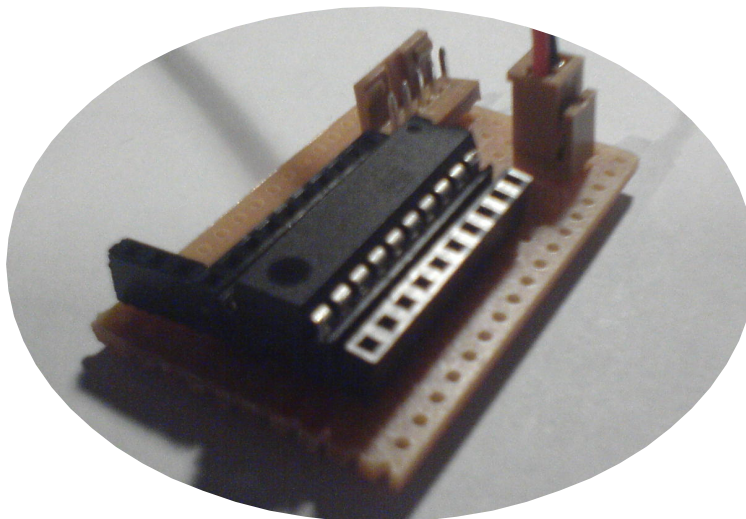


# **REALISIERUNG DER LOCAD-SCHALTUNGEN**

## **DURCH PROGRAMMIERUNG EINES**

### **AVR-MIKROCONTROLLERS (ATTINY2313)**



## INHALTSVERZEICHNIS

<b>1 Einführung</b>	<b>3</b>
1.1 Das Thema.....	3
1.2 Was ist ein AVR-Mikrocontroller?.....	3
1.2.1 Der ATTiny2313.....	4
1.2.2 Die Programmierung.....	5
1.2.2.1 Anschluss am PC.....	5
1.2.2.2 Die Compiler.....	6
1.2.2.3 Programmierung mit dem AVR-GCC.....	7
<b>2 Die Schaltungen</b>	<b>12</b>
2.1 Die umschaltbare Rechenschaltung.....	12
2.1.1 Aufbau.....	12
2.1.2 Der Programmcode.....	14
2.1.3 Erläuterung des Programmcodes.....	16
2.2 Erweiterte umschaltbare Rechenschaltung.....	19
2.2.1 Aufbau.....	19
2.2.2 Der Programmcode.....	22
2.2.3 Erläuterung des Programmcodes.....	24
<b>3 Fazit</b>	<b>26</b>
<b>4 Materialien</b>	<b>29</b>
4.1 Schaltskizzen und Fotos.....	30
<b>5 Verzeichnisse</b>	<b>38</b>
5.1 Literaturverzeichnis.....	38
5.2 Abbildungsverzeichnis.....	39
5.3 Tabellenverzeichnis.....	40



# 1 EINFÜHRUNG

## 1.1 Das Thema

Ich habe als Thema ein sehr praxisnahes Thema gewählt, da ich mich in meiner Freizeit auch gerne mit technischen Elementen auseinandersetze. Als Thema habe ich die Realisierung der im Seminarfach kennen gelernten, simulierten Schaltungen durch Programmierung eines programmierbaren AVR-Mikrocontrollers an einigen Beispielen gewählt. Darunter hat man sich vorzustellen, dass ich die Schaltungen nicht so nachbauen werde, wie sie im erhaltenen Skript<sup>1</sup> dargestellt sind, sondern werde ich sie, sehr einfach ausgedrückt, in Programmcode „übersetzen“. Ich werde somit z.B. den AVR-Mikrocontroller so programmieren, dass dieser zwei eingegebene Binärzahlen addiert und das Ergebnis ausgibt. Auf folgende Fragestellung möchte ich eine Antwort finden: Ist eine Realisierung der im Seminarfach kennen gelernten Schaltungen mit einem AVR-Mikrocontroller möglich?

## 1.2 Was ist ein AVR-Mikrocontroller?

Ein AVR-Mikrocontroller ist ein 8-Bit Mikrocontroller der Firma Atmel. Dieser lässt sich mithilfe eines PCs programmieren, um bestimmte Aktionen auszuführen. Je nach Modell besitzt der Controller unterschiedlich viele Pins („Beinchen“) an die externe Bauteile, wie LEDs, Schalter oder sogar LC-Displays angeschlossen werden können, um Ein- und Ausgaben zu realisieren. Die Grundfunktion der Mikrocontroller besteht also darin die Pins auf VCC (Versorgungsspannung) oder GND (Masse) zu schalten (Schaltzustand 0 oder 1) um externe Bauteile zu steuern. Die Modelle werden in ATTiny und ATmega gegliedert, wobei die ATTiny Controller die weniger leistungsfähigen und kostengünstigeren Controller darstellen. Somit lassen sich mit solchen Controllern auch die Schaltungen, die im Seminarfach eine Rolle spielen, nachbauen, da die Programmierung auch viele Rechenfunktionen unterstützt. Von der Firma Atmel gibt es auch vorgefertigte sog. „Evaluation Boards“ wie das



Abbildung 1: AVR-Mikrocontroller [Wikipedia]

<sup>1</sup> Karl-Heinz Loch: „Technische Informatik mit LOCAD 2002“

STK500. Dies ist ein Experimentierboard, das schon viele Schalter, LEDs und Erweiterungsmöglichkeiten per PIN-Leisten mitbringt. Diese liefern auch eine Anleitung und Programmierungssoftware mit sich. Ich werde hier aber ein selbst gelötetes Board mit einem ATTiny2313 Mikrocontroller benutzen, da dieses wesentlich kostengünstiger ist und für meine Zwecke ausreicht. Darauf wird im folgenden Kapitel noch weiter eingegangen.

### 1.2.1 Der ATTiny2313

Den Mikrocontroller, den ich benutzen werde, ist einer der Serie ATTiny. Zu diesem Mikrocontroller möchte ich noch einige Daten geben.

ATTINY2313
Basierend auf der RISC-Architektur
Befehlssatz: 120
Bis zu 20 MHz (normal 8 MHz)
2 KB interner programmierbarer Flashspeicher
128 Byte EEPROM
128 Byte interner SRAM
18 Programmierbare I/O Pins

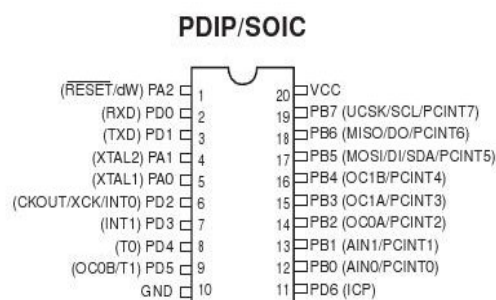


Abbildung 2: ATTiny2313 Aufbau [Datenblatt-AVR]

Tabelle 1: Daten des ATTiny2313 [Datenblatt-AVR]

In der rechten Skizze ist nun der Aufbau des ATTiny2313 dargestellt. Es sind 18 Pins mit der Bezeichnung  $PAX$ ,  $PBX$  und  $PDX$  zu erkennen. Diese 18 Pins können innerhalb des Programms als Ein- oder Ausgänge definiert werden um angeschlossene Taster oder LEDs anzusprechen. An die Pins GND und VCC wird die Spannungsquelle (in meinem Fall: universelles Schaltnetzteil) angeschlossen. Mit dem ATTiny2313 ist es auch möglich ein LC-Display zu steuern, jedoch sind dabei die Funktionen etwas begrenzt durch den 2KB großen Speicher. Für LC-Displays ist es praktischer den ATmega8 zu verwenden, da er mehr Speicher besitzt.

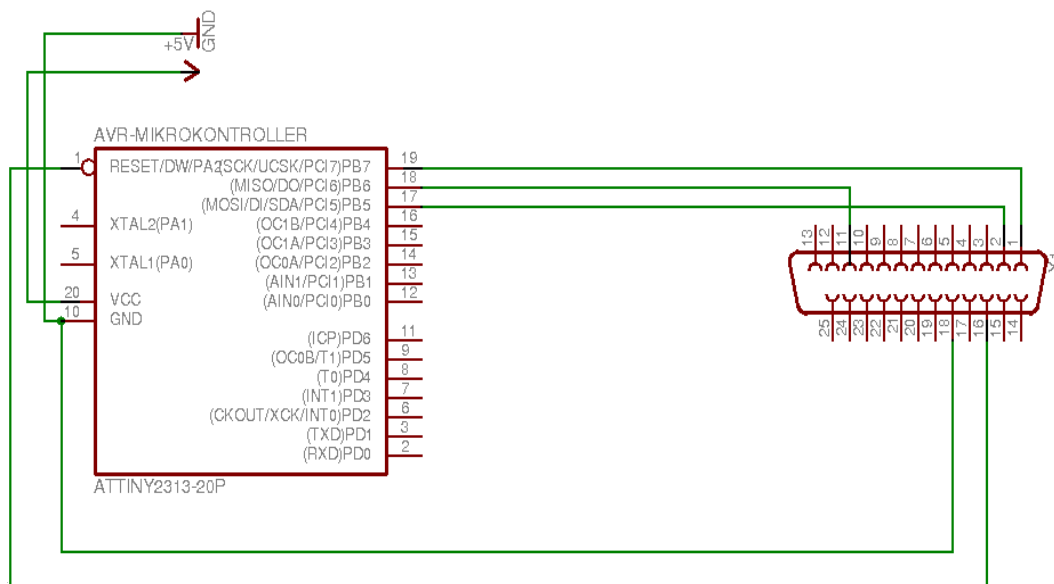
Die Angabe einer MHz Zahl in der Tabelle zeigt, dass der ATTiny2313, wie alle anderen AVR-Mikrocontroller, einen internen Taktgeber besitzt, welcher die Arbeitsgeschwindigkeit des Controllers bestimmt. Durch Anschließen externer Quarze (an die XTAL1/2 Pins) kann noch mehr Rechenleistung erreicht werden. Bei der Programmierung wird auf dem Flashspeicher (2KB) der kompilierte Programmcode gespeichert (RAM dient zur Zwischenspeicherung und EEPROM als zusätzlicher Speicher).

## 1.2.2 Die Programmierung

### 1.2.2.1 Anschluss am PC

Die Programmierung des AVR-Mikrocontrollers erfolgt generell über die 25 polige parallele Schnittstelle am PC, die früher auch zum Anschluss von Druckern benutzt wurde.

Seit noch nicht allzu langer Zeit gibt es auch „USB-Programmer“ zur Programmierung über USB. Des Weiteren ist es möglich mit dem AVR-Mikrocontroller über den seriellen (9 polig) Anschluss zu kommunizieren. Ich verwende die Programmierung über die 25 pol. parallele Schnittstelle, deshalb werde ich nun einmal eine Schaltskizze aufführen, die darstellt, wie der Controller an den PC angeschlossen wird.



Dies stellt den einfachsten, aber wirkungsvollen, Programmieradapter dar. Es gibt auch andere Programmieradapter, auch vom dem Hersteller Atmel, die eine höhere Kompatibilität und Geschwindigkeit bieten. In der folgenden Tabelle sind die PIN-Belegungen dargestellt.

PIN am Controller	PIN an der parallel Schnittstelle
17 (MOSI)	2
18 (MISO)	11
19 (SCK)	1
1 (RESET)	16
10 (GND)	18

Tabelle 2: PIN-Belegungen [Instruct]

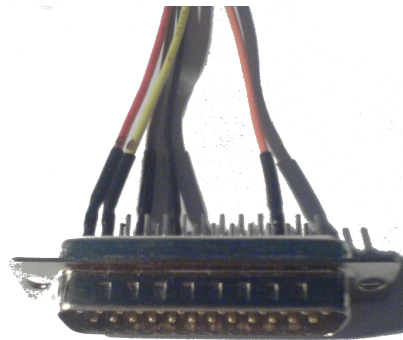


Abbildung 4: Kabel für die Programmierung (Foto)

### 1.2.2.2 Die Compiler

Nachdem nun deutlich geworden ist, wie der Controller an den PC angeschlossen wird, bleibt noch übrig, wie der Controller eigentlich programmiert wird. Also was für Compiler es gibt um den Programmcode in Maschinencode (in Form einer Hex-Datei) umzuschreiben und diesen auf den Controller zu übertragen.

Zum Programmieren des Controllers gibt es mehrere Compiler, die auf unterschiedlichsten Programmiersprachen basieren. Ich werde hier mal kurz die Wichtigsten darstellen.

Compiler	Sprache	Hersteller	Plattform
gcc-avr	C/C++	GNU (kostenlos)	Win32, Linux, Mac
AVR Studio	Assembler & C/C++	Atmel (kostenlos)	Win32
BASCOM	Basic	MCS Electronics (demo-version)	Win32
WinAVR	C/C++	(kostenlos)	Win32
SiSy AVR	Assembler & C/C++	Laser&Co Solutins	Win32

Tabelle 3: Wichtige verfügbare Compiler [Wikipedia]

Da ich selbst die Programmierung mit einem Linux System vornehme, habe ich mich für *gcc-avr* als Compiler entschieden. Der Compiler beinhaltet auch das schreiben des Maschinencodes auf den externen Controller (über *avrdude*). Alle aufgeführten Compiler außer der *gcc-avr* werden direkt mit einer Entwicklungsumgebung ausgeliefert. Das bedeutet für den *gcc-avr* Compiler, dass der Quellcode mit einem ganz normalen Texteditor erstellt wird. Für Anfänger ist es mit einer Entwicklungsumgebung einfacher einzusteigen und für größere Projekte sind Entwicklungsumgebungen durch ihre zahlreichen Möglichkeiten besser geeignet. Doch in meinem Fall reicht ein Texteditor vollkommen aus.

