**Diplomarbeit**

zum Thema

# Erhöhung der Kanalanzahl des Modular-EEG auf Mikrocontroller-Ebene

An der Fachhochschule Dortmund
im Fachbereich Informatik
erstellte Diplomarbeit
im
Studiengang Informatik
Vertiefungsrichtung Technische Informatik

zur Erlangung des Grades
Diplom-Informatiker (FH) von


Bastian Holtermann
geboren am 29.09.1979
(Matr.- Nr.: 7051425)


Betreuung: Prof. Dr. Engels

Dortmund, 27. März 2008

# Markenrechtlicher Hinweis

Die in dieser Arbeit wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenzeichen usw. können auch ohne besondere Kennzeichnung geschützte Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Sämtliche in dieser Arbeit abgedruckten Bildschirmabzüge unterliegen dem Urheberrecht (©) des jeweiligen Herstellers.

Das Wort Brockhaus ist für den Verlag Bibliographisches Institut & F.A. Brockhaus AG als Marke geschützt.

Atmel ist eine eingetragene Marke der Atmel Corporation

# Kurzfassung

In dieser Diplomarbeit wird das Modular-EEG-Gerät aus dem OpenEEG-Projekt von 6 auf 16 Kanäle erweitert. Diese Erweiterung wird durch die Verwendung eines ATmega1280 Mikrocontrollers möglich. Die nötigen Änderungen werden in dieser Arbeit beschrieben. Dazu zählen die Änderungen des Schaltplans, des Platinenlayouts und der Firmware.

# Abstract

In this diploma thesis the Modular-EEG device from the OpenEEG project will be expanded from 6 channel to 16 channel support. This is possible by using an ATmega1280 microcontroller. The changes made to the schematic diagram, the board layout and the firmware will be described in this work.

# Inhaltsverzeichnis


**1. Einleitung** **1**

1.1. Zielsetzung der Arbeit . . . . . 4

1.2. Vorgehensweise und Gliederung . . . . . 5

**2. Hauptteil** **6**

2.1. Machbarkeitsstudie . . . . . 6

2.1.1. Einführung in die Timing-Problematik . . . . . 7

2.1.2. Berechnung des Timings mit 6 Kanälen . . . . . 8

2.1.3. Berechnung des Timings mit 16 Kanälen . . . . . 12

2.1.4. Lösungsmöglichkeiten für das Timing-Problem . . . 13

2.1.5. Berechnung eines realisierbaren Timings . . . . . 14

2.2. Analyse der Firmware für den ATmega8 . . . . . 17

2.2.1. Programmiersprache C . . . . . 18

2.2.2. Verwendete Anweisungen . . . . . 18

2.2.3. Analyse . . . . . 21

2.3. Schaltplanänderungen . . . . . 35

2.4. Platinenlayout . . . . . 37

2.5. Bauteile . . . . . 40

2.6. Firmware . . . . . 45

2.6.1. Firmware für 6-Kanal-EEG mit ATmega1280 . . . . 45

2.6.2. Firmware für 16-Kanal-EEG mit ATmega1280 . . . 60

2.7. Software . . . . . 68

2.8. Tests . . . . . 70

**3. Abschluss** **74**

3.1. Fazit . . . . . 74

3.2. Ausblick . . . . . 74

**Literaturverzeichnis** **76**

**A. Original Firmware für ATmega8** **79**

**B. 6-Kanal-EEG Firmware für ATmega1280** **88**

**C. 16-Kanal Firmware für ATmega1280** **99**

# Abbildungsverzeichnis

2.1. Ausschnitt einer Tabelle aller Atmel-Mikrocontroller nach A/D-Kanälen sortiert . . . . . 7
2.2. Original-Schaltplan der Digital-Platine des Modular-EEG (Quelle: [Pro]) . . . . . 38
2.3. Schaltplan mit ATmega1280 . . . . . 39
2.4. Pin-Konfigurartion des ATmega8 (Quelle: [Atm07b]) . . . . 40
2.5. Pin-Konfigurartion des ATmega1280 (Quelle: [Atm07a]) . . 41
2.6. Oberseite des Original-Layout (Quelle: [Pro]) . . . . . 42
2.7. Oberseite des Layout für ATmega1280 . . . . . 42
2.8. Unterseite des Original-Layout (Quelle: [Pro]) . . . . . 43
2.9. Unterseite des Layout für ATmega1280 . . . . . 43
2.10. Oberseite der leeren Platine . . . . . 44
2.11. Unterseite der leeren Platine . . . . . 45
2.12. Oberseite der bestückten Platine . . . . . 46
2.13. Unterseite der bestückten Platine . . . . . 46
2.14. Screenshot aus Brainbay mit 16 Kanälen . . . . . 73

# Tabellenverzeichnis

2.1. Bestimmung des optimalen Quarz . . . . . 15
2.2. Bitmuster für die A/D-Wandler Kanalauswahl mit den Bits MUX0-MUX5 [Atm07a] . . . . . 58
2.3. Anordnung der Daten im Textpuffer ohne Komprimierung . 65
2.4. Anordnung der Daten im Textpuffer mit Komprimierung . . 66

# Kapitel 1.

# Einleitung

Computer werden in vielen Gebieten und für viele Zwecke eingesetzt. Für den sinnvollen Einsatz von Computern sind Schnittstellen unbedingt erforderlich. Computer können ohne Schnittstellen weder programmiert noch bedient werden und können auch keine Ergebnisse präsentieren. Je mehr Schnittstellen ein Computer hat, desto mehr Möglichkeiten gibt es, ihn einzusetzen. Gängige Schnittstellen, die jedem Computerbenutzer bekannt sein dürften, sind Tastatur, Maus und Monitor.

Eine spezielle Gruppe von Schnittstellen sind die BCI.[1] Diese Schnittstellen basieren darauf, Signale vom Gehirn zu erfassen und auf einem Computer zu verarbeiten. Ein Verfahren, mit dem man Signale vom Gehirn gewinnen kann, ist das EEG.

EEG ist die Abkürzung für das Elektroenzephalogramm[2], das Hirnstrombild. Die Gehirntätigkeit verursacht schwache elektrische Ströme, die für die Erstellung eines EEGs an verschiedenen Punkten der Kopfhaut mit Hilfe von Elektroden abgeleitet werden. Die Spannungsdifferenz und die Spannungsschwankungen werden verstärkt und entweder von einem Mehrkanalschreiber aufgezeichnet oder von einem Computer oder Mikrocontroller verarbei-

[1] BCI steht für Brain-Computer-Interface

[2] griechisch egképhalon = „was im Kopf ist“, „Gehirn“

tet.[3]

Ein EEG-Signal kann nach sechs abgrenzbaren Kurvenformen untersucht werden, die in unterschiedlichen Frequenzbereichen auftreten und den Zustand der Gehirnaktivität zeigen.

- Deltawellen treten im Schlaf auf und haben eine Frequenz von 1 bis 4 Hz.
- Thetawellen sind mit Kreativität, Spontanität, Ablenkbarkeit, Tagträumen, Depressionen und Angstzuständen verknüpft. Die Frequenz beträgt 4–8 Hz.
- Alphawellen haben eine Frequenz von 8–12 Hz und treten bei Meditation, innerer Ruhe und Friedlichkeit auf.
- Betawellen treten bei aktivem, aufmerksamem wachsein auf und haben eine Frequenz von 13 bis 21 Hertz.
- Hohe Betawellen werden mit Zuständen wie Höchstleistung, Sorge, Angst und Grübeln in Verbindung gebracht und treten bei einer Frequenz von 20–32 Hz auf.
- Gammawellen haben eine Frequenz von 38–40 Hz und sind mit Problem-Lösungs Aufgaben verknüpft.[4]

Man kann also mit dem EEG den Aktivitätsgrad von verschiedenen Hirnregionen[5] feststellen und auch Gehirnerkrankungen erkennen. Mit einem EEG können keine Gedanken gelesen werden.[6]

Da professionelle medizinische Geräte für den Hausgebrauch zu teuer sind um damit herum zu experimentieren wurde das OpenEEG-Projekt ins Leben

[3] vgl. [Bro07a] aus [Hol07] übernommen
[4] vgl. [Dem05]
[5] je nach Positionierung der Elektroden
[6] vgl. [Bro07a] aus [Hol07] übernommen

gerufen. Das Projekt beschäftigt sich mit der Entwicklung von do-it-yourself EEG-Geräten und richtet sich an jeden der sich Amateurhaft mit EEG beschäftigen möchte.[7]

Der erste Teil von OpenEEG, also „Open“, sagt aus, dass die Entwickler, die an diesem Projekt arbeiten, ihre Erfahrungen und ihr Wissen teilen. Dies erfolgt in diesem Fall auf der Internetseite des OpenEEG-Projekts[8], in dem dort Berichte, Anleitungen und vor allem die Quelltexte für Software und Schaltpläne für Hardware veröffentlicht werden. Die Inhalte dieser Internetseiten sind, soweit auf einzelnen Seiten nichts anderes vermerkt ist, unter der „creative commons Attribution-Sharealike 2.0“ Lizenz veröffentlicht.[9]

Die Lizenz legt fest, dass man das Werk vervielfältigen, verbreiten, öffentlich zugänglich machen und Bearbeitungen des Werkes anfertigen darf. Jedoch nur unter folgenden Bedingungen. Der Autor oder Rechteinhaber muss in der von ihm festgelegten Weise genannt werden und das neu entstandene Werk darf nur unter Verwendung von identischen oder vergleichbaren Lizenzbedingungen weitergegeben werden.[10]

Die Hardware des Projekts beschränkt sich bisher auf zwei EEG-Geräte. Zum einen das Modular-EEG, das schon fertig entwickelt ist, während meiner Projektarbeit[11] in Betrieb genommen und in dieser Diplomarbeit erweitert wird. Zum anderen das Soundcard-EEG, das wie der Name schon sagt, die Soundkarte als Schnittstelle zum Computer benutzt und dessen Entwicklung bisher nicht abgeschlossen ist.[12]

Das Modular-EEG verdankt seinen Namen dem modularen Aufbau. Es setzt sich aus einer Digital-Platine und bis zu drei Analog-Platinen zusam-

[7] vgl. [Pro]
[8] http://openeeg.sourceforge.net/
[9] vgl. [Lic] aus [Hol07] übernommen
[10] vgl. [Cre] aus [Hol07] übernommen
[11] [Hol07]
[12] vgl. [Pro] aus [Hol07] übernommen

men. Die Analog-Platinen haben die Aufgabe, die empfindlichen EEG-Signale zu verstärken und durch mehrere Filterstufen von Störungen zu befreien. Eine Analog-Platine ist mit zwei Kanälen ausgestattet. Auf der Digital-Platine befindet sich der Mikrocontroller, der die analogen Signale von den Analog-Platinen Digital-Signale umwandelt und per serieller Datenübertragung zu einem Computer sendet. Auf der Digital-Platine befindet sich ein Netzteilbereich, der durch Optokoppler vom Mikrocontroller-Teil galvanisch getrennt ist. Dieses Netzteil versorgt auch die Analog-Platinen mit Strom.[13]

## 1.1. Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist die Erhöhung der Kanalanzahl des Modular-EEG Gerätes auf 16-Kanäle. Dazu ist der ATmega8 Mikrocontroller durch einen größeren Mikrocontroller zu ersetzen. Um die Fehlerquellen gering zu halten, sollte nur das Nötigste geändert werden. Bestandteil dieser Arbeit sind

- die theoretische Vorbereitung,
- die Änderung des Schaltplans und des Platinen-Layouts für die Fertigung einer neuen Platine,
- die Anpassung der Firmware auf den neuen Mikrocontroller inklusive der Erweiterung auf mehr Kanäle und
- die Anpassung einer Software-Schnittstelle zur Überprüfung der Funktion.

Die Funktion der Digital-Platine wird mit Hilfe des PWM-Signals getestet, indem dieses auf alle Eingänge des Analog/Digital-Wandlers gelegt wird.

[13] vgl. [GNP+03]

## 1.2. Vorgehensweise und Gliederung

Im Hauptteil wird zuerst die theoretische Erreichbarkeit der Zielsetzung überprüft. Anschließend wird eine Firmware aus dem OpenEEG-Projekt detailliert analysiert. Danach wird auf die Änderungen am Schaltplan und am Platinen-Layout eingegangen. Darauf folgt ein Abschnitt über die Änderung der Firmware von ATmega8 auf ATmega1280 unter Beibehaltung der Arbeitsweise. Als nächstes wird die Firmware von 6-Kanal auf 16-Kanal Verwendung erweitert. Die letzten Abschnitte des Hauptteils befassen sich mit der Software-Schnittstelle und den Tests. Nach dem Hauptteil folgt ein Fazit und ein Ausblick. Neben anderen Quellen werden hauptsächlich die Firmware, der Schaltplan, das Platinenlayout und die Software-Schnittstelle aus dem OpenEEG-Projekt verwendet. Die wichtigste und am häufigsten verwendete Literatur sind die Datenblätter der Mikrocontroller ATmega8 und ATmega1280.

# Kapitel 2.

# Hauptteil

## 2.1. Machbarkeitsstudie

Vor Beginn der eigentlichen Arbeiten wird an dieser Stelle überprüft, ob das gesetzte Ziel theoretisch erreichbar ist.

Ein wichtiger Punkt für die Überprüfung der Machbarkeit ist die Berechnung, wie viel Zeit der Mikrocontroller braucht, um 16 Kanäle zu verarbeiten. In den nächsten Abschnitten folgen eine Einführung in die Timing-Problematik und Berechnungen der Timings für Sechs- und 16-Kanal-Betrieb.

