Programmierung eines AVR Mikroprozessors als Lernprojekt

Wir wollen anhand eines Anwendungsbeispiels nachvollziehen, wie man mit BASCOM einen AVR-Mikroprozessor ATmega8L programmieren kann und dabei die Grundelemente einer Programmiersprache (Basic) kennen lernt. Der Kurs richtet sich an Neulinge in diesem Metier.

Als Projektbeispiel haben wir uns das flackernde Lagerfeuer in einer Modelleisenbahnanlage vorgenommen. Eine rote LED soll wie ein Feuer unregelmässig flackern.

Inhalt

1. Grundsätzlicher Aufbau eines Mikroprozessors

Die Hardwarestruktur bzw. ein Blockschaltbild können unter diesem Link abgerufen werden:
http://modelleisenbahn-steuern.de/controller/atmega8/1-1-blockschaltbild-atmega8.htm

Hier werden nur die Teile, die wir in unserem "Projekt" brauchen, zusammenfassend vorgestellt.

Ein Mikroprozessor besteht im innersten Kern aus einer Recheneinheit. Diese Recheneinheit kann mit einer relativ kleinen Anzahl von digitalen Befehlen programmiert werden. Die Programmsprache, die der Prozessor versteht, ist nicht handlich. Daher gibt es höhere Programmiersprachen, die für uns viel einfacher zu verstehen sind. Eine solche Programmiersprache ist BASIC. Andere Programmiersprachen heissen C+ etc. Und zu diesen höheren Programmiersprachen gibt es Übersetzungsprogramme, die den Code auf die Sprache des Prozessors übersetzen. Einem solchen Übersetzungsprogramm sagt man Compiler.

Im Beispiel verwenden wir BASCOM. Für die kleinen AVR-Mikroprozessoren ist das eine gute Sache. Es gibt auch die Möglichkeit diese Prozessoren in C zu programmieren.

1.1. Hardwarestruktur des µP, das Inventar

CPU	Central Processing Unit. Die Recheneinheit. Beim ATmega8 werden 8bit parallel verarbeitet. Eine Fliesskommamultiplikation muss daher in vielen Schritten mit Behalte-Funktion berechnet werden, da aufs Mal nur 1 Stelle bearbeitet werden kann. Ein PC mit 64bit Architektur arbeitet da schon viel effizienter. Aber das kann uns hier einmal gleich sein.
32 Register	Die Register sind zur allgemeinen Verwendung. Wenn wir ein Programm mit BASCOM erstellen und kompilieren, dann wird sich der Compiler um Verwendung dieser Register kümmern. Als BASIC-Programmierer hat man mit ihnen also nichts zu tun. Programmiert man in einer Prozessor-nahen Programmiersprache wie Assembler, dann ist das anders und man muss sich um diese Register für die Aufbewahrung von Zwischenresultaten selbst kümmern.
Flash-Memory	Das Flash-Memory enthält den Programmcode, d.h. die Anweisungen, die unserem Gerät die Seele vermitteln. Das Memory kann mit dem Programmierstick beschrieben und gelöscht werden und einmal beschrieben bleibt der Inhalt netzausfallsicher gespeichert. In unserem Falle ist der Speicher 8k gross (ATmega8L). Das entspricht etwa 5 Seiten Schreibmaschinentext.
RAM-Speicher	Random Access Memory. Das RAM (auch SRAM, Statisches RAM) ist nicht netzausfallsicher wie das Flash-Memory, dafür kann es quasi unendlich viele Male beschrieben und gelöscht werden und es arbeitet schnell. Im RAM legt der Prozessor bzw. das Programm Zwischenresultate ab, die später wieder benötigt werden. Auch die aktuellen Werte aller verwendeten Variablen sind im RAM live abgespeichert. Das SRAM des ATmega8L ist 1kB gross (1024 Bytes). Als statisches RAM verliert es den Inhalt nicht, wenn der Prozessor nicht taktet, um Strom zu sparen. Es verliert den Inhalt erst, wenn die Speisespannung unter 3 Volt fällt.
EEPROM	Electrically Erasable Programmable Read Only Memory. Bei Geräten und vielen Applikationen möchte man einige Werte so gespeichert haben, dass sie bei Netzausfall nicht verloren gehen. Das können Einstellwerte sein oder Betriebszustände wie z.B. die Stellung einer Weiche oder eines Signals, damit der

Prozessor nach einem Netzausfall noch weiss, wo er vor dem Netzausfall stehen geblieben ist. Von dieser Art Memory gibt es im ATmega8L nur 512 Bytes. Das EEPROM kann vom Programm beschrieben und gelöscht werden und der Inhalt ist netzausfallsicher abgelegt. Die Zahl der Schreibzyklen ist nicht unendlich und ein Schreibvorgang braucht etwas Zeit im Vergleich zum RAM.

Zeitgeber 0

Der ATmega8L hat einen TIMER0, einen Hardware-Timer, der den Prozessortakt zählen kann. Der TIMER0 ist speziell für PWM-Ausgaben gedacht. PWM steht für Puls-Weite-Modulation. Mit ihr ist es möglich, ein analoges Ausgangssignal zu erzeugen. Wenn wir ein flackerndes Lagerfeuer haben wollen, dann müssen wir die rote LED in verschiedener wechselnder Helligkeit ansteuern können. Da brauchen wir die PWM-Ausgabe. Der TIMER0 kann von 0 auf 255 zählen und erzeugt immer einen Interrupt, wenn er oben angekommen ist. Wir können also ein Ausgangssignal nur in einer Stufigkeit von 255 Schritten erzeugen. Das ist für unsere Anwendung mehr als fein genug. Mehr zum TIMER0 siehe Programmbeispiele weiter unten.

Zeitgeber 1

Der ATmega8L hat einen TIMER1, einen Hardware-Timer, der den Prozessortakt zählen kann. Der TIMER1 ist für die Zeitsteuerung der Applikation gedacht, wo das erforderlich ist. Der TIMER1 kann von 0 auf 65'536 zählen und erzeugt immer einen Interrupt, wenn er oben angekommen ist. Wir brauchen diesen Timer um das Programm relativ genau in Zehntel- und ganzen Sekunden ablaufen zu lassen. Mit einem Quarz würde dieser Zähler eine exakte Uhrfunktion ermöglichen.

Die Ports

Ein PORT ist ein Register mit 8bit Ausgangs- und Eingangsschaltungen je Pin, die geeignet sind, um ankommende Signalzustände 0/1 abzufragen oder um Stellsignale 0/1 auszugeben. Die Ports führen auf die Pin des ATmega8L Die AVR-Prozessoren haben grundsätzlich 4 Ports zu 8 Bit. Unser µP hat nur 28 Pin und kann daher nur einen Teil der möglichen Port-Pins nach aussen führen.

PORTA fehlt ganz.
PORTB: Es werden alle 8 Ausgänge auf Pins hinausgeführt: PB0 … PB7
PORTC: Es werden nur die Ausgänge PC0 … PC5 auf Pins hinausgeführt.
PORTD: Es werden alle 8 Ausgänge auf Pins hinausgeführt: PD0 … PD7

Die Portausgänge können einzeln oder 8bit-gruppiert angesteuert werden. Einzeln über Begrenzungswiderstände kann man LEDs mit bis zu 20mA Strom direkt anschliessen. Relais und Motoren muss man über Optokoppler ansteuern, auch wegen der höheren Spannung dieser Lasten.

Jedes PORT und jedes einzelne Pin kann auch als Input konfiguriert werden. Damit kann man Tastensignale abfragen oder andere digitale 5V Signale - z.B. von einer Lichtschranke - auswerten.

Bemerkung: *Die Ports können auf mehrere Arten konfiguriert werden. Im Beispielprojekt beschränken wir uns auf die von mir aus gesehen einfachste Methode. Es gibt da noch ein Data-Direction-Register, mit dem die Ports oder Einzelpins auch als Output oder Input festgelegt werden können. Wir lassen das aber beiseite. Dieses Einfachheitsprinzip wenden wir im ganzen Projekt an. Das gilt speziell auch für die Timerfunktionen.*

LCD-Ausgabe

Nur zur Info: Mit den Ports kann man auch LCD-Displays ansteuern. Das brauchen wir in unserem Projekt einmal nicht.

Analog-Digitalwandler: Nur zur Info: Der Prozessor kann auch Messeingänge verarbeiten. Das brauchen wir in unserem Projekt nicht.