### 1.2.2.3 Programmierung mit dem AVR-GCC

Als Nächstes möchte ich die wichtigsten Befehle und Optionen des `gcc-avr` darstellen. Wie aus der Tabelle zu entnehmen ist, basiert der `gcc-avr` Compiler auf der Programmiersprache C/C++. Dies bedeutet es werden allgemein die von der Programmiersprache C bekannten Elemente verwendet (C++ basiert auf C). In der Programmiersprache C wird meistens eine Makefile zum Kompilieren verwendet. Diese enthält die nötigen Informationen für die Kompilierung. Durch Ausführen des Befehls 'make' (Installation von make vorausgesetzt, ist bei jedem gängigen C-Compiler dabei) in dem Verzeichnis, wo sich die Makefile Datei befindet, wird der Programmcode kompiliert. Ich werde da jetzt nicht weiter drauf eingehen, da dies eher nebensächlich ist.<sup>2</sup>

In dem nächsten Schritt erläutere ich einmal kurz die wichtigsten Inhalte und Befehle von C/C++, die bei der Programmierung verwendet werden.

Wörter die durch `/*` und `*/` eingeschlossen sind gelten als Kommentare, welche von dem Compiler nicht beachtet werden. Ebenfalls können ganze Zeilen kommentiert werden durch setzen von `//`. Zeilen, die mit einer Raute (`#`) beginnen, sind Anweisungen an den Präprozessor. Dadurch können Bibliotheken hinzugeladen oder bestimmte Definitionen vorgenommen werden (uvm.). Der Präprozessor bearbeitet den Quellcode, bevor dieser kompiliert wird. Es folgen ein paar (selbst) erdachte Beispiele:

```
1  /* Lädt die io Bibliothek, damit die
2     darin enthaltenen Funktionen und Definitionen
3     im Programm genutzt werden können */
4  #include <avr/io.h>
5
6  /* hier wird nun für den Präprozessor die
7     Zeichenkette MEINEVAR als 10 definiert. Der
8     Präprozessor ersetzt also im ganzen Programm
9     MEINEVAR durch 10 */
10 #define MEINEVAR 10
```

Des Weiteren möchte ich noch kurz die If-Bedingung und Schleifen ansprechen, da ein Programm für einen AVR-Mikrocontroller meist aus einer endlosen Schleife besteht, damit die eine Aktion die ausgeführt werden soll immer wiederholt wird. Bei normaler Programmierung ist es ja meist zu vermeiden solch eine endlose Schleife zu verwenden, da sich sonst ein Programm „aufhängen“ würde. Bei der AVR-Programmierung ist dies jedoch erwünscht.

<sup>2</sup> siehe beiliegende CD für eine Beispiel Makefile (-> Programmcode)

In diesem Beispiel stelle ich die Schleifen und die If-Bedingung dar:

```
1  /* Aufbau einer if-Bedingung */
2  if( Bedingung ){
3      //Anweisung
4  }
5  /* Aufbau einer while Schleife */
6  while( Bedingung ){
7      //Anweisung
8  }
```

Ich habe jetzt nur die while-Schleife angesprochen, da ich die for- und die do ... while Schleife in der folgenden Programmierung nicht verwende.

Der auszuführende Code steht in einem C-Programm immer in der Main-Funktion, die üblicherweise einen Integer zurück (zeigt fehlerhaften (Wert: 1) oder erfolgreichen (Wert: 0) Programmverlauf an).

```
1  int main() {
2      //Anweisungen
3      ...
4      return 0;
5  }
```

Generelle C-konforme Möglichkeiten habe ich gerade aufgeführt, als Nächstes komme ich zu den Besonderheiten der AVR-Programmierung. Konkret stellt das hauptsächlich die Bitmanipulationen der einzelnen Register über die bitweisen Operatoren dar.

Die 18 programmierbaren Pins des ATtiny2313 sind in Gruppen zusammengefasst, die man als PORT bezeichnet (z.B. PORTB). Die einzelnen Pins tragen daher die Bezeichnung PB0, PB1, ... PD0, ... PA0 (definiert in <avr/io.h>). Die Anzahl der Pins hängt von dem Controller-Modell ab (ATtiny oder ATmega). Nun kann man bestimmte Pins als Eingang oder als Ausgang setzen, je nach dem was das Programm leisten soll und was für externe Geräte angeschlossen werden. Dies erfolgt über das Richtungsregister, bei dem jedes Bit einem Pin zugeordnet ist und welches einfach wie eine Variable gesetzt werden kann. Man setzt also die Bits in dem Richtungsregister für den bestimmten Pin auf 0 oder 1, wobei 0 den Zustand Eingang darstellt und 1 den Zustand Ausgang. Beispiel:

```
1  /* Setzt das Richtungsregister von PORTA für
2  alle Bits („Beinchen“) auf Ausgang (1) */
3  DDRA = 0xff;
```

Etwas einfacher können die Register auch über Standard-C-konforme Bit-Operationen gesetzt werden, wobei auch deutlich wird welche Pins nun genau gesetzt werden.

```
1  /* Setzt PA0 und PA1 auf Ausgang */  
2  DDRA = ((1<<PA0) | (1<<PA1));
```

Was bei diesem Befehl der Mikrocontroller genau macht, möchte ich kurz darstellen. Die beiden kleiner als Zeichen (<<, Bit-Verschiebungsoperator) bewirken eine Verschiebung der 1 n-mal nach links. Dies bedeutet in Binärschreibweise für (1<<PA0) 0b00000001, da PA0 den ersten Bit des PORTA darstellt und somit den Wert 0 (definiert in der <avr/io.h> Header Datei) hat (keine Verschiebung nach links). Für (1<<PA1) ergibt dies nach demselben System dann 0b00000010. Diese beiden binären Werte werden nun bitweise Oder-verknüpft durch dieses Zeichen '|'. Daraus ergibt sich dann 0b00000011. Dieser binäre Wert wird dem DDRA-Richtungsregister zugeordnet. Dadurch gehen aber vorher gesetzte Werte verloren. Dies kann durch Oder-Verknüpfung bei der Zuordnung verhindert werden, falls dies erwünscht ist. Das sieht dann so aus:

```
1  DDRA |= ((1<<PA0) | (1<<PA1));
```

Nehmen wir mal an, DDRA hat am Anfang den Wert 0b00001100. Im letzten Beispiel ist herausgekommen, dass DDRA der Wert 0b00000011 zugeordnet wird beim Setzen von PA0 und PA1. Werden nun diese Werte Oder-verknüpft, ergibt sich 0b00001111 (neue Werte gesetzt und Vorherige erhalten geblieben).

Bis jetzt wurden nur Ausgänge im Richtungsregister gesetzt. Das Setzen von Eingängen verhält sich ähnlich. Dafür muss im Richtungsregister für das entsprechende Bit einfach eine 0 anstatt eine 1 gesetzt werden. Dies geschieht über Und-Verknüpfungen. Hier nun ein Beispiel:

```
1  /* Setzt PA0 und PA1 auf Eingang (0) */  
2  DDRA &= ~( (1<<PA0) | (1<<PA1) );
```

In der äußeren runden Klammer kommt wieder 0b00000011 als binärer Wert heraus (siehe vorheriges Beispiel). Vor der runden Klammer befindet sich jetzt jedoch noch diese Zeichen '~', welches eine Invertierung des binären Wertes zur Folge hat. Aus 0b00000011 wird also 0b11111100. Nehmen wir nun an, DDRA hat zu Anfang den Wert 0b00001111 (siehe letztes Beispiel). Werden diese beiden Werte nun Und-verknüpft, so erhält man 0b00001100. Dies zeigt also, dass die ersten beiden Bits auf 0 gesetzt wurden, was bedeutet, dass diese beiden nun als Eingänge agieren und genau dies war beabsichtigt. Mithilfe der Und-Verknüpfung erhält man also das gewünschte Ergebnis.



Kommen wir nun zur Aktivierung von Ausgängen. Um Ausgänge nun zu aktivieren, sie also mit der Eingangsspannung zu versorgen (VCC, in meinem Fall 5V), ist das Setzen der PORTx Register notwendig. Andernfalls besitzen die Ausgangspins eine Spannung von 0V (GND). Das Setzen dieser Bits an dem PORTx Register verhält sich genauso wie das Setzen der Richtungsregister. Hier ein Beispiel:

```
1  /* Aktiviert PA0 und PA1 */
2  PORTA = ((1<<PA0) | (1<<PA1));
```

Nun sind die Bits für PA0 und PA1 auf 1 gesetzt, wodurch jetzt die Eingangsspannung anliegt. Das Deaktivieren der Ausgänge verhält sich wie das Setzen der Eingänge eines Richtungsregisters:

```
1  PORTA &= ~(1<<PA0) | (1<<PA1);
```

Jetzt sind die beiden Bits wieder auf 0 gesetzt, wodurch nun keine Spannung anliegt.

Sind bestimmte Pins als Eingänge definiert und man möchte dort an diese Pins einen Schalter anschließen, so muss man etwas beachten. Damit der AVR-Mikrocontroller an den benannten Pins kein undefiniertes Signal bekommt, wenn der Schalter nicht geschlossen ist, müssen sog. Pull-Up Widerstände verwendet werden, die die meisten AVR-Mikrocontroller schon integriert haben. Ist der Schalter also offen bewirkt der Pull-Up Widerstand, dass der AVR-Mikrocontroller ein Signal von 1 erhält, da der Widerstand an VCC (Spannungsquelle) angeschlossen ist. Beim Schließen des Schalters erhält der AVR-Mikrocontroller also ein Eingangssignal von 0 (siehe Kapitel 4.1, Abb. 12 & 13). Das Aktivieren der Pull-Up Widerstände wird auch über die PORTx Register vollzogen, wenn der zu setzen Bit als Eingang definiert ist. Beispiel:

```
1  /* Aktiviert Pull-Up an PA0 & PA1 wenn diese
2  Eingänge sind */
3  PORTA = ((1<<PA0) | (1<<PA1));
4  /* Deaktivieren der Pull-Up Widerstände */
5  PORTA &= ~(1<<PA0) | (1<<PA1);
```

In einem Programm möchte man ja auch des Öffern auslesen, welchen Wert nun ein Eingang hat. Dies kann über die PINx Eingangsregister geschehen. Will man z.B. prüfen, ob ein Schalter geschlossen ist, um dann eine bestimmte Aktion auszuführen, so lässt sich das

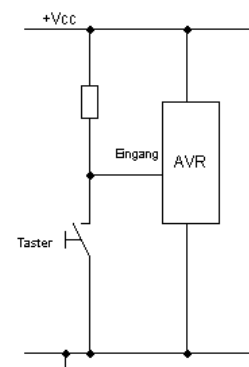


Abbildung 5: Pull-Up  
[AVR GCC TUT]



folgendermaßen realisieren:

```
1  if( !(PINA & (1<<PINA0)) ){  
2      // Anweisungen  
3  }
```

Ist ein Schalter geschlossen, so besitzt das entsprechende Bit den Wert 0, was durch die Pull-Up Widerstände zustande kommt. Die Und-Verknüpfung in der Klammer nimmt also den Wert 0 bei geschlossenem Schalter ein. Damit die if-Bedingung bei geschlossenem Schalter erfüllt wird, ist eine logische Invertierung nötig, die durch das Ausrufungszeichen vollzogen wird. Jetzt ist die Bedingung bei geschlossenem Schalter erfüllt und die Anweisungen der Bedingung werden ausgeführt.

Mithilfe dieser Erläuterungen sollte es jetzt nicht mehr so schwer sein der nun folgenden Programmierung zu folgen.

## 2 DIE SCHALTUNGEN

### 2.1 Die umschaltbare Rechenschaltung

Als Beispiel zur Realisierung der LOCAD-Schaltungen durch Programmierung eines AVR-Mikrocontrollers habe ich die umschaltbare Rechenschaltung gewählt, die im Skript für „Technische Informatik mit LOCAD 2002“ auf der Seite 41 zu finden ist. Diese Schaltung ist meiner Meinung nach am Besten geeignet, um daran die Möglichkeiten des AVR-Mikrocontrollers darzustellen. Im folgenden Kapitel werde ich den Aufbau der Schaltung in LOCAD sowie meinen Aufbau darstellen und dann auf die Programmierung eingehen.