Neben den theoretischen Voraussetzungen wird jedoch auch ein Mikrocontroller benötigt, der die entsprechende Anzahl an Analog/Digital-Wandler-Kanälen hat. Die bisher eingesetzten Mikrocontroller im Modular-EEG stammen von der Firma Atmel. Auf der Internetseite von Atmel ließen sich am 12.09.2007 noch alle Atmel-Mikrocontroller in einer Tabelle[1] auflisten und nach bestimmten Kriterien sortieren. In der Fassung vom 27.02.2008 ist die Anzahl der A/D-Kanäle für viele Mikrocontroller in der Tabelle nicht mehr eingetragen und daher an dieser Stelle nicht mehr für die Recherche nach dem geeigneten Mikrocontroller verwendbar. Ein Ausschnitt der nach A/D-Kanälen sortierten Tabelle vom 12.09.2007 ist in Abbildung 2.1 zu sehen.

[1] Parametric Product Table

Die Typen ATmega640, ATmega1280 und ATmega2560 eignen sich für diese Arbeit. Diese drei sind bis auf den Flash Speicher baugleich und werden im selben Datenblatt beschrieben.[2] Von den drei Typen sind bei Conrad oder Reichelt nur der ATmega1280 oder der ATmega2560 käuflich zu erwerben. Für den ATmega128RZBV und den ATmega256RZBV ließen sich keine Bezugsquellen finden. Um nicht alle drei Typen aufzuzählen, wird im weiteren Verlauf der Arbeit nur noch der ATmega1280 verwendet.

| Devices | Flash (Kbytes) | EEPROM (Kbytes) | SRAM (Bytes) | Max I/O Pins | F.max (MHz) | Vcc (V) | 16-bit Timers | 8-bit Timer | PWM (channels) | RTC | SPI | UART | TWI | ISP | 10-bit A/D (Channels) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ATmega2560 | 256 | 4 | 8192 | 86 | 16 | 1.8-5.5 | 4 | 2 | 16 | Yes | 1+USART | 4 | Yes | Yes | 16 |
| ATmega1280 | 128 | 4 | 8192 | 86 | 16 | 1.8-5.5 | 4 | 2 | 16 | Yes | 1+USART | 4 | Yes | Yes | 16 |
| ATmega256RZBV | 256 | 4 | 8192 | 86 | 16 | 1.8-5.5 | 4 | 2 | 16 | Yes | 1+USART | 4 | Yes | Yes | 16 |
| ATmega640 | 64 | 4 | 8192 | 86 | 16 | 1.8-5.5 | 4 | 2 | 16 | Yes | 1+USART | 4 | Yes | Yes | 16 |
| ATmega128RZBV | 128 | 4 | 8192 | 86 | 16 | 1.8-5.5 | 4 | 2 | 16 | Yes | 1+USART | 4 | Yes | Yes | 16 |
| AT90PWM2 | 8 | 0.5 | 512 | 19 | 16 | 2.7-5.5 | 1 | 1 | 7 | Yes | 1 | Yes | -- | Yes | 11 |

Abbildung 2.1.: Ausschnitt einer Tabelle aller Atmel-Mikrocontroller nach A/D-Kanälen sortiert

### 2.1.1. Einführung in die Timing-Problematik

In diesem Abschnitt werden die für die Berechnung des Timings wichtigen Punkte oberflächlich eingeführt. Ein Anhaltspunkt ist die aktuell eingesetzte Firmware für das 6-Kanal-Modular-EEG Gerät. Die detaillierte Analyse der Firmware folgt im Abschnitt 2.2 auf Seite 17.

Die Abtastrate legt fest, mit welcher Frequenz die EEG-Signale digitalisiert und zum Computer übertragen werden. Diese Frequenz beträgt zum Beispiel beim Modular-EEG 256 Hz.[3] Alle Kanäle werden 256-mal pro Sekunde von Analog zu Digital gewandelt und zum Computer übertragen. Die Umwandlung und die Datenübertragung dürfen zusammen nicht länger als

[2] vgl. [Atm07a]
[3] vgl. [HPR03]

3,90625 ms[4] dauern. Ansonsten wäre der erste Zyklus noch nicht vollständig abgearbeitet, bevor der nächste begonnen wird. Die Deadline wäre dann überschritten und das bedeutet, dass der Mikrocontroller nicht genug Zeit hat, das Programm ordnungsgemäß auszuführen.

Der Analog/Digital-Wandler wird beim Modular-EEG mit 10-Bit-Auflösung betrieben.[5] Die Geschwindigkeit des A/D-Wandlers wird durch den Systemtakt festgelegt. Der Systemtakt wird dafür durch einen für den A/D-Wandler separat vorhandenen Prescaler entsprechend skaliert. Die Frequenz des A/D-Wandlers darf bei 10-Bit Auflösung nicht mehr als 200 kHz betragen, um die maximale Auflösung zu erreichen. Die Umwandlungszeiten betragen für die erste Umwandlung 25 A/D-Takte und für die folgenden Umwandlungen jeweils 13 A/D-Takte.[6]

Die Datenübertragung erfolgt seriell über einen USART[7]. Die Datenübertragungsrate für den USART wird wie der A/D-Wandler Takt ebenfalls vom Systemtakt festgelegt und über einen Prescaler eingestellt.[8] Für die Datenübertragung wird pro Byte noch ein Start-Bit und ein Stop-Bit mit übertragen, dadurch werden pro Byte 10 Bit übertragen.[9]

### 2.1.2. Berechnung des Timings mit 6 Kanälen

In diesem Abschnitt wird das Timing für den Mikrocontroller ATmega8 mit der Firmware Version 0.5.4-p2 berechnet. Diese Firmware wurde in meiner Projektarbeit[10] zur Inbetriebnahme eines Modular-EEG Gerätes eingesetzt und dabei ausreichend getestet. Das p2 steht für Paket-Version 2, neben der

[4] siehe Gleichung 2.1
[5] vgl. [HPR03]
[6] vgl. [Atm07b]
[7] Universal Synchronous and Asynchronous serial Receiver and Transmitter
[8] vgl. [Atm07b]
[9] vgl. [HPR03]
[10] [Hol07]

es auch eine Paket-Version 3 gibt. Diese wird jedoch nicht von allen Programmen unterstützt und hier nicht thematisiert.

Wie oben beschrieben, beträgt die Abtastrate 256 Hz. Daraus resultiert nach Gleichung 2.1 eine Zeit von 3,90625 ms, in der ein Zyklus abgearbeitet sein muss.

$$\text{Zeit } t = \frac{1}{\text{Frequenz } f} = \frac{1}{256\,\frac{1}{\text{s}}} = 0{,}00390625\,\text{s} = 3{,}90625\,\text{ms} \tag{2.1}$$

Die Frequenz des A/D-Wandlers darf nicht mehr als 200 kHz betragen. Der Wert für den Prescaler ergibt sich somit aus der Ungleichung 2.2. Der Prescaler für den A/D-Wandler kann aber nur die Werte 2, 4, 8, 16, 32, 64 und 128 annehmen. Hier wird der Wert für den Prescaler also auf 64 festgelegt.

$$\begin{aligned} \text{A/D-Prescaler} &> \frac{\text{Systemtakt-Frequenz}}{\text{Max. Frequenz des A/D-Wandlers}} \\ &> \frac{7{,}3728\,\text{MHz}}{200\,\text{kHz}} \\ &> 36{,}864 \end{aligned} \tag{2.2}$$

Die Taktfrequenz des A/D-Wandlers beträgt 115,2 kHz und resultiert aus der Gleichung 2.3. Ein Taktzyklus des A/D-Wandlers benötigt nach Gleichung 2.4 8,68 $\mu$s.

$$\begin{aligned} \text{Frequenz des A/D-Wandlers} &= \frac{\text{System-Taktfrequenz}}{\text{A/D-Prescaler}} \\ &= \frac{7{,}3728\,\text{MHz}}{64} \\ &= 0{,}1152\,\text{MHz} \\ &= 115{,}2\,\text{kHz} \end{aligned} \tag{2.3}$$

$$\text{Zeit } t = \frac{1}{\text{Frequenz } f} = \frac{1}{115200\,\frac{1}{\text{s}}} = 8{,}68\,\mu\text{s} \tag{2.4}$$

Bei der Berechnung der Zeiten für die A/D-Wandlung werden in der Firmware für die jeweils erste von sechs Umwandlungen 26 A/D-Taktzyklen und für die folgenden 14 A/D-Taktzyklen zugrunde gelegt.[11] Nach Gleichung 2.5 beträgt die Zeit für die erste A/D-Wandlung 225,68 $\mu$s und nach Gleichung 2.6 die Zeit für jede folgende A/D-Wandlung 121,52 $\mu$s.Um die sechs Kanäle umzuwandeln, benötigt der A/D-Wandler nach Gleichung 2.7 insgesamt 0,83328 ms.

$$\begin{aligned}\text{Zeit für die erste A/D-Wandlung} &= 26\text{ Taktzyklen} \cdot 8{,}68\,\mu\text{s} \\ &= 225{,}68\,\mu\text{s}\end{aligned} \tag{2.5}$$

$$\begin{aligned}\text{Zeit für folgende A/D-Wandlungen} &= 14\text{ Taktzyklen} \cdot 8{,}68\,\mu\text{s} \\ &= 121{,}52\,\mu\text{s}\end{aligned} \tag{2.6}$$

$$\begin{aligned}\text{Gesamtzeit A/D-Wandlung} &= 1 \cdot 225{,}68\,\mu\text{s} + 5 \cdot 121{,}52\,\mu\text{s} \\ &= 833{,}28\,\mu\text{s} \\ &= 0{,}83328\,\text{ms}\end{aligned} \tag{2.7}$$

Pro Zyklus der Abtastrate wird ein Datenpaket gesendet. Dieses Paket enthält vier Byte für den Paketheader, zwei Byte für die Daten pro Kanal und ein Byte für den Paketfooter. Bei sechs Kanälen ist das Paket also 17 Byte groß. Bei der Datenübertragung werden pro Byte noch ein Start- und ein Stop-Bit mitgesendet. Die Gesamtgröße eines Paketes beträgt somit 170 Bit.

[11] vgl. [HPR03]

Bei der hier verwendeten Übertragungsrate von $57600\,\frac{\text{Bit}}{\text{s}}$ dauert die Übertragung nach Gleichung 2.8 2,95139 ms.

$$\begin{aligned}\text{Gesamtzeit Datenübertragung} &= \frac{\text{Datenmenge}}{\text{Übertragungsrate}} \\ &= \frac{170\,\text{Bit}}{57600\,\frac{\text{Bit}}{\text{s}}} \\ &= 0{,}00295139\,\text{s} \\ &= 2{,}95139\,\text{ms}\end{aligned} \tag{2.8}$$

Zieht man nun die Zeiten für die A/D-Wandlung und die Datenübertragung von der, für einen Abtast-Zyklus, zur Verfügung stehenden Zeit ab, so bleiben noch 0,12158 ms übrig.

$$\text{Restzeit} = 3{,}90625\,\text{ms} - 0{,}83328\,\text{ms} - 2{,}95139\,\text{ms} = 0{,}12158\,\text{ms} \tag{2.9}$$

In dieser Restzeit sind die Zeiten für die restlichen Anweisungen enthalten. Hier wird jedoch nicht die Dauer jeder einzelnen Anweisung analysiert, sondern die Anzahl der Single-Cycle Anweisungen berechnet, die in der Restzeit ausgeführt werden könnten. Single-Cycle bedeutet, dass die Anweisung mit einem Taktzyklus des Mikrocontrollers ausgeführt werden kann. Es gibt auch Anweisungen, die mehr als einen Taktzyklus für die Ausführung benötigen.

Bei einer Taktfrequenz von 7,3728 MHz dauert ein Taktzyklus $0{,}1356\,\mu\text{s}$. Es bleibt also noch Zeit für 896 Sigle-Cycle Anweisungen.

$$\text{Zeit } t = \frac{1}{\text{Frequenz } f} = \frac{1}{7372800\,\frac{1}{\text{s}}} = 0{,}1356\,\mu\text{s} \tag{2.10}$$

### 2.1.3. Berechnung des Timings mit 16 Kanälen

Hier wird das Timing, auf der Grundlage der obigen Rechnung, für 16 Kanäle berechnet.

Die Gesamtzeit für die A/D-Wandlung erhöht sich von 0,83328 ms auf 2,04848 ms.

$$\begin{aligned} \text{Gesamtzeit A/D-Wandlung} &= 1 \cdot 225{,}68\,\mu\text{s} + 15 \cdot 121{,}52\,\mu\text{s} \\ &= 2048{,}48\,\mu\text{s} \\ &= 2{,}04848\,\text{ms} \end{aligned} \tag{2.11}$$

Die Paketgröße steigt von 17 Byte auf 37 Byte, da pro Kanal zwei Byte übertragen und nun zehn Kanäle mehr verwendet werden. Durch die Start- und Stop-Bits entspricht ein Paket 370 Bit. Die Datenübertragung benötigt bei einer Übertragungsrate von 57600 $\frac{\text{Bit}}{\text{s}}$ eine Zeit von 6,42361 ms.

$$\begin{aligned} \text{Gesamtzeit Datenübertragung} &= \frac{\text{Datenmenge}}{\text{Übertragungsrate}} \\ &= \frac{370\,\text{Bit}}{57600\,\frac{\text{Bit}}{\text{s}}} \\ &= 0{,}00642361\,\text{s} \\ &= 6{,}42361\,\text{ms} \end{aligned} \tag{2.12}$$

Die Datenübertragung braucht schon mehr Zeit als zur Verfügung steht. Mit den bisherigen Einstellungen kann das Programm nicht ordnungsgemäß ausgeführt werden, wenn 16 Kanäle verarbeitet werden. Im folgenden Abschnitt werden Lösungsmöglichkeiten für dieses Problem aufgezählt.

### 2.1.4. Lösungsmöglichkeiten für das Timing-Problem

In diesem Abschnitt werden Möglichkeiten aufgezeigt, mit denen das Timing-Problem aus dem vorherigen Abschnitt gelöst werden könnte.

