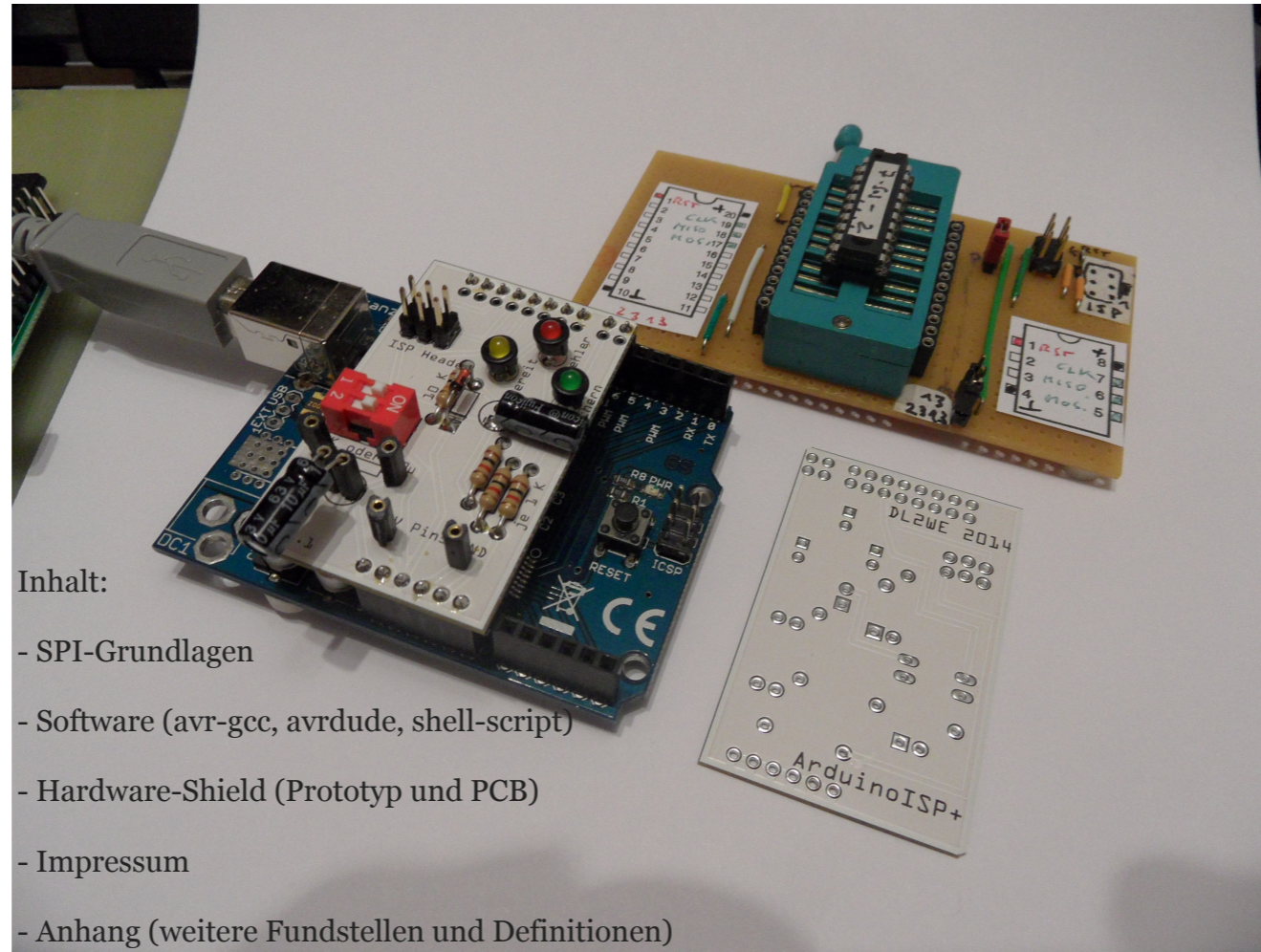


# Arduino als AVR-ISP am Raspberry PI



Inhalt:

- SPI-Grundlagen
- Software (avr-gcc, avrdude, shell-script)
- Hardware-Shield (Prototyp und PCB)
- Impressum
- Anhang (weitere Fundstellen und Definitionen)

ERFAHRUNGSBERICHT FÜR BASTLER

MIT GRUNDKENNTNISSEN AUS ELEKTRONIK UND PROGRAMMIERUNG

# Vorwort

Bei der Internet-Recherche zur Programmierung von Atmel-Prozessoren ergeben sich mehrere Alternativen, auch mittels des Arduino Bords. „Using an Arduino as an AVR ISP (In-System Programmer)“.

Die Fundstelle ist: <http://arduino.cc/en/Tutorial/ArduinoISP>. Eine ergiebige Informationsquelle rund um (auch Atmel-) Prozessoren ist: <http://www.mikrocontroller.net>.

Eine Zusammenstellung aller relevanten Fundstellen und Definitionen ist im Anhang nach dem Impressum zu finden. Diese Erfahrungen beziehen sich auf meinen Arduino Uno mit dem Prozessor ATmega 168. Die Übertragbarkeit auf andere Arduinotypen habe ich nicht testen können.

Die Faszination des „Minicomputer“ Raspberry Pi mit seinem Open Source Betriebssystem auf Unix/Linux-Basis mit professioneller Grafikschnittstelle hat auch mich in den Bann gezogen <http://RaspberryPi.org>. Die problemlose Installation der Entwicklungsumgebung (Arduino API) mit dem über die USB-Schnittstelle angeschlossenen Arduino als ISP-Programmer erleichterte mir die Entscheidung loszulegen.

Die unmittelbare Nutzung der GPIO-Pins des Raspberry Pi sollte möglich sein, wird hier aber nicht versucht (GPIO General Purpose Input Output).

Wie immer lag der „Teufel im Detail“. Durch eine kleinere Hardwarebastelei und Shell-Scripts zur Vereinfachung der Handeingaben habe ich jetzt eine schlanke Lösung mit vorhandener preisgünstiger Technik, die relativ leicht nachvollziehbar sein soll. Der Einstieg in diese Materie produziert vermutlich bei dir sofort eigene Ideen, viel Spaß!

DL2WE, Hannover 7. November 2014

Der Weg über den Arduino ist für mich reizvoll, weil

1. Arduino bei mir vorhanden und mir länger vertraut ist,
2. ein Beispiel-Sketch ArduinoAVR von Randall Bohn mich animiert hat,
3. ich mit dem Raspberry Pi experimentiere und die Arduino-API problemlos zu installieren ist,
4. mit der Installation der ArduinoAPI auch der avr-gcc Compiler und avrdude mitinstalliert werden, die sofort im Terminal des Raspberry Pi direkt ansprechbar sind,
5. C in allen Varianten eine universelle Programmiersprache ist, deren Compiler sehr effiziente Codes erzeugen,
6. der Bau des mini-Shield als Prototyp mit ISP-Kabel eine nette Bastelei ist und damit auch viele weitere Atmel-Prozessoren programmierbar werden,
7. die Open Source Produkte keine Lizenzverstöße vermuten lassen,
8. die Annahme, dass ArduinoAVR-ISP mit dem Shield an allen Rechnern nutzbar sein sollte, die die ArduinoAPI installieren können,
9. der Raspberry Pi durch interessante Funktionen der Atmel-Prozessoren erweitert werden kann und
10. das Zusammentreffen mehrerer Fachdisziplinen immer wieder eine persönliche Herausforderung ist.

---

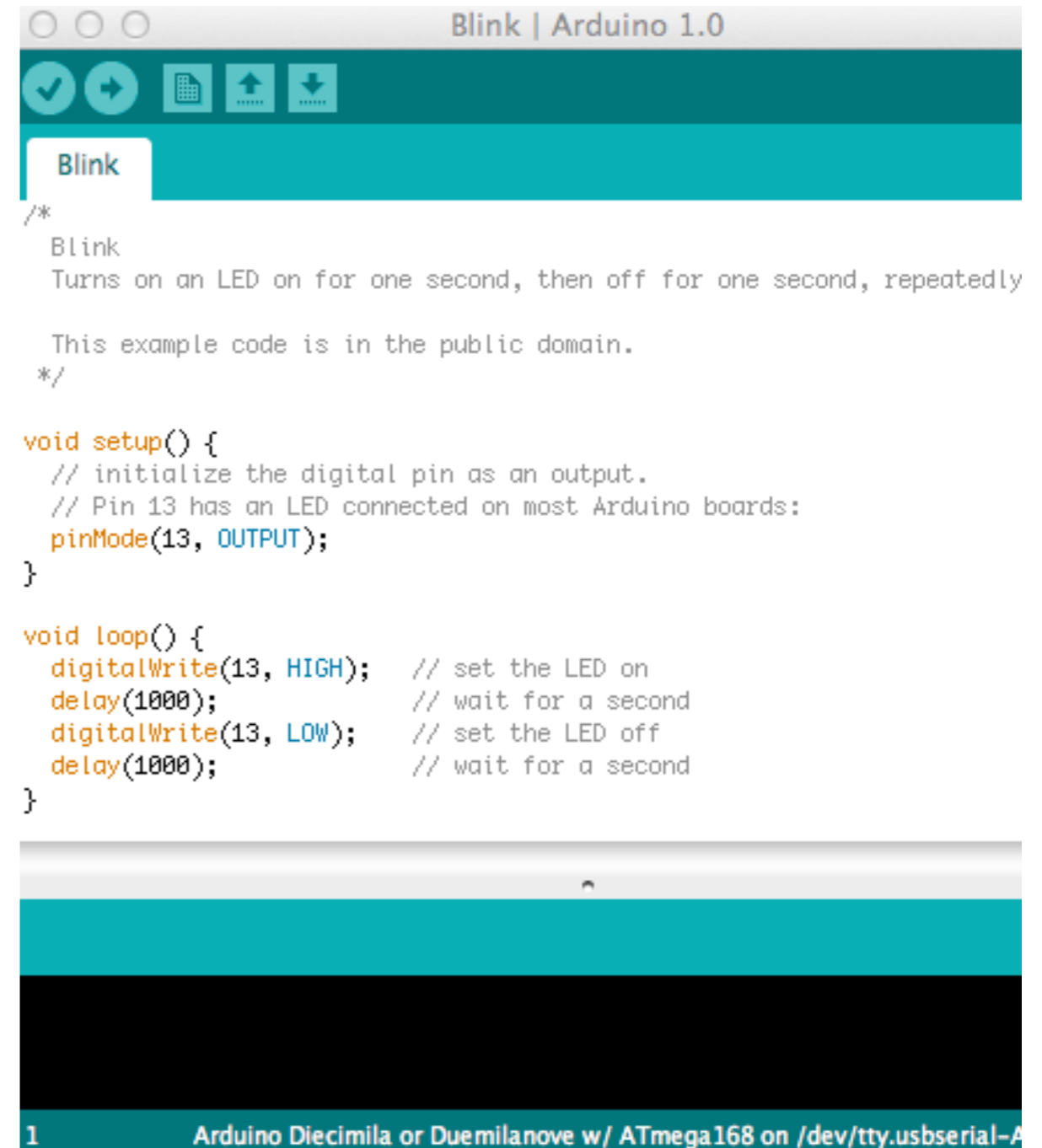
## DAS SOLLTEST DU MITBRINGEN

---

Du besitzt und nutzt den „Minicomputer Raspberry Pi“ mit dem neuesten Betriebssystem Wheezy (Basis ist Debian-Linux). Die Arduino-IDE (integrated development environment - integrierte Entwicklungsumgebung) konnte von dir installiert werden.

Nach dem Aufruf der IDE, zeigt sich ein leerer Sketch (Programmierbereich), in den hier der Blink-Sketch geladen wurde oder auch eingetippt werden kann. Auch die Verbindung über die (virtuelle serielle) USB-Verbindung steht. Jetzt lässt sich der Sketch kompilieren, laden und ausführen. Das alles solltest du bereits gemacht haben.

Außerdem sind Grundfertigkeiten im Elektronikbasteln und Mikroprozessoren, Programmierkenntnisse und die Lust auf Neues von Vorteil.



The screenshot shows the Arduino IDE interface. The title bar reads "Blink | Arduino 1.0". The code editor displays the following code:

```
/*
 * Blink
 * Turns on an LED on for one second, then off for one second, repeatedly
 *
 * This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH); // set the LED on
  delay(1000);           // wait for a second
  digitalWrite(13, LOW); // set the LED off
  delay(1000);           // wait for a second
}
```

Below the code editor, the serial monitor is visible, showing the output of the program. The status bar at the bottom indicates the board is "1 Arduino Diecimila or Duemilanove w/ ATmega168 on /dev/tty.usbserial-A".