1.2. Der PWM-Ausgang bzw. der "Analog"-Ausgang

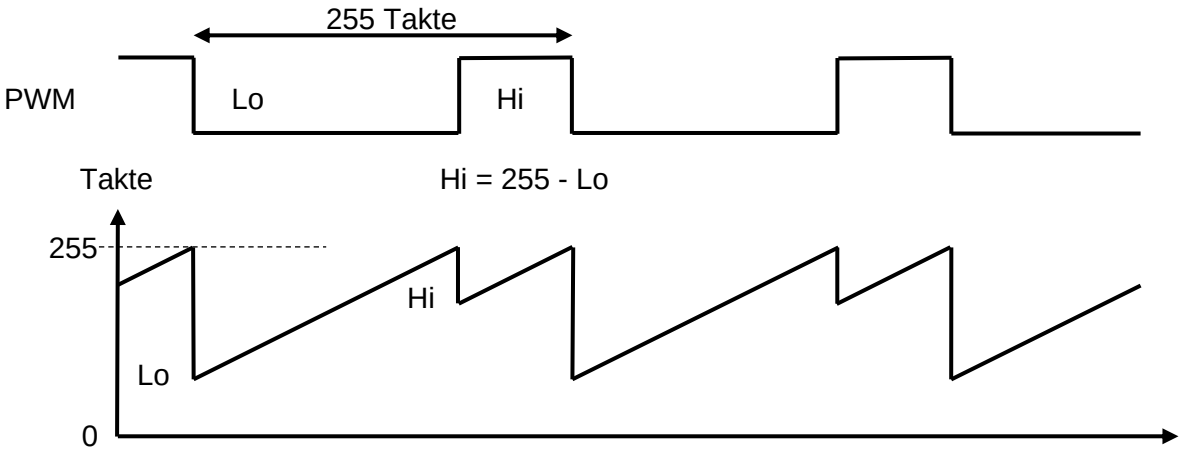
PWM steht für Pulse-Width-Modulation oder Puls-Weiten Modulation
Der Ausgang ist eine schnelle Folge von Impulsen und Lücken. Da die Pulsfolge sehr schnell ist, merkt das Auge nichts von dem Ein- und Ausschalten des Lichts. Damit dies erfüllt ist, muss die Pulsfolge eine Frequenz höher als 50Hz haben. Wenn die LED 50% leuchten soll, dann sind Puls und Lücke gleich gross.

Der Abstand zwischen 2 positiven Pulsflanken ist immer gleich. D.h. die Frequenz ist konstant und sie wird im µP von einem Hardwaretimer TIMER0 erzeugt.

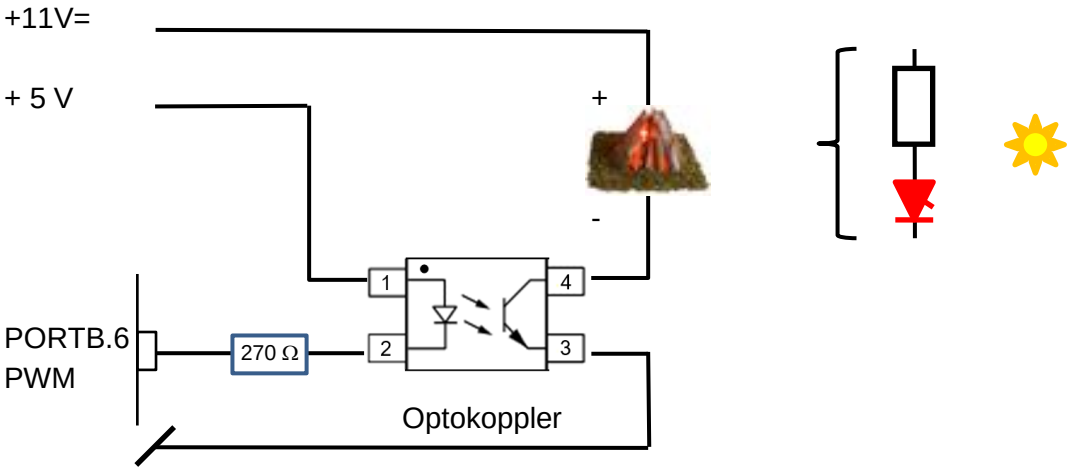
Das Pulsverhältnis gibt man dem Timer z.B. mit den beiden Variablen Lo und Hi (Byte) vor. Der Timer zählt regelmässig von 0 auf 255. Man kann den Timer auf einen Anfangswert setzen. Von diesem zählt er immer aufwärts bis er 255 erreicht. Hat er 255 erreicht, löst er einen Interrupt aus und man springt dann in eine Subroutine, wo man dem Timer einen neuen Startwert zwischen 0 … 255 eingibt.

Lo und Hi werden aus der Helligkeit im zyklischen Programmteil jede Zehntelssekunde neu berechnet: Lo = 255 - Helligkeit und Hi = 255 - Lo.

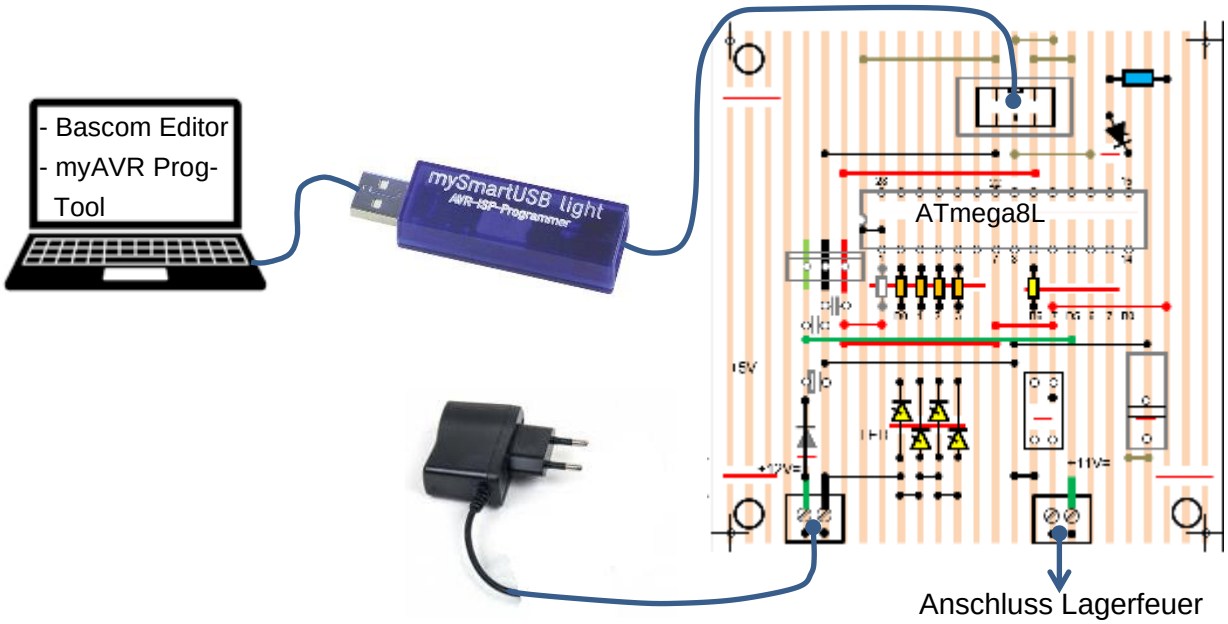
<u>PWM mit Timer0</u>
Das Wechselsignal PWM hat eine konstante Periodendauer aber unterschiedliche Lo und Hi-Zustände. Lo + Hi müssen zusammen immer 255 Taktzyklen ergeben.



PWM-Ausgangsschaltung

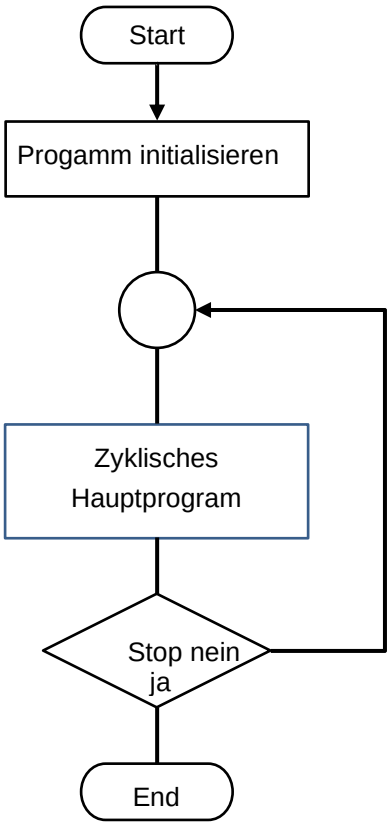


2. Programmierumgebung



Das BASCOM-Programm wird mit dem BASCOM-Editor auf dem PC erstellt. Dann wird das Programm kompiliert. Mit dem Hilfsprogramm myAVR-Prog-Tool wird das kompilierte Programm in Binär- oder HEX-Format über den Interface-Stick mySmartUSB auf den Prozessor auf der Platine geschrieben. Das Programm kann ohne 12V Anschluss geschrieben werden. Der Print mit den Ausgängen und den LEDs funktioniert aber erst, wenn die Speisung 12V vom Steckernetzgerät auch anliegt.
Die Sprachreferenz für BASCOM findet man hier: https://avrhelp.mcselec.com/

3. Üblicher Programmablauf



Prozessortyp angeben
Variablen deklarieren
Konstanten deklarieren
PORTs konfigurieren, Ein- und Ausgänge
Timer konfigurieren
Initialwerte von Variablen setzen.