Die Abtastrate legt den Zeitrahmen fest, in dem alles abgearbeitet werden muss, um eine ordnungsgemäße Funktion des Programms zu gewährleisten. Die Abtastrate muss nach dem Abtasttheorem von Harry Nyquist mindestens das $2\frac{1}{2}$ fache der höchsten Frequenz betragen. Das Signal muss dabei durch einen Tiefpassfilter begrenzt werden. Es muss bestimmt werden, bis zu welcher Frequenz die Signale wichtig sind und welche Signalanteile für die Auswertung nicht benötigt werden.[12]

Wie in der Einleitung beschrieben, haben die Gammawellen mit 42 Hz die höchste Frequenz, die zum Beispiel beim Neurofeedback Training relevant ist.[13]

Die Abtastrate könnte nach dem Nyquist-Theroem bis auf ein Minimum von 105 Hz reduziert werden. Je höher jedoch die Abtastrate ist desto besser lässt sich das analoge Signal digital abbilden. Die meisten Programme aus dem OpenEEG-Projekt verwenden 256 Hz als Standard-Abtastrate.

Bei der Datenübertragung kann die Geschwindigkeit durch Verändern des Prescalers problemlos von 57600 $\frac{\text{Bit}}{\text{s}}$ auf 115200 $\frac{\text{Bit}}{\text{s}}$ erhöht werden. Dadurch würde sich die Zeit für die Übertragung auf die Hälfte reduzieren.

Weiterhin werden pro Kanal zwei Byte übermittelt. Der A/D-Wandler hat aber eine maximale Auflösung von zehn Bit. Es bleiben demnach pro Kanal sechs Bit im Datenpaket frei. Bei 16 Kanälen sind das immerhin 12 Byte. Durch Komprimierung könnten sich zum Beispiel zwei Kanäle drei Byte teilen.[14] Der Paketumfang wäre dann nur noch 29 Byte anstelle von 37 Bytes. Ein weiteres Byte lässt sich auch im Paketheader sparen, wenn das Byte

[12] vgl. [Stu04]
[13] vgl. [Dem05]
[14] siehe Tabelle 2.4 auf Seite 66

für die Paketversion weggelassen wird.

Die maximale Frequenz des A/D-Wandlers von 200 kHz ist mit 115,2 kHz noch nicht voll ausgeschöpft. Für eine Erhöhung der Frequenz muss die Systemtaktfrequenz erhöht werden. Diese Frequenz wird durch ein Quarz auf der Platine festgelegt. Die Tabelle 2.1 dient der Bestimmung der optimalen Taktfrequenz. In ihr werden für die Frequenzen der handelsüblichen Quarze die zugehörigen Datenübertragungsraten und Frequenzen des A/D-Wandlers dargestellt. Die Tabelle enthält nur Quarzfrequenzen zwischen 1000 und 16000 kHz und davon nur diejenigen, mit der sich Anhand des Prescalers eine Datenübertragungsrate von 115200 $\frac{\text{Bit}}{\text{s}}$ einstellen lässt. Die optimale Frequenz beträgt 11059,2 kHz, da bei dieser Frequenz die höchste A/D-Frequenz , bei gleichzeitig eingestellter Übertragunsgrate von 115200 $\frac{\text{Bit}}{\text{s}}$, erzielt werden kann.

### 2.1.5. Berechnung eines realisierbaren Timings

Hier erfolgt die Berechnung eines möglichen Timings, mit dem die Funktion für 16 Kanäle sichergestellt werden kann. Die Abtastrate bleibt unverändert und der Prescaler für den A/D-Wandler kann ebenfalls auf 64 eingestellt bleiben. Durch die Erhöhung der Systemtaktfrequenz auf 11,0592 MHz wird die Frequenz des A/D-Wandlers auf 172,8 kHz erhöht.

$$\begin{aligned} \text{Frequenz des A/D-Wandlers} &= \frac{\text{System-Taktfrequenz}}{\text{A/D-Prescaler}} \\ &= \frac{11{,}0592\,\text{MHz}}{64} \\ &= 0{,}1728\,\text{MHz} \\ &= 172{,}8\,\text{kHz} \end{aligned} \tag{2.13}$$

Die Zeit für einen A/D-Taktzyklus vermindert sich auf 5,787 $\mu$s.

Tabelle 2.1.: Bestimmung des optimalen Quarz

| Quarz | Übertragungsrate mit Prescaler | | | | | | A/D-Frequenz mit Prescaler | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Frequenz | 16 | 32 | 64 | 80 | 96 | 128 | 8 | 16 | 32 | 64 | 128 |
| 1843,2 | 115,2 | 57,6 | 28,8 | 23,04 | 19,2 | 14,4 | - | 115,2 | 57,6 | 28,8 | 14,4 |
| 3686,4 | 230,4 | 115,2 | 57,6 | 46,08 | 38,4 | 28,8 | - | - | 115,2 | 57,6 | 28,8 |
| 7372,8 | 460,8 | 230,4 | 115,2 | 92,16 | 76,8 | 57,6 | - | - | - | 115,2 | 57,6 |
| 9216,0 | 576,0 | 288,0 | 144,0 | 115,2 | 96,0 | 72,0 | - | - | - | 144 | 72 |
| **11059,2** | 691,2 | 345,6 | 172,8 | 138,2 | **115,2** | 86,4 | - | - | - | **172,8** | 86,4 |
| 14745,6 | 921,6 | 460,8 | 230,4 | 184,3 | 153,6 | 115,2 | - | - | - | - | 115,2 |

$$\text{Zeit } t = \frac{1}{\text{Frequenz } f} = \frac{1}{172800\,\frac{1}{\text{s}}} = 5{,}787\,\mu\text{s} \tag{2.14}$$

Damit ändert sich auch die Gesamtzeit für die A/D-Wandlung von 2,04848 ms auf 1,365732 ms.

$$\begin{aligned}\text{Zeit für die erste A/D-Wandlung} &= 26\text{ Taktzyklen} \cdot 5{,}787\,\mu\text{s} \\ &= 150{,}462\,\mu\text{s}\end{aligned} \tag{2.15}$$

$$\begin{aligned}\text{Zeit für folgende A/D-Wandlungen} &= 14\text{ Taktzyklen} \cdot 5{,}787\,\mu\text{s} \\ &= 81{,}018\,\mu\text{s}\end{aligned} \tag{2.16}$$

$$\begin{aligned}\text{Gesamtzeit A/D-Wandlung} &= 1 \cdot 150{,}462\,\mu\text{s} + 15 \cdot 81{,}018\,\mu\text{s} \\ &= 1365{,}732\,\mu\text{s} \\ &= 1{,}365732\,\text{ms}\end{aligned} \tag{2.17}$$

Die Paketgröße wird durch Komprimierung nach dem Schema in Tabelle 2.4 auf Seite 66 von 37 Byte auf 29 Byte verringert und das Byte für die Paketversion im Header wird ebenfalls eingespart. Insgesamt ist ein Paket dann 28 Byte groß. Die Datenübertragungsrate wird auf 115200 $\frac{\text{Bit}}{\text{s}}$ erhöht. Diese Maßnahmen kürzen die Zeit für die Datenübertragung von 6,42361 ms auf 2,43056 ms.

$$\begin{aligned}\text{Gesamtzeit Datenübertragung} &= \frac{\text{Datenmenge}}{\text{Übertragungsrate}} \\ &= \frac{280\,\text{Bit}}{115200\,\frac{\text{Bit}}{\text{s}}} \\ &= 0{,}00243056\,\text{s} \\ &= 2{,}43056\,\text{ms}\end{aligned} \tag{2.18}$$

Es bleibt noch eine restliche Zeit von 0,109958 ms. Bei 11,0592 MHz dauert ein Taktzyklus 0,0904 $\mu$s. Die restliche Zeit reicht demnach für 1216 Single-Cycle Anweisungen.

$$\text{Restzeit} = 3{,}90625\,\text{ms} - 1{,}365732\,\text{ms} - 2{,}43056\,\text{ms} = 0{,}109958\,\text{ms} \tag{2.19}$$

$$\text{Zeit } t = \frac{1}{\text{Frequenz } f} = \frac{1}{11059200\,\frac{1}{\text{s}}} = 0{,}0904\,\mu\text{s} \tag{2.20}$$

Dieses Timing ermöglicht die Verarbeitung von 16 Kanälen mit einer Abtastrate von 256 Hz.

## 2.2. Analyse der Firmware für den ATmega8

Für die Änderungen des Schaltplans und des Platinenlayouts ist das Verständnis der Arbeitsweise notwendig. Weiterhin muss für die Änderung der Firmware, für die Verwendung auf einem anderen Mikrocontroller, die Wirkung jeder Anweisung in der Original-Firmware klar sein. Aus diesen Gründen erfolgt hier eine detaillierte Analyse der Firmware. Zunächst wird auf die verwendete Programmiersprache und anschließend auf die häufig verwendeten Anweisungen eingegangen.

### 2.2.1. Programmiersprache C

Die Firmware ist in der Programmiersprache C geschrieben. Die Sprache C wurde von Dennis Ritchie und Brian Kernighan aus der Programmiersprache B weitermodifiziert, die von Ken Thompson entwickelt wurde. Veröffentlicht wurde C das erste Mal im Jahr 1978. Der erste Standard namens ANSI-C wurde 1989 vom American Nationals Standards Institute festgelegt. Ein paar Elemente aus der Weiterentwicklung von C zu C++ sind in den ISO-Standard C99 von C aus dem Jahre 1999 aufgenommen worden.[15]

C ist eine maschinennahe und eine höhere Programmiersprache. Auf der einen Seite kann mit C hardwareorientiert programmiert und somit die Eigenschaften einer speziellen Hardware optimal ausgenutzt werden. Und auf der anderen Seite kann ein C Programm, welches nicht so maschinennah programmiert ist, auf einer anderen Computerarchitektur oder einem anderen Betriebssystem verwendet werden. Dafür muss es aber von einem Compiler neu übersetzt werden. Der Sprachumfang von C umfasst 31 reservierte Worte und beinhaltet dennoch Elemente aus höheren Programmiersprachen. Weiterhin stehen 40 verschiedene Operatoren für die Verknüpfung von Variablen und Zahlen zur Verfügung.[16]

Das Hauptprogramm ist wie alle Unterprogramme und Programmmodule als Funktion realisiert. Wobei das Hauptprogramm die einzige Funktion ist, die sich selbst aufruft. Dieser Selbstaufruf der Funktion *main()* erfolgt beim Programmstart.[17]

### 2.2.2. Verwendete Anweisungen

An dieser Stelle werden verschiedene Anweisungen erklärt, die in der Firmware verwendet werden. Dadurch kann bei der Analyse auf die wiederholte

[15] vgl. [Bro07b]
[16] vgl. [Bro07b]
[17] vgl. [Bro07b]

Erklärung der Funktionsweise verzichtet werden.

Die *#include*-Anweisung gefolgt von einem Dateinamen bewirkt, dass der Inhalt der angegebenen Datei an Stelle der *#include*-Anweisung in das Programm geschrieben wird.[18]

Mit *#define* werden symbolische Konstanten definiert. Mit der Anweisung

```
#define NUMCHANNELS 6
```

wird der symbolischen Konstanten *NUMCHANNELS* der Wert 6 zugewiesen. *NUMCHANNELS* steht hier für die Anzahl der verwendeten A/D-Kanäle. Im Programm wird an allen Stellen, an denen die Anzahl der Kanäle angegeben wird, statt der Zahl die symbolische Konstante eingetragen. Der Vorteil von symbolischen Konstanten ist, dass, bei einer nötigen Änderung des Wertes, nur an einer zentralen Stelle und nicht im ganzen Programm verstreut der Wert angepasst werden muss. Außerdem wird das Programm dadurch besser lesbar.[19]

Die Funktion *outb(ZIEL, WERT)* dient der Ausgabe des Wertes WERT in ZIEL.[20] Diese Funktion wird in der Firmware für das Schreiben in Register verwendet.

Mit *inp(PORT)* werden Daten vom angegebenen Port gelesen.[21]

Die Anweisung *BV(BIT)* wird durch die eingebundene Datei *avr/sfr_defs* in (1 « (BIT)) umgewandelt.[22] Es wird also eine 1 um *BIT* stellen nach links verschoben. Die 1 steht dann an genau der Stelle, an der sich das Bit im jeweiligen Register befindet.

*sbi(REGISTER, BIT)* ist eine Funktion, die im angegebenen Register das

[18] vgl. [Wol06]
[19] vgl. [Wol06]
[20] vgl. [AVR07]
[21] vgl. [AVR07]
[22] vgl. [AVR07]

angegebene Bit auf den Wert 1 setzt.[23]

Das gegenteilige wird mit der Funktion *cbi(REGISTER, BIT)* bewirkt. Das Bit wird im Register gelöscht, also mit dem Wert 0 versehen.[24]

Die folgenden Anweisungen werden bei der Übersetzung des Quellprogramms in ein ausführbares Programm ausgewertet. Anweisungen, die zwischen *#if defined (__AVR_ATmega8__)* und *#else* stehen werden nur dann übersetzt, wenn das Programm für einen AVR ATmega8 Mikrocontroller übersetzt wird. In allen anderen Fällen werden die Anweisungen zwischen *#else* und *#endif* übersetzt und somit in das ausführbare Programm übernommen.

```
#if defined (__AVR_ATmega8__)
// Anweisungen
#else
// Anweisungen
#endif
```

Ähnliches gilt bei dieser Verzweigung, die jedoch bei der Ausführung des Programms und nicht bei der Übersetzung des Programms ausgewertet wird. Diesmal müssen die jeweiligen Anweisungen, die ausgeführt werden sollen, in geschweifte Klammern gefasst werden. Ist die *BEDINGUNG* erfüllt[25], dann werden die Anweisungen zwischen den Klammern nach *if(BEDINGUNG)* ausgeführt.

```
if (BEDINGUNG)
{
// Anweisungen
}
else
{
// Anweisungen
```

[23] vgl. [AVR07]
[24] vgl. [AVR07]
[25] wahr oder 1

```
}
```

Das waren die Anweisungen, die am häufigsten in der Firmware verwendet werden. Die restlichen werden in der nun folgenden Analyse der Firmware beschrieben.