# In-System Programmer

### ZUSAMMENSPIEL DER SOFTWARE

1. Der Arduino ist „lediglich“ ein intelligenter Baustein zwischen `avr-gcc/avrdude` und dem zu programmierenden Mikroprozessor
2. **avr-gcc** (Compiler) wird im Terminalfenster des Raspberry Pi Rechners ausgeführt
3. **avrdude** (Transferprogramm) ebenfalls!
4. Der ArduinoISP-Sketch empfängt die Ausgabe von `avrdude` über den SerialPort (im Menü: Tools-SerialPort auszuwählen) und leitet sie mit dem sog. „STK500-Protokoll“, das Atmel-Prozessoren „verstehen“, an diese zur Programmierung des Programmspeichers weiter.

Zuerst lade den Sketch von Randall Bohn, der im Menü unter File-Examples-ArduinoISP zu finden ist. Nach der Übersetzung (Compilierung/Verifizierung) und Hochladen (Upload) in den Programmbereich des Arduino beginnt der Sketch abzulaufen (bei mir z.B. im ATmega168). Sind die Leuchtdioden wie vorgegeben angeschlossen, so „atmet“ die Herzschlag (heartbeat) genannte Diode, der Sketch wartet auf Arbeit. So weit so gut!

Der Arduino IDE Sketch muß bleiben wo er ist, im ATmega168 meines Arduino und auf Input warten, d.h. die IDE des Arduino wird nicht mehr benutzt (es sei „nur“ für die C-Quellcode-Eingabe). Die Softwarewerkzeuge „avr-gcc“ und „avrdude“ sind jetzt die zu nutzenden Programme. Der jeweilige Aufruf erfolgt per Befehl an der Konsole/Terminalfenster des Raspberry Pi. Der C-Quellcode (Sourcecode) wird mit einem geeigneten Editor erstellt, ich benutze TextWrangler unter Apple OSX, Notepad++ unter Windows und beim Raspberry Pi den Texteditor Nano bzw. die IDE „Geany“, alles Open Source Software. Du nutzt wahrscheinlich einen Editor deiner Wahl.

Die „Arbeitskette“ besteht aus verdrahteter Hardware, Software-Aufrufen im Terminal und dem Sketch ArduinoISP:

Raspberry Pi --->USB--->Arduino <---ISP-Pins--->Mikro-  
Terminal                      ArduinoISP                      prozessor  
  
- avr-gcc  
  
- avrdude

## ISP VERKABELUNG AM MIKROPROZESSOR

Die SPI-Schnittstelle arbeitet synchron mit einer gemeinsamen Taktleitung für Sender und Empfänger. Der Arduino-ISP Sketch arbeitet als SPI-Master, der den angeschlossenen Slave steuert. Diese Master/Slave Rollen werden nicht nur für die Programmierung von Atmel-Prozessoren vorgegeben, sondern sind allgemein für diese Art Kommunikation definiert. Beispiele sind das Auslesen von Temperaturchips, Porterweiterungschips, Analog/Digitalwandlerchips, usw.

Daneben gibt es weitere Schnittstellen, die mit weniger Verkabelung auskommen z.B. die I2C-Schnittstelle. Bei **SPI** haben die Pins folgende genormte Bedeutungen:

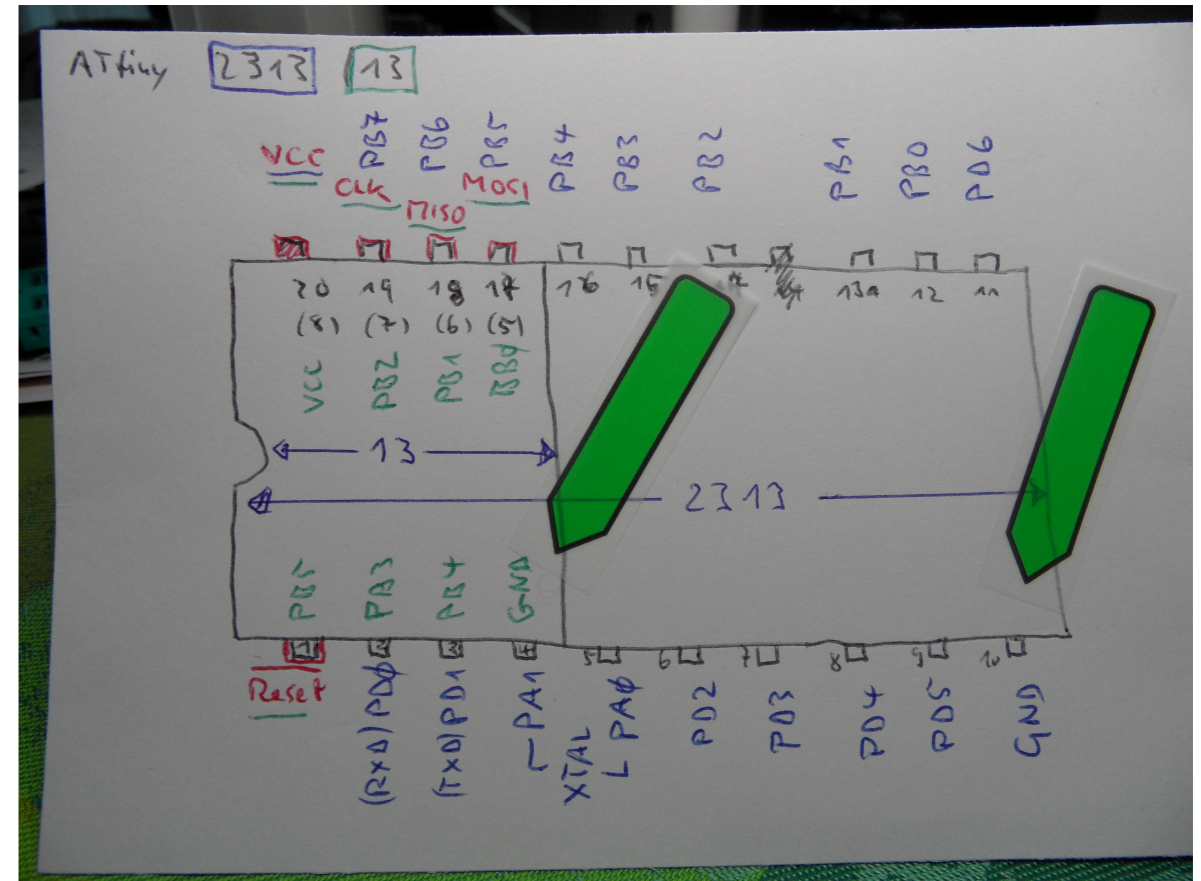
CLK      gemeinsame Taktleitung

MOSI    Master Out, Slave In

MISO    Master In, Slave Out

RST      Reset am Slave

Der ArduinoISP Sketch sorgt dafür, dass diese gleichnamigen Pins des zu programmierenden Mikroprozessors mit dem richtigen „Protokoll“ angesteuert werden, wenn die Datendatei mit dem Mikroprozessor-Code über USB von avrdu- de „angeliefert“ wird.



Vergleich der Pinbelegung beim ATtiny13 und ATtiny2313 aus dem Datenblatt von Atmel. Der Chip ist **Slave** in der SPI-Schnittstelle.

Die Pins für die Programmierung (rot: VCC, CLK, MISO, MOSI und Reset) sind identisch. Lediglich ist der Anschluss GND (Ground) beim ATtiny13 auf Pin 4 bzw. beim ATtiny2313 auf Pin 10 zu finden.

Es ist nur jeweils ein Prozessor zur Zeit programmierbar.

# ISP VERKABELUNG AM ARDUINO

## ArduinoISP

```
// this sketch turns the Arduino into a AVRISP
// using the following pins:
// 10: slave reset
// 11: MOSI
// 12: MISO
// 13: SCK

// Put an LED (with resistor) on the following pins:
// 9: Heartbeat - shows the programmer is running
// 8: Error - Lights up if something goes wrong (use red if that makes sense)
// 7: Programming - In communication with the slave
//
// October 2009 by David A. Mellis
// - Added support for the read signature command
//
// February 2009 by Randall Bohn
// - Added support for writing to EEPROM (what took so long?)
// Windows users should consider WinAVR's avrdude instead of the
// avrdude included with Arduino software.
//
// January 2008 by Randall Bohn
// - Thanks to Amplificar for helping me with the STK500 protocol
// - The AVRISP/STK500 (mk I) protocol is used in the arduino bootloader
// - The SPI functions herein were developed for the AVR910_ARD programmer
// - More information at http://code.google.com/p/mega-isp
```

Im Sketch ArduinoISP sind die Pin-Belegungen des **Masters** für ISP angegeben. Jetzt ist die Verkabelung prinzipiell mit gleichnamigen Pins am Slave möglich.

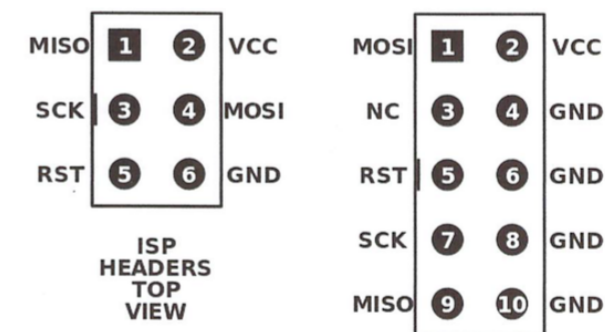
Nachdem der ArduinoISP-Sketch kompiliert und in den Arduinoprozessor hochgeladen wurde, sollte die „Herzschlagdiode“ am Pin 9 die Bereitschaft anzeigen.

Im <http://arduino.cc/en/Tutorial/ArduinoISP> findet sich der Hinweis auf die Veränderung der Schlagfrequenz `delay(40)` in `delay(20)` abhängig vom Arduino-Modell. Auch der Anschluß von  $10\ \mu\text{F}$  an „reset“ des Arduino (disable autoreset „**after** uploading the sketch“) ist hier angegeben.

Darüberhinaus sind nach den Datenblättern der Prozessoren Zeitbedingungen für ein sauberes Power-on-Reset-Signal am zu programmierenden Prozessor herzustellen. Geeignet hierfür ist ein RC-Kreis aus  $10\text{K}\Omega$  und 1 bis  $10\ \mu\text{F}$  mit Freilaufdiode.

Es bietet sich an, diese Zusatzbeschaltung mit den drei Dioden und auch einen Übergang zu einem Programmierkabel samt Leuchtdioden auf einem „Mini-Shield“ unterzubringen.

Das Programmierkabel sollte der Standardbelegung von ISP-Steckern/Buchsen entsprechen:



## SOFTWARE

### TERMINALFENSTER

Das Terminalfenster des Raspberry Pi (z.B. Betriebssystem Wheezy) erlaubt die Eingabe von Befehlen, die ausgeführt werden, z.B. ls (entspricht dem DIR in der DOS-Konsole). Die Befehle können durch Parameterübergabe spezifiziert werden, z.B.

```
ls -l
```

```
drwxr-xr-x  4 weingarten  staff    136 28 Mär  2013 Applications
drwx-----+ 42 weingarten  staff   1428 13 Aug  08:39 Desktop
drwx-----+ 92 weingarten  staff   3128 13 Aug  08:36 Documents
drwx-----+ 178 weingarten  staff   6052 29 Jul  17:49 Downloads
```

usw.

Wechsel des Arbeitsverzeichnisses (cd untergeordnetes Verzeichnis oder cd .. für das übergeordnete Verzeichnis). Bei meinem Raspberry Pi speichere ich die Programmcodes für AT-tiny13-Projekte (hier als Beispiel das Projekt blink) in

```
cd /home/pi/CProgramme/atmel/attiny13/blink
```

Der Befehl ls würde das Quellprogramm blink.c anzeigen.

Ein Compileraufruf wird auf Dateien dieses Verzeichnisses angewandt, also avr-gcc blink.c findet die Quelldatei.

Die Arbeitsschritte für den Compiler

#### **avr-gcc**

1. Sketch-Datei blink.c (Quellcode) kompilieren
2. Es entsteht eine sog. Objektdatei blink.o
3. Weitere Funktionen z.B. aus Bibliotheken werden dazu definiert, es entsteht eine Link-Datei blink.elf
4. Hieraus wird schließlich die Upload-Datei blink.hex erstellt. Deren Code im Intel-Hex-Format „versteht“ der Mikroprozessor als Programmcode.

Die Arbeitsschritte des Transferprogrammes

#### **avrdude**

mit der zuvor erzeugten blink.hex werden dafür sorgen, dass der Programmcode in den Mikroprozessor transferiert wird.

Alle Arbeitsschritte werden in sog.

#### **Scripts**

zusammengefasst, damit der manuelle Tippaufwand und Schreibfehler sich in Grenzen halten.

Es folgt mein Kochrezept zur Erstellung einer in den Programmbereich eines Prozessors hochladbaren Datei.