Hier wird die Hauptarbeit gemacht
Es können auch Unterprogramme vorkommen, die z.B. das Gerät in einen Testmodus schicken.
Eingänge werden zyklisch abgefragt
Ausgänge werden nach Bedarf geschaltet

3.1. Programm initialisieren

Compilerbefehle beginnen mit $

Auf der ersten Zeile im Programmcode muss der Prozessortyp angegeben werden. In unserem Falle muss beim ATmega8L immer stehen:

```
$regfile = "m8def.dat"
```

Dieser Befehl verweist auf die Dat-Datei, die für das Kompilierern von unserem Prozessor zuständig ist.

Variablen deklarieren

In allen modernen Programmen und auch in BASCOM muss man die Variablen, die man brauchen will, zu Beginn des Programmes deklarieren. D.h. man muss angeben, wie die Variable heissen soll und von welchem Datentyp sie sein soll. Es gibt Datentypen, die 1 Bit, 1 Byte, 2 Byte oder 4 Byte etc. Platz brauchen im Memory, um einen Einzelwert abzuspeichern. Wenn man Datentypen wählt, die nur so viel Speicherplatz brauchen wie nötig, dann ist das optimal für den µP.

Eine einwertige Variable festlegen bzw. dimensionieren. (Ja / Nein, ein Flag, ON / OFF) (Bit und Boolean sind erlaubt.)

```
Dim New_time as Boolean
      oder auch
Dim X01 as Bit
```

Eine Ganzzahl-Variable mit 8 Bit = 1 Byte

```
Dim Auswahl as Byte
```

Eine Ganzzahl-Variable mit 2 Byte
Eine Ganzzahl-Variable mit 4 Byte

```
Dim Register as Integer
Dim XY as Long
```

Eine Fliesskommazahl einfacher Genauigkeit
Eine Fliesskommazahl doppelter Genauigkeit

```
Dim A as Single
Dim B as Double
```

Buchstaben, Textteile. Nach dem Komma steht die Anzahl Zeichen der Variablen. In die Variable kann ich auch z.B. "kaufen!" einschreiben. Es dürfen nur nicht mehr als 8 Buchstaben sein.

```
Dim Kommentar as String * 8
Kommentar = "zu teuer"
```

Ein Byte im EEPROM an Adresse 1 festlegen

```
Dim Ee_Auswahl As Eram Byte At 1
```

Intermezzo: Mit Variablen rechnen

Beachte, dass Berechnungen und Zuweisungen nur innerhalb des gleichen Datentyps möglich sind. Man kann nur eine Single- mit einer Single-Variable multiplizieren. Wenn es unausweichlich ist, dass man verschiedene Datentypen miteinander verarbeiten muss, dann muss man zuerst alle auf den gleichen Typ umwandeln. Wir werden nur folgender Umwandlung begegnen:
Wir werden einen Zufallsgenerator brauchen um das Flackern zu erzeugen. Dieser Generator läuft mit einem Register, das als Integer (2 Byte) dimensioniert ist. Für das Flackern am Ausgang zur LED brauchen wir aber nur die 8 niederwertigen Bits dieses Registers in der Variablen Helligkeit, die als Byte dimensioniert wurde. Wir werden diese mit folgendem Befehl herausholen:

```
Helligkeit = low(Register)
```

Konstanten deklarieren

Braucht man konstante Zahlenwerte im Programmcode, dann ist es von Vorteil, wenn man den Konstanten einen Namen gibt und den zugehörigen Zahlenwert im Initialisierungsabschnitt zuweist. Das macht den Programmcode lesbarer. Logisch kommt es aufs Gleiche hinaus, wenn man jeweils im Programm den Zahlenwert einschreibt. Wird die Konstante mehrfach gebraucht, dann muss man

im Falle der Konstanten-Deklaration bei der Initialisierung den Zahlenwert nur an einem Ort ändern und er ist dann bei jedem Vorkommen im Programm automatisch auch geändert.

```
Const Timerstart = 63976
Const True = 1
Const False = 0
Const EIN = 0     *
Const AUS = 1     *
```

* In unserer Applikation wird es so sein, dass eine LED an einem Ausgangsport des µPs dann leuchtet, wenn der Ausgang auf 0 ist, weil der Prozessor gegen 0 grössere Ströme am Ausgang liefern kann. Logisch 0 bedeutet dann LED EIN. Dies ist verkehrt und leuchtet nicht ganz ein, aber wenn wir die Konstanten EIN und AUS verwenden, dann setzen diese das Ausgangsport richtig und der Programmcode lässt sich besser lesen.

PORTs konfigurieren

Alle 7 Pins von PORTD als Ausgänge konfigurieren

```
Config PORTD = Output
```

Pin 0 von PORTB als Eingang mit Pullup-Widerstand konfigurieren. Ein Eingang wird mit PIN und ein Ausgang mit PORT bezeichnet. Für den Pullup-Widerstand wird die PORT-Eigenschaft verwendet.
Dem PINB.0 wird der Alias-Name "Key" gegeben, damit das Programm lesbarer wird.

```
Config PINB.0 = Input
PORTB.0 = 1  'Pullup-Widerstand aktiv
Key Alias PINB.0
```

PWM-Ausgang auf PORTB.6 konfigurieren:
Man kann das PORT konfigurieren und dann den Alias-Namen vergeben. In umgekehrter Reihenfolge geht es aber auch.

```
PWM Alias PORTB.6
Config PWM = Output
'Bemerkungen kann man nach einem Apo-
'stroph einfügen und diese haben
'keine programmtechnische Wirkung.
```

Timer1 konfigurieren

Wir haben gelernt, dass der Timer1 als Zeitbasis für Uhrzeit-abhängige Funktionen einer Applikation benutzt werden kann. Im Initialisierungsteil des Programmes braucht es folgende Festlegungen:

Dimensionierung von Variablen und Konstanten
Timer1 zählt vom Startwert jeweils bis 65536, löst oben angekommen einen Interrupt aus und beginnt wieder beim Startwert neu zu zählen.

```
Const Timerstart = 63974
Dim New_time as Boolean
```

Timer1 mit Vorteiler 64 für den Prozessortakt konfigurieren für eine Zeitbasis von 0.1 sec

```
Config TIMER1 = Timer, Prescale = 64
```

Label der Subroutine `Timer1_isr` , die beim Zählerreset abgearbeitet werden soll.

```
On TIMER1 Timer1_isr
```

Setzen der Initialisierungswerte

```
Timer1 = Timerstart
New_time = False
```

Aktivieren des Timers

```
Enable TIMER1
```

Aktivieren, der Interrupts (Der Befehl gilt für alle Interrupts im Programm)

```
Enable INTERRUPTS
```

Ab diesem Moment im Programm läuft der Timer1. Siehe auch Realisierungsbeispiel Abs 8

TIMER0 für den PWM-Ausgang konfigurieren

Dimensionierung der benötigten Variablen

```
Dim Helligkeit as Byte
Dim Lo as Byte
Dim Hi as Byte
```

Den PWM-Ausgang auf PORTB.6 festlegen

Konfigurierung des Timers

```
PWM Alias PORTB.6
Config PWM = Output
```

Konfigurierung des Timers

```
'Timer0. PWM-Timer
Config TIMER0 = Timer, Prescale=64
On Timer0 Timer0_isr
Enable TIMER0
```

Setzen der Initialisierungswerte der Variablen

```
Helligkeit = 63
Lo = 255 - Helligkeit
Hi=255 - Lo
PWM = 1
```

Mit `Enable INTERRUPTS` kann der Timer seine Arbeit aufnehmen.
Siehe auch Realisierungsbeispiel Abs 8.2

```
Enable INTERRUPTS
```

4. Strukturelemente der Programmierung

4.1. Interrupts

Jeder Mikroprozessor sollte die Möglichkeit haben, für dringende Operationen den normalen Lauf unterbrechen zu können. Man sagt dem INTERRUPT. Die Tastatur des PCs muss immer sofort Wirkung zeigen. D.h. jeder Tastenanschlag unterbricht laufende Arbeiten zugunsten der Entgegenahme des neuen Tastensignals. Oder ein digitaler Wecker muss jede Minute prioritär nachschauen, ob eine Weckfunktion nach dem Minutenwechsel fällig wird.