### 2.2.3. Analyse

Die Analyse der Original-Firmware findet in der Reihenfolge des Programmablaufs statt. Die Anweisungen, die im Abschnitt Definitionen dargestellt sind, werden sogar schon vor dem eigentlichen Programmstart beim Übersetzen des Quelltextes in ein ausführbares Programm abgearbeitet. Die Anweisungen, die bei jeder Ausführung eines Programms als erstes ausgeführt werden, stehen in der Hauptfunktion *main()*, welche sich beim Programmstart selbst aufruft.

In dieser Analyse wird nur erläutert, was durch welche Anweisung bewirkt wird. Auf die Darstellung von weiteren Optionen, welche mit den Registern noch konfiguriert werden können, wird hier verzichtet, da dies im Rahmen dieser Arbeit nicht zu leisten ist. Die Konzentration liegt auf der Funktionsweise der Firmware.

#### Definitionen

Der Inhalt der folgenden Dateien wird bei der Übersetzung in das Programm eingebunden. Da diese Dateien zum Teil auch *#inlude*-Anweisungen mit weiteren Dateien enthalten und nicht alle Dateien für das Verständnis des Programms erwähnt werden müssen, werden hier nicht alle Dateien behandelt.

```
#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>
```

```
#include <avr/signal.h>
```

Die Datei *avr/io.h* bindet die für den Mikrocontroller entsprechenden IO-Definitionen in das Programm ein. Dazu gehören die folgenden weiteren Dateien. Die Datei *avr/iom8.h* wird nur eingebunden, wenn das Programm für einen AVR ATmega8-Mikrocontroller übersetzt wird. Mit einer ganzen Reihe von solchen Bedingungen in der Datei *avr/io.h* wird für jeden Mikrocontroller die richtige IO-Definitions Datei gefunden.[26]

```
#include <avr/sfr_defs.h>
#elif defined (__AVR_ATmega8__)
#include <avr/iom8.h>
#endif
#include <avr/portpins.h>
#include <avr/common.h>
#include <avr/version.h>
#include <avr/fuse.h>
#include <avr/lock.h>
```

*avr/sfr_defs.h* beinhaltet die Definitionen für die Funktionen, mit denen auf die *Special Function Register* zugegriffen werden kann. Darunter ist auch die oben erwähnte Funktion *BV()*. Dadurch wird eine Abstraction von den spezifischen AVR Anweisungen für den Zugriff auf die Register mit Spezialfunktionen erreicht. So kann zum Beispiel im Programm direkt auf PORTA zugegriffen werden. Der Compiler ersetzt beim Übersetzen *PORTA* durch den Funktionsaufruf *_SFR_IO8(0x02)* mit der zugehörigen Adresse für Port A.

```
#define PORTA    _SFR_IO8(0x02)
```

In der Datei *avr/iom8.h* wird unter anderem zu jedem Registernamen die zugehörige Speicheradresse und zu jedem Bitnamen die zugehörige Bitposition im Register definiert.[27]

[26] vgl. [MWW07]
[27] vgl. [Mic02]

Als nächstes folgt die Definition der symbolischen Konstanten. *NUMCHANNELS* steht für die Anzahl der A/D-Kanäle, *HEADERLEN* für die Länge des Paket-Headers in Byte und *PACKETLEN* für die Länge des Datenpaketes insgesamt. Die Paketlänge setzt sich aus zwei Byte für jeden Kanal plus der Länge des Paketheaders plus einem Byte für den Paket-Footer zusammen. *SAMPFREQ* steht für die Abtastrate und *TIMER0VAL* für den Wert, den der Timer 0 erhält, um die Abtastrate zu realisieren. Die Zahl 7372800 in *TIMER0VAL* ist die Frequenz des Quarzes, der die Taktfrequenz des Mikrocontrollers bestimmt. Die detaillierte Beschreibung der Berechnung von *TIMER0VAL* findet sich weiter unten auf Seite 31.

```
#define NUMCHANNELS 6
#define HEADERLEN 4
#define PACKETLEN (NUMCHANNELS * 2 + HEADERLEN + 1)
#define SAMPFREQ 256
#define TIMER0VAL 256 - ((7372800 / 256) / SAMPFREQ)
```

Nun folgen einige Variablen, die für das ganze Programm gültig sind. *channel_order[]* ist eine konstante[28] Liste aus Werten vom Datentyp *char*. Die Liste legt fest, in welcher Reihenfolge die A/D-Kanäle bearbeitet werden. Die Reihenfolge entspricht der Zuordnung der Kanäle zu den Analog-Platinen. Kanal 0 und 3 sind auf der ersten Platine, 1 und 4 auf der zweiten und 2 und 5 auf der dritten Analog-Platine.

Der Datentyp *char* ist ein Byte groß und kann für einzelne Zeichen oder für Ganzzahlen zwischen -128 und +127 verwendet werden.[29]

*TXBuf* ist ebenfalls eine Liste. Diese Liste dient als Puffer für die Paketdaten, die diesmal vom Datentyp *uint8_t* sind. Die Länge, also die Größe des Puffers, wird durch *PACKETLEN* festgelegt.

[28] const
[29] vgl. [Wol06]

Das Schlüsselwort *volatile* bewirkt, dass der Wert einer Variablen bei jedem Zugriff neu aus dem Speicher gelesen wird und der Compiler beim Übersetzen des Programms keine Optimierungen an diesem Variablenzugriff vornimmt.[30]

Weiterhin bewirkt *volatile*, dass die Variable im Speicher und nicht in einem Register gespeichert wird. Dies ist notwendig, wenn mit, in einem C-Programm eingebetteten, Assembler-Anweisungen auf die Variable zugegriffen werden soll.[31]

*uint8_t* steht für unsigned integer 8 bit type und kann für vorzeichenlose, also nur positive, Ganzzahlen von 0–255 verwendet werden.[32]

Die Variable *TXIndex* wird zur Indizierung des Paketpuffers und *CurrentCh* wird zur Indizierung der Kanalliste verwendet.

```
char const channel_order[]= { 0, 3, 1, 4, 2, 5 };

/** The transmission packet */
volatile uint8_t TXBuf[PACKETLEN];

/** Next byte to read or write in the transmission packet. */
volatile uint8_t TXIndex;

/** Current channel being sampled. */
volatile uint8_t CurrentCh;
```

### Initialisierung in der Hauptfunktion

In diesem Abschnitt werden die Anweisungen analysiert, die beim Programmstart einmalig ausgeführt werden.

Zuerst werden die ersten vier Byte des Textpuffers geschrieben. In die ersten zwei Byte kommen Werte für die Synchronisierung mit der Software. Die

[30] vgl. [Wol06]
[31] vgl. [Fli01]
[32] vgl. [AVR07]

Software kann anhand dieser aufeinanderfolgenden und festgelegten Werte den Anfang eines Datenpaketes erkennen. Im dritten Byte des Puffers wird die Nummer der Paketversion untergebracht. Dies zeigt der Software an, in welcher Reihenfolge die Daten in einem Datenpaket gespeichert sind. Und das vierte Byte enthält einen Zähler für die Pakete von 0–255, der bei jedem Paket um eins hochgezählt wird. Hier wird der Paketzähler mit 0 initialisiert.

```
//Write packet header and footer
TXBuf[0] = 0xa5;          //Sync 0
TXBuf[1] = 0x5a;          //Sync 1
TXBuf[2] = 2;             //Protocol version
TXBuf[3] = 0;             //Packet counter
```

Danach werden die Ports D und B initialisiert:
*DDRD* ist das Data Direction Register für Port D. Analog dazu ist *DDRB* für Port B. Diese Register legen für die einzelnen Pins am jeweiligen Port fest, ob diese als Eingang oder als Ausgang verwendet werden sollen. Durch setzen eines Bits auf 1 wird der zugehörige Pin als Ausgang definiert. Die Register werden mit 0 initialisiert. Daher sind alle Pins von allen IO-Ports beim Programmstart als Eingang konfiguriert.[33]

Hier werden die Pins PD7, PD6 und PD1 von Port D und die Pins PB2, PB1 und PB0 von Port B als Ausgang umkonfiguriert.

```
outb(DDRD, 0xc2);
outb(DDRB, 0x07);
```

*PORTD* steht für das Port D Data Register. Durch die folgenden Anweisungen werden für alle Pins an Port D und Port B, die als Eingang konfiguriert sind, interne Pull-up Widerstände aktiviert.[34]

```
outb(PORTD, 0xff);
```

[33] vgl. [Atm07b]
[34] vgl. [Atm07b]

```
outb(PORTB, 0xff);
```

Als nächstes wird als Sleep Mode *Idle* ausgewählt:
*MCUCR* ist das MCU Control Register, *SE* ist das Sleep Enable Bit und *SM2–SM0* sind die Sleep Mode Bits. Im *Idle Mode* wird die CPU gestoppt, aber SPI, USART, A/D-Wandler, Timer und Interrupts funktionieren weiter. So kann der Mikrocontroller durch Interrupts geweckt werden, die in dieser Firmware vom USART, dem A/D-Wandler, dem Timer 0 und dem Timer 1 ausgelöst werden.[35]

```
outb(MCUCR,
    (inp(MCUCR) | BV(SE)) & (~BV(SM0) | ~BV(SM1) | ~BV(SM2)));
```

Nun erfolgt die Initialisierung des A/D-Wandlers:
*ADMUX* ist das ADC Multiplexer Selection Register, in dem die MUX0-MUX5 Bits, die REFS0 + REFS1 Bits und das ADLAR Bit. Diese Bits werden beim Einschalten mit 0 initialisiert. Über die MUX Bits wird der Kanal und mit den REFS Bits die Quelle der Referenzspannung für die nächste A/D-Wandlung festgelegt. REFS steht für Reference Selection Bits und ADLAR für ADC Left Adjust Result. Als Quelle für die Referenzspannung wird der Pin AREF ausgewählt. ADCH und ADCL sind die Datenregister in denen das Ergebnis der A/D-Wandlung gespeichert wird. Das ADLAR Bit legt die Sortierung der Daten in den Datenregistern fest. Durch ADLAR = 0 werden die Bits 9 und 8 in ADCH und die Bits 7 bis 0 in ADCL gespeichert. Durch setzten des ganzen Registers *ADMUX* auf 0 wird auch der Kanal 0 für die erste A/D-Wandlung ausgewählt.[36]

*ADCSR* steht für ADC Control and Status Register, *ADPS2-ADPS0* sind die ADC Prescaler Select Bits, *ADIF* ist das ADC Interrupt Flag und *ADEN* ist das ADC Interrupt Enable Bit. Durch setzen der ADPS2 und des ADPS1

[35] vgl. [Atm07b]
[36] vgl. [Atm07b]

Bits auf 1 wird der Prescaler für den A/D-Wandler auf 64 eingestellt. Das ADIF Bit zeigt einen bestehenden Interrupt des A/D-Wandlers an, wenn es 0 ist. Wird es auf 1 gesetzt, werden die Interrupts zurück gesetzt. Mit ADEN wird der A/D-Wandler eingeschaltet.[37]

```
outb(ADMUX, 0);          //Select channel 0
//Prescaler = 64, free running mode = off, interrupts off.
outb(ADCSR, BV(ADPS2) | BV(ADPS1));
sbi(ADCSR, ADIF);        //Reset any pending ADC interrupts
sbi(ADCSR, ADEN);        //Enable the ADC
```

Initialisierung des USART:
Der Wert 7, der für eine Übertragungsrate von 57600 $\frac{\text{Bit}}{\text{s}}$, in die USART Baud Rate Register *UBRRL* und *UBRRH* geschrieben werden muss resultiert aus Gleichung 2.21. *UCSRA* bis *UCSRC* sind die USART Control and Status Register.[38]

$$\begin{aligned} \text{UBRR} &= \frac{\textit{Systemtakt frequenz}}{16 \cdot \text{Übertragunsrate}} - 1 \\ &= \frac{7372800\,\frac{1}{\text{s}}}{16 \cdot 57600\,\frac{\text{Bit}}{\text{s}}} - 1 \\ &= 7 \end{aligned} \tag{2.21}$$

Im UCSRA Register wird nur ein Bit bei der Systemstart mit 1 initialisiert. Das ist das USART Data Register Empty Bit, welches anzeigt, ob der Sendepuffer leer ist. Ist das Bit 1, kann ein Interrupt ausgelöst werden. Das wird durch löschen des Bits verhindert.

Das Register UCSRC teilt sich eine Speicheradresse mit dem UBRRH Register. Durch wenn das USART Register Select Bit *URSEL* in der Anwei-

[37] vgl. [Atm07b]
[38] vgl. [Atm07b]

sung 1 ist, wird in das UCSRC geschrieben. Die USART Character Size Bits *UCSZ2–UCSZ0* legen durch setzten von UCSZ1 und UCSZ0 auf 1 fest, dass acht Datenbits pro Frame übertragen werden. Ein Frame beginnt mit einem Start-Bit und endet mit einem Stop-Bit.