#### 2.1.1 Aufbau

Zunächst füge ich die Schaltung aus dem LOCAD-Skript ein:

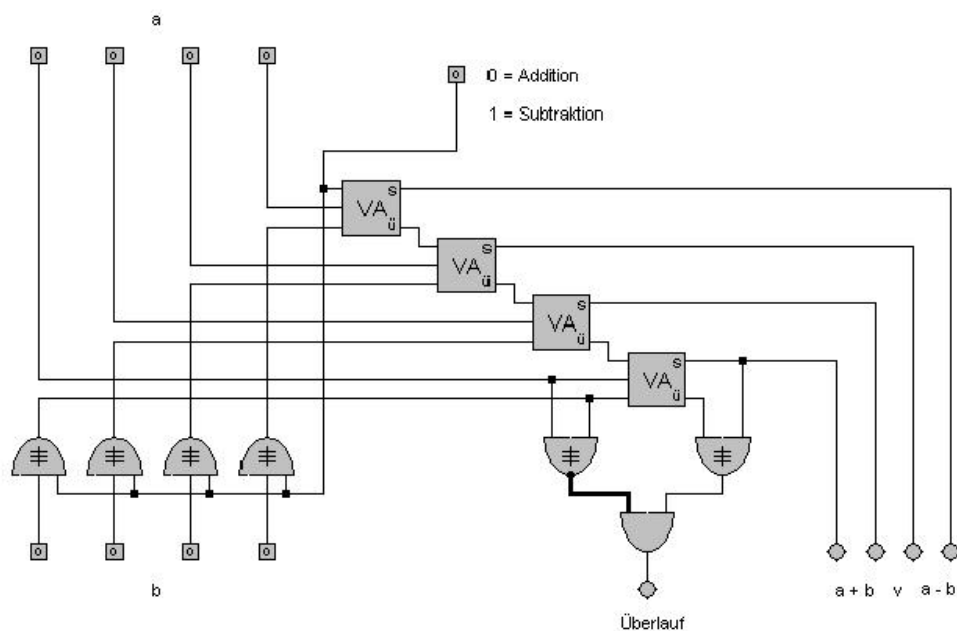


Abbildung 6: Umschaltbare Rechenschaltung, [LOCAD-Skript]: S. 41

Diese Schaltung ist ja aus dem Seminarfach bekannt. Sie addiert oder subtrahiert 2 Zahlen, die binär über Schalter eingegeben werden. Die parallele Addition/Subtraktion aller Summandenbits wird über die 4 Volladdierer gewährleistet. Falls subtrahiert werden soll (Schalter auf 1) dienen die Exklusiv-Oder Glieder am Summanden b als Inverter, die somit das Einerkomplement bilden, ebenfalls wird auch noch eine 1 am ersten

Volladdierer addiert, wodurch das Zweierkomplement gebildet wird, was der Gegenzahl entspricht. Die 4 LEDs zeigen am Ende das Ergebnis der Rechnung an, wobei dabei nur der Zahlenbereich von -8 bis +7 dargestellt werden kann. Wird dieser überschritten, wird das durch die Überlaufs-LED angezeigt, was bedeutet, dass die Rechnung nicht zulässig und das angezeigte Ergebnis falsch ist.

Als Nächstes folgt die Schaltskizze meines Aufbaus mit dem Mikrocontroller:

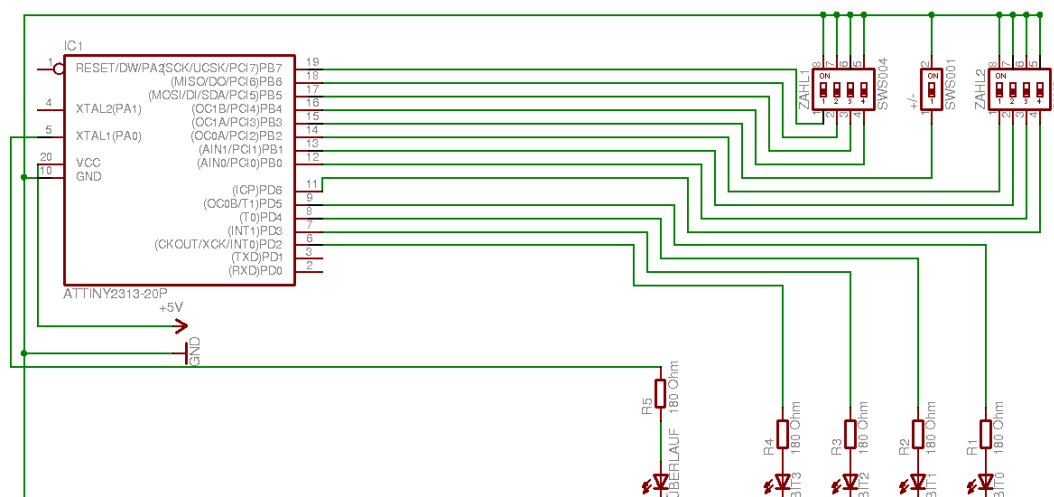


Abbildung 7: Umschaltbare Rechenschaltung (Schaltskizze)

Auf der linken Seite der Schaltskizze ist das Schaltsymbol des ATtiny2313 zu erkennen. Die Anordnung der Pins entspricht dabei nicht der Realität. Auf der rechten Seite im oberen Teil kann man die für die Eingabe genutzten DIP-Schalter erkennen. Die zwei 4-poligen DIP-Schalter dienen dabei für die Eingabe der beiden zu addierenden oder subtrahierenden Zahlen. Ob nun addiert oder subtrahiert wird, wird mit dem einzelnen DIP-Schalter kontrolliert. In der unteren Hälfte auf der rechten Seite sind die für die Ausgabe vorgesehenen LEDs aufgeführt. Die LED, die etwas abseits ihren Platz hat, visualisiert den Überlauf. Die anderen LEDs stellen die Ausgabe des Ergebnisses in binärer Form dar. Im Gegensatz zu der LOCAD-Schaltung wird hier die Rechnung nun von dem AVR-Mikrocontroller übernommen und nicht von den Volladdierern. Wie dies nun möglich ist, stelle ich im nächsten Kapitel dar. Weitere original Bilder des Aufbaus sowie der Schaltskizze in einem größeren Format und der Schaltung im Einsatz sind unter Kapitel 4: 'Materialien' zu finden.

## 2.1.2 Der Programmcode

```
1 // PINB4 - PINB7 -> summand 1
2 // PIND6 + PINB0 - PINB2 -> summand 2
3 // PINB3 -> +/- choice
4 // PIND5 - PIND2 -> ergebnis output
5 // PINA0 -> Überlauf
6
7 /* CPU geschwindigkeit definieren und nötige
8 Bibliotheken importieren */
9 #define F_CPU 1000000UL
10 #include <avr/io.h>
11 #include <stdint.h>
12
13 int main(){
14 /* Definieren der Ausgänge */
15 DDRD = ((1<<PD5) | (1<<PD4) | (1<<PD3) |
16 (1<<PD2)); /* nötige PIND ausgang */
17 DDRA = 0xff; /* alle PINA ausgang */
18 /* Definieren der Eingänge */
19 DDRB = 0x00; /* alle PINB eingang */
20 PORTB = 0xff; /* aktiviere alle Pull-
21 Up widerstände an PINB */
22 DDRD &= ~(1<<PD6); /* PD6 als Eingang */
23 PORTD = (1<<PD6); /* aktiviere Pullup an
24 PD6 */
25
26 /* nötige variablen definieren */
27 int8_t sum1 = 0;
28 int8_t sum2 = 0;
29 int8_t choice = 1;
30
31 /* Main-Loop */
32 while(1){
33 /* setzen der ausgänge auf 0 */
34 PORTD &= ~(((1 << PD5)) | ((1 << PD4)) |
35 ((1 << PD3)) | ((1 << PD2)));
36
37 /* ersten summanden/zahl auslesen (dezimal) */
38 sum1 = 1*(!(PINB & (1<<PINB4)))+
39 2*(!(PINB & (1<<PINB5)))+
40 4*(!(PINB & (1<<PINB6)))+
41 8*(!(PINB & (1<<PINB7)));
42 /* zweiten summanden/zahl auslesen (dezimal) */
43 sum2 = 1*(!(PIND & (1<<PIND6)))+
44 2*(!(PINB & (1<<PINB0)))+
45 4*(!(PINB & (1<<PINB1)))+
46 8*(!(PINB & (1<<PINB2)));
47
48 /* prüfe: Addition oder Subtraktion */
49 if( !(PINB & (1<<PINB3)) ){
50 sum1 = sum1-sum2; /* in sum1 wird
51 das ergebnis gespeichert */
```

```
52 } else {
53     sum1 = sum1+sum2;
54 }
55
56 /* Auf Ueberlauf prüfen und ggf. Überlauf LED
57 aktivieren -> PA0 */
58 if((sum1>7) | (sum1<-8)){
59     PORTA = (1<<PA0);
60 }
61 else {
62     PORTA &= ~(1<<PA0);
63 }
64
65 /* Prüfen ob ergebnis negativ ist */
66 if(sum1 < 0){
67     sum1++;           /* addieren von 1
68                     (Zweierkomplement) */
69     choice = 0;       /* Bewirkt invertierung
70                     (einerkomplement) */
71 } else {
72     choice = 1;       /* keine invertierung */
73 }
74
75 /* dual in binärzahl umwandeln und ausgeben */
76 if(sum1%2 == choice){
77     PORTD |= (1 << PD5);
78 }
79
80 sum1 = sum1/2;
81
82 if(sum1%2 == choice){
83     PORTD |= (1 << PD4);
84 }
85
86 sum1 = sum1/2;
87
88 if(sum1%2 == choice){
89     PORTD |= (1 << PD3);
90 }
91
92 sum1 = sum1/2;
93
94 if(sum1%2 == choice){
95     PORTD |= (1 << PD2);
96 }
97 } /* Ende Main-Loop */
98 /* dieser Punkt wird nie erreicht, da es eine
99 endlos-Schleife ist */
100 return(0);
101 }
```

### 2.1.3 Erläuterung des Programmcodes

Das war nun der Programmcode zur Realisierung der umschaltbaren Rechenschaltung. Diesen möchte ich nun etwas näher erläutern.

In den Zeilen 1 bis 5 habe ich in Form von Kommentaren kurz notiert, an welche Pins die einzelnen Schalter zur Eingabe und LEDs zur Ausgabe angeschlossen sind, um immer schnell sehen zu können, an welchen Pin welches Bauteil angeschlossen ist.

Das Hinzufügen der Bibliotheken mit den vordefinierten Funktionen und Werten findet in Zeile 9 bis 11 statt (io.h: Definitionen der Register etc.; stdint.h: Definitionen der Integer-Datentypen wie int8\_t). Dort wird auch die Taktfrequenz definiert um mögliche Fehler während der Laufzeit zu verhindern.

In Zeile 13 tritt die Main-Funktion auf, die Bestandteil jedes C-Programmes ist.

Nun kommt es in Zeile 15 bis 23 zum Setzen der Bits der einzelnen Register, um sozusagen den Mikrocontroller auf die Zusammenarbeit mit den externen Geräten einzustellen (für Näheres siehe Kapitel 1.2.2.3). Für die Realisierung der umschaltbaren Rechenschaltung werden zunächst 2 Variablen benötigt zum Speichern der eingegebenen Zahlen, die addiert oder subtrahiert werden sollen. Hinzu kommt noch eine Variable, die in der weiteren Programmierung für die Invertierung der Ausgabe zuständig ist, falls diese einen negativen Wert annimmt, wodurch das Einerkomplement gebildet wird, was nötig ist, um die binäre Gegenzahl zu ermitteln. Diese werden in Zeile 27 bis 29 deklariert und definiert. In Zeile 32 beginnt nun der Hauptteil des Programmes die sog. Main-Loop. Dabei handelt es sich um eine endlose while-Schleife, die immer wieder die Addition oder Subtraktion der Zahlen durchführt, um zu gewährleisten, dass bei Änderung der Eingabezahlen sofort das Ergebnis angepasst wird. In Zeile 34 werden, bedingt durch die endlose Wiederholung, als Erstes die verwendeten Ausgänge auf „Low“ gesetzt, was bedeutet, dass die LEDs an den Ausgängen nicht leuchten, damit beim erneuten Ausführen der Schleife das Ergebnis nicht durch vorherige Werte der Ausgänge beeinflusst wird. Da die Schleife so schnell hintereinander abläuft, ist dieses An- und Ausschalten der LEDs nicht zu beobachten, welches jedoch für eine reibungslose Programmierung notwendig ist.

Das Auslesen der eingegebenen Zahlen erfolgt in Zeile 38 bis 46. Die

Zahlen werden von dem Programm als Dezimalzahl ausgelesen, um das Addieren und Subtrahieren zu vereinfachen. Beträgt zum Beispiel der Status an dem Schalter PB4 0, bedeutet dies, dass er geschlossen ist (siehe „Pull-Up“, Kapitel 1.2.2.3). Falls er geschlossen ist, soll dies als eine 1 in binärer Schreibweise interpretiert werden, somit wird der Statuswert invertiert und mit dem entsprechenden dezimalen Wert multipliziert, der für den jeweiligen Summandenbit steht ( $2^0$ ,  $2^1$ ,  $2^2$ , ...). Diese Umrechnung erfolgt nach demselben Prinzip, wie es im Kapitel 2.1.1.1 des Skripts für „Technische Informatik mit LOCAD 2002“ vorgestellt wird. Somit werden beide eingegebenen Zahlen ausgelesen und in den vordefinierten Variablen gespeichert.

Die Addition und Subtraktion findet nun in Zeile 49 bis 54 statt. Der Schalter, der an den PINB3 angeschlossen ist, kontrolliert die Rechenaktion. Falls dieser geschlossen ist, soll subtrahiert werden und falls dieser offen ist, soll addiert werden. Dies wird in der if-Bedingung ausgedrückt. Das Ergebnis wird jeweils in der sum1 Variable gespeichert um die Nutzung eine weiteren Variable zu vermeiden, was sonst weiteren Speicherbedarf zur Folge hätte.