Wir benutzen 2 Interruptroutinen in unserer Anwendung. Die erste kommt von unserem Hardware-Timer 1, der Zeitbasis für das Programm in Zehntelsekunden. Und was nach einem Interrupt gemacht werden soll, steht in der Subrouten nach dem Label: Timer1_isr:

Entsprechend geht es auch mit dem TIMER0, der für das PWM-Signal immer beim Zählerstand 255 einen Interrupt auslöst, der zur Routine nach folgendem Label führt: Timer0_isr:

In einer Interrupt-Routine sollte möglichst nur das Nötigste stehen, damit das zyklische Programm weitermachen kann. Mit anderen Worten, muss nach einem Interrupt eine längere Arbeit gemacht werden, dann soll diese innerhalb des normalen zyklischen Ablaufs abgewickelt werden. Der Interrupt definiert nur den Start.

4.2. Die Subroutine GOSUB

Es gehört zum guten Programmierstil, Programme in kurze, funktional zusammengehörende Programmteile zu unterteilen. Dazu muss man die Möglichkeit haben, in der Programmiersprache solche Programmteile voneinander zu separieren. Basic in seinen Anfängen hat dazu die Subroutine bereitgestellt und es gibt die Subroutine in allen Programmiersprachen. Wenn ich z.B. in einem Programm einen Testmodus habe, der nur selten und nach Bedarf ausgeführt wird, dann kann ich diesen in eine Subroutine stecken und im Hauptprogramm steht davon nur ein Hinweis, wie wir im

folgenden Beispiel sehen werden. In einer Subroutine sind alle für das Hauptprogramm deklarierten und initialisierten Variablen gültig.

Hauptprogramm

Wenn Taste Key gedrückt, wechselt das Programm zum Testmodus über.

```
Do          'zyklisches Hauptprogrmm

  If Key = 0 Then
    PORTD = &H00
    GOSUB Testmodus
    Sechsminuten = 104
    Exit For
  End if
  . . .
Loop

Testmodus:
  Sechsminuten = 0
  'Hier folgt alles, was im Test
  'abgearbeitet werden soll.
  . . .
Return
```

Das Label `Testmodus:` gibt an, wo die Subroutine beginnt.
Dann folgt alles, was in der Subroutine gemacht werden muss.
Mit `Return` kehrt das Programm zum Hauptprogramm zurück und arbeitet dort mit der nächsten Instruktion nach dem GOSUB-Befehl (`Sechsminuten = 104`) weiter. In unserem einfachen Beispiel kommen nur Subroutinen vor, die von einem Interrupt ausgelöst werden. Das Programm ist sonst so kurz, dass auf Subroutinen zum Zweck der Unterteilung des Programmes in überschaubare Befehlsblöcke verzichtet wurde. Bei professionellen Programmierern hat die Subroutine einen fahlen Beigeschmack, sie verwenden stattdessen Prozeduren.

4.3. Die Prozedur CALL

Prozeduren unterscheiden sich von Subroutinen dadurch, dass lokale Variablen innerhalb einer Prozedur abgeschirmt vom Hauptprogramm initialisiert werden können. Im Prozedur-Aufruf muss man aber die Variablen, die innerhalb der Prozedur bearbeitet werden sollen, der Prozedur übergeben. Man kann die Variablen so übergeben, dass die Prozedur diese für das Hauptprogramm neu berechnet oder man übergibt die Variablen so, dass nur ihr Wert innerhalb der Prozedur verwendet wird und dass der Variablenwert im Hauptprogramm unangetastet bleibt. In unserem kleinen Lernprojekt brauchen wir noch keine Prozeduren.

4.4. Die Funktion DECLARE FUNCTION

Man kann in BASCOM auch Funktionen deklarieren. Ganz einfach könnte die Funktion $y = x^2$ sein. Mit der Deklaration gibt man der Funktion einen Namen, z.B. Quadrat und gibt auch an, welche Variable von der Funktion verarbeitet werden soll. Bei der Deklaration gibt man den Rechnungsgang ein, der zur Funktion gehört. Im Programm kann man dann die Funktion direkt wie folgt verwenden: Y = Quadrat(Kantenlaenge). Man kann mit einer Funktion wie mit einer Variablen rechnen: Volumen = Quadrat(Kantenlaenge) * Hoehe

Bei einer Funktion kann man wie bei einer Prozedur auch mehrere Variablen übergeben. Im Prinzip wie folgt: Volumen(Grundflaeche, Hoehe)

In unserem einfachen Beispiel brauchen wir das nicht.

5. Programmierung

In diesem Abschnitt behandeln wir primär Befehle, die im Beispielprojekt später auch verwendet werden. Wer dieses Beispiel begriffen hat, kann später selbstständig weitere Befehle und Möglichkeiten in der Sprachreferenz von BASCOM nachschlagen, ausprobieren und anwenden und sich so in die Programmiertechnik weiter einarbeiten.

5.1. Zahlendarstellung

Dezimalzahlen werden mit Punkt als Dezimaltrenner geschrieben 123.45

Bits werden mit 0 und 1 dargestellt. Wenn man im Programm Alias-Namen einsetzt, kann man False = 0 und True = 1 verwenden.

Binärzahlen werden häufig als HEX-Zahlen geschrieben. Auch Programmcode wird zuweilen im HEX-Format weitergegeben. Der Vorteil des HEX-Formats ist, dass nur 16 verschiedene Zeichen vorkommen, und dass diese nicht Gefahr laufen, wegen länderspezifischen Tastaturnormen nicht verstanden zu werden, ausgenommen natürlich Chinesisch und Arabisch.

HEX-Zahlen

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dezimal															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Schreibweise im Programmcode:

&H00	&H01	&H02	&H03	&H04	etc	.	.	.	.	.	.		&H0D	&H0E	&H0F

Weitere Beispiele: &HFFFF = 65'535 dezimal, oder &H8000 = 32'768 dezimal.
Wissenschaftliche Taschenrechner können Dezimalzahlen in HEX-Zahlen umrechnen.

Binärzahlen können auch in Bit-Form geschrieben werden. Jede Stelle entspricht einer Potenz zur Basis 2. Einer Variablen vom Typ Byte mit 8 Bit kann wie folgt ein Binärwert zugewiesen werden:

Variable = &B00000001 (Dezimal 1) oder Variable = &B00100110 (Dezimal 38)
Variable = 38 ist im Code auch möglich.

5.2. Operatoren

Zum Rechnen gibt es die Operatoren + - * /
für Addition, Subtraktion, Multiplikation, Division
$y = x^2$ y = x ^2

Ganzzahloperatoren für die Division \ MOD
Der Rückschräger dividiert die Ganzzahl durch eine Ganzzahl und gibt das Resultat ohne Kommastelle als Ganzzahl aus. MOD gibt den Rest der gleichen Operation aus. Diese Operationen lassen sich mit Datentypen Byte, Integer und Long ausführen.
Also 27 \ 5 = 5
Und 27 MOD 5 = 2

Vergleiche: > < = <= <=
A < B, A > B, A = B, A kleiner oder gleich B: A <= B etc.

Bemerkung: In C-Sprache unterscheidet man den Vergleich auf Gleichheit von einer Zuweisung:

Gleichheit von A und B wird so abgefragt: A == B, und eine Zuweisung wird so dargestellt A = 27.

Logische Operatoren AND OR XOR
Diese Operatoren könne Bit-weise oder auch Byte-weise angewendet werden.