*TXEN* heißt Transmitter Enable und aktiviert den USART Sender.[39]

```
#if defined (__AVR_ATmega8__)
    outb(UBRRH,0);                  //Set speed to 57600 bps
    outb(UBRRL,7);
    outb(UCSRA, 0);
    outb(UCSRC, BV(URSEL) | BV(UCSZ1) | BV(UCSZ0));
    outb(UCSRB, BV(TXEN));
```

Jetzt wird der Timer 0 für die Abtastrate initialisiert:
*TCNT0* ist das Timer/Counter 0 Register, das dessen Zählwert enthält. Der Zähler wird gelöscht. *TCCR0* steht für Timer/Counter 0 Control Register welches nur drei Clock Select Bits (CS02 bis CS00) enthält. Diese Bits legen die Quelle für den Zählertakt fest, die hier auf den Systemtakt mit einem Prescaler von 256 eingestellt wird. Durch Auswahl der Quelle wird der Zähler gestartet. Wenn das Timer/Counter 0 Overflow Flag *TOIE0* im Timer/-Counter Interrupt Mask Register *TIMSK* auf 1 gesetzt wird und globale Interrupts aktiviert sind, dann ist der Timer 0 Überlauf Interrupt eingeschaltet. Dass heißt, dass unter dieser Voraussetzung beim nächsten Timer 0 Überlauf ein Interrupt ausgelöst würde. Globale Interrupts sind zu diesem Zeitpunkt noch nicht aktiviert.[40]

```
//Initialize timer 0
outb(TCNT0, 0);             //Clear it.
outb(TCCR0, 4);             //Start it. Frequency = clk / 256
outb(TIMSK, BV(TOIE0));     //Enable the interrupts.
```

[39] vgl. [Atm07b]
[40] vgl. [Atm07b]

Danach folgt optional die Initialisierung des PWM Signals durch den Aufruf der Funktion *pwm_init()*:

```
//Initialize PWM (optional)
pwm_init();
```

Der folgende Funktionsaufruf ist in der Datei *interrupt.h* definiert, die zu Beginn eingebunden wurde. Die Definition wandelt diesen in einen Inline-Assembler-Aufruf um, der wiederrum in [Atm05] beschrieben ist. Letztendlich wird durch diese Anweisung das Global Interrupt Flag im Status Register des Mikrocontrollers gesetzt und dadurch werden globale Interrupts aktiviert. Beim nächsten Überlauf von Timer 0 wird somit der Timer 0 Overflow Interrupt ausgelöst.

```
sei();
```

Zum Abschluss der Initialisierungsphase wird der Prozessor angehalten, in dem der Mikrocontroller in den Schlaf-Modus versetzt wird. Es handelt sich hier wieder um eine Inline-Assembler-Anweisung, die hier in einer Endlos-Schleife ausgeführt wird. Der Mikrocontroller wird immer wieder in den Schlaf-Modus versetzt.

```
while (1)
    {
        __asm__ __volatile__ ("sleep");
    }
```

#### PWM-Testsignal

Für das PWM-Testsignal wird der Timer 1 verwendet. Die Anweisungen in diesem Abschnitt werden durch den Funktionsaufruf *pwm_init()* in der Hauptfunktion ausgeführt.

*OCR1A* bedeutet Output Compare Register A für Timer 1. Der Wert in den Registern *OCR1AH* und *OCR1AL* wird ständig mit dem Zählerwert des Timers 1 verglichen. Durch die Timer/Counter 1 Control Register *TCCR1A* und *TCCR1B* wird die Funktion des Timers konfiguriert. *WGM* steht für Waveform Generation Mode. Die Bits WGM13 bis WGM10 legen den Funktionsmodus des Timer 1 fest. Durch setzten der Bits *WGM11* und *WGM10* auf 1 wird der phasenkorrekte 10-Bit PWM Modus ausgewählt. Durch die Bitkombination der Compare Output Mode Bits *COM1A1* und *COM1A0* mit COM1A1 = 1 und COM1A0 = 0 wird festgelegt, dass der Pin OC1A, bei Übereinstimmung von OCR1A mit dem Zählerwert, beim Hochzählen des Timers auf 0 und beim Runterzählen auf 1 gesetzt wird. Der Zähler 1 zählt in dieser Konfiguration von 0 bis 1023 hoch und dann wieder bis 0 runter. Wenn er den Wert 512 erreicht, wird beim hochzählen OC1A auf 0 und beim runterzählen auf 1 gesetzt. *CS* steht für Clock Select. Durch setzen von *CS12* auf 1 wird als Quelle für den Timer 1 der Prescaler mit einer Einstellung von 256 gewählt. Nach Gleichung 2.22 beträgt die Frequenz des Signals, durch Festlegen des Vergleichswertes auf die Mitte des Wertebereiches, 14,076 Hz. Der Wert 2046 kommt dadurch zustande, dass der Pin nur bei jeder zweiten Übereinstimmung umgeschaltet wird.[41]

$$\begin{aligned} \text{PWM-Frequenz} &= \frac{\frac{\textit{Systemtaktfrequenz}}{\text{Prescaler}}}{2046} \\ &= \frac{\frac{7372800\,\frac{1}{\text{s}}}{256}}{2046} \\ &= 14{,}076\,\frac{1}{\text{s}} \end{aligned} \tag{2.22}$$

```
#if defined (__AVR_ATmega8__)
    outb(OCR1AH, 2);        // Set OCR1A = 512
```

[41] vgl. [Atm07b]

```
    outb(OCR1AL,0);
    // Set 10-bit PWM mode
    outb(TCCR1A, BV(COM1A1) + BV(WGM11) + BV(WGM10));
    // Start and let run at clk / 256 Hz.
    outb(TCCR1B, (1 << CS12));
```

#### Interruptroutine für die Abtastrate

Die Funktion *SIGNAL(SIG_OVERFLOW0)* wird jedesmal aufgerufen, wenn der Zähler von Timer 0 überläuft, also einen Wert von 255 überschreitet und wieder von 0 anfängt. Die Frequenz des Timer 0 wurde in der Hauptfunktion auf die Systemtaktfrequenz geteilt duch 256 eingestellt und beträgt somit 28800 Hz. Der Wert von Timer 0 wird 28800 mal pro Sekunde erhöht. Würde der Timer immer von 0 bis 255 zählen, so läge die Abtastrate nach Gleichung 2.24 bei 112,5 Hz. Um die Abtastrate auf 256 Hz zu erhöhen muss die Anzahl der Zählerschritte bis zum Überlauf verringert werden. In Gleichung 2.25 wird der Wert 112,5 für die nötige Anzahl der Schritte bis zum Überlauf für eine Abtastrate von 256 Hz bestimmt. Da der Timer jedoch nur Ganzzahlen annimmt, wird die Nachkommastelle abgeschnitten und die Abtastrate somit auf 257 Hz festgelegt.[42]

$$\begin{aligned} \text{Timer 0 Frequenz} &= \frac{Systemtakt frequenz}{\text{Prescaler}} \\ &= \frac{7372800\,\frac{1}{\text{s}}}{256} \\ &= 28800\,\frac{1}{\text{s}} \end{aligned} \tag{2.23}$$

[42] vgl. [Atm07b]

$$\begin{aligned} \text{Abtastrate} &= \frac{\text{Timer 0 Frequenz}}{\text{Zählerschritte bis Überlauf}} \\ &= \frac{28800\,\frac{1}{\text{s}}}{256} \\ &= 112{,}5\,\frac{1}{\text{s}} \end{aligned} \tag{2.24}$$

$$\begin{aligned} \text{Zählerschritte bis Überlauf} &= \frac{\text{Timer 0 Frequenz}}{\text{Abtastrate}} \\ &= \frac{28800\,\frac{1}{\text{s}}}{256} \\ &= 112{,}5\,\frac{1}{\text{s}} \end{aligned} \tag{2.25}$$

```
//Reset timer to get correct sampling frequency.
outb(TCNT0, TIMER0VAL);
```

Die folgenden Anweisungen setzen den Index für die Kanalliste auf 0 zurück, erhöhen den Paketzähler im Datenpuffer um 1 und schreiben in das letzte Byte des Puffers die digitalen Werte von Port D. Diese Werte werden um zwei Stellen nach rechts verschoben um die Werte von PD5 bis PD2 rechtsbündig zu haben. Anschließend wird noch mit *0x0F* maskiert. Das heißt, dass alle bis auf die rechten vier Bits gelöscht werden.

```
CurrentCh = 0;
// Write header and footer:
// Increase packet counter (fourth byte in header)
TXBuf[3]++;
//Get state of switches on PD2..5,
//if any (last byte in packet).
TXBuf[2 * NUMCHANNELS + HEADERLEN] = (inp(PIND) >> 2) &0x0F;
```

Als letztes wird in dieser Interruptroutine, durch löschen und setzen von verschiedenen Interupts, sichergestellt, welche Interruptroutine als nächstes ausgeführt wird. *UDRIE* ist das USART Data Register Empty Interrupt Enable Bit, welches hier gelöscht wird, um einen USART Interrupt zu verhindern. Durch Setzen des *ADIF* auf 1 werden bestehende A/D-Wandler-Interrupts gelöscht und mit dem ADC Interrupt Enable Bit *ADIE* wird ein A/D-Interrupt ausgelöst, der als Anzeige für eine abgeschlossene A/D-Wandlung dient.[43]

```
cbi(UCSRB, UDRIE);          //Ensure UART IRQ's are disabled.
sbi(ADCSR, ADIF);           //Reset any pending ADC interrupts
sbi(ADCSR, ADIE);           //Enable ADC interrupts.
```

### A/D-Wandlung

Die Interruptroutine *SIGNAL(SIG_ADC)* reagiert auf den Interrupt, der eine abgeschlossene A/D-Wandlung anzeigt.[44] Diese Funktion wird immer wieder von neuem aufgerufen, bis das *ADIE* Bit gelöscht wird.

Zunächst wird eine lokale Variable mit Namen *i* deklariert, die zur Indizierung der Datenbytes im Puffer dient. Diese Variable ist nur in dieser Interruptroutine gültig und wird bei jedem Aufruf neu berechnet. Für den ersten Durchlauf hat sie den Wert 4. Mit den folgenden Anweisungen werden die untersten acht Bit des Ergebnisses der letzten A/D-Wandlung in das sechste Byte und die restlichen zwei Bit in das fünfte Byte des Puffers geschrieben. Anschließend wird der Index für die Kanalliste um eins erhöht.

```
volatile uint8_t i;
i = 2 * CurrentCh + HEADERLEN;
TXBuf[i+1] = inp(ADCL);
TXBuf[i] = inp(ADCH);
```

[43] vgl. [Atm07b]
[44] vgl. [AVR07]

```
CurrentCh++;
```

Wenn der Index für die Kanalliste kleiner ist als die Anzahl der Kanäle, wurden noch nicht alle Kanäle verarbeitet. In diesem Fall wird der Wert für den nächsten Kanal in den A/D-Wandler-Multiplexer gespeichert und somit ausgewählt. Sind alle Kanäle verarbeitet worden, so wird wieder der Kanal 0 als erster Kanal ausgewählt und der A/D-Wandler-Interrupt deaktiviert. Dann wird das erste Byte des Puffers in das USART Daten Register *UDR* geschrieben, der USART-Interrupt aktiviert und der Index für den Puffer auf 1 gesetzt. Durch das schreiben der Daten in den das *UDR* Register werden die Daten gesendet.

```
if (CurrentCh < NUMCHANNELS)
{
    //Select the next channel.
    outb(ADMUX, channel_order[CurrentCh]);
    //The next sampling is started automatically.
}
else
{
    //Prepare next conversion, on channel 0.
    outb(ADMUX, channel_order[0]);
    // Disable ADC interrupts to prevent
        // further calls to SIG_ADC.
    cbi(ADCSR, ADIE);
    // Hand over to SIG_UART_DATA, by starting
    // the UART transfer and enabling UDR IRQ's.
    outb(UDR, TXBuf[0]);
    sbi(UCSRB, UDRIE);
    TXIndex = 1;
}
```

### Datenübertragung

In der Interruptroutine für den A/D-Wandler wird der USART Data Register Empty Interrupt aktiviert, durch den die die Interruptroutine für die Datenübertragung aufgerufen wird. Zuerst wird das nächste Byte an Daten übertragen, dann wird der Index für den Puffer um 1 erhöht und danach wird überprüft, ob alle Daten für ein Paket gesendet wurden. Wenn alle Daten gesendet wurden, wird der USART Data Register Empty Interrupt deaktiviert. Durch den nächsten Timer 0 Überlauf wird dann wieder die Interruptroutine für die Abtastrate[45] aufgerufen und der Zyklus beginnt von vorne. Ansonsten wird diese Funktion solange wieder aufgerufen, bis alle Daten gesendet wurden.[46]

```
SIGNAL(SIG_UART_DATA)
{
    outb(UDR, TXBuf[TXIndex]);   //Send next byte
    TXIndex++;
        //See if we're done with this packet
    if (TXIndex == PACKETLEN)
    {
            //Disable SIG_UART_DATA interrupts.
        cbi(UCSRB, UDRIE);
        //Next interrupt will be a SIG_OVERFLOW0.
    }
}
```

## 2.3. Schaltplanänderungen

Neben der Original-Firmware stehen auch die Schaltpläne der Platinen im Internet zum Download zur Verfügung. Die Schaltpläne wurden mit Eagle

[45] siehe Seite 31
[46] vgl. [Atm07b]

gezeichnet. Dieses Programm ist in einer eingeschränkten Version kostenlos im Internet erhältlich. Für die Herstellung einer Platine wird das Platinen-Layout benötigt. Eagle verknüpft den Schaltplan mit dem Layout und sieht vor, dass zuerst der Schaltplan und dann das Layout fertiggestellt wird.

Die Abbildung 2.2 zeigt den Original-Schaltplan, in dem noch der Mikrocontroller AT90S4433P verwendet wird. Dieser Mikrocontroller wurde von Atmel durch den ATmega8 ersetzt. Diese beiden Mikrocontroller haben die gleiche Pin-Konfiguration. In Abbildung 2.3 ist der geänderte Schaltplan mit dem ATmega1280 Mikrocontroller dargestellt. Um Platz für den Mikrocontroller zu schaffen werden die umliegenden Schaltungsteile verschoben.

Die Belegung von J102 wird nicht grundlegend geändert, um die Kompatibilität zum Original-Modular-EEG zu erhalten. Die Zuordnung der Pins in J102 wird an die Pin-Konfiguration des ATmega1280, unter Berücksichtigung der Lage jedes Pin am Gehäuse, angepasst. Die Funktionsweise bleibt aufgrund der folgenden Anpassungen in der Firmware die gleiche. So werden die A/D-Kanäle 10–15 anstelle der Kanäle 0–5 an J102 angeschlossen.