Da mit den 4 LEDs zur binären Ausgabe nur ein bestimmter Zahlenbereich ausgegeben werden kann, (von -8 bis +7) muss signalisiert werden, wenn dieser überschritten wird und somit das Ergebnis nicht korrekt sein kann. Dies erfolgt über die Überlaufs-LED. In der LOCAD-Schaltung tritt dies ein, wenn „die höchst wertigen Summandenbits gleich sind“<sup>3</sup> und wenn „der letzte Übertrag ungleich dem höchstwertigen Ergebnisbit ist“<sup>3</sup>. Da die Rechnung in meiner Programmierung über das dezimale System erfolgt, kann einfach geprüft werden, ob das Ergebnis möglicherweise den Zahlenbereich überschreitet. Dies wird in Zeile 58 bis 63 überprüft. Dort wird bei Überschreitung des Zahlenbereichs die Überlaufs-LED aktiviert.

An dieser Stelle des Programms ist der Rechengvorgang abgeschlossen und es geht in den Ausgabevorgang über. Dabei muss die dezimale Zahl, die in der sum1 Variable gespeichert ist, wieder in eine Dualzahl umgewandelt werden. Dies erfolgt gemäß der im Skript für „Technische Informatik mit LOCAD 2002“ kennen gelernte Methode<sup>4</sup>. Die Dezimalzahl wird durch 2 dividiert, wobei der Divisionsrest (der immer 0 oder 1 ist) die Dualzahl darstellt. Über den Module Operator (%) kann in C der

---

3 [LOCAD-Skript]: S. 38

4 siehe Kapitel 2.1.1.2 im [LOCAD-Skript]

Divisionsrest einer Division ausgegeben werden. Falls dieser 1 ist, wird dies durch eine leuchtende LED visualisiert. Dies ist in den if-Bedingungen ab Zeile 76 ausgedrückt. Nachdem der Divisionsrest ermittelt wurde, muss die Zahl noch mit 2 dividiert werden, damit der nächste Divisionsrest ermittelt werden kann. Dies folgt jeweils auf die if-Bedingungen. Damit wären nun alle positiven Werte für das Ergebnis abgedeckt. Da das Ergebnis auch negativ sein kann, muss also noch geprüft werden, ob das Ergebnis negativ ist, um daraufhin das Zweierkomplement zu bilden.

Das Zweierkomplement wird gebildet durch Addieren von 1 zu dem Ergebnis und durch Invertierung der Ergebnisbits. Die Überprüfung ob das Ergebnis negativ ist findet in Zeile 66 statt. Falls es negativ ist, wird dort zu dem Ergebnis 1 addiert. Gleichzeitig wird die 'choice' Variable gleich 0 gesetzt. Dies hat die Invertierung zur Folge, da nun die darauf folgenden if-Bedingungen nicht mehr erfüllt sind, wenn der Divisionsrest 1 ist, sondern nur wenn der Divisionsrest 0 ist. Das binäre Ergebnis wird also invertiert (Einerkomplement). Da vorher schon eine 1 addiert wurde erhalten wird das Zweierkomplement, was der Gegenzahl entspricht und somit werden negative Ergebnisse, auf einfache Art und Weise, von dem Programm korrekt ausgegeben.

An dieser Stelle ist auch die Erläuterung des Programmcodes abgeschlossen. Wenn die Ausgabe erfolgt ist, folgt im Code nur noch das Ende der Main-Loop. Doch der darauf folgende Code wird nie erreicht, da es sich ja um eine endlose Schleife handelt, deren Bedingung immer erfüllt ist.

Dieses Programm bietet also dieselben Funktionen wie die LOCAD-Schaltung.



## 2.2 Erweiterte umschaltbare Rechenschaltung

Da im Seminarfach auch die 7-Segment-Anzeigen zum Zuge kamen, habe ich die umschaltbare Rechenschaltung noch so erweitert, dass die Ausgabe über eine 7-Segment-Anzeige erfolgt. Damit möchte ich darstellen, was mit dem AVR-Mikrocontroller noch so möglich ist. Ich habe auch eine Schaltung mit LOCAD erstellt, die eine Ausgabe über eine 7-Segment- oder Ziffernanzeige ermöglicht. Dies erwies sich als nicht ganz unkompliziert, aber darauf werde ich im nächsten Unterkapitel eingehen.

### 2.2.1 Aufbau

Ich füge zunächst die erstellte LOCAD-Schaltung ein:

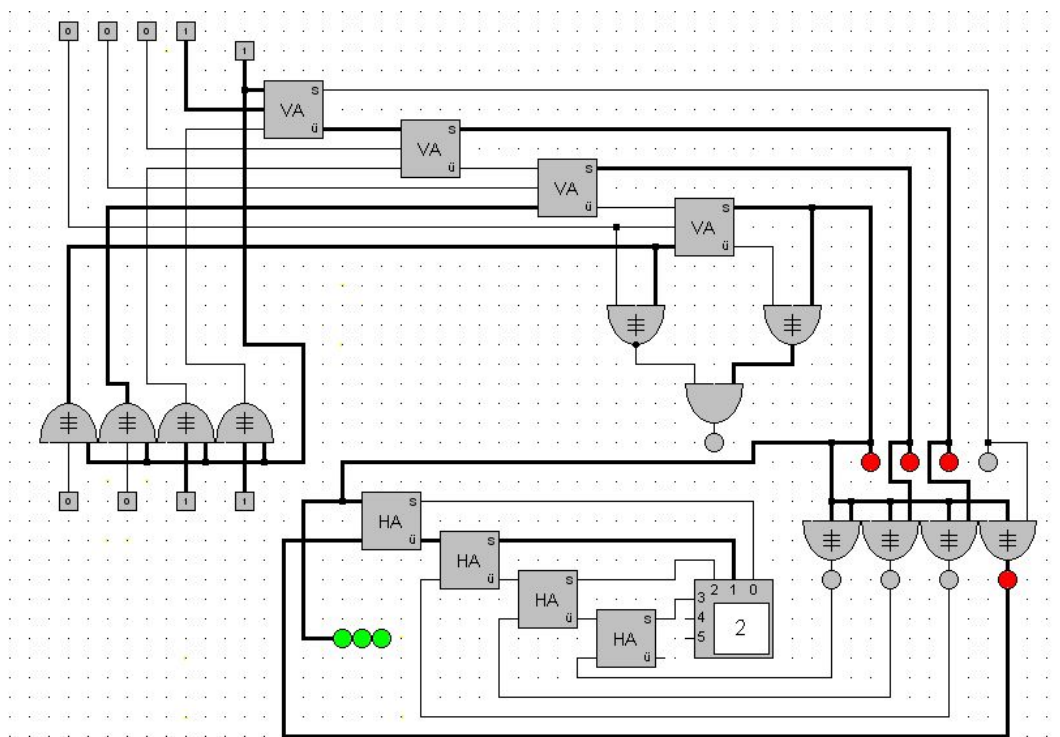


Abbildung 8: Umschaltbare Rechenschaltung mit Ziffernanzeige (LOCAD-Schaltung)

In der oberen Hälfte der Schaltung ist die umschaltbare Rechenschaltung zu erkennen. Die untere Hälfte ist nun noch weiter ausgefüllt mit Halbaddierern und der Ziffernanzeige, die ich einer 7-Segment-Anzeige, bestehend aus LEDs, aus Platzmangel vorgezogen habe. Bei der Ausgabe über eine Ziffernanzeige war das Problem die Ausgabe der negativen Zahlen. Diese können ja nicht richtig von der Ziffernanzeige ermittelt werden, da ja erst die Gegenzahl (Zweierkomplement) gebildet werden muss. Genau dies übernimmt der neu hinzugekommene Teil aus

Halbaddierern und Exklusiv-Oder Gliedern. Eine negative Zahl kann man an dem höchsten Bit erkennen. Hat dieser den Wert 1, dann handelt es sich um eine negative Zahl. Falls die Zahl negativ ist, soll also das Zweierkomplement gebildet werden, damit die Zahl richtig angezeigt werden kann. Falls die Zahl positiv ist, soll jedoch nichts passieren. Durch Verbindung der Exklusiv-Oder Glieder an den höchsten Ergebnisbit agieren diese als Inverter, falls dieser Bit den Wert 1 hat. Also falls die Zahl negativ ist, wird das Ergebnis invertiert, was das Einerkomplement darstellt. Dies wird durch den Gliedern folgende LEDs visualisiert. Damit ist aber noch nicht die Gegenzahl erreicht. Es muss noch eine 1 addiert werden. Dies übernehmen die Halbaddierer parallel für alle Bits. Der oberste Halbaddierer ist mit dem noch nicht invertierten höchsten Ergebnisbit verbunden. Also falls dieser 1 ist (negative Zahl), wird 1 addiert, falls nicht wird nichts (0) addiert. Somit wird, falls die Zahl positiv ist, die Zahl nicht verändert. Genau wie bei den Exklusiv-Oder Glieder, die auch nur zum Inverter werden falls der höchste Ergebnisbit 1 ist. An die Ausgänge der Halbaddierer ist nun die Ziffernanzeige angeschlossen. Falls das Ergebnis nun negativ ist, wird erst die Gegenzahl gebildet, damit die Ziffernanzeige das Ergebnis gleich korrekt anzeigt, falls es positiv ist, wird das Ergebnis unberührt weitergegeben. Wenn es sich um eine negative Zahl handelt, wird dies auch noch über die 3 grünen LEDs visualisiert, die ein Minuszeichen darstellen sollen und leuchten, falls es sich um eine negative Zahl handelt. Somit stellt diese erweiterte umschaltbare Rechenschaltung das Additions- oder Subtraktionsergebnis zweier Zahlen über eine Ziffernanzeige dar.

Der Aufbau mit dem AVR-Mikrocontroller sieht nun folgendermaßen aus:

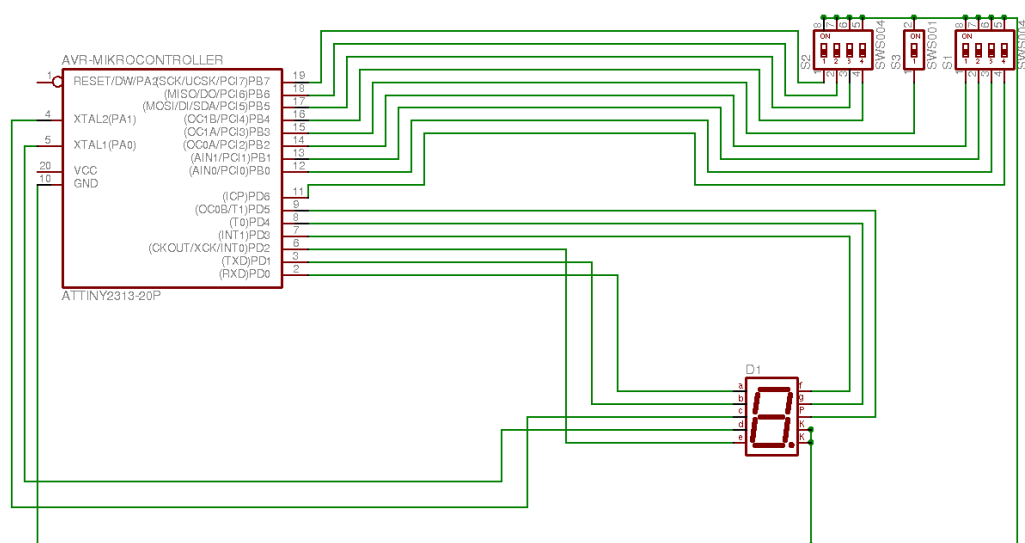


Abbildung 9: erweiterte Umschaltbare Rechenschaltung (Schaltskizze)

Diese Schaltung ähnelt also auch der Vorherigen (Abb.7), nur das jetzt keine LEDs zur Ausgabe benutzt werden sondern eine 7-Segment-Anzeige. Durch den Anschluss einer 7-Segment-Anzeige und des Eingabepanels sind auch alle I/O Pins des ATtiny2313 ausgeschöpft. Ich möchte noch einmal kurz auf den Aufbau der 7-Segment-Anzeige eingehen, da dieser in der Programmierung auch eine Rolle spielt.

Wie es der Name schon sagt, besteht eine 7-Segment-Anzeige aus 7 Segmenten. Diese werden mit Buchstaben bezeichnet (a-g, im Uhrzeigersinn). Ebenfalls besitzt eine 7-Segment-Anzeige auch noch einen Dezimalpunkt.