A und B seien ein Bit:
A = 1, B = 1 → A AND B = 1
A = 1, B = 0 → A AND B = 0

A = 1, B = 0 → A OR B = 1
A = 0, B = 0 → A OR B = 0

Exklusiv ODER bzw. Vorzeichenmultiplikation"
A = 0, B = 1 → A XOR B = 1
A = 1, B = 1 → A XOR B = 0
A = 0, B = 0 → A XOR B = 0
A = 1, B = 0 → A XOR B = 1

AND-Verknüpfung von 2 Bytes. Das ginge auch mit Datentypen Integer und Long

```
A =          &B00101101
B =          &B10001010
C = A OR B   &B10101111
```

5.3. Programmbefehle

Eine Schleife eine bestimmte Anzahl mal durchlaufen FOR NEXT

Zum Beispiel ein Schieberegister um 8 Schritte weiterschieben.

```
For N = 1 To 8
  'Schieberegister um 1 Schritt
  'weiterschieben
Next N
```

Man kann eine FOR-NEXT-Schleife auch mittendrin mit EXIT verlassen, wenn nötig. Dies gilt auch für die Schleifen von DO und WHILE und auch für Prozeduren (EXIT SUB) und Funktionen (EXIT FUNCTION)

```
For N = 1 To 8
  'Normale Aufgaben in der Schleife
  If Fehler = True then EXIT FOR
  . . .
Next N
```

Die Endlosschleife DO LOOP

Das Programm geht von Do bis Loop und kehrt dann wieder nach Do zurück usw.

```
Do
  . . .
Loop
```

Man kann bei Loop auch eine Abbruchbedingung einfügen

```
Do
  . . .
Loop until abbrechen = True
```

Man kann auch eine Bedingung am Anfang einer Endlosschleife festlegen. Der Befehl heisst dann WHILE …. WEND

```
While immerweiterso = True
  . . .
Wend
```

Eine Bedingung zum Verzweigen: IF THEN ELSE END IF

Einen Programmteil nur ausführen, wenn die Bedingung erfüllt ist.

```
If New_time = True then
  . . .
End if

If Bedingung 1 Then
  . . .
Else if Bedingung 2 Then
  . . .
Else
  . . .
End if

If Key=0 AND Keyabgefragt=False then
  . . .
```

Allgemeiner:
Es können mehrere verschiedene Bedingungen untereinandergesetzt werden. Else behandelt optional den Fall, wenn keine der vorangegangenen Bedingungen erfüllt war.

Als Bedingung dürfen auch mehrere Vergleiche gleichzeitig angestellt werden

Was man in BASCOM nicht kann und in höheren Programmiersprachen aber möglich ist, dass man innerhalb der Bedingung noch Berechnungen ausführt.
Das geht also nicht: `If A * B > 200 then . . .`

Fallweise unterschiedliche Operationen durchführen: SELECT CASE

Im Fall Case 0 und Case else soll die Helligkeit nicht verändert werden:
'Helligkeit = Helligkeit
Es dürfte auch einfach nichts in der Zeile stehen.
Es werden von `Auswahl` die Werte 0, 1, 2, 3 verwendet. Andere Werte sollten nicht vorkommen.

```
Select Case Auswahl
  Case 0
    'Helligkeit = Helligkeit
  Case 1
    Helligkeit = Helligkeit OR &B00001111
  Case 2
    Helligkeit = Helligkeit OR &B00011111
  Case 3
    Helligkeit = Helligkeit AND &B00111111
  Case else
    'Helligkeit = Helligkeit
End Select
```

Die höherwertigen oder niederwertigen Bits einer Integer-Variable selektieren. LOW HIGH

Low(Variable)

High(Variable)

```
Dim Helligkeit as Byte
Dim Register as Integer

Helligkeit = low(Register)

Register      = &B0110101101000101
Helligkeit    =         &B01000101

high(Register) = &B01101011
```

Wartezeit WAIT bzw. WAITMS

Will man das Programm an einem bestimmten Ort eine bestimmte Zeitlang unterbrechen, dann lässt sich dies mit WAITMS bewerkstelligen. WAITMS 20 bedeutet, dass das Programm 20 Millisekunden lang stehen bleibt.

Diese Instruktion ist unter Informatikern verpönt. Sie unterbricht den Prozessor und damit auch z.B. die Interrupt-Routinen. Eine Uhr-Funktion im Programm könnte dadurch auch unterbrochen werden. Und wenn man die Funktion braucht, dann sollte man sie nur mit ganz kurzen Zeiten verwenden. Längere Zeiten soll man auf jeden Fall z.B. von einem 1/10-Sekundentakt ableiten.

Die WAIT-Funktion lässt sich im BASCOM-Simulator schlecht simulieren. Der Simulator bleibt praktisch stehen.

Die Bits in einem Register nach rechts schieben SHIFT

Für den Zufallsgenerator im Projekt brauchen wir ein rückgekoppeltes 15-Bit Schieberegister. Bei jedem Takt muss man alle Bits um einen Schritt nach rechts verschieben.
Right gibt die Schieberichtung an. Gegenteil: Left
1 gibt an, dass alle Bits um 1 Platz verschoben werden.

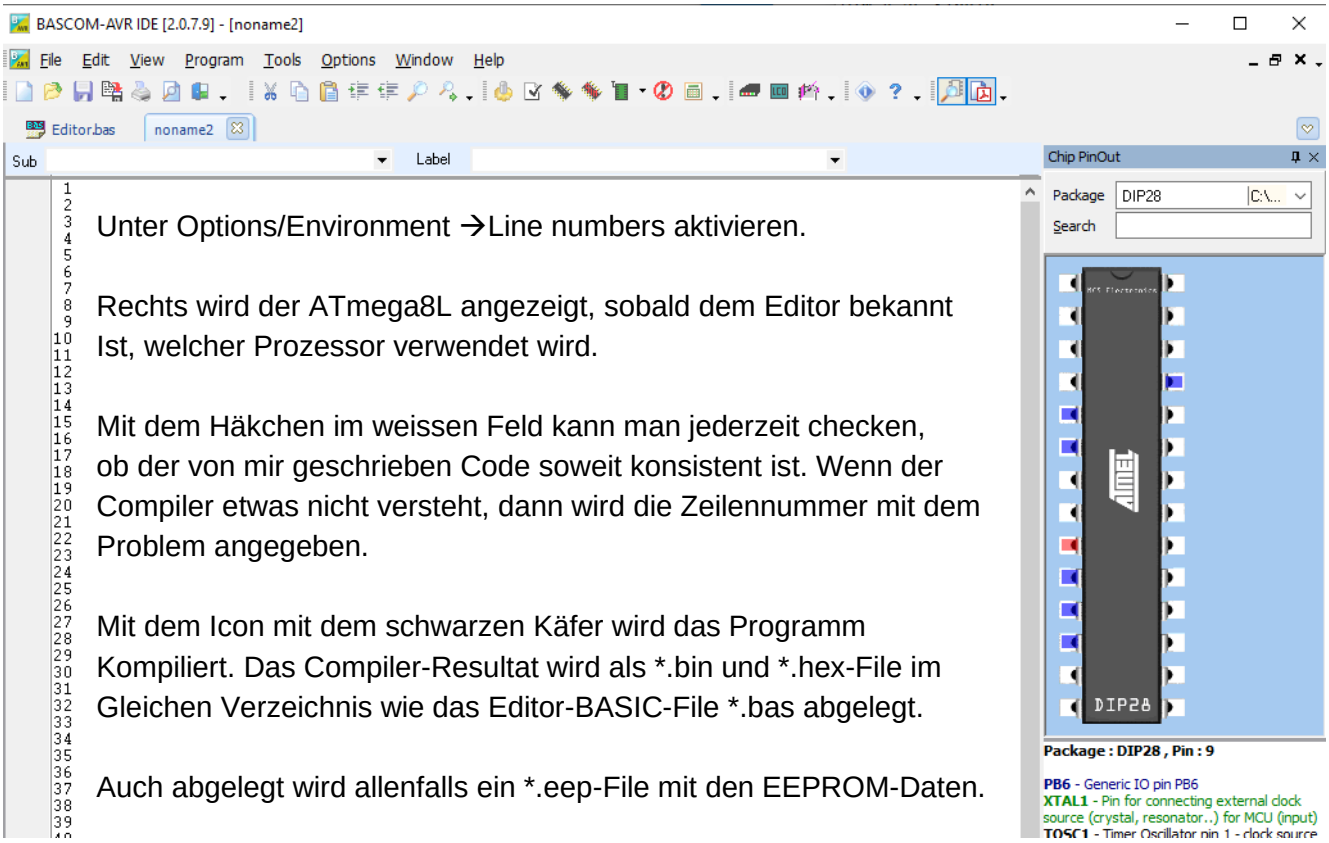
```
Dim Register as Integer

. . .
Register = &B1000011010110011
Shift Register, right, 1
. . .

Resultat-> &B0100001101011001
```

Beachte: Das niederwertigste Bit wird beim Schiebevorgang rechts aus dem Register hinausgeworfen. Und, der erste Platz ist nach einer Schiebung mit 0 besetzt.

6. Der BASCOM-Editor



Unter Options/Environment →Line numbers aktivieren.

Rechts wird der ATmega8L angezeigt, sobald dem Editor bekannt Ist, welcher Prozessor verwendet wird.

Mit dem Häkchen im weissen Feld kann man jederzeit checken, ob der von mir geschrieben Code soweit konsistent ist. Wenn der Compiler etwas nicht versteht, dann wird die Zeilennummer mit dem Problem angegeben.