Die Quellcode-datei hat den Typ \*.c, die wir als erstes Kompilieren müssen, z.B.:

```
avr-gcc -mmcu=attiny2313 -Os -c t2313_1.c
```

Der Parameter -mmcu=attiny2313 teilt dem Compiler den Prozessortyp mit. Es entsteht die Objektcode-datei t2313\_1.o, die noch mit weiterem Code zu vervollständigen ist:

```
avr-gcc t2313_1.o -o t2313_1.elf
```

Es entsteht der Dateityp t2313\_1.elf der Arbeitskette, die als prozessor-geeignete Form t2313\_1.hex für den Upload in den Prozessor umzuwandeln ist:

```
avr-objcopy -O ihex -j .text -j .data t2313_1.elf t2313_1.hex
```

Das war es schon, genutzt wird die zuletzt erzeugte Datei im Transferprogramm avrdude.

Ich habe zur Vereinfachung und die spätere Nutzung von Scripts einige Festlegungen bei mir getroffen (die jeder für sich anders festlegen kann):

1. Je nach Prozessortyp wird unterhalb /home/pi/CProgramme/atmel ein Verzeichnis je Prozessor angelegt, z.B. /home/pi/CProgramme/atmel/attiny2313
2. Dort sind allgemeine Dokumentationen und Scripte für diesen µProzessor gespeichert (z.B. für Kompilierung, verifizieren und auslesen und Hochladen in den Programmbereich), also eine Verzeichnisebene oberhalb einzelner Vorhaben.
3. Beispielordner attiny2313 mit allgemeinen Informationen zum Prozessor im Ordner doku, einem Vorhaben t2313\_1 und den Befehlen für „flashen“, „lesen“ und „verifizieren“:



4. Die einzelnen Vorhaben werden also getrennt gespeichert. Darin ist die C-Code-Datei t2313\_1.c mit einem geeigneten Editor zu erstellen. Ebenfalls sind hier (bei mir) die speziellen Vorhabensdokumentationen abzulegen.
5. Die durch Einsatz des Compilers entstehenden Dateien t2313\_1.o, t2313\_1.elf und auch später t2313\_1.hex befinden sich ebenfalls im Projektordner t2313\_1.



## AVRDUDE TRANSFERPROGRAMM

<http://savannah.nongnu.org/projects/avrdude>

avrdude wurde bei der Installation der Arduino-IDE mitgeliefert, da sie es selbst nutzt.

Der Aufruf erfolgt im Terminalfenster.

Wir nutzen mit avrdude nur den im Prozessor ausführbaren Code \*.hex, der im vorigen Abschnitt erzeugt wurde. Hilfreich kann das Tutorial sein:

<http://ladyada.net/learn/avr/avrdude.html>

avrdude -p  $\mu$ Prozessortyp (z.B. Atmega168)

-c Programmierooltyp (z.B. avr911)

-P Schnittstelle (z.B. /dev/ttyUSB0)

-b 19200 (z.B. serielle Geschwindigkeit/Baudrate)

-U z.B. flash:w:attiny13.hex

Mögliche  $\mu$ Prozessortypen und Programmierooltypen können aus der Dokumentation auf der Webseite des Projektes oder direkt aus einem „falschem“ Aufruf: avrdude 123 auf Grund der Fehlermeldung entnommen werden ;-)

Der Parameter -U flash:w:attiny13.hex enthält z.B. die Anweisung den flash-Speicher zu füllen (w=schreiben, r=lesen, v=verifizieren) und die hex-Datei für den upload.

Anstelle des Flashspeichers lässt sich auch der Fuse-Inhalt und der EEPROM-Inhalt definieren:

-U <memtype>:r|w|v:<filename><:filetype>

Dabei wird der File-Typ meist weggelassen, da er als Standard i=intel hex angenommen wird. Der memtype kann noch sein:

eprom ---> interner Speicher z.B. für Konstanten

hfuse ---> high Fuse

lfuse ---> low Fuse

efuse ---> extended Fuse

Die Fuseprogrammierung solltest du nur angehen, wenn du genau weißt, was du tust.

Ich belasse es hier bei der sicheren Seite der Programmierung des Flash-Speichers, da der Safemode (sicherer Modus) für die Fuse-Bits standardmäßig eingestellt ist. Mit den Fuse-Bits kann der Chip z.B. gegen Auslesen oder Neuprogrammierung geschützt werden, was aber auch die Zahl zerschossener Chips erhöhen könnte. Die Fuse-Bits sind im Datenblatt des Herstellers angegeben und auch in den genannten Fundstellen beschrieben.

## Kommandosprache im Terminalfenster

Ein Shell-Script enthält Programmaufrufe/Befehle der Reihe nach, vergleichbar mit den unter Windows evtl. noch bekannten \*.bat-Dateien. Erstellt wird ein Script mit einem Texteditor deiner Wahl. Ich habe nicht vor, die Scriptsprache und das „Ausführbar-machen“ näher zu erläutern, nur soviel verkürzt:

***chmod +x atmega8-lesen --> ausführbar machen***

***./atmega8-lesen --> mit dieser Eingabe starten.***

Fundstelle zur Programmierung in der Scriptsprache:

[http://de.wikibooks.org/wiki/Linux-Praxisbuch:\\_Shellprogrammierung](http://de.wikibooks.org/wiki/Linux-Praxisbuch:_Shellprogrammierung)

Folgende Programmaufrufe sind am Beispiel der Programmierung des ATtiny2313 mit dem t2313\_1.c zu verketteten („Kochbuch“):

1. **avr-gcc** -mmcu=attiny2313 -Os -c t2313\_1.c -o t2313\_1.o
2. **avr-gcc** t2313\_1.o -o t2313\_1.elf
3. **avr-objcopy** -O ihex -j .text -j .data t2313\_1.elf t2313\_1.hex
4. **avrdude** -p t13 -c avrisp -U flash:w:t2313\_1.hex  
-P /dev/ttyUSB0 -b 19200

Der Aufruf für die Flash-Programmierung soll (bei mir)

```
./t2313-flashen t2313_1
```

ohne Typangabe .c erfolgen. Parameter selbst dürfen kein Minuszeichen enthalten! Im Folgenden gehe ich das Script zum Hochladen einer .hex in den ATtiny2313 zeilenweise durch. Die erste Zeile ist für die bash-shell des Raspberry Pi vorzusehen. Die zweite und dritte Zeile dienen meiner Dokumentation. Das #-Zeichen wird am Anfang einer Kommentarzeile verwendet, die selbst nicht ausgeführt wird.

```
#!/bin/bash
```

```
#Stand: 11. Aug. 2014
```

```
#Datei: /home/pi/CProgramme/atmel/attiny2313/t2313-flashen
```

Jetzt ist der Parameter zu überprüfen, damit nichts z.B. durch Tippfehler schief geht:

```
 $# ---> Anzahl der übergebenen Parameter muss 1 sein
```

```
 echo „...Text...“ ---> Textausgabe im Terminalfenster
```

```
 exit 1 ---> Scriptende mit Fehlercode 1
```

```
 if .... then .... fi ---> Bedingungsklausel
```

```
 # *****Parameter checken*****
```

```
 if [ $# != 1 ] ; then
```

```
 echo "---> DateinameQuelltext (ohne Dateityp .c) fehlt"
```

```
 exit 1
```

```
 fi
```

Mit dem Parameter `-e` wird die Existenz der Quellcodedatei im Projektordner gleichen Namens überprüft. `$1` ist der erste Übergabeparameter im Scriptaufruf.

```
if [ -e $1/$1.c ]; then
    echo "---> Quellcode $1/$1.c existiert"
else
    echo "---> Quellcode $1/$1.c existiert nicht"
    exit 1
fi
```

Jetzt erfolgt der Aufruf des Compilers, der ohne Fehler war, wenn  `$?`  Null ist. Im Script wird ein Befehl mit `$(Befehl)` ausgeführt. Der Parameter `t2313_1` wird zu `t2313_1/t2313_1.c` und `t2313_1/t2313_1.o` zusammengesetzt (deshalb meine Festlegungen, dass Projektordner und Sourcecode namensgleich sein sollen):

```
# *****Kompilieren*****

$(avr-gcc -mmcu=attiny2313 -Os -c $1/$1.c -o $1/$1.o)

if [ $? != 0 ]; then
    echo "---> Kompilierung nicht ok"
    exit 1
else
    echo "---> Kompilierung ok"
fi
```

Die nächsten Arbeitsschritte sind im Aufbau analog:

Die Erstellung der `*.elf` und `*.hex` Dateien ist in allen Fällen dieselbe:

```
$(avr-gcc $1/$1.o -o $1/$1.elf)
```

und

```
$(avr-objcopy -O ihex -j .text -j .data $1/$1.elf $1/$1.hex)
```

Das Beiwerk der IF-Abfragen kannst du als Copy/Paste erstellen (habe ich auch so gemacht).

```
# *****Programmieren Flash*****
```

```
$(avrdude -p t2313 -c avrisp -U flash:w:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

usw.

Das Script habe ich für das Verständnis in wesentlichen Teilen im nächsten Abschnitt abgedruckt und im darauf Folgenden auch das Terminal-Ausgabeprotokoll, das du im Terminal nach dem Scriptaufruf siehst. Wenn es bei dir anders aussieht, sollte der Grund auf diese Weise auffindbar sein.

Im Wesentlichen also Schreibaarbeit. Eleganter sollte es mit intelligenteren Scripts, d.h. Abfrage des Prozessors, des Programmers usw. im Dialog erfolgen können. Mir erschien es aber so „Anfänger-sicherer“, für jeden Prozessortyp Scripts einzeln auszutesten und zu vervielfältigen. Copy und Paste!

## Zusammenfassung der genutzten Aufrufe mit Parametern für die vier von mir genutzten Prozessoren:

**# \*\*\*\*\*Kompilieren\*\*\*\*\***

```
$(avr-gcc -mmcu=attiny2313 -Os -c $1/$1.c -o $1/$1.o)
```

```
$(avr-gcc -mmcu=attiny13 -Os -c $1/$1.c -o $1/$1.o)
```

```
$(avr-gcc -mmcu=atmega168 -Os -c $1/$1.c -o $1/$1.o)
```

```
echo " avr-gcc -mmcu=atmega8 -Os -c $1/$1.c -o $1/$1.o"
```

Die Erstellung der \*.elf und \*.hex Dateien ist in allen Fällen dieselbe:

```
$(avr-gcc $1/$1.o -o $1/$1.elf)
```

und

```
$(avr-objcopy -O ihex -j .text -j .data $1/$1.elf $1/$1.hex)
```

**# \*\*\*\*\*Programmieren Flash\*\*\*\*\***

```
$(avrdude -p t2313 -c avrisp -U flash:w:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p t13 -c avrisp -U flash:w:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p m168 -c avrisp -U flash:w:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p m8 -c avrisp -U flash:w:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

**# \*\*\*\*\*Auslesen Flash\*\*\*\*\***

```
$(avrdude -p t2313 -c avrisp -U flash:r:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p t13 -c avrisp -U flash:r:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p m168 -c avrisp -U flash:r:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p m8 -c avrisp -U flash:r:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

**# \*\*\*\*\*Verifizieren Flash\*\*\*\*\***

```
$(avrdude -p t2313 -c avrisp -U flash:v:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p t13 -c avrisp -U flash:v:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p m168 -c avrisp -U flash:v:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

```
$(avrdude -p m8 -c avrisp -U flash:v:$1/$1.hex -P /dev/ttyUSB0 -b 19200)
```

Die Ausgabe zur Ansicht z.B. von t2313\_1.hex im Terminal erfolgt z.B. durch:

```
$(more | t2313_1.hex)
```