Die Digital-Eingänge PD2–PD5 werden durch PA3–PA6 ersetzt. Bei den digitalen Ausgängen wird die alternative Port-Funktion bei der Auswahl der Pins teilweise mit berücksichtigt. Dies ist besonders wichtig bei PB1, der die alternative Port-Funktion für den Output Compare Pin von Timer 1 hat und das PWM-Signal hier ausgegeben wird. Die alternativen Port-Funktionen werden in der Pin-Konfiguration in Klammern dargestellt. Die Pin-Konfiguration des ATmega8 ist in Abbildung 2.4, die des ATmega1280 in Abbildung 2.5 zu sehen. PB0 und PB2 werden aufgrund der alternativen Port-Funktionen *ICP1* und *OC1B* anderen Ports zugewiesen, die die selbe alternative Funktion haben. PB0 wird PD4, PB1 wird PB5 und PB2 wird PB6. Bei den anderen Digital-Ausgängen, die mit J102 verbunden sind, kann irgend ein Port verwendet werden. PD7 wird PD5.

Die Programmier-Schnittstelle J101 wird entsprechend der alternativen Port-Funktionen von *MISO*, *MOSI*, *SCK* und */RESET* mit PB1–PB3 und dem

Reset-Pin verbunden.

Eine weitere Schnittstelle *J103* wird eingefügt um die erweiterten A/D-Kanäle anschließen zu können. Dabei ist berücksichtigt, dass die Kanäle 0–5 über ein Flachbandkabel mit drei Analog-Platinen verbunden werden kann und dabei auch die Versorgungsspannungen für die Analog-Platinen mit geführt werden. Es bleiben noch vier Kanäle übrig, die auf zwei Analog-Platinen verteilt werden können. Dabei ist darauf zu achten, dass bei dem Verbindungskabel, auf der Seite der Analog-Platinen eine Reihe zwischen ADC7 und ADC8 frei bleiben muss, um die Zuordnung der Kanäle zu den Analog-Platinen nach dem bisherigen Schema beizubehalten.

Für jeden A/D-Kanal ist ein RC-Tiefpass vorhanden.

## 2.4. Platinenlayout

Da die kostenlose Eagle-Version nur Platinengrößen bis 10 cm x 8 cm erlaubt, werden die Maße hier nicht überschritten. Dadurch wird Sichergestellt, dass andere Entwicklern im OpenEEG-Projekt die Möglichkeit haben das Platinen-Layout zu ändern. Die Original-Platine erfüllt bereits diese Maximal-Maße.

Aus Platzgründen wurden einige Widerstände und Kondensatoren durch SMD-Bauteile ersetzt. Dies sind alle 470 Ω Vorwiderstände an J102, alle 7,5 kΩ Widerstände für den RC-Tiefpass an den A/D-Kanälen und ein 10 kΩ Widerstand. Die ersetzten Kondensatoren, mit 220 nF sind diejenigen, welche zum RC-Tiefpass gehören.

Die folgenden Abbildungen zeigen die Veränderung des Platinenlayouts. Abbildung 2.6 zeigt das Original-Layout von der Oberseite. Entsprechend ist in Abbildung 2.7 das neue Layout zu sehen.

Die selbe Reihenfolge für die Unterseite, erst das Original in Abbildung 2.8, dann das neue Layout in Abbildung 2.9.

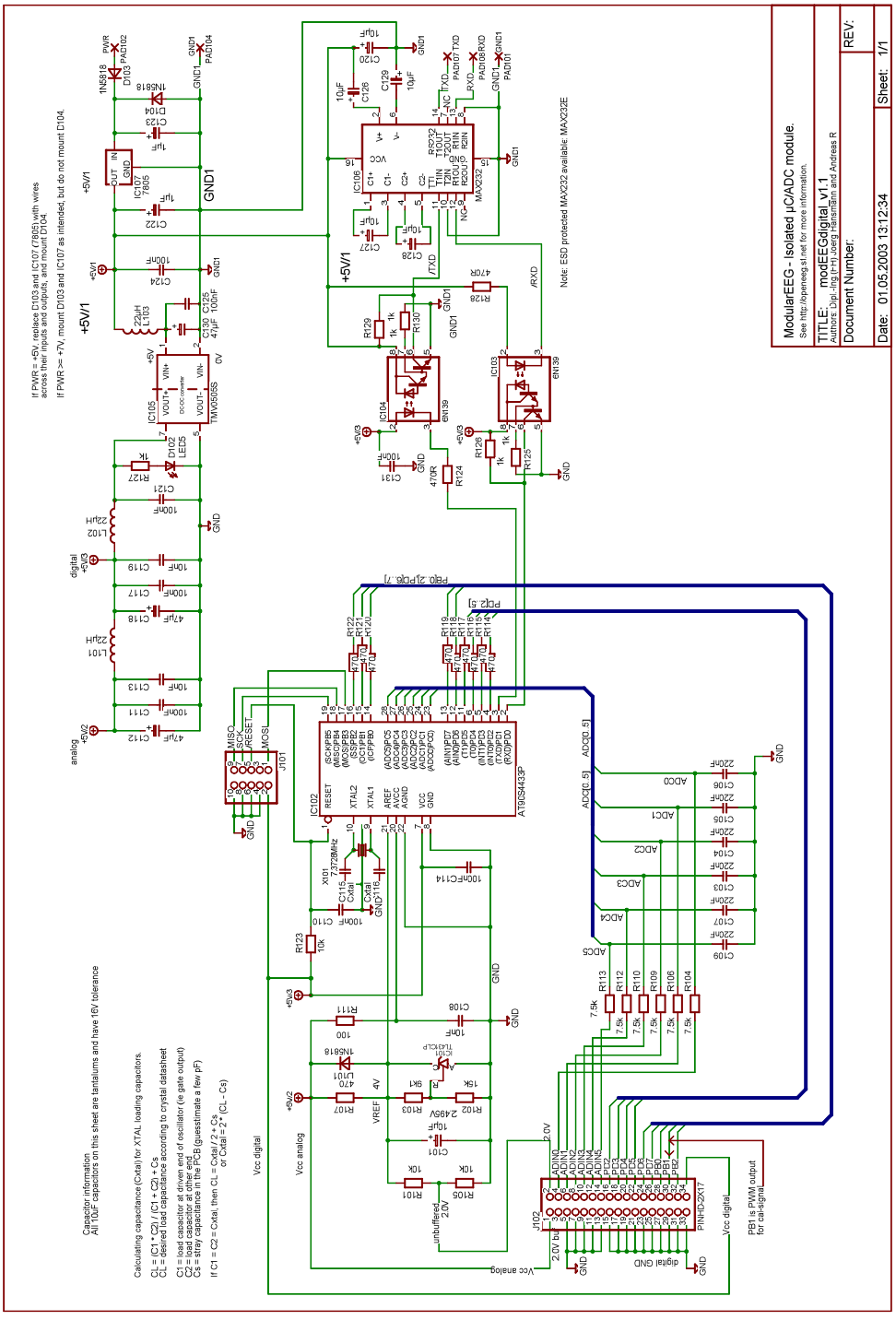


Abbildung 2.2.: Original-Schaltplan der Digital-Platine des Modular-EEG (Quelle: [Pro])

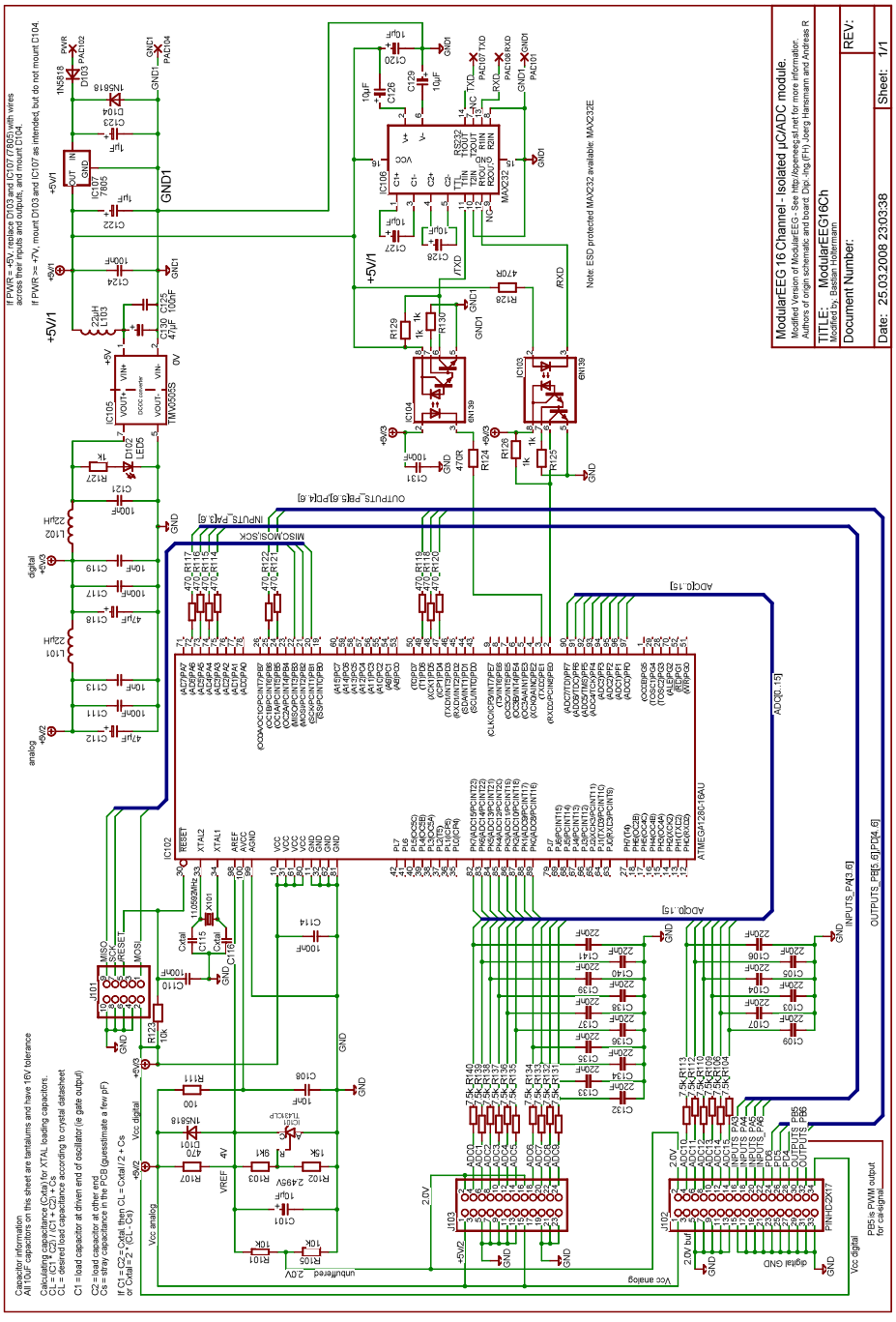


Abbildung 2.3.: Schaltplan mit ATmega1280

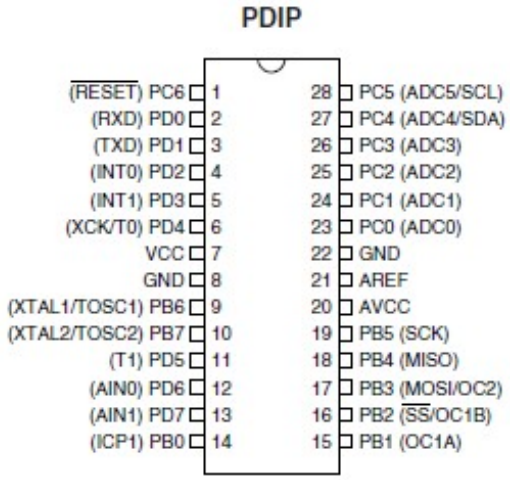


Abbildung 2.4.: Pin-Konfigurartion des ATmega8 (Quelle: [Atm07b])

## 2.5. Bauteile

Da der ATmega1280 mit einem Abstand von 5 mil zwischen den Pins sehr filigran ist, hat sich bei der Bestellung der Platine herausgestellt, das nicht jede Firma eine solche Platine fertigen kann. Die meisten Firmen geben einen Mindestabstand von 6 mil für die Fertigung an. Ein mil entspricht einem tausendstel inch, also 0,0254 mm.

Die leere Platine ist in den Abbildungen 2.10 und 2.11 zu sehen.

Im Internet [47] kann ein Archiv heruntergeladen werden, in dem neben den Schaltplänen auch Bauteilelisten enthalten sind. In den Listen sind auch die Bestellnummern von mehreren Lieferanten hinterlegt.