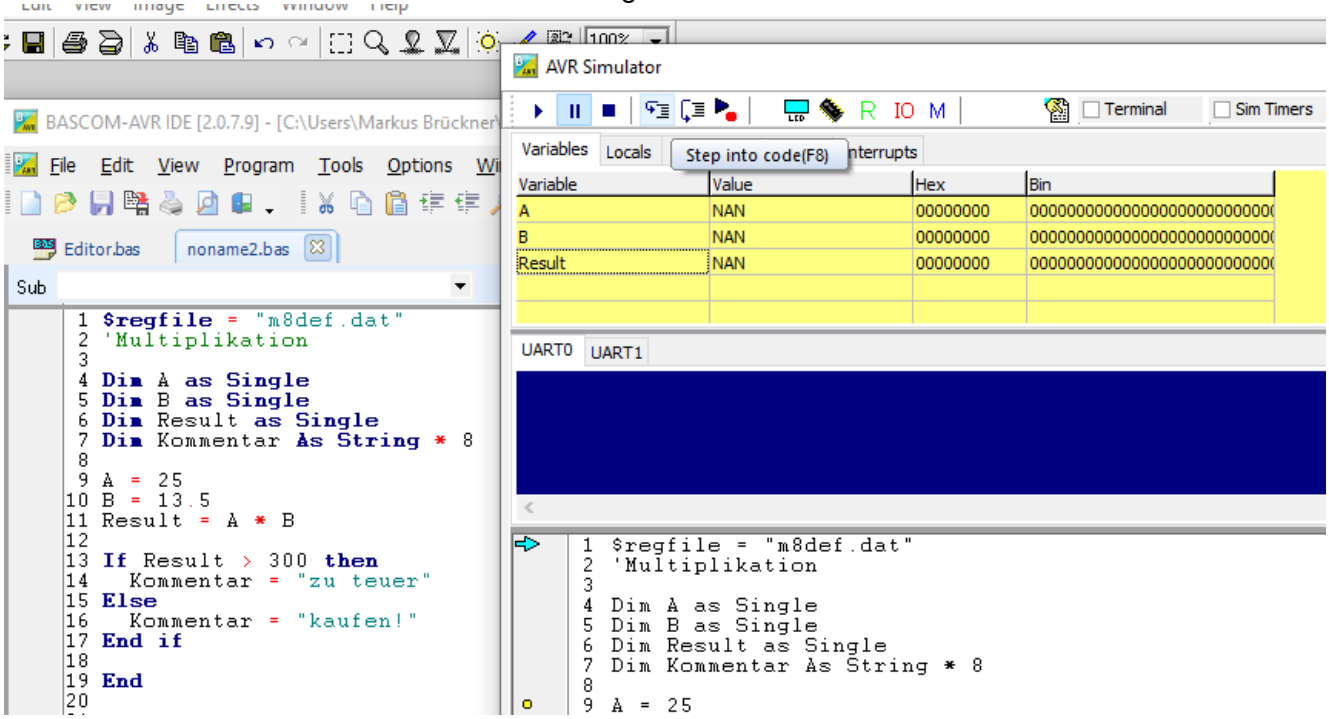
Mit dem Icon mit dem schwarzen Käfer wird das Programm Kompiliert. Das Compiler-Resultat wird als *.bin und *.hex-File im Gleichen Verzeichnis wie das Editor-BASIC-File *.bas abgelegt.

Auch abgelegt wird allenfalls ein *.eep-File mit den EEPROM-Daten.

Das bin- oder hex-File wird mittels myAVR_ProgTool auf den Chip ins Flash-Memory geschrieben.

6.1. Der Simulator

Mit dem roten Käfer-Icon wird der Simulator aufgerufen.

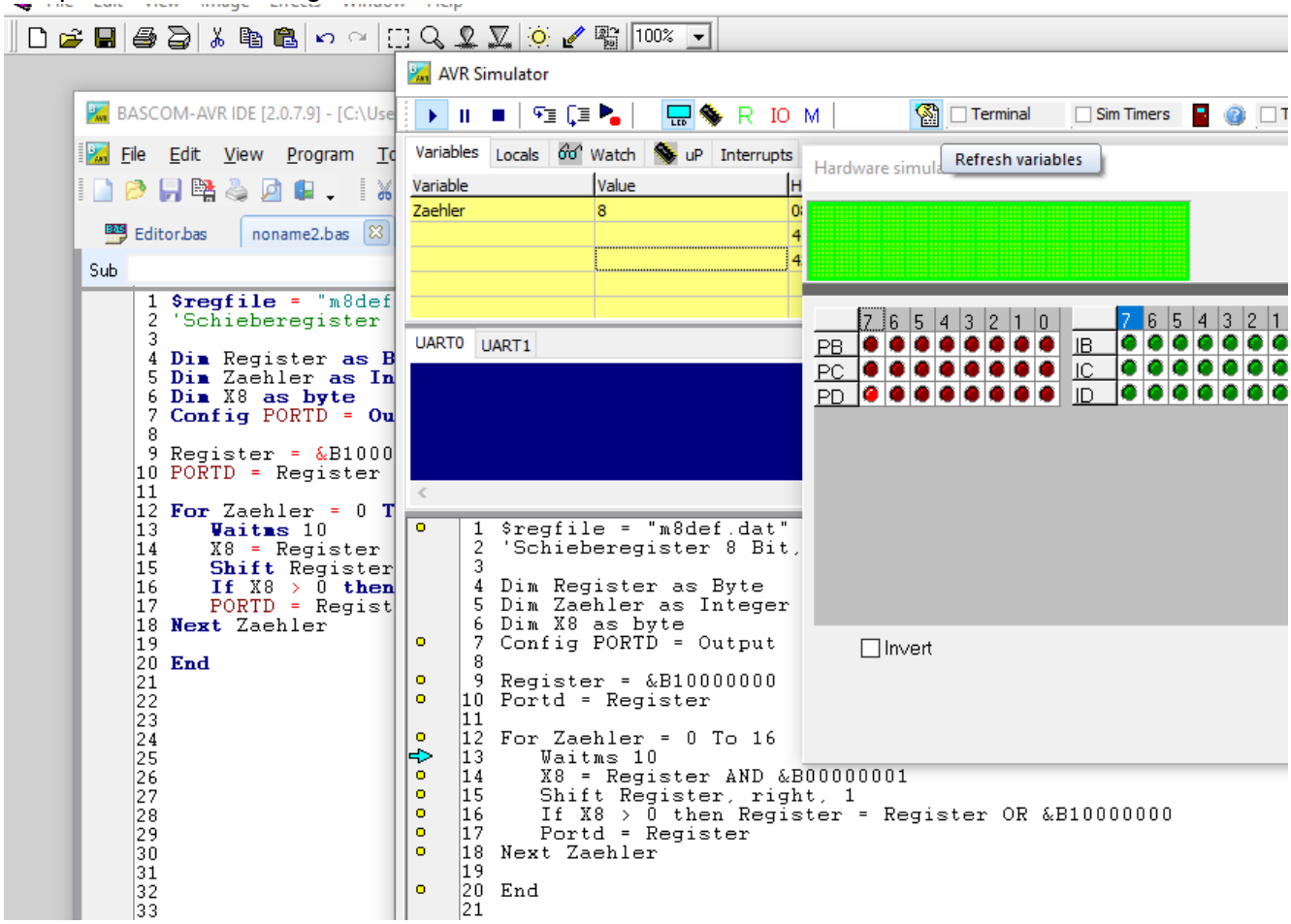


Das Programm kann nun schrittweise [Icon Step into code] durchgegangen werden und die manuell unter Variable eingegebenen Variablen werden bei jedem Schritt angezeigt. Ein String kann nicht

angezeigt werden. Wir sehen aber trotzdem, wie das Programm nach dem Vergleich `If Result > 300 then` verzweigt.