## t2313-flaschen

Als konkretes Beispiel für die Programmierung des Flashspeichers eines ATtiny2313 drucke ich das script t2313-flaschen aus, wobei das C-Projekt den Namen t2313\_1 hat und das C-Quellprogramm nach meinen Festlegungen den Namen t2313\_1.c haben muß:

```
#!/bin/bash

#Stand: 11. Aug. 2014

#Scriptdatei: /home/pi/CProgramme/atmel/t2313/t2313-flaschen

echo "Je nach Prozessortyp wird unterhalb /home/pi/CProgramme/atmel ein"
echo "Verzeichnis je µProzessor angelegt, z.B. attiny2313/"
echo "Dort sind allgemeine Scripte für diesen µP gespeichert"
echo "Jeweils ein Projektordner je zu brennendem *.hex ist darin anzulegen."
echo "Dieser enthält die gleichlautenden Dateien *.c, *.obj, usw."
echo "Also z.B. t2313_1 als Projekt-Pfad und darin t2313_1.c, t2313_1.hex, usw."
echo ""
echo "Scriptaufruf: <t2313-flaschen DateinameC_QuellDatei> ohne Typ .c"
echo "der Pfad $1 kann so automatisch hinzugefügt werden, z.B. $1/$1.c"
echo ""
```

```
# Parameter $1 darf kein Minuszeichen enthalten!

# <t2313> in avrdude sowie <-mmcu=attiny2313> in avr-gcc

# *****Parameter checken*****

if [ $# != 1 ] ; then

echo "---> Parameter -DateinameQuelltext ohne Dateityp .c fehlt"

exit 1

fi

if [ -e $1/$1.c ]; then

echo "---> Quellcode $1/$1.c existiert"

else

echo "---> Quellcode $1/$1.c existiert nicht"

exit 1

fi

# *****Kompilieren*****

echo "---> Kompilierung $1/$1.c startet:"

echo " avr-gcc -mmcu=attiny2313 -Os -c $1/$1.c -o $1/$1.o"

$(avr-gcc -mmcu=attiny2313 -Os -c $1/$1.c -o $1/$1.o)

if [ $? != 0 ] ; then

echo "---> Kompilierung nicht ok"

exit 1
```

```

else

echo "---> Kompilierung ok"

fi

# *****Linkdatei erstellen*****

echo "---> Erstellen $1/$1.elf beginnt"

$(avr-gcc $1/$1.o -o $1/$1.elf)

if [ $? != 0 ] ; then

echo "---> Linken nicht ok"

exit 1

else

echo "---> $1/$1.elf ok"

fi

# *****hex-Datei erstellen*****

echo "---> Erstellen $1/$1.hex startet"

$(avr-objcopy -O ihex -j .text -j .data $1/$1.elf $1/$1.hex)

if [ $? != 0 ] ; then

echo "---> $1/$1.hex nicht ok"

exit 1

else

echo "---> $1/$1.hex ok"

fi

```

```

# *****attiny2313 programmieren*****

echo "---> $1/$1.hex an Prozessor senden"

$(avrdude -p t2313 -c avrisp -U flash:w:$1/$1.hex -P /dev/ttyUSB0 -b 19200)

if [ $? = 0 ] ; then

echo "---> Programmierung ok"

else

echo "---> Programmierung nicht ok"

exit 1

fi

# *****Verifizierung*****

echo "---> start Verifizierung mit:"

echo "---> $(avrdude -p t2313 -c avrisp -U flash:v:$1/$1.hex -P /dev/ttyUSB0 -b 19200)"

$(avrdude -p t2313 -c avrisp -U flash:v:$1/$1.hex -P /dev/ttyUSB0 -b 19200)

if [ $? = 0 ] ; then

usw...

```

## AUSGABEPROTOKOLL NACH AUFRUF IM TERMINAL

### ./t2313-flaschen t2313\_1

Je nach Prozessortyp wird unterhalb /home/pi/CProgramme/atmel ein

Verzeichnis je  $\mu$ Prozessor angelegt, z.B. attiny2313/

Dort sind allgemeine Scripte für diesen  $\mu$ P gespeichert

Jeweils ein Projektordner je zu brennendem \*.hex ist darin anzulegen.

Dieser enthält die gleichlautenden Dateien \*.c, \*.obj, \*.elf, usw.

Also z.B. t2313\_1 als Projekt-Pfad und darin t2313\_1.c, t2313\_1.hex, usw.

Scriptaufruf: <t2313-flaschen DateinameC\_QuellDatei> ohne Typ .c

der Pfad t2313\_1 kann so automatisch hinzugefügt werden, z.B.

t2313\_1/t2313\_1.c

---> Quellcode t2313\_1/t2313\_1.c existiert

---> Kompilierung t2313\_1/t2313\_1.c startet:

```
avr-gcc -mmcu=attiny2313 -Os -c t2313_1/t2313_1.c -o t2313_1/t2313_1.o
```

---> Kompilierung ok

---> Erstellen t2313\_1/t2313\_1.elf beginnt

---> t2313\_1/t2313\_1.elf ok

---> Erstellen t2313\_1/t2313\_1.hex startet

---> t2313\_1/t2313\_1.hex ok

---> t2313\_1/t2313\_1.hex an Prozessor senden

avrdude: AVR device initialized and ready to accept instructions

Reading |

##### | 100%

0.06s

avrdude: Device signature = 0x1e910a

avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed

To disable this feature, specify the -D option.

avrdude: erasing chip

avrdude: reading input file "t2313\_1/t2313\_1.hex"

avrdude: input file t2313\_1/t2313\_1.hex auto detected as Intel Hex

avrdude: writing flash (36 bytes):

Writing | #####

| 100% 0.15s

avrdude: 36 bytes of flash written

avrdude: verifying flash memory against t2313\_1/t2313\_1.hex:

avrdude: load data flash data from input file t2313\_1/t2313\_1.hex:

avrdude: input file t2313\_1/t2313\_1.hex auto detected as Intel Hex

avrdude: input file t2313\_1/t2313\_1.hex contains 36 bytes

avrdude: reading on-chip flash data:

Reading |

##### | 100%

0.09s

```
avrdude: verifying ...

avrdude: 36 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done. Thank you.

---> Programmierung ok

---> start Verifizierung mit:

---> avrdude -p t2313 -c avrisp -U flash:v:t2313_1/t2313_1.hex -P /dev/ttyUSB0 -b
19200

avrdude: AVR device initialized and ready to accept instructions

Reading |
##### | 100%
0.06s

avrdude: Device signature = 0x1e910a

avrdude: verifying flash memory against t2313_1/t2313_1.hex:

avrdude: load data flash data from input file t2313_1/t2313_1.hex:

avrdude: input file t2313_1/t2313_1.hex auto detected as Intel Hex

avrdude: input file t2313_1/t2313_1.hex contains 36 bytes

avrdude: reading on-chip flash data:

Reading |
##### | 100%
0.09s

avrdude: verifying ...

avrdude: 36 bytes of flash verified

avrdude: safemode: Fuses OK
```

avrdude done. Thank you.

Verifizierung ok

Inhalt t2313\_1/t2313\_1.hex:

:1000000080E187BB88BBE0EAF3EC3197F1F718BADF

:10001000F1E02DE136E0F15020403040E1F700C042

:040020000000FoCF1D

:00000001FF

### **Anmerkung:**

Falls du das Terminalprotokoll bei der Abarbeitung in einer Textdatei ablegen möchtest, so ist es mit dem > Zeichen wie folgt möglich:

```
./t2313-flaschen t2313_1 > protokoll.txt
```

Die Ausgaben werden in die Datei protokoll.txt umgeleitet, die du dann beliebig verwenden kannst.



## ABSCHNITT 6

### OHNE TERMINALFENSTER GEHT ES AUCH

### DIE ARDUINO IDE SELBST ALS ENTWICKLUNGSWERKZEUG FÜR ANDERE AVR-TYPEN VERWENDEN

#### INHALT DIESES KURZEN AUSBLICKES

1. Die Arduino IDE selbst wird zur Programmierung, Compilierung und Hochladen der HEX-Datei von Projekten (mit dem selben Prozessor - bei mir der ATmega 168) mittels ArduinoISP benutzt
2. Zusätzliche Prozessoren z.B. der ATtiny Chips über neue Borddefinitionen ansprechen. Dabei sind nicht alle Sprachmöglichkeiten der IDE verwendbar (z.B. geringere Pin-Anzahl usw.)
3. Vorteil: du musst dich nicht mit avr-gcc, avrdude und scripts auseinandersetzen. Es bleibt fast bei der gewohnten Vorgehensweise
4. Nachteil: du musst dich weniger mit den unter der IDE liegenden Prozessen befassen. Bei gewolltem „black box“ Verständnis ist das sogar ein Vorteil

Derselbe Prozessor, wie im genutzten Arduino lässt sich mit dem Arduino Bord und der IDE selbst programmieren

<http://arduino.cc/en/Tutorial/ArduinoISP> und

<http://arduino.cc/en/Tutorial/ArduinoToBreadboard>

Wenn du bei Suchmaschinen nach <ArduinoISP ATtiny> nachschlägst, gibt es eine Reihe von Fundstellen, die beschreiben, wie du auch z.B. den ATtiny 2313 auf diese Weise programmieren kannst. Mir haben z.B. diese Fundstellen gefallen und weitergeholfen:

<http://technikhobby.blogspot.de/2013/01/die-kleinen-attiny-chips-im-arduino-ide.html>

<http://technikhobby.blogspot.de/2013/01/arduinoisp-avr-chips-programmieren.html>

Hier wird auch der Ersatz vom 10 Microfarad durch 110 Ohm gezeigt (verhindern des „auto reset“ auf meinem shield).

Die Methode, z.B. ATtiny-Prozessoren mit ArduinoISP und der Arduino IDE anzusprechen, besteht in der Hinzufügung weiterer Bords, die eben die anderen Prozessoren beinhalten. Wie und wo Borddefinitionen aus dem Internet geladen werden und in der IDE verfügbar gemacht werden, ist im Einzelnen genau erklärt.

Ich habe Versuche mit dem 2313 gemacht und muss sagen, dass es mit Einschränkungen (Pins, Funktionen) auch funktioniert. Meine Hardware, wie im nächsten Kapitel beschrieben, findet dabei genauso Verwendung.

## ABSCHNITT 7

### DER VOLLSTÄNDIGKEIT WEGEN:

#### DIE ENERGIA IDE ALS ENTWICKLUNGSWERKZEUG FÜR MSP430 PROZESSOREN VERWENDEN

##### INHALT DIESES KURZEN AUSBLICKES

1. Die Energia IDE wurde 2013 entwickelt. Der Prozessor MSP430 der Firma Texas Instruments ist energiesparend, schnell und mit einer 16-Bit Architektur versehen
2. Die Entwicklung mit der Energia IDE ist bewusst fast identisch zu AVR IDE, so sind zum Teil gleiche Programme ablauffähig
3. Vorteil: du musst dich nicht mit neuen Entwicklungswerkzeugen für den ganz anderen Prozessortyp befassen
4. Beispiele sind wie beim Arduino Bord für das Starter Kit vorhanden. Das Starterkit ist preiswerter, der Prozessorchip lässt sich analog zum AVR ohne Starterkit betreiben!

Die RISC Architektur erlaubt Rechengeschwindigkeiten und Energieeffizienz zu optimieren, wie sie im professionellen Umfeld gefordert sind. Der Prozessor lässt sich „schlafend stellen“ mit minimalstem Stromverbrauch. „Das MSP430 Mikrocontroller Buch“ mit ISBN 978-3-89576-236-9 von Marian Walter und Stefan Tappertshofen beschreibt die vielfältigen Möglichkeiten der MSP430-Familie. Es existiert analog zum Arduino ein Stück Hardware von TI, das über USB von der IDE zur Programmentwicklung dient: das „launch pad“.



The screenshot shows the Energia IDE interface. The top window, titled "sketch\_nov25a | Energia 0101E0010", displays a C++ sketch with the following code:

```
void setup()
{
  // put your setup code here, to run once:
}

void loop()
{
  // put your main code here, to run repeatedly:
}
```

The bottom window shows a terminal output with a red header bar and a black body. The text in the terminal reads: "4 LaunchPad w/ msp430g2553 (16MHz) on /dev/tty.uart-11FF427A4D192D16".

## Einige Fundstellen:

[https://de.wikipedia.org/wiki/TI\\_MSP430](https://de.wikipedia.org/wiki/TI_MSP430)

<http://www.ti.com/tool/msp430-gcc-opensource>

Die Prozessor-Architektur ist eine auf einem internen Bus basierende

<https://de.wikipedia.org/wiki/Von-Neumann-Architektur>

Die Open Source Entwicklungsumgebung:

Während die Original-IDE von TI auf Windows zugeschnitten ist (war?), ist energia auch auf Linux und OSX portiert:

<http://energia.nu>

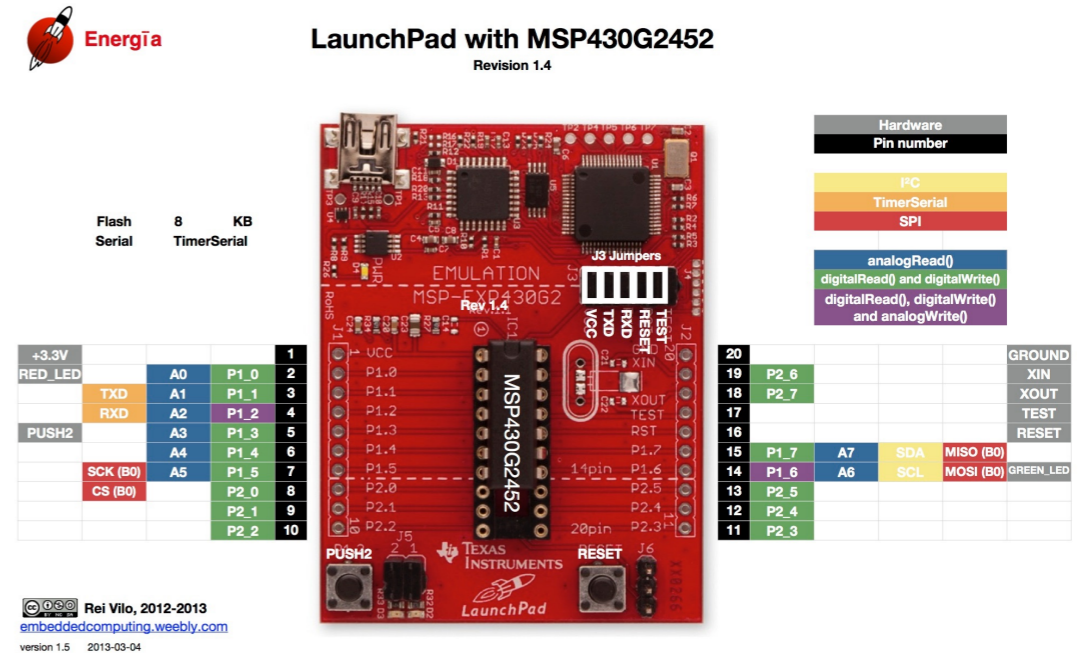
Eine Installation auf Mac OS ist bei mir etwas „speziell“ gewesen, wie es oft mit nicht über den app-store geladenen Programmen vorkommt:

<http://energia.nu/guide/>

Ich kann auch nicht sagen, ob es sich um eine ausgereifte und sichere Software handelt. Das Aussehen und die Beispiele sind für anfängliches Kennenlernen der MSP-Familie und Prototyping aber sehr gut geeignet. Mir ist dieser interessante Prozessor, der sogar Operationsverstärker integriert hat und damit auch in die Signalverarbeitung reicht, bei einem Workshop von Gerrit (Amateurfunk-Rufzeichen DL9GFA) in Hannover praktisch begegnet. Die Original-Entwicklungsumgebung von TI war auf meinem einzigen Windows-Rechner, ei-

nem Netbook, nicht wirklich gut nutzbar. Umso erfreuter war ich jetzt über die Fundstelle der energia IDE im Internet.