Folgendermaßen ist die Anzeige an den AVR-Mikrocontroller angeschlossen:

Pin am AVR	Segment
PD0	a (7)
PD1	b (6)
PA1	c (4)
PA0	d (2)
PD2	e (1)
PD3	f (9)
PD4	g (10)
PD5	DP (5)
GND	(3, 8)

Tabelle 4: Anschluss am Mikrocontroller

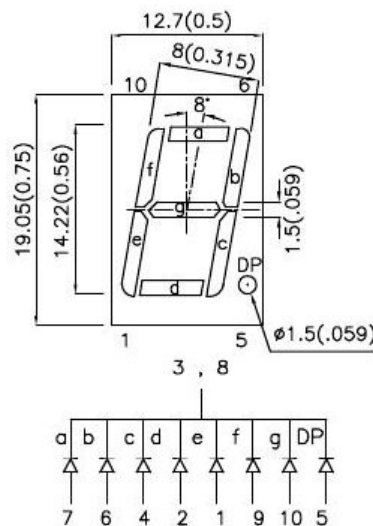


Abbildung 10: 7-Segment-Anzeige [Datenblatt-7]

## 2.2.2 Der Programmcode

```
1  #define F_CPU 1000000UL
2  #include <avr/io.h>
3  #include <stdint.h>
4  /* einfachere Nutzung der Pins durch den
5     praeprozessor*/
6  #define SA PD0
7  #define SB PD1
8  #define SC PA1
9  #define SD PA0
10 #define SE PD2
11 #define SF PD3
12 #define SG PD4
13 #define DP PD5
14
15 int main(){
16     DDRD = 0xff;          /* alle PIND ausgang */
17     DDRA = 0xff;          /* alle PINA ausgang */
18     DDRB = 0x00;          /* alle PINB eingang */
19     PORTB = 0xff;          /* aktiviere alle Pull-Up an
20                               PINB */
21     DDRD &= ~(1<<PD6);    /* PD6 als Eingang */
22     PORTD = (1<<PD6);     /* aktiviere Pullup an
23                               PD6 */
24
25     /* Speicherung der Eingabezahlen*/
26     int8_t sum1 = 0;
27     int8_t sum2 = 0;
28     while(1){
29
30         /* ersten summanden auslesen */
31         sum1 = 1*(!(PINB & (1<<PINB4)))+
32               2*(!(PINB & (1<<PINB5)))+
33               4*(!(PINB & (1<<PINB6)))+
34               8*(!(PINB & (1<<PINB7)));
35         /* zweiten summanden auslesen */
36         sum2 = 1*(!(PIND & (1<<PIND6)))+
37               2*(!(PINB & (1<<PINB0)))+
38               4*(!(PINB & (1<<PINB1)))+
39               8*(!(PINB & (1<<PINB2)));
40
41         /* addieren oder subtrahieren */
42         if( !(PINB & (1<<PINB3)) ){
43             sum1 = sum1-sum2;    /* in sum1 wird das
44                                   ergebnis gespeichert */
45         } else {
46             sum1 = sum1+sum2;
47         }
48         /* Prüfen ob ergebnis negativ ist */
49         if(sum1 < 0){
50             PORTD |= (1 << DP);  /* Dezimalpunkt
51                                   signalisiert negativ */
52         }
```

```
52     sum1= sum1-(2*sum1); /* sum1 in pos.  
53                          gegenzahl umwandeln */  
54 } else {  
55     PORTD &= ~(1 << DP);  
56 }  
57  
58 /* Ausgabe über die 7-Segm.-Anzeige */  
59 switch( sum1 ) {  
60  
61     case 0:  
62         PORTD |= ( (1 << SA) | (1 << SB) |  
63                   (1 << SE) | (1 << SF) );  
64         PORTD &= ~( 1 << SG );  
65         PORTA |= ( (1 << SC) | (1 << SD) );  
66         break;  
67     case 1:  
68         PORTD |= (1 << SB);  
69         PORTD &= ~( (1 << SA) | (1 << SE) |  
70                   (1 << SF) | (1 << SG) );  
71         PORTA |= (1 << SC);  
72         PORTA &= ~(1 << SD);  
73         break;  
74     case 2:  
75         PORTD |= ( (1 << SA) | (1 << SB) |  
76                   (1 << SE) | (1 << SG) );  
77         PORTD &= ~( 1 << SF );  
78         PORTA |= (1 << SD);  
79         PORTA &= ~(1 << SC);  
80         break;  
81     case 3:  
82         PORTD |= ( (1 << SA) | (1 << SB) |  
83                   (1 << SG) );  
84         PORTD &= ~( (1 << SE) | (1 << SF) );  
85         PORTA |= ( (1 << SC) | (1 << SD) );  
86         break;  
87     case 4:  
88         PORTD |= ( (1 << SB) | (1 << SG) |  
89                   (1 << SF) );  
90         PORTD &= ~( (1 << SA) | (1 << SE) );  
91         PORTA |= (1 << SC);  
92         PORTA &= ~(1 << SD);  
93         break;  
94     case 5:  
95         PORTD |= ( (1 << SA) | (1 << SF) |  
96                   (1 << SG) );  
97         PORTD &= ~( (1 << SE) | (1 << SB) );  
98         PORTA |= ( (1 << SC) | (1 << SD) );  
99         break;  
100    case 6:  
101        PORTD |= ( (1 << SA) | (1 << SF) |  
102                  (1 << SG) | (1 << SE) );  
103        PORTD &= ~( 1 << SB );  
104        PORTA |= ( (1 << SC) | (1 << SD) );
```

```
105         break;
106     case 7:
107         PORTD |= ( (1 << SA) | (1 << SB) );
108         PORTD &= ~( (1 << SE) | (1 << SF) |
109                     (1 << SG) );
110         PORTA |= (1 << SC);
111         PORTA &= ~(1 << SD);
112         break;
113     case 8:
114         PORTD |= ( (1 << SA) | (1 << SB) |
115                     (1 << SE) | (1 << SF) | (1 << SG) );
116         PORTA |= ( (1 << SC) | (1 << SD) );
117         break;
118     case 9:
119         PORTD |= ( (1 << SA) | (1 << SB) |
120                     (1 << SF) | (1 << SG) );
121         PORTD &= ~( 1 << SE );
122         PORTA |= ( (1 << SC) | (1 << SD) );
123         break;
124     default:
125         PORTD |= (1 << SG);
126         PORTD &= ~( (1 << SA) | (1 << SB) |
127                     (1 << SE) | (1 << SF) );
128         PORTA &= ~( (1 << SC) | (1 << SD) );
129         break;
130 }
131 }
132 return(0);
133 }
```

### 2.2.3 Erläuterung des Programmcodes

Dies war nun der Programmcodes zu der erweiterten umschaltbaren Rechenschaltung. Vom Kern her unterscheidet dieser sich nicht von der umschaltbaren Rechenschaltung, es ist nur die Ausgabe verändert worden, daher werde ich nur die Änderungen erläutern.

Hinzugekommen sind in Zeile 6 bis Zeile 13 die Definitionen, um während der Programmierung den Überblick über die Segmente der 7-Segment-Anzeige zu behalten. Dadurch kann ich im Programm einfach SA für Segment A etc. verwenden und muss nicht immer nachschauen, welcher Pin jetzt an welchem Segment angeschlossen ist (siehe auch Tabelle 4 & Abb. 10). Für dieses Programm werden nur noch die beiden Zahlvariablen gebraucht, die 'choice' Variable entfällt, da sie für die Einerkomplementbildung genutzt wurde, was bei der 7-Segment-Anzeige nicht nötig ist. Das Auslesen der eingegebenen Zahlen als Dezimalzahlen erfolgt nach demselben Prinzip (Z. 31-39). Daraufhin werden die Zahlen addiert oder subtrahiert (Z. 42-47). Mit einer 7-Segment-Anzeige können nur positive

Zahlen dargestellt werden. In meinem Fall habe ich nun den Dezimalpunkt, den jede 7-Segment-Anzeige besitzt, dazu verwendet anzuzeigen, ob eine Zahl negativ ist oder nicht. Ich musste den Dezimalpunkt verwenden, da keine weitere LED mehr angeschlossen werden kann (alle I/O Pins sind vergeben). Falls das Ergebnis also negativ ist, leuchtet der Dezimalpunkt und das Ergebnis wird zur besseren Weiterverarbeitung in die positive Gegenzahl umgewandelt (Z. 49-56).

Ab Zeile 59 findet nun die Ausgabe über die 7-Segment-Anzeige statt. Dort habe ich mich für die switch und case Anweisung entschieden, andernfalls hätte ich eine große Anzahl von aufeinanderfolgenden if-Bedingung verwenden müssen. Für solche Sachen kann man immer sehr gut die switch Anweisung verwenden. Die switch Anweisung überprüft eine Variable auf ihren Inhalt (über die case Anweisungen) und führt daraufhin vordefinierte Anweisungen aus. In meinem Fall wird falls sum1 den Wert 0 hat werden die Anweisungen in Zeile 62-66 ausgeführt usw. Die switch Anweisung stellt also eine Mehrfachauswahl dar. Das 'break;' am Ende jedes Code-Blockes, der auf das 'case' folgt, bewirkt, dass die folgenden case Anweisungen gar nicht mehr beachtet werden (Sprung ans Ende der switch Anweisung). In einer switch Anweisung kann man auch über die 'default' Anweisung festlegen, was passieren soll, falls die Variable keine der mit 'case' abgefragten Werte besitzt. Somit kann ich mithilfe der 'default' Anweisung den Überlauf signalisieren. Denn falls sum1 keinen Wert von 0-9 besitzt, wird in meinem Programm die 'default' Anweisung ausgeführt, was eine Überschreitung des möglichen Ausgabewerts darstellt. Es werden also nach jedem 'case 0:', 'case 1:', ... die Segmente aktiviert und deaktiviert um die jeweilige Zahl auf der 7-Segment-Anzeige darzustellen. Als 'default' habe ich festgelegt, dass nur Segment g (siehe Abb. 10) leuchtet, somit leuchtet bei Überlauf auch nur das Segment in der Mitte. Da eine 7-Segment-Anzeige Zahlen von 0 bis 9 darstellen kann und das Programm die Zahlen auf dezimaler Basis addiert, kann mit diesem Aufbau ein Ergebnis von -9 bis 9 ausgegeben werden (Negative werden über den Dezimalpunkt dargestellt). Die dazugehörige LOCAD-Schaltung (siehe Abb. 7) kann nur Zahlen von -8 bis +7 korrekt ausgegeben, was durch die Beschränkung der 4 stelligen Dualzahl zustande kommt.

### 3 FAZIT

Als Fazit zu der Fragestellung, ob man mit einem AVR-Mikrocontroller die im Seminarfach kennengelernten Schaltungen realisieren kann, kann man nach der Darstellung in Kapitel 2 mit einem deutlichen 'Ja' beantworten.

Sogar die Erweiterung der umschaltbaren Rechenschaltung, die in LOCAD nicht so leicht zu realisieren ist, lässt sich mit dem AVR-Mikrocontroller einfach nachbauen. Doch dort sind bald die Grenzen des ATtiny2313 erreicht. Aber der ATtiny2313 ist auch einer der „kleineren“ AVR-Mikrocontroller. Mit „größeren“ Mikrocontrollern wie den ATmega Modellen lassen sich noch größere Projekte realisieren. Diese besitzen noch mehr programmierbare Pins, Flashspeicher sowie RAM etc.

ATmega Modelle werden deshalb auch nicht umsonst in so manchen Roboter-Baukästen verwendet. Denn es besteht auch die Möglichkeit des Anschlusses von verschiedensten Sensoren und die Anbindung in andere Stromkreise über Transistoren und Relais etc.

Durch geschickte Multiplexer<sup>5</sup> ähnliche Schaltungen ist es aber auch möglich die wenigen Pins der „kleineren“ AVR-Mikrocontrollern besser auszunutzen. Mit dem sog. „Charlieplexing“, welches noch effektiver ist, ist es sogar möglich mit nur 5 Pins 20 LEDs zu kontrollieren, da dabei alle 3 Schaltzustände eines Pins genutzt werden (Output-High (VCC), Output-Low (GND), Input). Mithilfe von multiplexerartigen Schaltungen können an 16 Pins acht 7-Segment-Anzeigen betrieben werden. Die entsprechende charlieplex Methode kommt mit nur 9 Pins aus (siehe Abb. 33 & 34). Daran wird die Effektivität sehr deutlich.

AVR-Mikrocontroller sind also sehr vielseitig und erfreuen sich dank recht einfacher Handhabung auch an vielen Nutzern im Hobby-Bereich, sie kommen aber auch in professionellen Projekten zum Einsatz. Ein Beispiel dafür wäre ein, auf einem leistungsfähigen AVR-Mikrocontroller basierender, Mini-Web-Server<sup>6</sup>, der dank eines Realtek Netzwerkbausteins Zugriff auf das Internet erhält. Des Weiteren gibt es mehrere AVR-MP3-Player<sup>7</sup> Projekte, wo ein AVR-Mikrocontroller mithilfe eines MP3/WAV-Decoder Bausteins, einer Flashspeicherkarte oder einer Festplatte und einem Display als MP3-Player agiert. Dies sind recht große

<sup>5</sup> siehe [LOCAD-Skript]: S. 54 ff.

<sup>6</sup> Beispiel-Projekt-HP: <http://www.ulrichradig.de/>

<sup>7</sup> Beispiel-Projekt-HP: <http://www.myplace.nu/mp3/>



Projekte, die meist auch auf leistungsfähigen und teureren Mikrocontrollern, wie dem ATmega128, basieren. So etwas ist mit dem in meinem Projekt genutzten Mikrocontroller (ATTiny2313) natürlich nicht möglich.

Mir ist während der Programmierung aufgefallen, dass in der AVR-Programmierung auch die Inhalte des Seminarfaches eine nicht unerhebliche Rolle einnehmen. Ein gutes Beispiel dafür sind die beim Manipulieren der Bits der einzelnen Register verwendeten bitweisen Operatoren, die die Programmiersprache C auch noch gegenüber den arithmetischen, relationalen und logischen Operatoren besitzt. Darunter fallen die Und-Funktion (&), die Oder-Funktion (|) und das Komplement (~). Diese werden im Skript für „Technische Informatik mit LOCAD 2002“ ganz zu Anfang behandelt (Kapitel 1.2.1–1.2.3). Diese Funktionen sind Teil der Programmiersprache C, da es sich bei C auch um eine sog. „Middle-Level“-Computersprache handelt. „C wird deswegen so genannt, weil es die besten Elemente der High-Level-Sprachen (Pascal, Basic) mit den Kontrollmöglichkeiten und der Flexibilität der Assemblersprache kombiniert“<sup>8</sup>. Diese bitweisen Operatoren stammen also eindeutig aus der Assemblersprache, für die es auch einige Compiler zur Programmierung von AVR-Mikrocontrollern gibt (siehe Kapitel 1.2.2.2, Tabelle 3).