6.2. Ausgangsports im Simulator darstellen

Beispiel 8-Bit Schieberegister

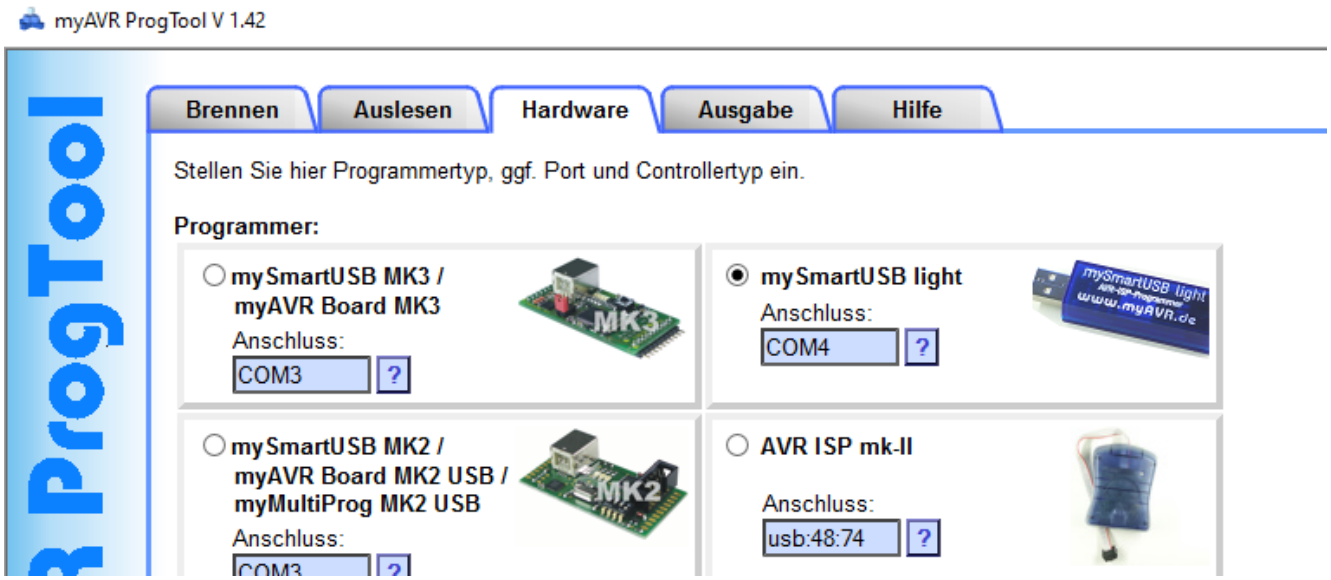


Damit Portd bei jedem Schritt neu angezeigt wird, muss auch [Refresh variables] aktiviert sein.
Will man das Programm neu simulieren, muss man ■ und dann wieder ▶ drücken.

7. myAVR_PogTool

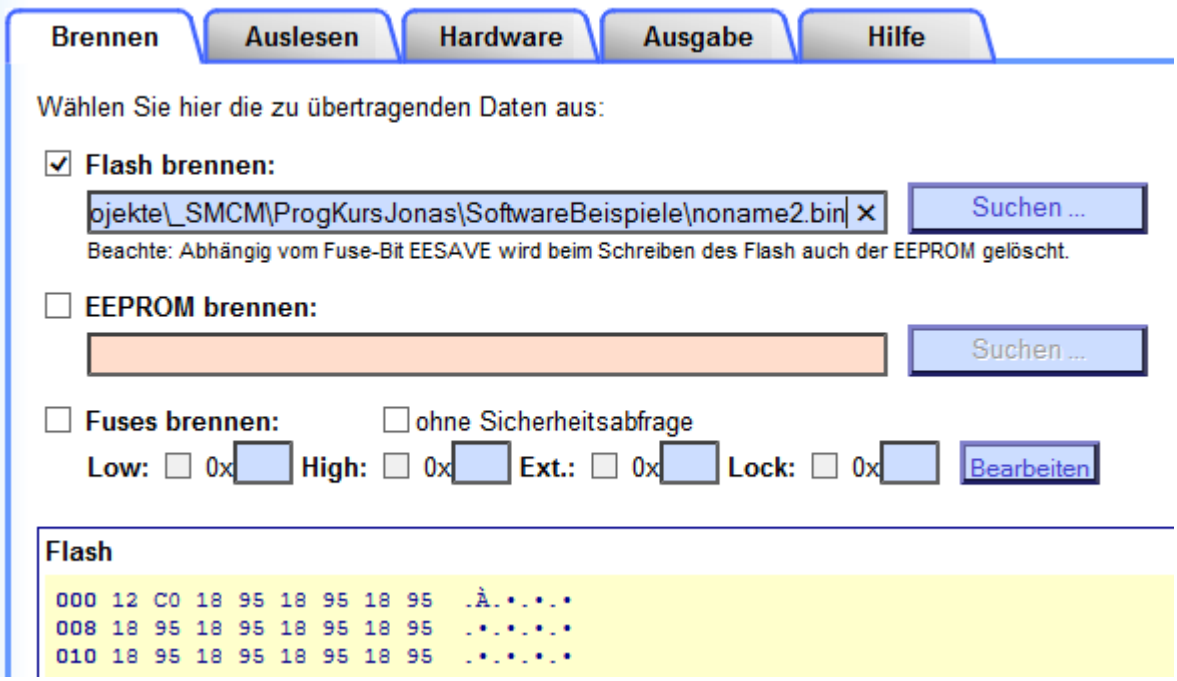
7.1. Hardware-Einstellung

Interface und COM-Schnittstelle eingeben.



7.2. Das Programm in die Applikation laden

Das *.bin-File aufrufen und dann Taste Brennen(F5) drücken. Wenn mySmartUSB über den ISP-Stecker mit der Applikation verbunden ist und alles richtig geht, sollte das Programm jetzt auf die Applikation geschrieben werden.



Beachte: Ich habe schon festgestellt, dass nach Programmänderungen und Neu-Kompilieren (Schwarzes Käfer-Icon) das Flash mit der alten Programm-Version beschrieben wurde. In diesem Falle habe ich abwechselnd das *.hex-File und dann wieder das *.bin File unter "Flash brennen" aufgerufen. Dies hat geholfen.

8. Realisierungsbeispiele

8.1. Realisierungsbeispiel Timer1

Berechnung Timerstart bei Prescale = 64:
Bei einer Prozessorfrequenz von 1MHz und Prescale 64 zählt der Timer pro Sekunde
1'000'000 / 64 = 15'625 Takte.
Will man einen Zehntelsekundentakt, dann ist der Wert noch durch 10 zu teilen:
15'625 / 10 = 1'562
Der Timer braucht zum Abzählen einer 1/10-tel Sekunde 1562 Takte bei Prescale 64.
Den Timerstart muss man daher wie folgt setzen
Timerstart = 65'536-1'562 = 63'974
Dimensionierung von Variablen und Konstanten

New_time ist ein Flag das bei jedem Interrupt auf `True` gesetzt wird und das im zyklischen Hauptprogramm jeweils auf `False` gesetzt wird, sobald alle Anweisungen, die jede Zehntelssekunde abgearbeitet werden sollen, erledigt sind

Konfigurierung TIMER1

Label der Subroutine, die beim Zählerreset abgearbeitet werden soll

Timer1, Zeitbasis 0.1 sec

Befehle im zyklischen Hauptprogramm, die alle 1/10-Sekunden abgearbeitet werden sollen
Mit dem Flag `New_time` wird dafür gesorgt, dass die Anweisungen zwischen If und End If jede Zehntelsekunde genau nur 1mal durchlaufen werden.

Die Interrupt-Routine

```
Const Timerstart = 63974

Dim New_time as Boolean









Config TIMER1 = Timer, Prescale = 64

On TIMER1 Timer1_isr

Timer1 = Timerstart
New_time = False
Enable TIMER1
Enable INTERRUPTS

Do
   If New_time = True then
      . . .
      New_time = False
   End if
Loop


Timer1_isr:
   Timer1 = Timerstart
   New_time = True
Return
```

8.2. Realisierungsbeispiel Timer0

Die Puls-Weiten Modulation für den LED-Ausgang "flackerndes Lagerfeuer"

Der Ausgang ist wie schon gesagt eine schnelle Folge von Impulsen und Lücken. Nach unserer Konvention leuchtet die LED, wenn der Ausgang auf 0 steht und sie ist dunkel, wenn der Ausgang auf logisch 1 (5V Vcc) steht. Wenn die LED nicht leuchten soll, dann bleibt der Ausgang dauernd auf 1. Soll er 100% leuchten, dann ist der Ausgang dauernd auf 0.

./.