Das Pin-Mapping einiger Launch-Pads ist z.B. hier zu finden:



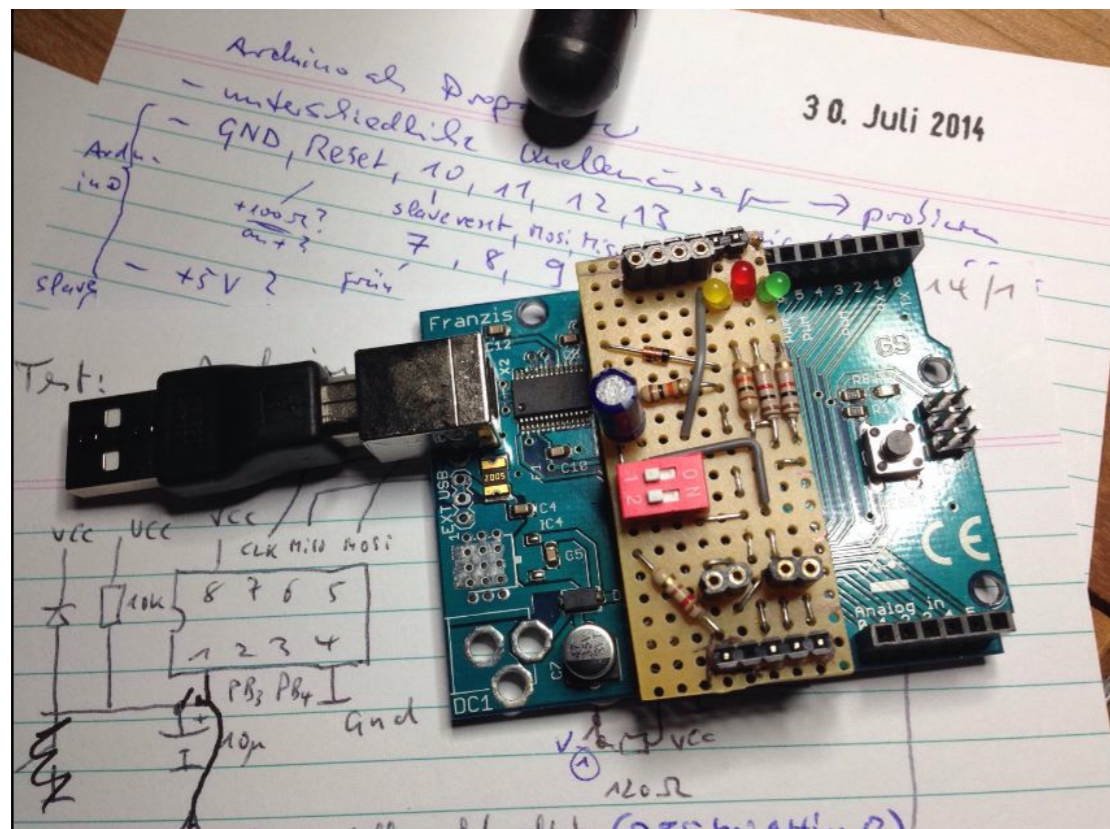
[http://energia.nu/Guide\\_MSP430LaunchPad.html](http://energia.nu/Guide_MSP430LaunchPad.html)

Dies mag als Anreiz genügen, weitere Quellen zu finden, der freie Platz ist für das Aufschreiben gerade recht ;-)

# Hardware

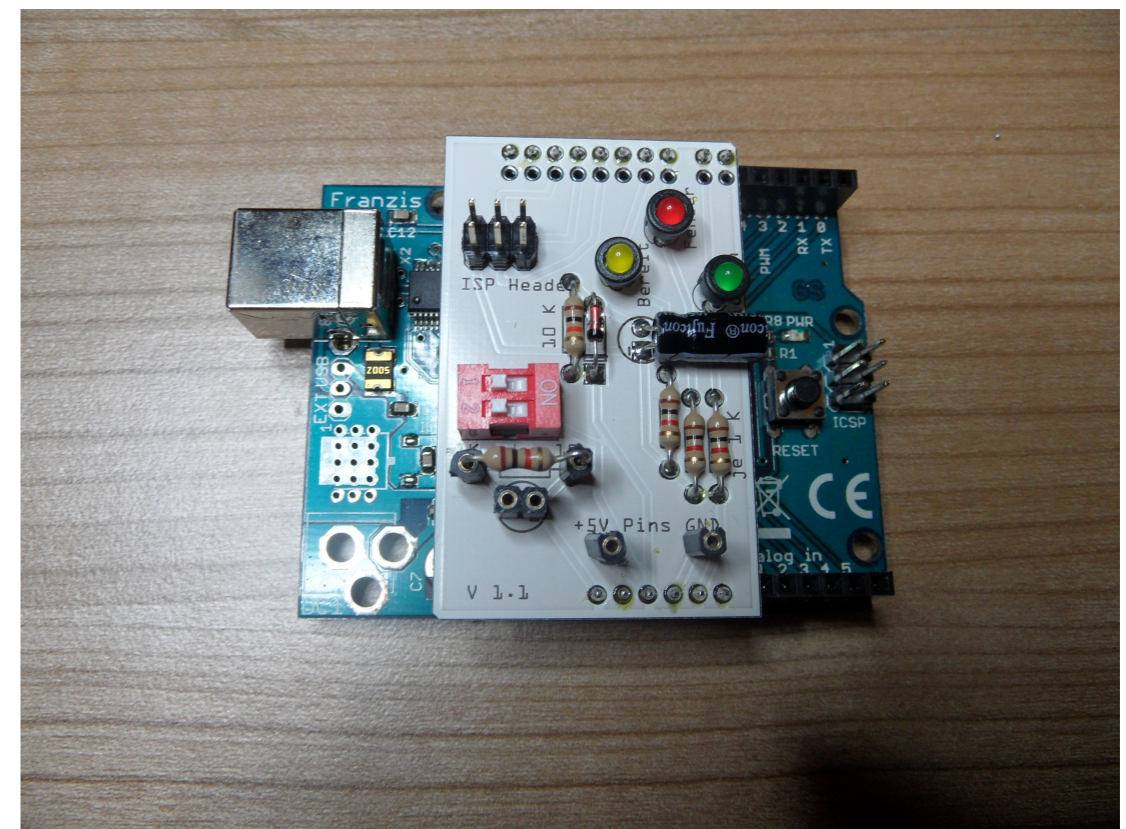
Es ging mir darum, vorhandene Hardware (Raspberry Pi und Arduino) zu nutzen und gegebenenfalls um kleinere Basteleien zu ergänzen. Nach der Internetrecherche war nicht klar, wie das Ergebnis aussehen sollte.

Die Verwendung von fertigen kommerziellen USB-Programmern ist sicher schneller erfolgreich. Aber ich wollte das Ganze auch einigermaßen verstanden haben.



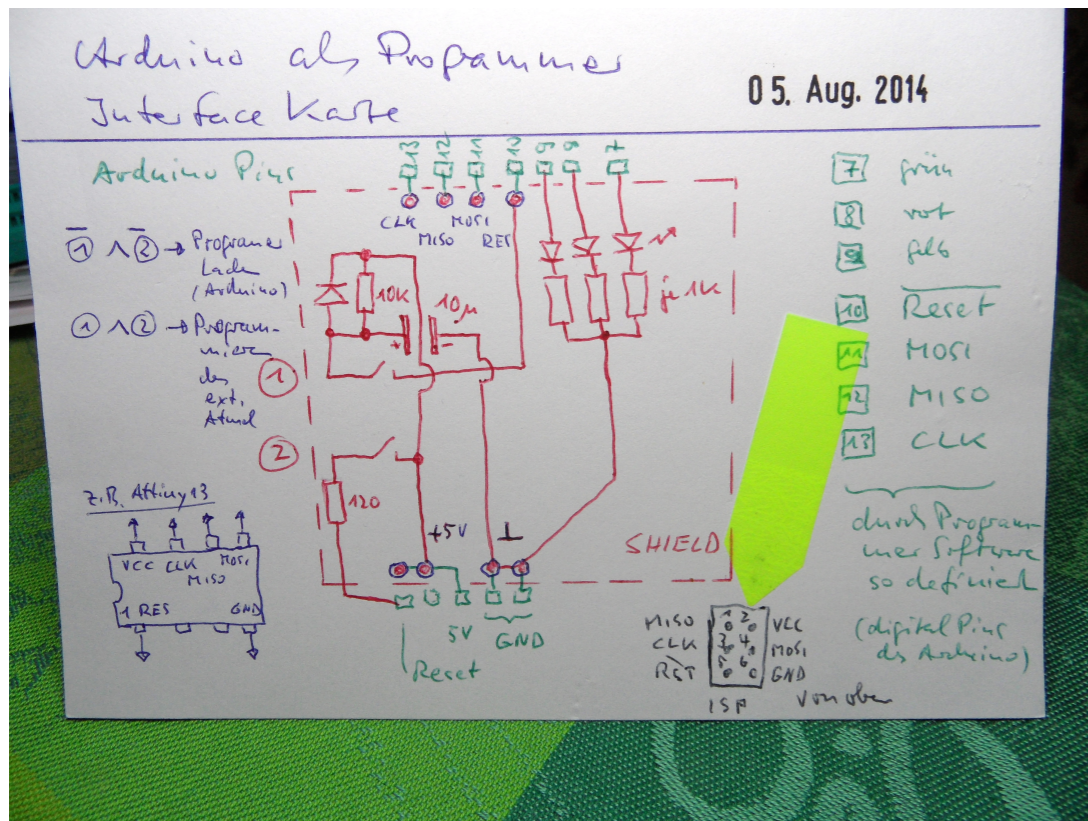
Bis die diversen Fundstellen verarbeitet waren und damit praktische Tests durch fliegende Verdrahtung begonnen werden konnten, waren etliche Fehlversuche auszuwerten. Dass es hinterher meinem Empfinden nach so einfach aussah, hat mich veranlasst, es zu dokumentieren. Den Aufwand dafür habe ich total unterschätzt ;-)

Der Prototyp auf Lochraster im Bild und die überarbeitete Version auf doppelseitiger Platine (PCB):