Somit habe ich durch das Seminarfach auch Erkenntnisse erlangt, die mir beim Verständnis der AVR-Programmierung sehr hilfreich waren (weiteres Beispiel: der Multiplexer).

Die AVR-Programmierung ist derzeit meist auch Teil des Bachelor Informatikstudiums, daran wird deren Bedeutung für die Technik sicher auch nochmal deutlich.

Letzten Endes muss ich sagen, dass ich mit meinem Thema sehr zufrieden bin und es mir viel Spaß bereitet hat, diese eher wissenschaftliche Arbeit zu verfassen. Ich denke, dass meine gewählte Arbeitsweise zu diesem Thema recht gut gelungen ist, da am Ende das gewollte Ergebnis ohne Einschränkungen erreicht wurde, nämlich die Realisierung der LOCAD-Schaltungen durch Programmierung eines AVR-Mikrocontrollers (ATTiny2313). Zu Anfang war natürlich eine Einarbeitung notwendig, jedoch hielt sich dies in Grenzen, da ich durch private

---

<sup>8</sup> [C++]: S. 28

Programmierung die Sprache C schon kennengelernt hatte und die Bezüge aus der technischen Informatik wurden im Seminarfach behandelt.

Die Facharbeit war in vielerlei Hinsicht bereichernd für mich. Ich habe gelernt eine wissenschaftliche Arbeit zu verfassen, was später im Studium auch eine Rolle spielt. Des Weiteren habe ich noch einige neue Aspekte der C-Programmierung kennengelernt, darunter fallen auch die bitweisen Operatoren, die in anderen Programmen nicht eine so große Rolle spielen, wie in der AVR-Programmierung. Und ich habe noch weitere Dinge im Bezug auf die technische Informatik erfahren.

## 4 MATERIALIEN

In diesem Kapitel möchte ich einerseits die von mir genutzten Materialien auflisten und andererseits noch ein paar Fotos/Abbildungen einfügen, die so in den fließenden Text nicht mit eingegangen sind. Darunter fallen auch die Schaltskizzen als ganzseitiger Ausdruck.

Hier nun eine Liste der genutzten Materialien:

- Aoyue 937 Lötstation mit 0,5 mm starkem Lötzinn
- Universelles Schaltnetzteil (1,5 – 12V; 500mA)
- 10-adriges Flachbandkabel und anderes Kabel
- Rote und grüne Standard LEDs
- ATTiny2313 Mikrocontroller
- 180 Ohm Widerstände
- 7-Segment-Anzeige mit gemeinsamer Kathode (Höhe: 14,2mm)
- Stiftleisten und Buchsenleisten (Raster: 2,54mm)
- 4-polige DIP-Schalter
- Kurzhubtaster
- Schrumpfschlauch
- Punkt/Streifenraster Platine

Für das Erstellen der LOCAD-Schaltungen wurde LOCAD 2004 genutzt. Die Schaltskizzen des AVR-Mikrocontroller wurden erstellt mit EAGLE (<http://www.cadsoft.de>). Des Weiteren wurde noch Crocodile Clip genutzt. Die Programmierung erfolgte mithilfe des gcc-avr Compilers (<http://www.nongnu.org/avr-libc/>) auf einem Linux System (Kubuntu 7.04).

Es folgen die Schaltskizzen und weitere Fotos.

## 4.1 Schaltskizzen und Fotos

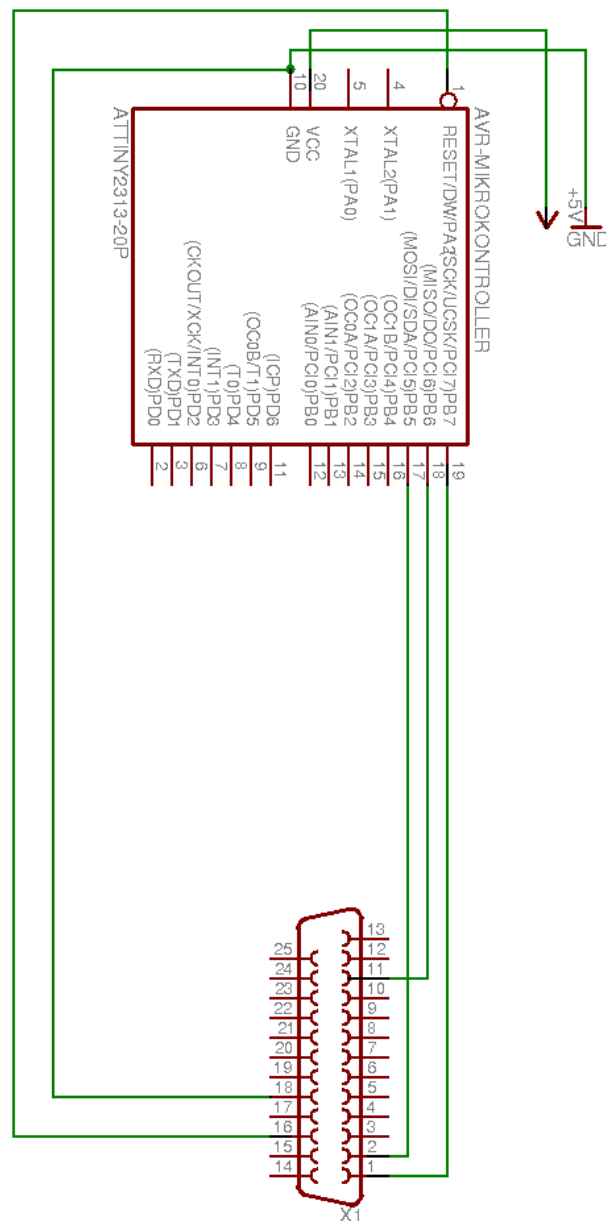


Abbildung 11: Verbindung zum PC ( Schaltskizze)

Abbildung 12:  
Pull-Up  
Widerstand  
(Schalter  
offen)

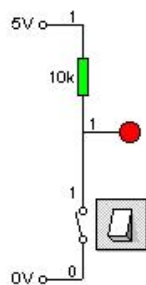
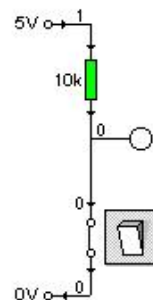


Abbildung  
13: Pull-Up  
Widerstand  
(Schalter  
geschlossen)



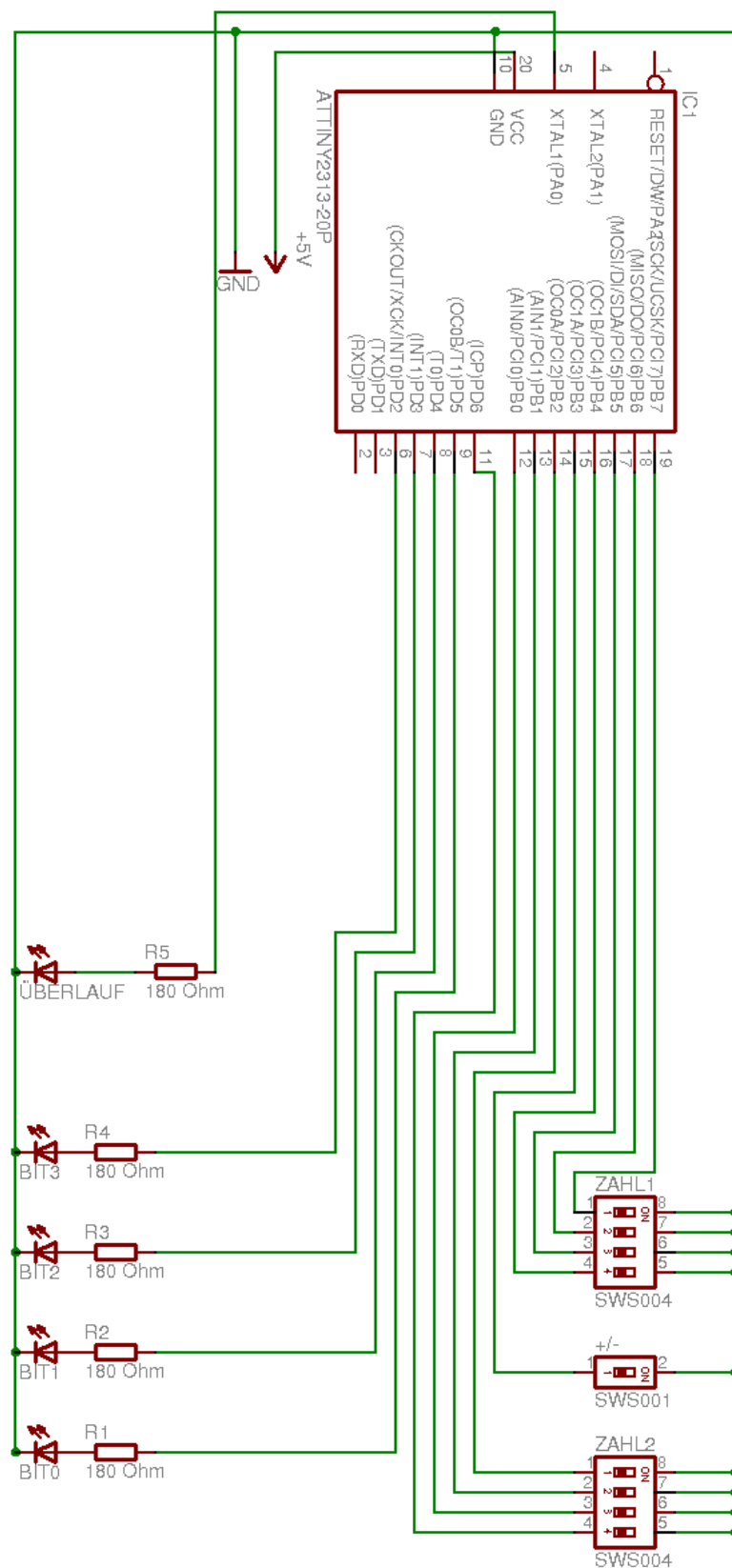


Abbildung 14: Umschaltbare Rechenschaltung (Schaltskizze)

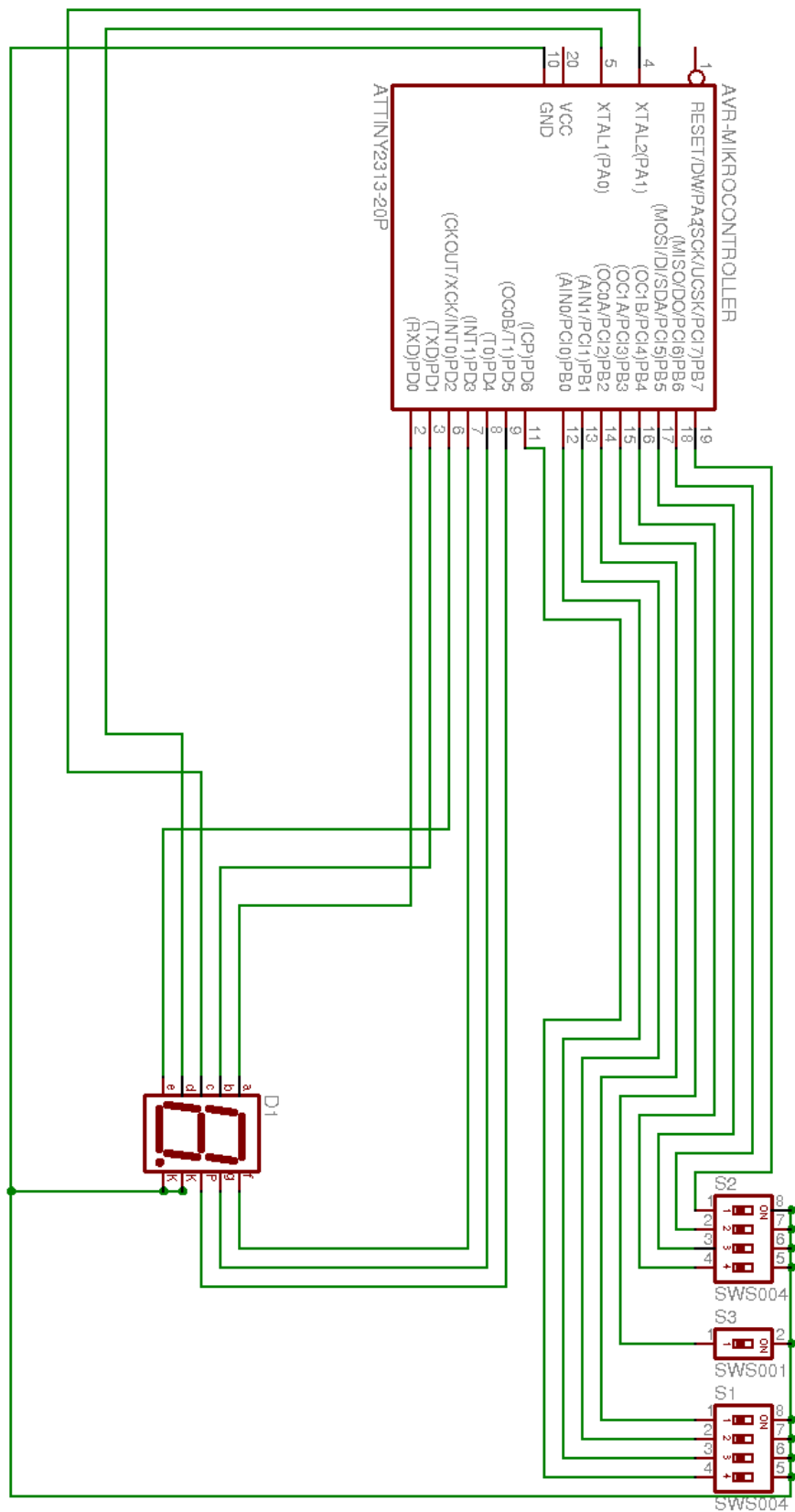


Abbildung 15: Erweiterte umschaltbare Rechenschaltung (Schaltskizze)