Die Bauteile, die nicht in den Listen aus dem Projekt stehen sind die SMD-Bauteile. Nach Rücksprache mit Joerg Hansmann[48] werden für die SMD-Widerstände 0,125 Watt-Typen in der Bauform 0805 verwendet. Die

[47] http://sourceforge.net/project/showfiles.php?group_id=35817
[48] einem der Entwickler des Modular-EEG

**Figure 1-1.** TQFP-pinout ATmega640/1280/2560

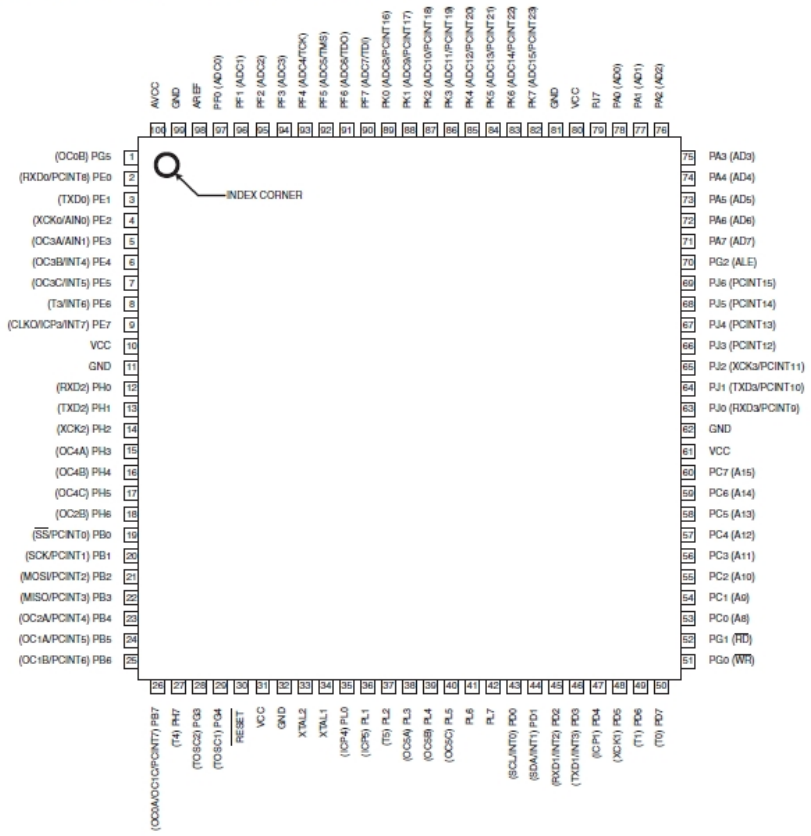


Abbildung 2.5.: Pin-Konfigurartion des ATmega1280 (Quelle: [Atm07a])

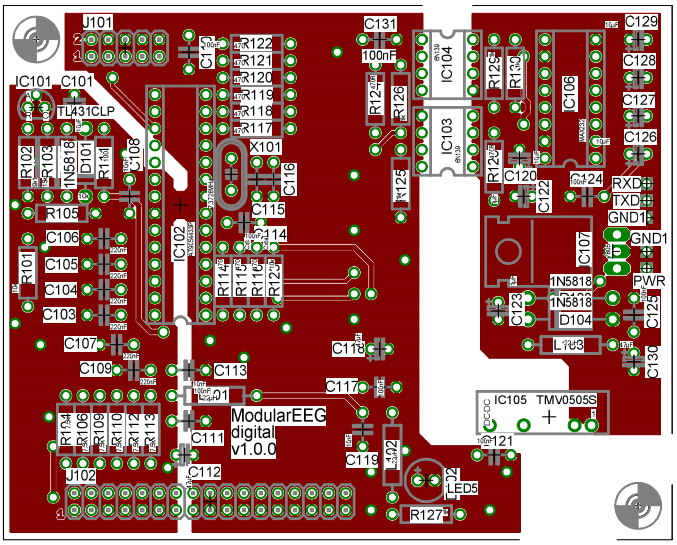


Abbildung 2.6.: Oberseite des Original-Layout (Quelle: [Pro])

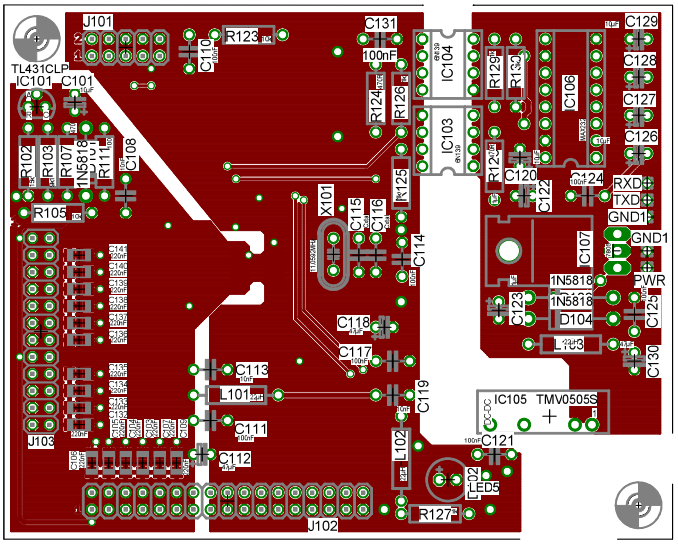


Abbildung 2.7.: Oberseite des Layout für ATmega1280

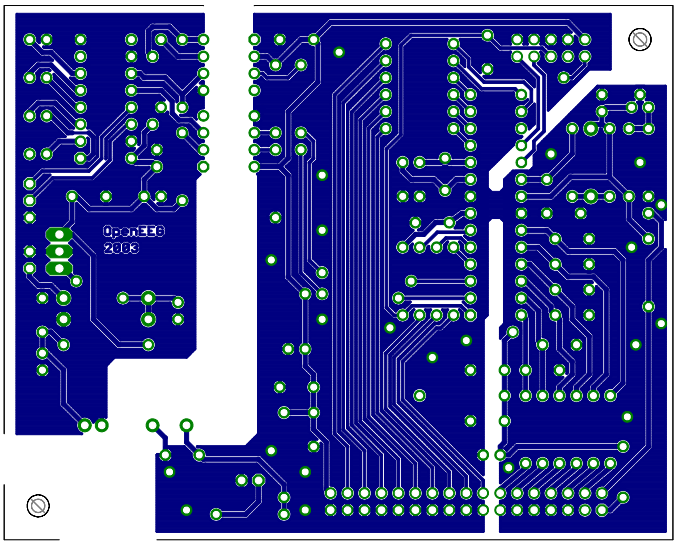


Abbildung 2.8.: Unterseite des Original-Layout (Quelle: [Pro])

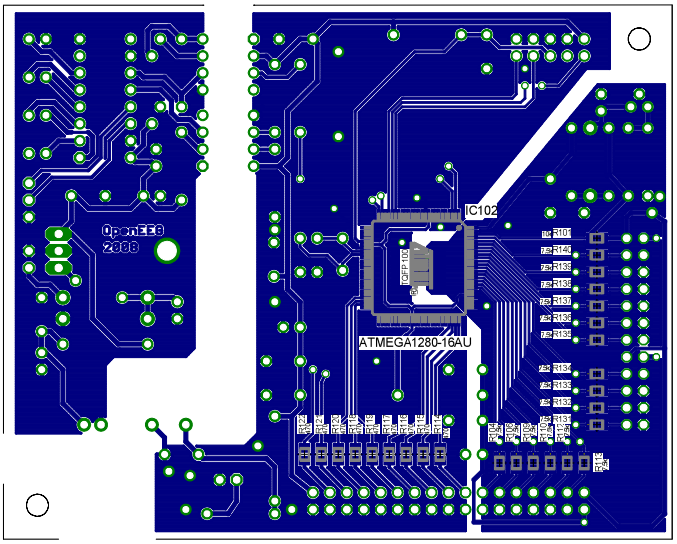


Abbildung 2.9.: Unterseite des Layout für ATmega1280

Kondensatoren sind in der Bauform 1206. Für den Mikrocontroller kann ein ATmega1280, ein ATmega640 oder ein ATmega2560 verwendet werden.

Die Abbildungen 2.12 und 2.13 zeigen die bestückte Platine.

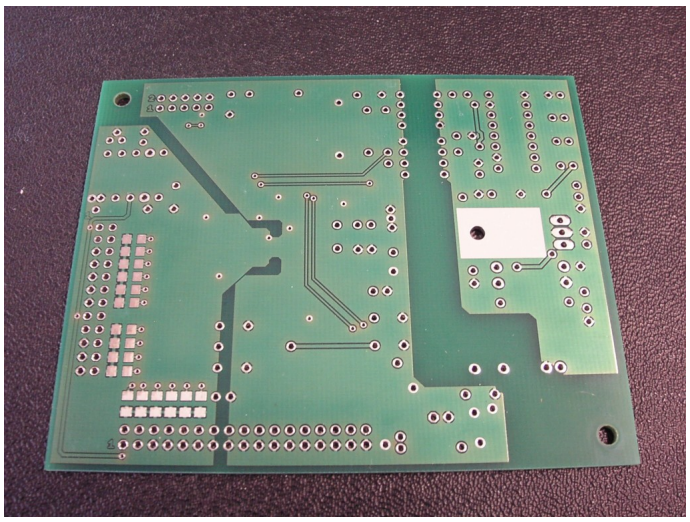

Abbildung 2.10.: Oberseite der leeren Platine

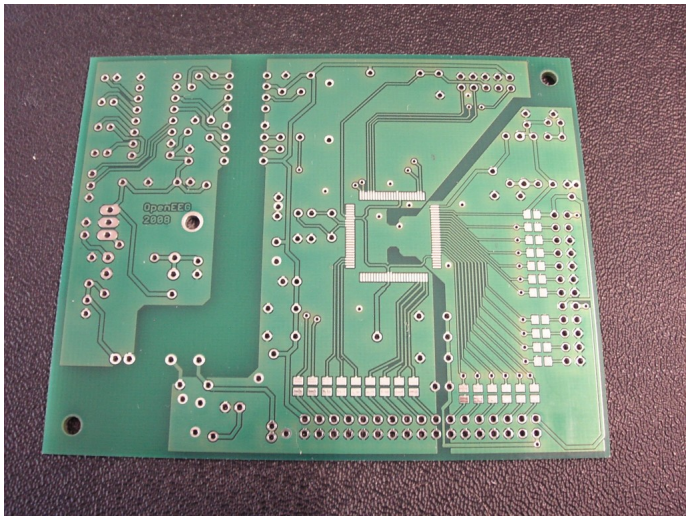


Abbildung 2.11.: Unterseite der leeren Platine

## 2.6. Firmware

Zunächst wird die Firmware für das 6-Kanal-EEG für den ATmega1280 geändert. Dies hat den Vorteil, dass die Platine mit der bisherigen Software aus dem OpenEEG-Projekt getestet werden kann. Anschließend wird die Firmware für 16-Kanal Betrieb angepasst.

### 2.6.1. Firmware für 6-Kanal-EEG mit ATmega1280

Die Änderung der Firmware erfolgt in mehreren Schritten. Im ersten Schritt werden alle Bezeichner an den ATmega1280 angepasst. Dazu gehören zum Beispiel die Bezeichnungen der Register, der Bits in den Registern und der Interrupts. Der zweite Schritt wird dadurch notwendig, dass der aktuelle AVR-Compiler AVR-GCC v4.2.2 eine Vielzahl von Fehlern anzeigt. Ein Beispiel für einen solchen Fehler sieht so aus: *undefined reference to 'outb'*. Diese Fehler kommen durch Anweisungen zustande, die in Programmen

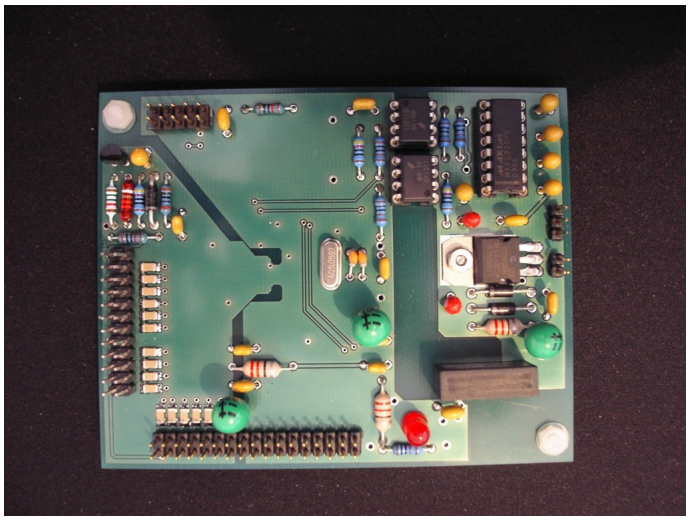

Abbildung 2.12.: Oberseite der bestückten Platine

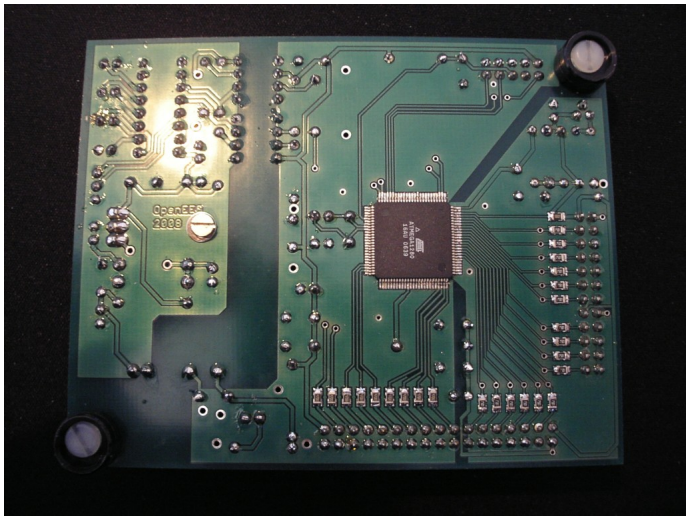

Abbildung 2.13.: Unterseite der bestückten Platine

nicht mehr verwendet werden sollen.
Die Anweisungen, die der Compiler bemängelt, sind:

```
sbi(Register, Bit);
cbi(Register, Bit);
inp(Register);
outb(Ziel, Wert);
```

Um die Änderungen am Code deutlich sichtbar zu machen, wird pro Änderung zuerst der Code aus der Original Firmware für den ATmega8 und darunter der Code für die Firmware für den ATmega1280 abgebildet. Für den ATmega1280 sind jeweils zwei Zeilen Code angegeben, wobei die obere, auskommentierte Zeile, aus der Änderung im ersten Schritt stammt und die zweite Zeile den endgültigen Code beinhaltet. Die Reihenfolge der Änderungen entspricht, wie bei der Analyse der Firmware, der Reihenfolge des Programmablaufs. Eine detaillierte Beschreibung der Firmware befindet sich im Abschnitt Analyse der Firmware. Die komplette 6-Kanal-EEG-Firmware für den ATmega1280 befindet sich im Anhang. Die Zeilen für die Kompatibilität mit dem AT90S4434 werden hier nicht berücksichtigt.