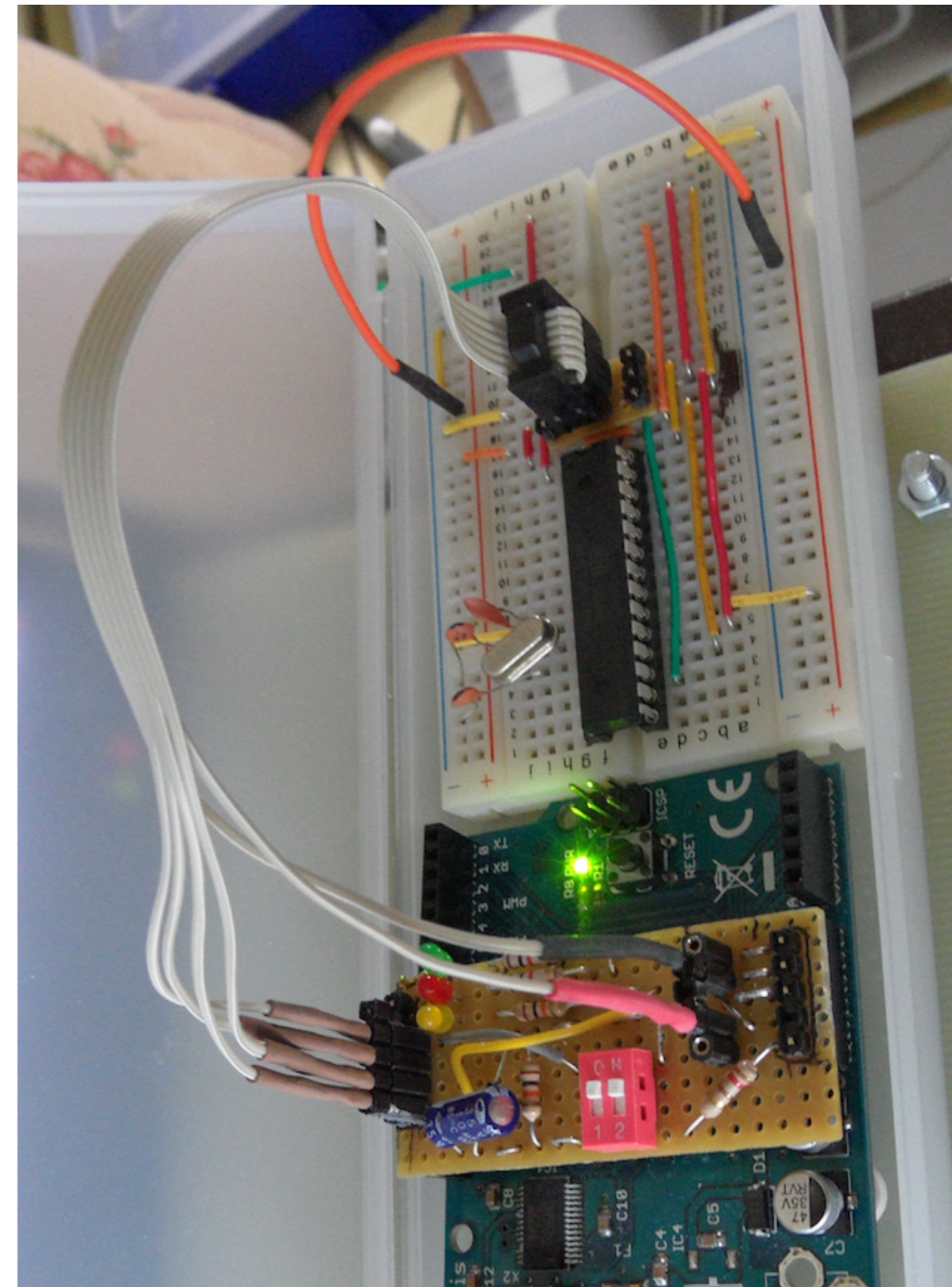
# Shield Prototyp

Die beschriebene Zusatzbeschaltung kann über Schalter ausgeschaltet werden - dann ist Arduino kein ISP mehr.



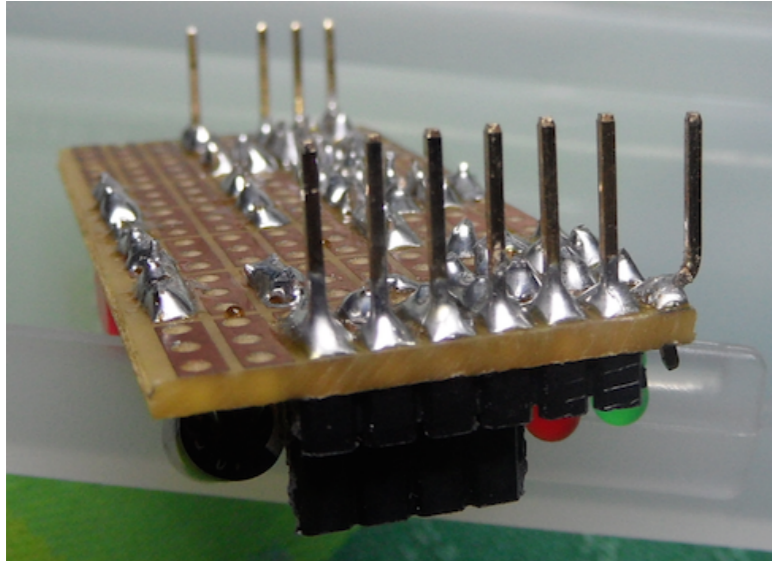
Dadurch und mit dem Übergang auf ein Programmierkabel wird das sonst entstehende Kabelgewirr vermieden und es sieht auch noch sauber aus ;-).

s1 und s2 offen: ArduinoISP-Sketch Compilieren und Laden  
 s1 und s2 danach schließen: Das Shield ist bereit



Das Bild zeigt das aufgesetzte Mini-Shield und Programmierkabel mit 6-poligem ISP-Verbinder zum Steckbrett, in dem ein ATmega8 eingesetzt ist.

Eine Besonderheit ist der nicht zu erwartende Pin-Abstand zwischen Pin 7 und 8. Hier wird der rechte Pin des Shields „verbogen“:



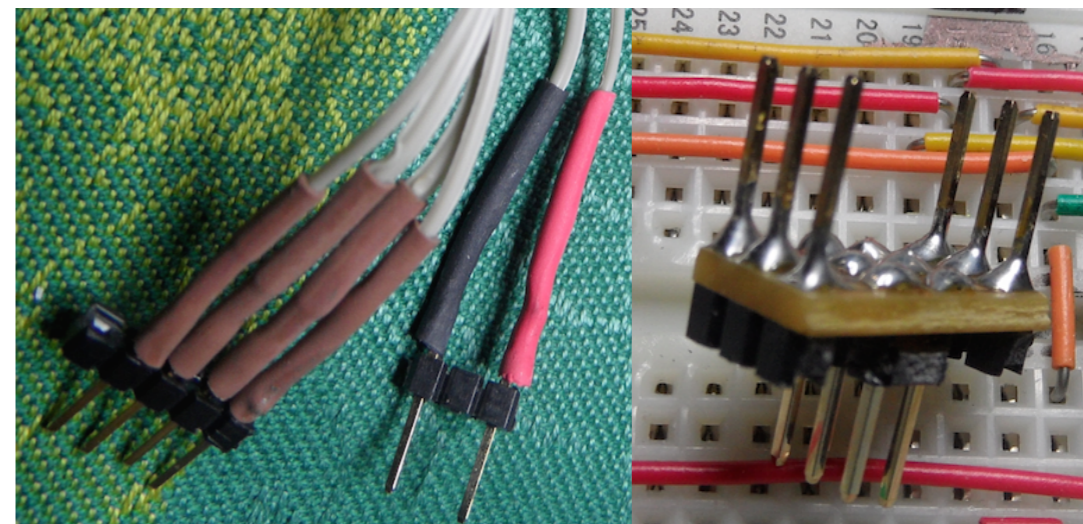
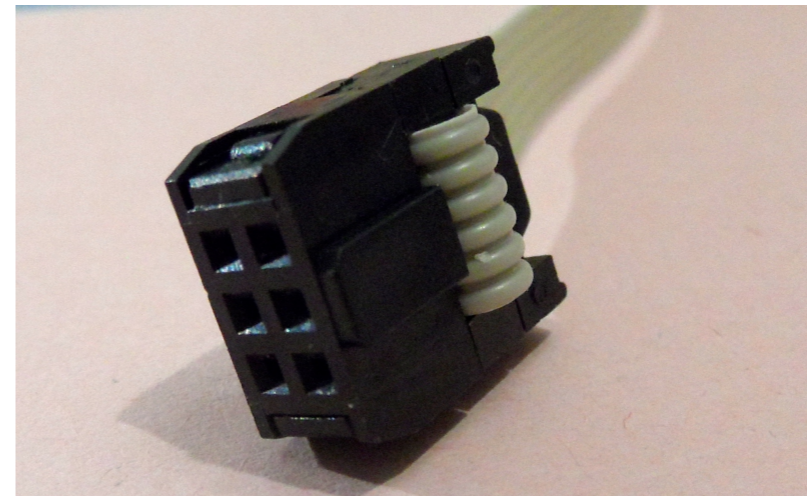
Möglich wäre auch eine Leerfassung für ATtiny13/2313 auf dem Shield selbst. Dann wäre die Nutzung für andere Prozessoren eingeschränkt - im Prinzip kann ArduinoAVR-ISP sehr viele unterschiedliche AVR-Prozessoren ansprechen. Ein Flachbandkabel kann speziell den Übergang vom Shield auf eine Norm ISP Beschaltung (Stecker/Buchse) ermöglichen. Dadurch ist das Shield universeller einsetzbar.

Jeweils von links: Im Hintergrund sind die Shield-Pins „Reset Fixierung“, +5V und zweimal GND zu sehen. Vorne CLK, MISO, MOSI, Reset, gelb, rot, grüne Dioden. Verkabelt werden aber nur die im ISP vorgesehenen sechs Anschlüsse:

+5V (VCC), GND (Ground), CLK (Clock), MISO (MasterInSlaveOut), MOSI (MasterOutSlaveIn) und RST (Reset).

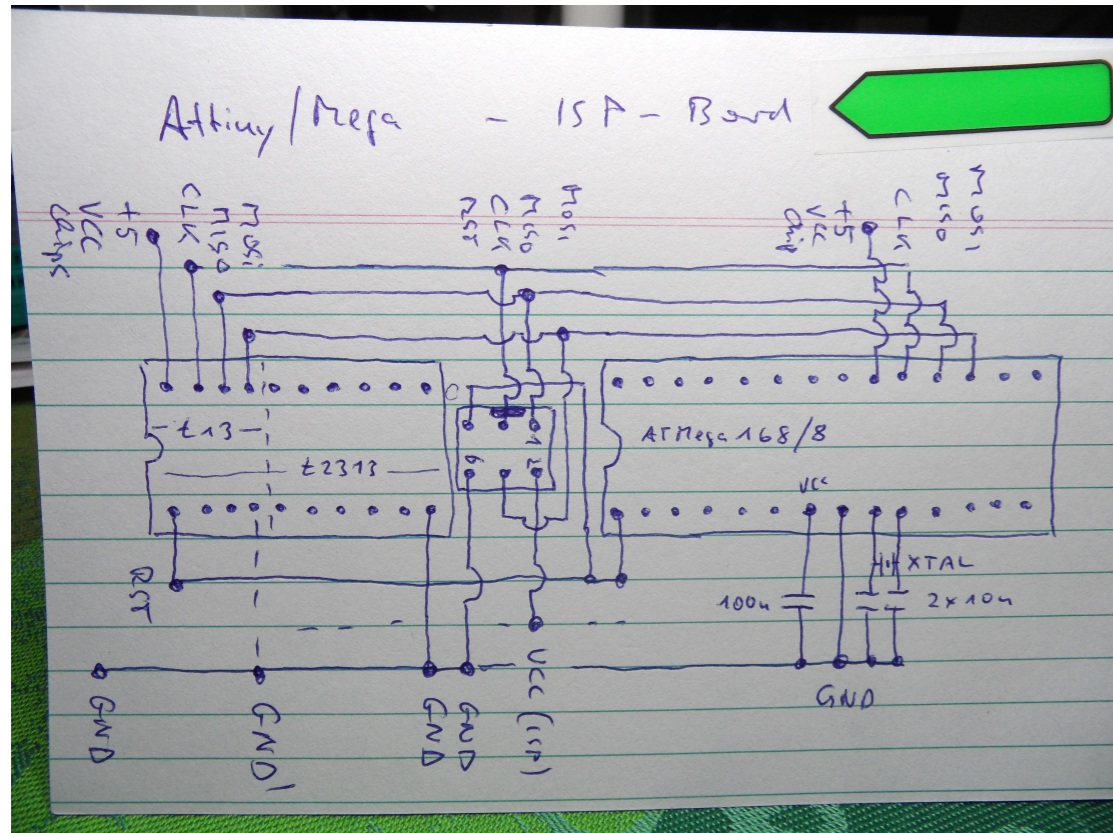
Das Flachband wird auf der einen Seite in der 6-poligen Buchse aufgeklemmt und am anderen Ende sinngemäß auf Stecker gelötet und mit Schrumpfschlauch isoliert. Damit ist das Kabel universell nutzbar.

Ich habe auch einen Adapter für ein Steckbrett erstellt, um ISP (untere Stifte) zu verbreitern (obere Stifte) für die Aufnahme im Steckbrett.