Dimensionierung der Variablen

Konfigurierung des PWM-Ausgangs

Setzen der Initialisierungswerte der Variablen

Konfigurierung des Timers

Befehle im zyklischen Hauptprogramm

Subroutine, die über Interrupt vom TIMER0 aufgerufen wird.
TIMER0 zählt ab Lo aufwärts

TIMER0 zählt ab Hi aufwärts

```
Dim Helligkeit as Byte
Dim Lo as Byte
Dim Hi as Byte

PWM Alias PORTB.6
Config PWM = Output

Helligkeit = 63
Lo = 255 - Helligkeit
Hi=255 - Lo
PWM = 1

Config TIMER0 = Timer, Prescale = 64
On Timer0 Timer0_isr
Enable TIMER0
Enable INTERRUPTS

Do
   . . .
   Helligkeit = low(Register)
   Lo = 255 - Helligkeit
   Hi = 255 - Lo
   . . .
Loop

Timer0_isr:
If PWM = 1 then
     PWM = 0
     TIMER0 = Lo
   Else
     PWM = 1
     Timer0 = Hi
   End if
Return
```

8.3. Realisierungsbeispiel: EEPROM initialisieren, beschreiben und lesen

Wir könnten den Tastschalter auf dem Print dazu verwenden, die Helligkeit und den Helligkeitsunterschied des Feuer-Flackerns in verschiedenen Konfigurationen abzurufen. Die letzte Einstellung der Auswahl bleibt immer netzausfallsicher erhalten.

Die Verwendung des EEPROMS muss gleich zu Beginn mit einem Compilerbefehl angekündigt werden. Man darf hier nicht mehr Platz belegen als die 512 Byte, über die der EEPROM-Speicher verfügt.

```
$regfile = "m8def.dat"

$eeprom
Data &HFF                'Adresse 0
Data &B00000000, &HFF, &HFF, &HFF,
     &HFF, &HFF, &HFF    'Adresse 1-7
Data "Lager-  feuer   Jonas"
$data
```

./.

Dimensionierung einer EEPROM-Variablen zum Lesen und Schreiben in diesen Speicherbereich:
Die Variable heisse Ee_Auswahl und sei vom Datentyp Byte (8Bit).
At 1 bedeutet, dass die Variable ab Adresse 1 im EEPROM-Speicher abgelegt ist.

```
Dim Ee_Auswahl As Eram Byte At 1
```

Bemerkung: Auf Adresse 0 ist oben &HFF eingeschrieben. Die Adresse wird in unserem Beispiel nicht verwendet.

Den Wert von Ee_Auswahl beim Hochstart aus dem EEPROM **lesen** und auf dem PORTD binär anzeigen:

```
Auswahl = Ee_Auswahl
PORTD = Auswahl XOR &HFF
```

Zyklisches Hauptprogramm:
Alle Zehntelssekunde wird `New_time = True` gesetzt. Wenn der Key nicht gedrückt ist, dann wird hier `Keyabgefragt = False` gesetzt.

Nach einem entprellten Tastendruck einen neuen Wert von Auswahl ins EEPROM **schreiben** und auf dem PORTD anzeigen:

Die Zuweisung von `Auswahl` zu `Ee_Auswahl` löst das Schreiben des Wertes ins EEPROM ab Adresse 1 aus.

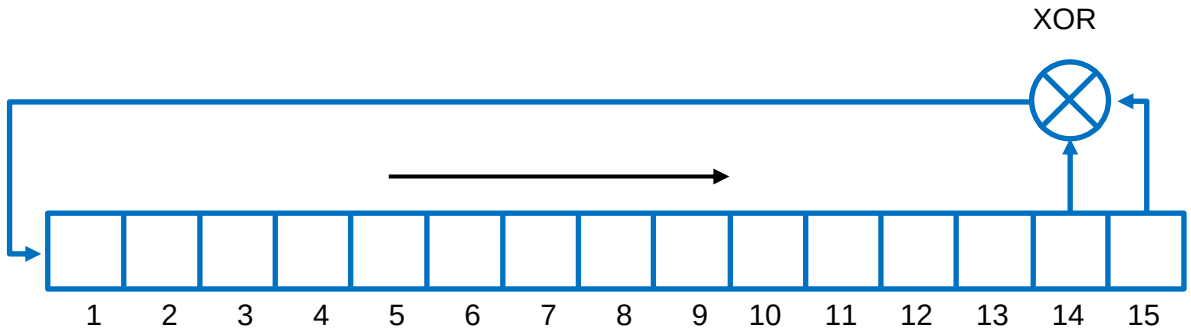
```
Do
  If New_time = True then
    If Keyabgefragt = True AND Key = 1 then
      Keyabgefragt = False
    End if
    New_time = False
  End if
  . . .
  If Key = 0 AND Keyabgefragt = False then
     Waitms 20
     If Key = 0 then
        Incr Auswahl
        If Auswahl >= 4 then Auswahl = 0
        Ee_Auswahl = Auswahl
        PORTD = Auswahl XOR &HFF
     End if
     Keyabgefragt = True
  End if
  . . .
Loop
```

Beachte: Im Beispiel wird `Auswahl` zur Anzeige auf einem PORT mit angeschlossenen LEDs mit XOR invertiert, damit Einsen des Wertes zum Aufleuchten der LEDs führen und umgekehrt Nullen die LEDs ablöschen.

Und noch: Weshalb musste Key "entprellt" werden?
Ein Taster hat einen mechanischen Kontakt mit Kontaktfeder. Beim Schliessen kann es durchaus vorkommen, dass der Taster schliesst und aber in Millisekunden danach nochmals zurückschnellt und wieder kurz öffnet. Die Entprellung soll dafür sorgen, dass in einem solchen Fall die Taste nur einmal abgefragt wird. Auf einer PC-Tastatur würden ohne Entprellung immer wieder Buchstaben mehrfach angeschlagen erscheinen. Als Zeit für eine Entprellung wähle ich 20ms. Nach dieser Zeit hat sich der Kontakt beruhigt und die Gefahr der Mehrfachabfrage ist vorbei.

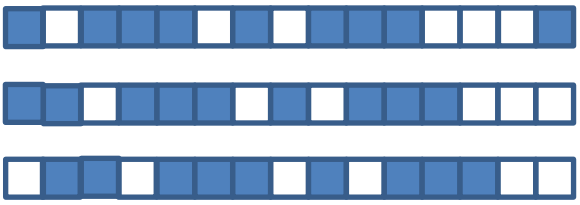
9. Applikations-Algorithmus: Realisierung eines Zufallsgenerators

In der Verschlüsselungstechnik sind vielerorts quasizufällige Bit-Folgen gefragt. So etwas ähnliches brauchen wir für ein flackerndes Lagerfeuer auch. Man verwendet dazu z.B. rückgekoppelte Schieberegister, möglichst Maximallängen-Schieberegister. Wir verwenden hier ein 15 Bit Register. Dieses Register erzeugt eine Sequenz der Länge 2 ^15 - 1 = (32'768 – 1) = 32'767. Nach 32'767 Takten beginnt die Sequenz wieder von vorne. Alle Bits=0 kommt in der Sequenz als einziger Zustand nicht vor.



Funktion: Bei jedem Takt verschieben sich alle Zelleninhalte um 1 Position nach rechts.
Der Inhalt der Position 14 und 15 wird vor dem Taktschlag ausgelesen und mit XOR verknüpft. Beim Taktschlag wird das Resultat dieser XOR-Verknüpfung in Zelle 1 eingelesen. Der Inhalt der Zelle 15 fällt beim Taktschlag rechts aus dem Register hinaus.

Beispiel: Die Zellinhalte des Registers in 3 aufeinanderfolgenden Takten:



Software-Realisierung mit einer Integer-Variablen `Register`:
Das Register hat 16 Zellen, aber wir brauchen davon nur die Zellen 1-15. Das stört uns nicht.

Zur Zwischenspeicherung der Zustände in Zelle 14 und 15 brauchen wir die Bit-Variablen `X14bit` und `X15bit`. Da wir die Integer-Variable nicht direkt mit einer Bit-Variablen verrechnen können, machen wir den Umweg über die zwei Integer-Variablen `X14` und `X15`. Wir maskieren Register mit Einer AND-Operation so, dass im einen Fall nur Bit 14 und im anderen Fall nur Bit 15 übrigbleibt. Ist diese Variable dann >0, wissen wir, das `X14bit` auf 1 oder entsprechend das `X15bit` auf 1 gesetzt werden muss.

Die XOR-Verknüpfung von `X14bit` mit `X15bit` legen wir in `X01bit` ab.

```
X15 = Register AND &B0000000000000010
If X15 > 0 then X15bit = 1 Else X15bit = 0
X14 = Register AND &B0000000000000100
If X14 > 0 then X14bit = 1 Else X14bit = 0
X01bit = X15bit XOR X14bit
Shift Register, right, 1
If X01bit = 1 then Register = Register OR &B1000000000000000
```

In der letzten Zeile wird der Wert von `X01bit` nach der Shift-Operation in der ersten Zelle von `Register` abgelegt.

Software Projekt "Flackerndes Lagerfeuer" mit einem ATmega8L

Beispiel: Rechenbeispiel und 8 Bit Register sind unter Simulator angegeben worden.

Sprachreferenz
https://avrhelp.mcselec.com/