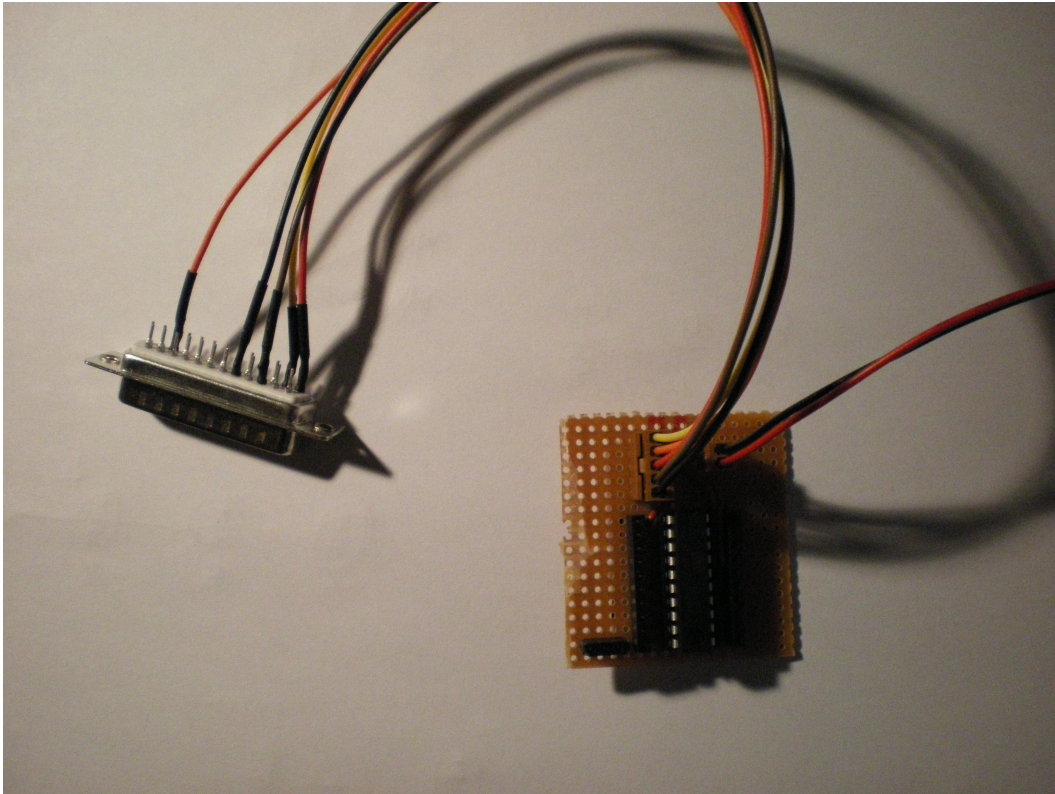


Abbildung 16: Verbindung zum PC ( Foto)

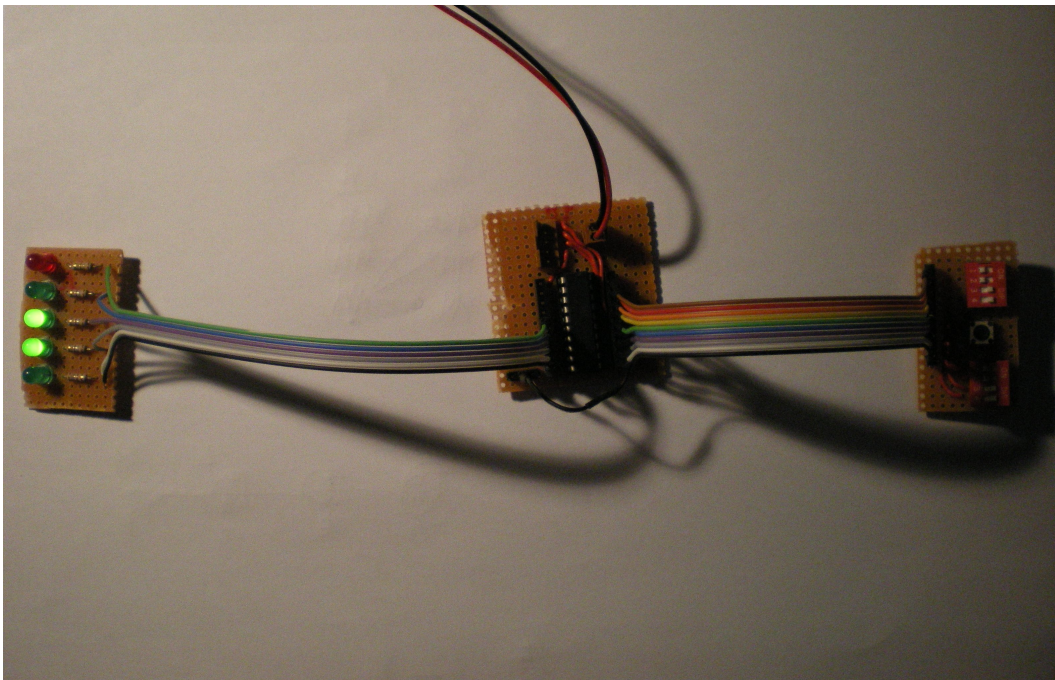


Abbildung 17: Umschaltbare Rechenschaltung (Foto)



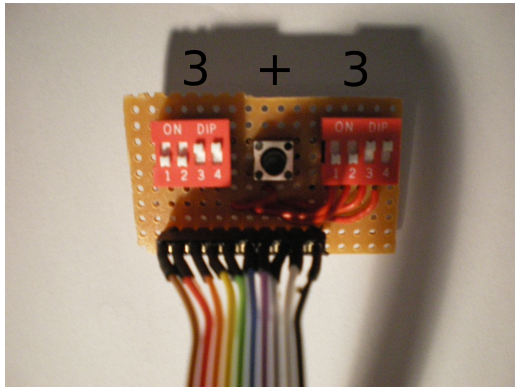


Abbildung 18: Zahleneingabe - umschaltbare Rechenschaltung

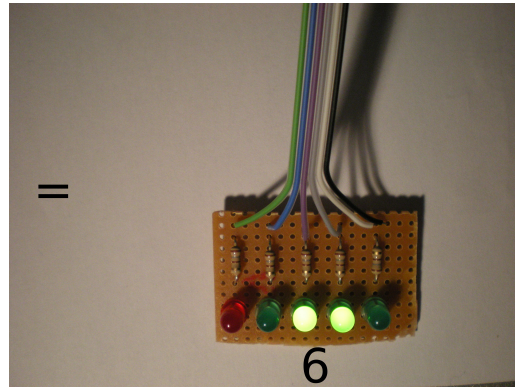


Abbildung 19: LED-Ausgabe - umschaltbare Rechenschaltung

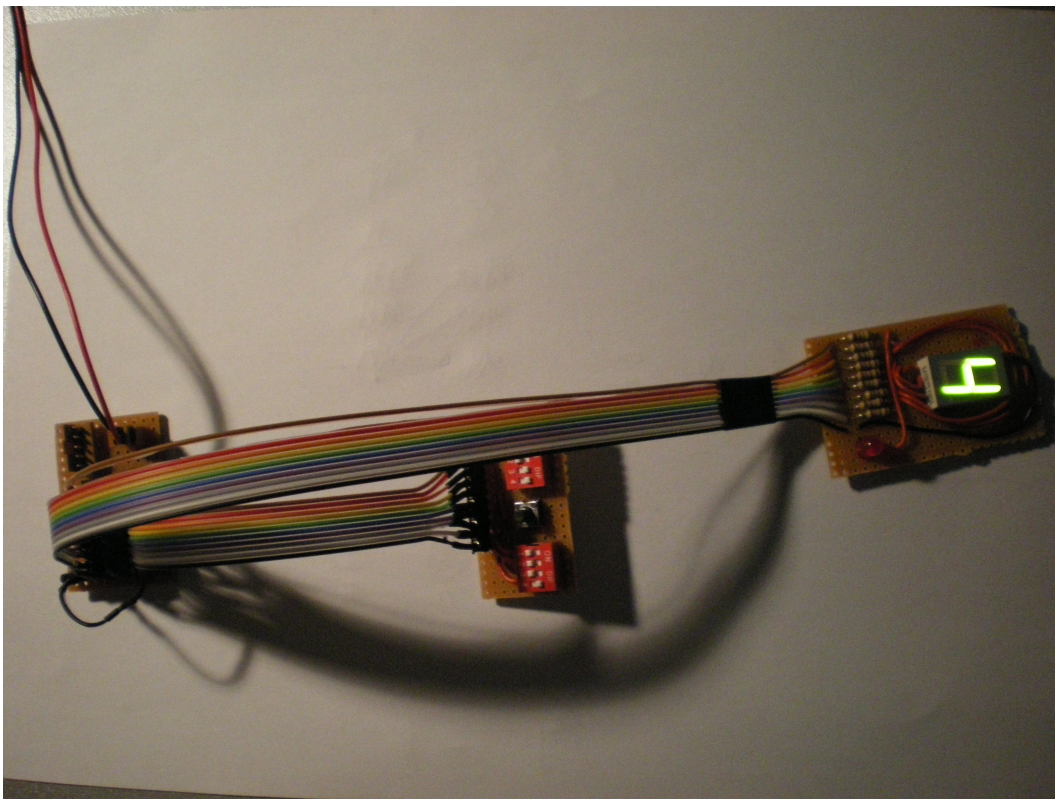


Abbildung 20: Erweiterte umschaltbare Rechenschaltung (Foto)