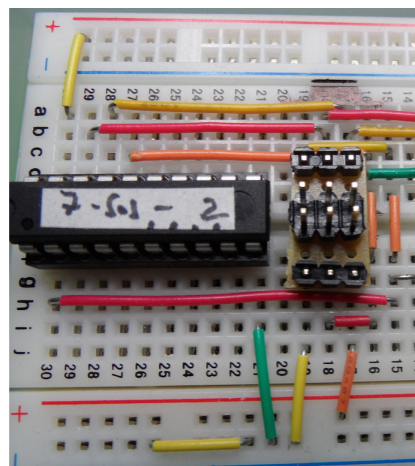


## STECKBRETT UND WEITERE SCHRITTE

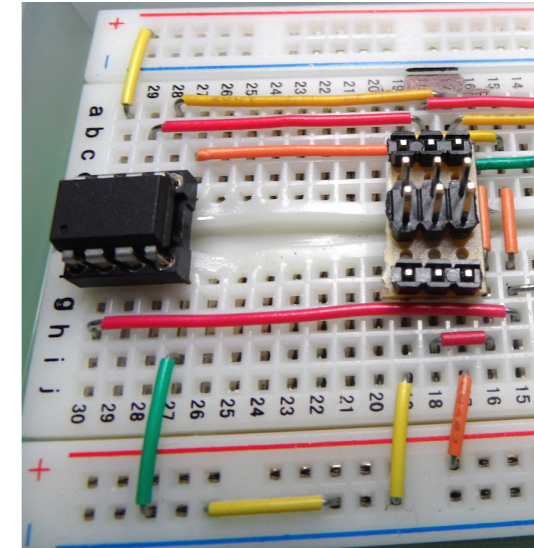
Die teilweise identische Pinbelegung der ATtiny 2313 und 13 sowie die identische Pinbelegung der ATmega 168 und 8 erlaubt die Nutzung von ISP auf einem kleinen Steckbrett:



ATtiny2313



ATtiny13

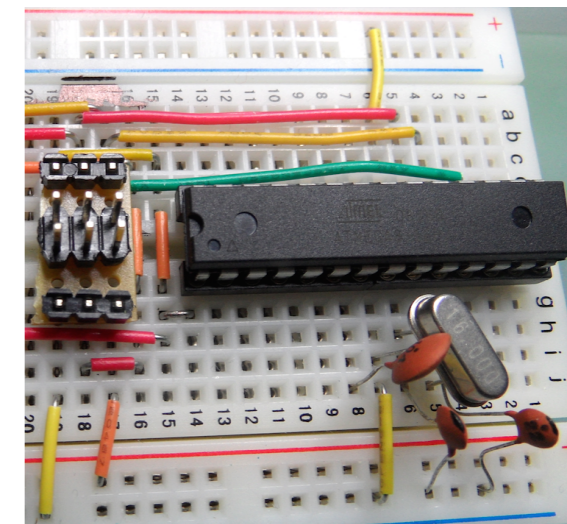


GND (grüner Draht) wurde verlegt, mehr nicht.

Die „Kunststoffschmelze“ verursachte ein kurzzeitig falsch herum eingesteckter ATtiny2313 (VCC und GND vertauscht)! Der Chip war nicht zerstört - hält also eine Menge aus ;-). Das Steckbrett funktioniert auch danach noch für den 2313 fehlerfrei.

ATmega168 bzw.

ATmega8

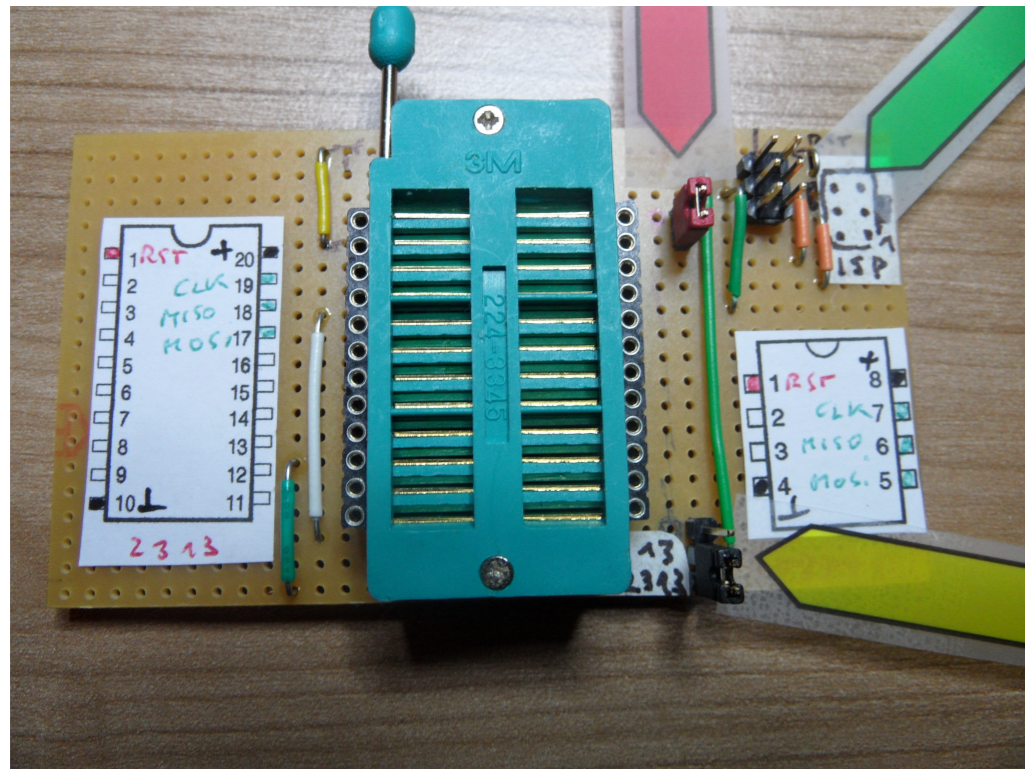


Nach Datenblatt ist es sinnvoll, einen externen 16MHZ Quarz bei diesen Prozessortypen auf dem Steckbrett einzusetzen.

## WEITERE SCHRITTE

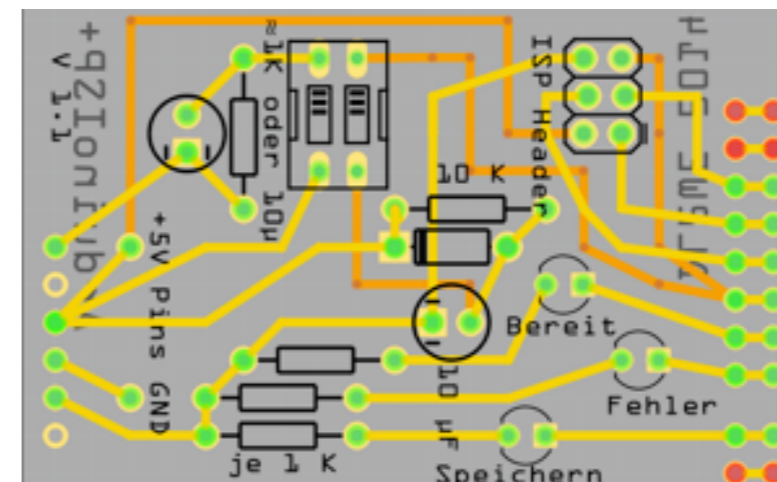
### NACH DEM PROTOTYPING

1. Meine Absicht ist, die ATtiny-Typen häufiger einzusetzen. Daher habe ich einen Nulldruck-Sockel auf einer Rasterplatine fest verdrahtet. Beide Typen sollen oben eingesetzt werden, d.h. die Masse (0 Volt) muss veränderbar sein. Dies wird mit der Steckbrücke (gelber Pfeil) erreicht (13 oben, 2313 unten überbrücken). Der rote Pfeil deutet auf eine rote Steckbrücke, die die + 5 Volt aus dem ISP-Anschluss (grün) trennen kann. Da alle Pins parallel auf Buchsenleisten geführt sind, lassen sich auch Tests gleich mit dieser Anordnung fliegend verdrahten. Die Handhabung dieser Anordnung hat sich bei mir



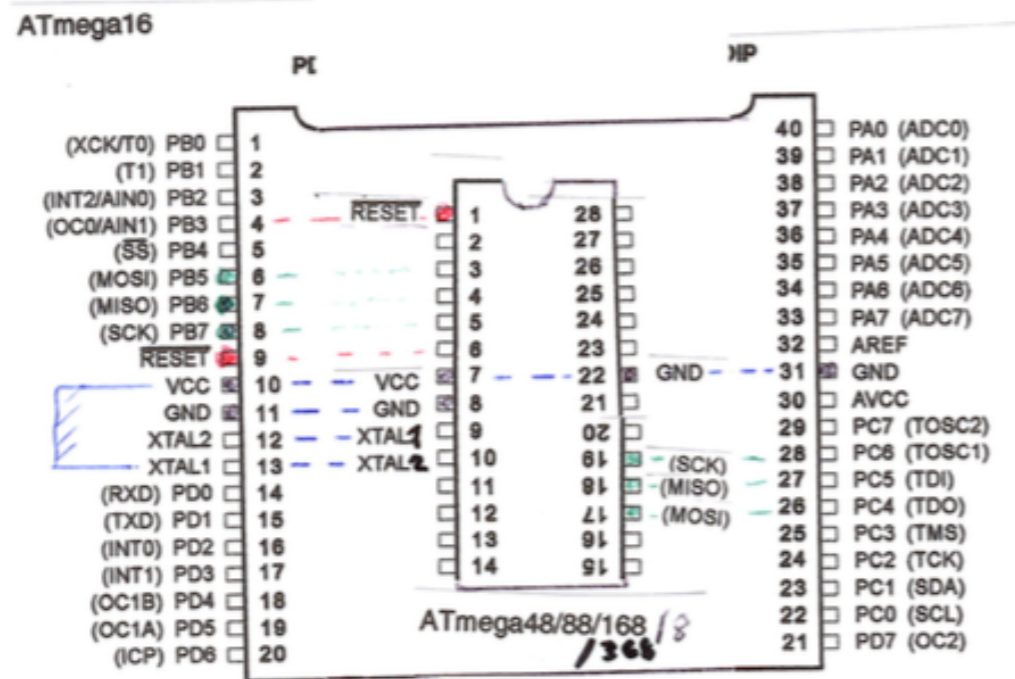
bewährt. Nulldrucksockel sind nicht besonders preiswert, aber schonen die Pins.

2. Überarbeitung des Shield- und Steckbrett-Layouts. <http://fritzing.org> ist die Adresse für ein Dokumentations- und Layoutprogramm (PCB-Design), das die Fachhochschule Potsdam initiierte. Es ist m.E. einfach zu nutzen, eine reichhaltige Bibliothek für Bauteile ist enthalten. Insbesondere berücksichtigt das Programm Arduino und Steckbrett für die Schaltungsdarstellung. Mittels Fritzing entstand ein Shield mit normgerechtem ISP-Anschluß. Das zweiseitige Layout vermeidet Drahtbrücken. Die Platine habe ich von dort direkt in Auftrag geben können (gelbe Bahnen sind oben, orange Bahnen auf der Unterseite). Neben den DIP-Schaltern findest du zum Experimentieren Pins (gut 100 Ohm oder 10 Mikrofarad für das Verhindern von „auto resets“ des Arduino beim Programmieren). Nach heutigem Stand empfehle ich, die 10 Mikrofarad der bereits zitierten Dokumentation (<http://arduino.cc/en/Tutorial/ArduinoISP>) einzusetzen.

