltdauer von 0,25 sec, da sie für das Erkennen des Interrupts allemal reicht.

Weiter ist zu berücksichtigen, dass der mögliche zeitliche Abstand zwischen zwei aufeinander folgenden Einschaltbefehlen so kurz wie möglich sein sollte, wenn man jeden Einschaltbefehl sicher erfassen will. Dies ist zwar bei einem Ölbrenner normalerweise nicht so wichtig, weil die Abstände recht groß sind. Ich empfehle hier aber ebenfalls den kürzest möglichen Sendeabstand von 8 sec (siehe Bedienungsanleitung Seite 18).

Was ist die Lichtempfindlichkeit des Sensors betrifft, so ist diese ebenfalls einstellbar (Bedienungsanleitung, Seite 19). Dieser Einstellwert muss in der Praxis im Experiment gefunden werden. Sein Wert hängt sowohl von der Lichtstärke der Leuchtquelle wie auch von evtl. gegebenem Streulicht ab. Zur Vermeidung von Fehlschaltungen sollte Streulicht durch entsprechende Einbaumaßnahmen möglichst vermieden werden.

---Schluss---