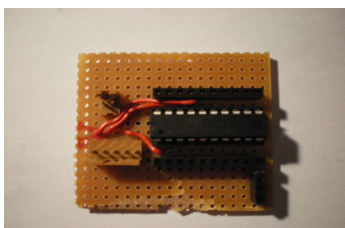


Abbildung 21: ATTiny2313 - 1

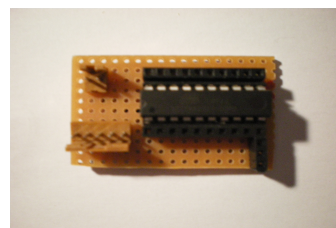


Abbildung 22: ATTiny2313 - 2



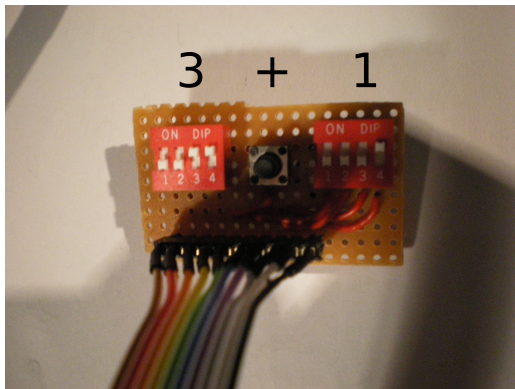


Abbildung 23: Zahleneingabe - erweiterte umschaltbare Rechenschaltung

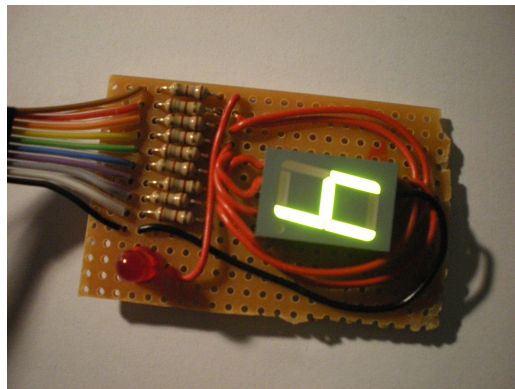


Abbildung 24: 7-Segm. Ausgabe - erweiterte umschaltbare Rechenschaltung

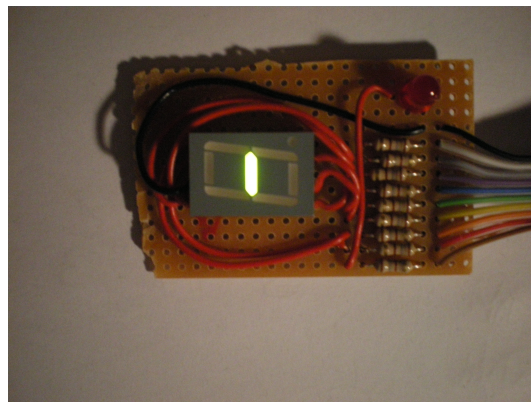


Abbildung 25: 7-Segm. Ausgabe bei Überlauf

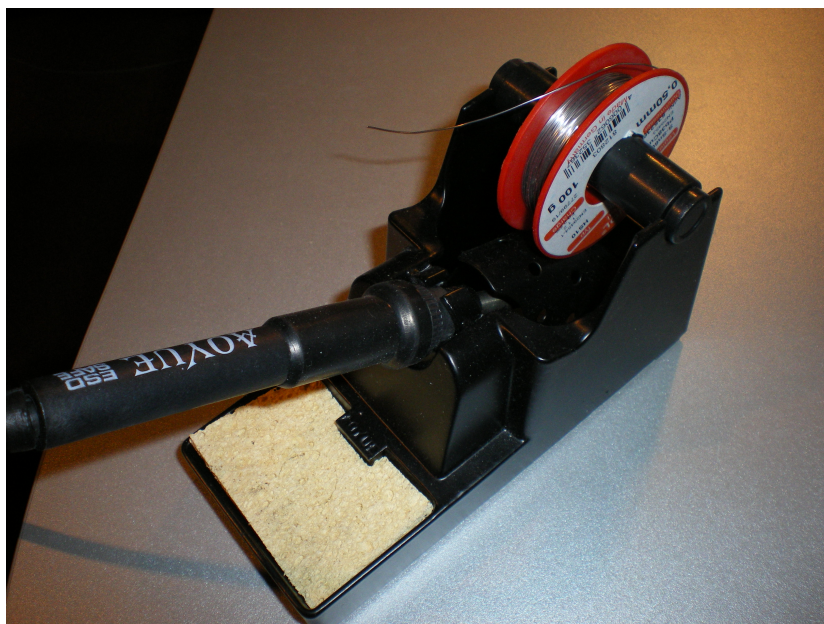


Abbildung 26: AOYUE Lötkolben mit Ständer und Lötzinn

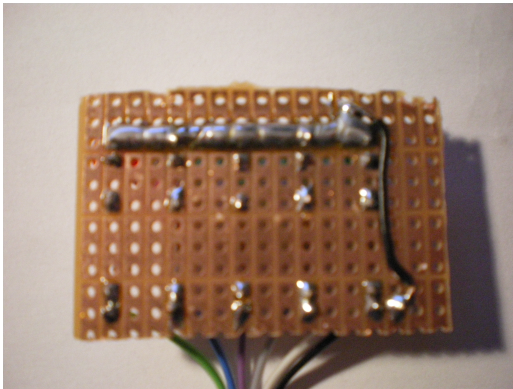


Abbildung 27: LED-Ausgabe  
(Platinenunterseite)

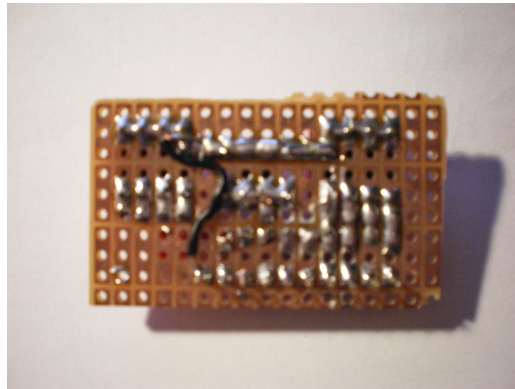


Abbildung 28: Zahleneingabe  
(Platinenunterseite)

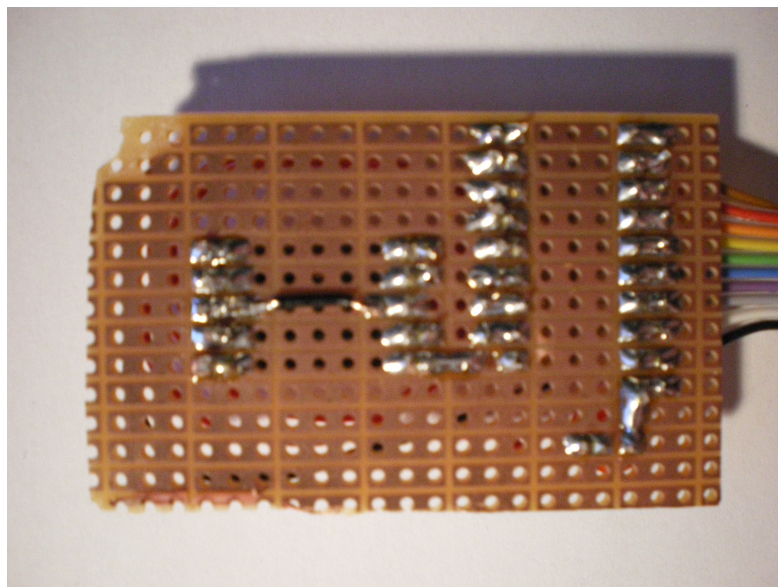


Abbildung 29: 7-Segment-Anzeige (Platinenunterseite)

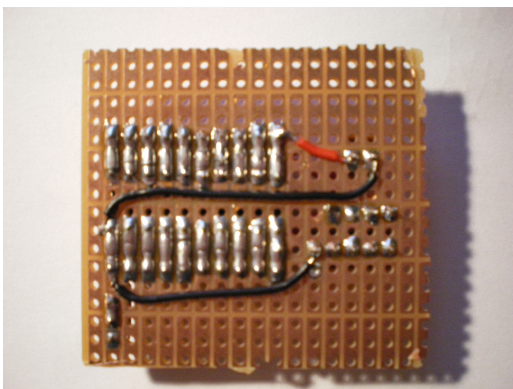


Abbildung 30: ATTiny2313-1  
(Platinenunterseite)

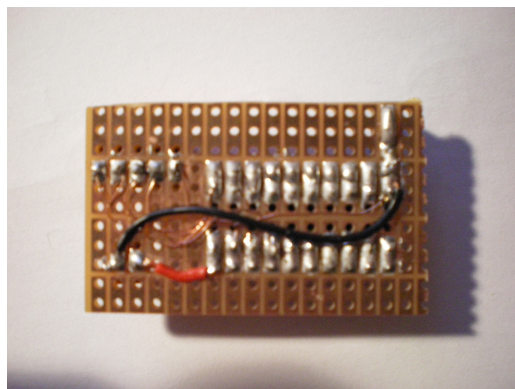


Abbildung 31: ATTiny2313-2  
(Platinenunterseite)





Abbildung 32: Genutzte Materialien

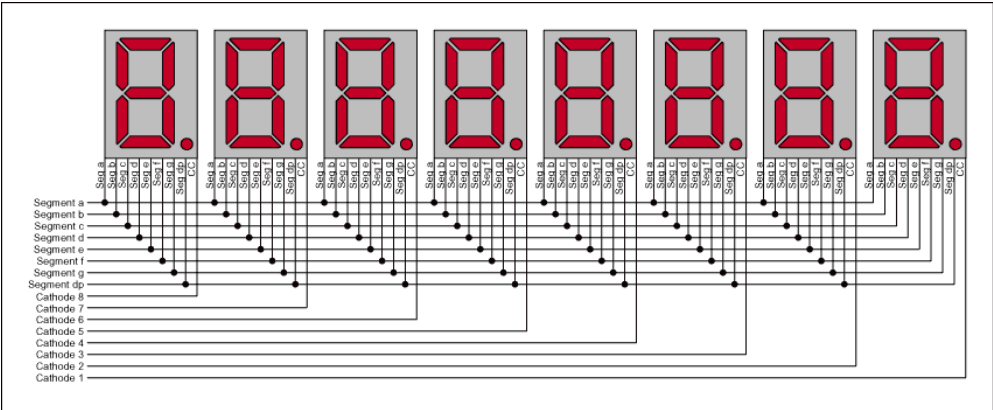


Abbildung 33: 7-Segment-Anzeigen Multiplexer ähnlich, [Charlieplex]

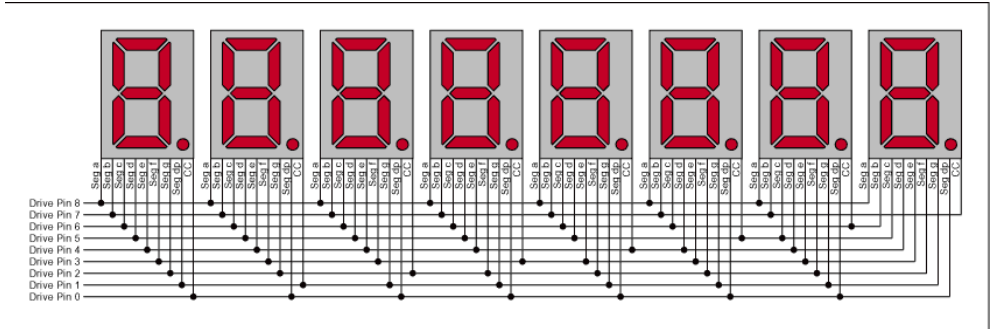


Abbildung 34: 7-Segment-Anzeigen Charlieplexer ähnlich, [Charlieplex]

## 5 VERZEICHNISSE

### 5.1 Literaturverzeichnis

\*[Wikipedia]: Wikipedia.org: Atmel AVR.

URL: [http://de.wikipedia.org/wiki/Atmel\\_AVR](http://de.wikipedia.org/wiki/Atmel_AVR) (13.02.08)

\*[AVR-GCC-TUT]: Andreas Schwarz: AVR-GCC-Tutorial.

URL: <http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial>  
(09.02.08)

\*[Datenblatt-AVR]: Atmel: ATtiny2313 Preliminary.

URL: [http://www.atmel.com/dyn/resources/prod\\_documents/doc2543.pdf](http://www.atmel.com/dyn/resources/prod_documents/doc2543.pdf)  
(04/06)

\*[Datenblatt-7]: 7-Segment-Anzeigen Datenblatt. URL:

[http://www.micropik.com/provisional/PDF/SC56-11GWA\(V5\)%5B1%5D.pdf](http://www.micropik.com/provisional/PDF/SC56-11GWA(V5)%5B1%5D.pdf)

[Instruct]: Ghetto Programming: Getting started with AVR microprocessors on the cheap.

URL: <http://www.instructables.com/id/Ghetto-Programming%3a-Getting-started-with-AVR-micro/> (18.11.06)

\*[LOCAD-Skript]: Karl-Heinz Loch: Technische Informatik mit LOCAD 2002. 2. Auflg. Erkelenz: 2002.

[C++]: Herbert Schildt: C++: Die professionelle Referenz. Bonn: mitp, 2004. ISBN 3-8266-1367-8

\*[Charlieplex]: Charlieplexing - Reduced Pin-Count LED Display Multiplexing.

URL: [http://www.maxim-ic.com/appnotes.cfm/appnote\\_number/1880](http://www.maxim-ic.com/appnotes.cfm/appnote_number/1880)  
(10.02.03)

---

\* siehe auch mitgelieferte CD -> Dokumente (als PDF oder HTML)

## 5.2 Abbildungsverzeichnis

AVR-Mikrocontroller [Wikipedia]....3	(Foto).....33
ATTiny2313 Aufbau [Datenblatt- AVR].....4	Zahleneingabe - umschaltbare Rechenschaltung.....34
Anschluss an der parallelen 25 pol. Schnittstelle (Schaltskizze).....5	LED-Ausgabe - umschaltbare Rechenschaltung.....34
Kabel für die Programmierung (Foto).....6	Erweiterte umschaltbare Rechenschaltung (Foto).....34
Pull-Up [AVR GCC TUT].....10	ATTiny2313 - 1.....34 ATTiny2313 - 2.....34
Umschaltbare Rechenschaltung, [LOCAD-Skript]: S. 41.....12	Zahleneingabe - erweiterte umschaltbare Rechenschaltung...35
Umschaltbare Rechenschaltung (Schaltskizze).....13	7-Segm. Ausgabe - erweiterte umschaltbare Rechenschaltung...35
Umschaltbare Rechenschaltung mit Ziffernanzeige (LOCAD-Schaltung) .....19	7-Segm. Ausgabe bei Überlauf...35 AOYUE LötKolben mit Ständer und Lötzinn.....35
erweiterte Umschaltbare Rechenschaltung (Schaltskizze)..20	LED-Ausgabe (Platinenunterseite) .....36
7-Segment-Anzeige [Datenblatt-7] .....21	Zahleneingabe (Platinenunterseite) .....36
Verbindung zum PC ( Schaltskizze) .....30	7-Segment-Anzeige (Platinenunterseite).....36
Pull-Up Widerstand (Schalter offen) .....30	ATTiny2313-1 (Platinenunterseite) .....36
Pull-Up Widerstand (Schalter geschlossen).....30	ATTiny2313-2 (Platinenunterseite) .....36
Umschaltbare Rechenschaltung (Schaltskizze).....31	Genutzte Materialien.....37
Erweiterte umschaltbare Rechenschaltung (Schaltskizze)..32	7-Segment-Anzeigen Multiplexer ähnlich, [Charlieplex].....37
Verbindung zum PC ( Foto).....33	7-Segment-Anzeigen Charlieplexer ähnlich, [Charlieplex].....37
Umschaltbare Rechenschaltung	

### **5.3 Tabellenverzeichnis**

Daten des ATTiny2313 [Datenblatt-AVR].....	4
PIN-Belegungen [Instruct].....	6
Wichtige verfügbare Compiler [Wikipedia].....	6
Anschluss am Mikrocontroller.....	21