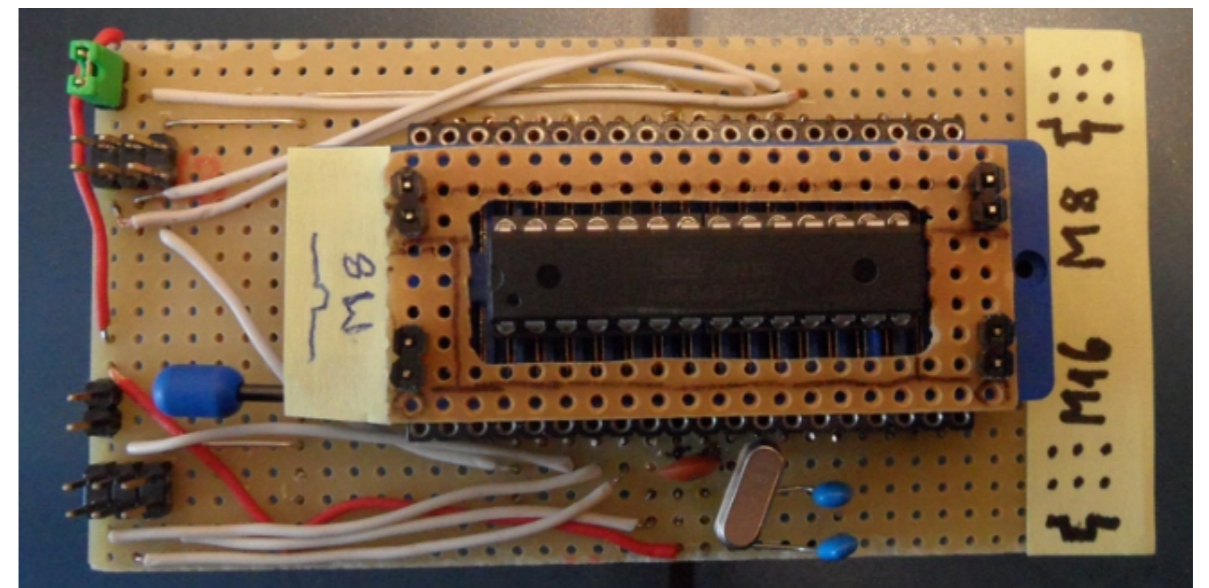
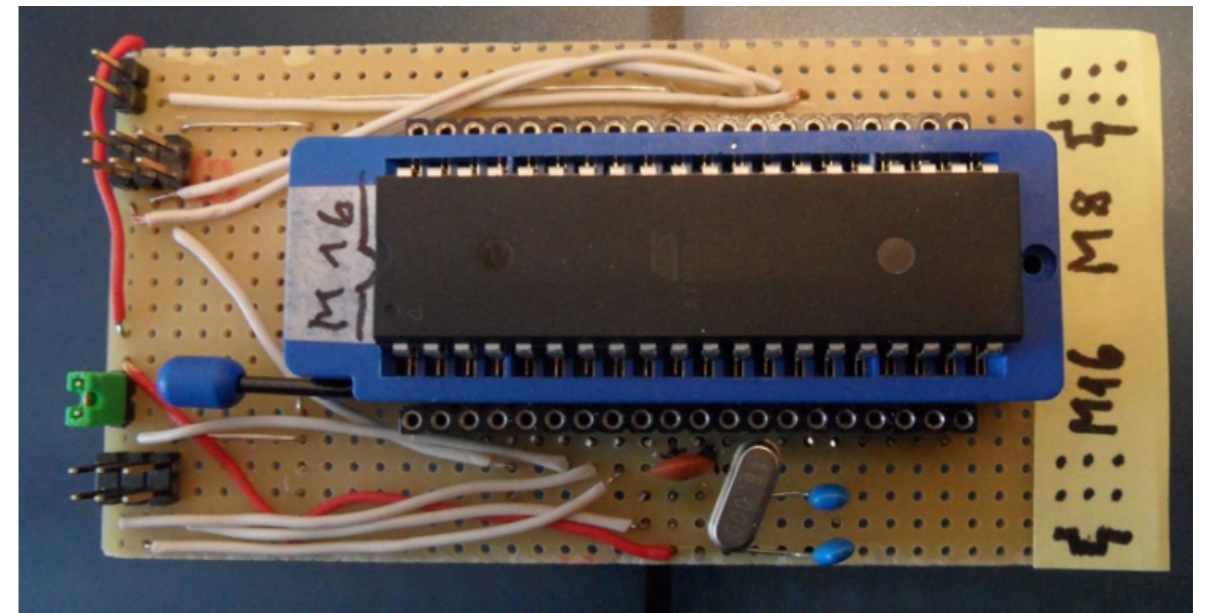




3. Für die Programmierung der ATmega-Reihe kann derselbe Ansatz für einen Nulldrucksockel auf Rasterplatine zum Erfolg führen: die ähnliche Pin-Belegung zeigt es, ein Sockel mit 40 Pins ist dafür vorzusehen. Der ATmega368 z.B. beginnt fast pingleich durch eine Verschiebung, wie angezeigt. So können VCC, GND, XTAL1 und XTAL2 mit derselben Verdrahtung genutzt werden. Es bietet sich mir an, zwei ISP-Zugänge mit den Pins für Reset (rot), SCK, MISO und MOSI (jeweils grün) einzurichten. Ebenfalls möchte ich die VCC aus den ISP-Header mit einer Steckbrücke trennen können. Die bewährte Stiftleiste für die herausgeführten Prozessor-Pins soll ebenfalls vorhanden sein.



4. Folgende Bilder zeigen die Realisierung. Eine Hardware-Schablone sorgt für die korrekte Verschiebung des Prozessors. Die grüne Steckbrücke wird zur Trennung bzw. korrekten VCC-Verbindung genutzt.



# Impressum und Anhang

Buchtitel: **Arduino als AVR-ISP am Raspberry Pi**

Buchtyp: **PDF-Format**

ISBN: **keine**      Version: **1.7**      (1.0 am 10.März 2015)

Verfasser: **DL2WE**      e-mail: [dl2we@darc.de](mailto:dl2we@darc.de)

Veröffentlicht: 10. März 2015 auf <http://dl2we.darc.de>

Nutzungsrechte:

**Übernahme von Inhalten bei Quellenangabe dieser Ausarbeitung, mit der Bitte einer Information an den Verfasser.**

Namensrechte:

**Namen und Marken werden ohne Gewährleistung der freien Verwendbarkeit benutzt, da der Inhalt zu reinen Lehrzwecken nichtkommerzieller Art gedacht ist.**

Haftung und Gewähr:

**Es wird keine Haftung und Gewähr für die Funktion und Richtigkeit der Inhalte übernommen.**

Urheberrechte Dritter:

**Wegen der frei verfügbaren Quellen und Open Source Software sind Urheberrechtsverletzungen nicht zu vermuten. Falls es mir dennoch unbeabsichtigt passiert ist, bitte ich um eine e-mail, um diesen Fehler umgehend zu korrigieren.**

**Fundstellen im Internet, die das Thema weitgehend und grundsätzlich abdecken:**

Minicomputer Raspberry Pi: <http://RaspberryPi.org>

Datenblätter der Atmel-Prozessoren: <http://atmel.com>

„Using an Arduino as an AVR ISP (In-System Programmer)“: <http://arduino.cc/en/Tutorial/ArduinoISP>

Eine ergiebige Informationsquelle rund um (auch Atmel-) Prozessoren ist: <http://www.mikrocontroller.net>

Compiler avr-gcc: <http://www.nongnu.org/avr-libc/>

avrdude Transferprogramm und Tutorial:  
<http://savannah.nongnu.org/projects/avrdude>

<http://ladyada.net/learn/avr/avrdude.html>

Dokumentations- und Layoutprogramm: <http://fritzing.org>

Shell-Programmierung:

[http://de.wikibooks.org/wiki/Linux-Praxisbuch:\\_Shellprogrammierung](http://de.wikibooks.org/wiki/Linux-Praxisbuch:_Shellprogrammierung)

## ANHANG

Zusätzliche Informationen und Definitionen, Fundstellen

=====

**AVR**      [http://en.wikipedia.org/wiki/Atmel\\_AVR](http://en.wikipedia.org/wiki/Atmel_AVR)

„The creators of the AVR give no definitive answer as to what the term "AVR" stands for.[3] However, it is commonly accepted that AVR stands for Alf (Egil Bogen) and Vegard (Wollan)'s RISC processor.[5] Note that the use of "AVR" in this article generally refers to the 8-bit RISC line of Atmel AVR Microcontrollers.“

**Fritzing**      <http://en.wikipedia.org/wiki/Fritzing>

„Fritzing is an open source software initiative to support designers and artists ready to move from physical prototyping to actual product. It was developed at the University of Applied Sciences of Potsdam.“

„The software is created in the spirit of the Processing programming language and the Arduino microcontroller and allows a designer, artist, researcher, or hobbyist to document their Arduino-based prototype and create a PCB layout for manufacturing.“      <https://processing.org>

## GND

Im Allgemeinen die Bezeichnung für Ground (Masse), entspricht oft dem Minuspol der Stromversorgung (Null Volt). Siehe auch VCC.

## Harvard-Architektur, Von-Neumann-Architektur

<http://de.wikipedia.org/wiki/Harvard-Architektur>

<http://de.wikipedia.org/wiki/Von-Neumann-Architektur>

Die Besonderheit der Atmel „AVR Microcontroller“ liegt in der Harvard-Architektur: Datenspeicher und Programm-Anweisungsspeicher sind getrennt voneinander angelegt.

**Objektdatei**      [http://en.wikipedia.org/wiki/Object\\_code](http://en.wikipedia.org/wiki/Object_code)

„Object code, or sometimes an object module, is what a computer compiler produces. In a general sense object code is a sequence of statements or instructions in a computer language, usually a machine code language (i.e., 1's and 0's) ...

Object files can in turn be linked to form an executable file or library file. In order to be used, object code must either be placed in an executable file, a library file, or an object file.“

**Open Source** [http://en.wikipedia.org/wiki/Open\\_source](http://en.wikipedia.org/wiki/Open_source)

„Generally, open source refers to a computer program in which the source code is available to the general public for use and/or modification from its original design. Open-source code is typically a collaborative effort where programmers improve upon the source code and share the changes within the community so that other members can help improve it further.“

**PCB** [http://en.wikipedia.org/wiki/Printed\\_circuit\\_board](http://en.wikipedia.org/wiki/Printed_circuit_board)

### **Seriell RS 232**

<http://encyclopedia.thefreedictionary.com/rs232>

„An RS-232 serial port was once a standard feature of a personal computer, used for connections to modems, printers, mice, data storage, uninterruptible power supplies, and other peripheral devices. However, the low transmission speed, large voltage swing, and large standard connectors motivated development of the Universal Serial Bus, which has displaced RS-232 from most of its peripheral interface roles“.

### **Shell, Scriptsprache**

[http://en.wikipedia.org/wiki/Shell\\_script](http://en.wikipedia.org/wiki/Shell_script)

Kommandozeilen-Interpreter, der in einem Terminal arbeitet:

„A shell script is a computer program designed to be run by the Unix shell, a command line interpreter.[1] The various dialects of shell scripts are considered to be scripting languages.

Typical operations performed by shell scripts include file manipulation, program execution, and printing text.“

### **STK-500 Protokoll**

[http://en.wikipedia.org/wiki/Atmel\\_AVR](http://en.wikipedia.org/wiki/Atmel_AVR)

„The STK500 starter kit and development system features ISP and high voltage programming (HVP) for all AVR devices, either directly or through extension boards. The board is fitted with DIP sockets for all AVRs available in DIP packages.“

### **Universal Serial Bus (USB)**

<http://en.wikipedia.org/wiki/USB>

„is an industry standard developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication, and power supply between computers and electronic devices.

Released in January 1996, USB 1.0 specified data rates of 1.5 Mbit/s (Low Bandwidth or Low Speed) and 12 Mbit/s (Full Bandwidth or Full Speed). It did not allow for extension cables or pass-through monitors, due to timing and power limitations.

USB 2.0 was released in April 2000, adding a higher maximum signaling rate of 480 Mbit/s called High Speed, in addition to the USB 1.x Full Speed signaling rate of 12 Mbit/s.

USB 3.0 standard was released in November 2008, defining a new SuperSpeed mode. A USB 3.0 port, usually colored blue, is backward-compatible with USB 2.0 devices and cables.

The new SuperSpeed bus provides a fourth transfer mode at 5.0 Gbit/s (raw data rate), in addition to the modes supported by earlier versions.

The USB 3.1 standard increases the signaling rate to 10 Gbit/s, double that of USB 3.0, ...“

## USB-Hub

Ein HUB stellt mehrere einzelne USB-Anschlüsse über **einen** gemeinsamen USB-Anschluß am PC zur Verfügung. Bei einem passiven HUB teilen sich die Anschlüsse den verfügbaren Strom, dagegen hat bei einem aktiven HUB (mit eigener Stromversorgung) jeder Anschluß die volle elektrische Leistung wie ein Einzelanschluß.

**VCC** [http://en.wikipedia.org/wiki/IC\\_power-supply\\_pin](http://en.wikipedia.org/wiki/IC_power-supply_pin)

„Almost all integrated circuits (ICs) have at least two pins that connect to the power rails of the circuit in which they are installed. These are known as the power-supply pins. However, the labeling of the pins varies by IC family and manufacturer.“

Im Allgemeinen die Bezeichnung für den Pluspol der Stromversorgung (5 Volt beim Arduino). Allerdings sind auch 3,3 Volt gebräuchlich, die oft aus 5 Volt auf dem Prozessor-Bord gewonnen werden. Während einige Prozessoren für Betriebsbereiche der VCC „von/bis“ tolerant sind, wird der Prozessor auf dem **Raspberry Pi-Bord** bei mehr als 3,3 Volt z.B. an seinen externen Anschlüssen unweigerlich **zerstört**. Deshalb die Datenblätter beachten!

**DL2WE** [www.darc.de](http://www.darc.de)

Ist ein von der Bundesnetzagentur an Dr.-Ing. Werner Weingarten vergebenes Rufzeichen, das an der Teilnahme des Amateurfunk-Dienstes berechtigt. Dieses Hobby ermöglicht mir den Kontakt zu technik-begeisterten Menschen, die im gemeinnützigen Verein Deutscher Amateur-Radio-Club e.V. organisiert sind.

Stand: 28. August 2015    Version 1.7