#### Definitionen

Die Headerdatei *avr/signal.h* soll laut Compiler nicht mehr verwendet werden. Statt dessen soll *avr/interrupt.h* eingebunden werden. Da diese schon eingebunden ist, reicht es, die Anweisung

```
#include <avr/signal.h>
```

auszukommentieren.

```
//#include <avr/signal.h>
```

Die Berechnung der Konstanten TIMER0VAL beinhaltet die Taktfrequenz des Mikrocontrollers. Da der ATmega1280 hier mit 11,0592 MHz und nicht

wie der ATmega8 mit 7,372800 MHz getaktet wird, muss der Wert 7372800 in

```
#define TIMER0VAL 256 - ((7372800 / 256) / SAMPFREQ)
```

auf 11059200 geändert werden. TIMER0VAL ist der Wert, der für den Timer 0 eingestellt wird, um eine Abtastrate von 256 Hz zu generieren.

```
#define TIMER0VAL 256 - ((11059200 / 256) / SAMPFREQ)
```

Die Reihenfolge der Kanäle wird dahingehend angepasst, dass nun die A/D-Kanäle 10–15 an J102 angeschlossen sind und nicht mehr die Kanäle 0–5. Diese Änderung hat sich durch das Platinenlayout ergeben.
Alt:

```
char const channel_order[]= { 0, 3, 1, 4, 2, 5 };
```

Neu:

```
char const channel_order[]= { 10, 13, 11, 14, 12, 15 };
```

Die Kanalreihenfolge entspricht der Kanalzuordnung auf den Analog-Platinen. Kanal 10 und Kanal 13 entsprechen zwei Kanälen auf der selben Analog-Platine.

### Main-Funktion

Da der ATmega8 nur drei Ports und der ATmega1280 elf Ports hat, muss die Konfiguration der Ports in der Firmware erweitert werden. Die Ports haben bei den beiden Mikrocontrollern nicht die gleichen Funktionen, so dass bei der Konfiguration auf den jeweiligen Schaltplan zurückzugreifen ist.

Atmel empfiehlt für nicht angeschlossene Pins den internen Pull-up-Widerstand zu aktivieren, damit der Pin einen definierten Zustand erhält. Durch das Setzen aller Bits auf 1 im Port Register des Ports wird für alle Pins, die durch das Data Direction Register als Eingang gekennzeichnet sind, der

interne Pull-up-Widerstand aktiviert.[49]

Port A existiert beim ATmega8 nicht und hat hier die Funktion für die vier digitalen Eingänge von J102. Es können somit alle Pins von Port A als Eingang konfiguriert werden. Die Data Direction Register werden mit 0 initialisiert. Aus Gründen der Übersicht erfolgt hier dennoch die explizite Anweisung, das Data Direction Register auf 0 zu setzen.

```
// outb(DDRA, 0x00);
DDRA = 0x00;
```

An Port B werden PB5 und PB6 als Ausgänge verwendet. PB5 dient hierbei als Ausgang für das PWM-Kalibrierungssignal. Weiterhin sind die Signale für die Programmierung (SCK, MOSI, MISO) an Port B Pins 1–3 angeschlossen. Diese werden hier, analog zu der Firmware des ATmega8, als Eingang konfiguriert.

```
// outb(DDRB, 0x60);
DDRB = 0x60;
```

Die Ports C, G, H, J und L sind gar nicht mit der Schaltung verbunden und sollten als Eingänge konfiguriert werden, damit die Pull-up-Widerstände aktivierbar sind.

```
// outb(DDRC, 0x00);
DDRC = 0x00;
// outb(DDRG, 0x00);
DDRG = 0x00;
// outb(DDRH, 0x00);
DDRH = 0x00;
// outb(DDRJ, 0x00);
DDRJ = 0x00;
// outb(DDRL, 0x00);
DDRL = 0x00;
```

[49] vgl. [Atm07a]

Die Pins PD4 bis PD6 sind als weitere digitale Ausgänge vorgesehen. Die anderen Pins von Port D sind nicht angeschlossen.

```
//outb(DDRD, 0x70);
DDRD = 0x70;
```

Port E wird nur für die serielle Datenübertragung verwendet. Pin 0 ist für RxD und als Eingang zu konfigurieren. Pin 1 ist für TxD vorgesehen und somit Ausgang. Andere Pins sind nicht angeschlossen.

```
//outb(DDRE, 0x02);
DDRE = 0x02;
```

Die Ports F und K dienen als analoge Eingänge für den A/D-Wandler. Beim ATmega8 ist das der Port C, der nicht explizit konfiguriert wird. An dieser Stelle wird ebenfalls darauf verzichtet.

Nun folgen die Anweisungen für die Aktivierung der Pull-up-Widerstände bei den Pins, die als Eingang konfiguriert sind.

```
//outb(PORTA, 0xff);
PORTA = 0xff;
//outb(PORTB, 0xff);
PORTB = 0xff;
//outb(PORTC, 0xff);
PORTC = 0xff;
//outb(PORTD, 0xff);
PORTD = 0xff;
//outb(PORTE, 0xff);
PORTE = 0xff;
//outb(PORTG, 0xff);
PORTG = 0xff;
//outb(PORTH, 0xff);
PORTH = 0xff;
```

```
//outb(PORTJ, 0xff);
PORTJ = 0xff;
//outb(PORTL, 0xff);
PORTL = 0xff;
```

Die Auswahl und die Aktivierung der Sleep-Modi erfolgt beim ATmega1280 über das Sleep Mode Control Register (SMCR) und nicht über das MCU Control Register (MCUCR). Die Anweisung

```
outb(MCUCR,(inp(MCUCR) | BV(SE)) &
                        (~BV(SM0) | ~BV(SM1) | ~BV(SM2)));
```

muss abgewandelt werden in

```
//outb(SMCR,(inp(SMCR) | BV(SE)) &
                        //(~BV(SM0) | ~BV(SM1) | ~BV(SM2)));
SMCR |= (1 << SE);
SMCR &= ~((1 << SM0) | (1 << SM1) | (1 << SM2));
```

.

Die Anweisung

```
outb(ADMUX, 0);             //Select channel 0
```

wählt Kanal 0 als aktuellen Eingang für den A/D-Wandler aus. Da in dieser Arbeit die ersten Kanäle, die jedoch die Kanäle 10–15 sind, und der ATmega1280 die Kanäle mit 5 Bits anstelle von 3 Bits adressiert, wird die Anweisung durch folgende ersetzt.

```
//Select channel 10
//outb(ADMUX, (inp(ADMUX) | BV(MUX1)) &
          //(~BV(MUX0) | ~BV(MUX2) | ~BV(MUX3) | ~BV(MUX4));
ADMUX |= (1 << MUX1);
ADMUX &=
    ~((1 << MUX0) | (1 << MUX2) | (1 << MUX3) | (1 << MUX4));
//outb(ADCSRB,(inp(ADCSRB) | BV(MUX5)));
```

```
ADCSRB |= (1 << MUX5);
```

Der ATmega8 verfügt nur über das ADC Control and Status Register A (ADCSRA), welches mit ADCSR angesprochen werden kann. Die folgenden Anweisungen

```
outb(ADCSR, BV(ADPS2) | BV(ADPS1));
sbi(ADCSR, ADIF);          //Reset any pending ADC interrupts
sbi(ADCSR, ADEN);          //Enable the ADC
```

werden hier jeweils durch ein A ergänzt. Der Prescaler bleibt gleich.

```
//outb(ADCSRA, BV(ADPS2) | BV(ADPS1));
ADCSRA |= (1 << ADPS2) | (1 << ADPS1);
//sbi(ADCSRA, ADIF);       //Reset any pending ADC interrupts
ADCSRA |= (1 << ADIF);     //Reset any pending ADC interrupts
//sbi(ADCSRA, ADEN);       //Enable the ADC
ADCSRA |= (1 << ADEN);     //Enable the ADC
```

Da der ATmega1280 über vier USARTs und der ATmega8 nur über einen verfügt müssen die Registernamen für den USART mit seiner Nummer ergänzt werden. In dieser Arbeit wird der USART 0 verwendet. Durch die Anweisungen

```
outb(UBRRH,0);                  //Set speed to 57600 bps
outb(UBRRL,7);
```

wird bei einer Frequenz von 7,372800 MHz die Datenübertragungsrate auf 57600 bps festgelegt. Bei der hier verwendeten Frequenz von 11,0592 MHz würde die Datenübertragungsrate bei gleicher Prescaler-Einstellung 86400 bps betragen. Mit folgenden Anweisungen wird der USART-Prescaler wieder auf 57600 bps eingestellt.

```
//outb(UBRR0H,0x00);    //Set speed to 57600 bps
UBRR0H = 0x00;          //Set speed to 57600 bps
//outb(UBRR0L,0x0b);
```

```
UBRR0L = 0x0b;
```

In den USART Control and Status Registern A–C (UCSRA bis UCSRC) sind auch die Bezeichnungen der Bits mit der Nummer des USART zu ergänzen. Das USART Register Select Bit (URSEL) entfällt beim ATmega1280, da hier die USART-Register jeweils eine eigene Adresse haben. Beim ATmega8 haben sich das UCSRC Register und das UBRRH Register eine Adresse geteilt. Die Auswahl, welches Register gerade bearbeitet wird, erfolgt dabei über das URSEL Bit.

```
outb(UCSRA, 0);
outb(UCSRC, BV(URSEL) | BV(UCSZ1) | BV(UCSZ0));
outb(UCSRB, BV(TXEN));
```

wird geändert in:

```
//outb(UCSR0A, 0);
UCSR0A = 0x00;
//outb(UCSR0C, BV(UCSZ01) | BV(UCSZ00));
UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);
//outb(UCSR0B, BV(TXEN0));
UCSR0B |= (1 << TXEN0);
```

Die Adresse des Timer/Counter-Register für Timer 0 bleibt gleich, aber der *outb()* Anweisungen sollte ersetzt werden.

```
//outb(TCNT0, 0);     //Clear it.
TCNT0 = 0x00;         //Clear it.
```

Der ATmega1280 hat zwei Timer/Counter Control Register pro Timer. Für Timer 0 sind das TCCR0A und TCCR0B. Die Geschwindigkeit des Timer 0 wird durch die Clock Select Bits CS00–CS02 bestimmt. Diese befinden sich im Register TCCR0B.

```
outb(TCCR0, 4);          //Start it. Frequency = clk / 256
```

```
//outb(TCCR0B, 4);        //Start it. Frequency = clk / 256
TCCR0B = (1 << CS02);     //Start it. Frequency = clk / 256
```

Das Timer/Counter Interrupt Mask Register (TIMSK) heißt beim ATmega1280 TIMSK0.

```
outb(TIMSK, BV(TOIE0));       //Enable the interrupts.
```

```
//outb(TIMSK0, BV(TOIE0));    //Enable the interrupts.
TIMSK0 |= (1 << TOIE0);       //Enable the interrupts.
```

### PWM Initialisierung

In der pwm_init Funktion müssen alle Anweisungen ersetzt werden, da diese veraltet sind. Die veralteten Zeilen sind hier auskommentiert dargestellt. Zusätzlich sei hier kurz erwähnt, dass sich die Frequenz des Kalibrierungssignals durch die Erhöhung der Taktfrequenz des Mikrocontrollers von ca. 14 Hz auf 21,11 Hz erhöht.

```
//outb(OCR1AH,2);     // Set OCR1A = 512
OCR1AH = 0x02;        // Set OCR1A = 512
//outb(OCR1AL,0);
OCR1AL = 0x00;
// Set 10-bit PWM mode
//outb(TCCR1A, BV(COM1A1) + BV(WGM11) + BV(WGM10));
// Set 10-bit PWM mode
TCCR1A |= (1 << COM1A1) | (1 << WGM11) | (1 << WGM10);
// Start and let run at clk / 256 Hz.
//outb(TCCR1B, (1 << CS12));
// Start and let run at clk / 256 Hz.
TCCR1B |= (1 << CS12);
```

### Interruptroutine für die Abtastrate

Die Funktion SIGNAL(SIG_OVERFLOW0) muss aufgrund der Änderung der Interruptnamen umbenannt werden.

```
SIGNAL(SIG_OVERFLOW0)
```

```
//SIGNAL(SIG_OVERFLOW0)
ISR(TIMER0_OVF_vect)
```

Diese Funktion wird durch den Timer 0 ausgelöst, der für die Abtastrate von 256 Hz zuständig ist.

Hier wieder eine veraltete Anweisung:

```
//Reset timer to get correct sampling frequency.
//outb(TCNT0, TIMER0VAL);
//Reset timer to get correct sampling frequency.
TCNT0 = TIMER0VAL;
```

Die Auswahl des aktuellen Kanals wird wieder auf den ersten Kanal zurückgesetzt. Die Zeile

```
CurrentCh = 0;
```

muss nicht geändert werden, da CurrentCh zur Indizierung der Kanalnummer in channel_order[] verwendet wird.

Die vier digitalen Eingänge, deren Zustand mit zum PC übertragen wird, befinden sich nicht mehr an Port D 2–5, sondern an Port A 3–6. Die Verschiebung der Bits muss auch angepasst werden, um die Werte ganz nach rechts zu verschieben und korrekt zu maskieren.

```
//Get state of switches on PD2..5,
//if any (last byte in packet).
TXBuf[2 * NUMCHANNELS + HEADERLEN] = (inp(PIND) >> 2) &0x0F;
```

```
//Get state of switches on PA3..6,
//if any (last byte in packet).
//TXBuf[2 * NUMCHANNELS + HEADERLEN] = (inp(PINA) >> 3) &0x0F;
TXBuf[2 * NUMCHANNELS + HEADERLEN] = (PINA >> 3) &0x0F;
```

Die Adresse des USART-Interrupts ist nun in Register UCSR0B und heißt UDRIE0.

```
cbi(UCSRB, UDRIE);              //Ensure UART IRQ's are disabled.
```

```
//cbi(UCSR0B, UDRIE0);          //Ensure UART IRQ's are disabled.
UCSR0B &= ~(1 << UDRIE0);       //Ensure UART IRQ's are disabled.
```

Das ADC Control and Status Register (ADCSR) heißt nun ADCSRA.

```
sbi(ADCSR, ADIF);               //Reset any pending ADC interrupts
sbi(ADCSR, ADIE);               //Enable ADC interrupts.
```

```
//sbi(ADCSRA, ADIF);            //Reset any pending ADC interrupts
ADCSRA |= (1 << ADIF);          //Reset any pending ADC interrupts
//sbi(ADCSRA, ADIE);            //Enable ADC interrupts.
ADCSRA |= (1 << ADIE);          //Enable ADC interrupts.
```

#### Interruptroutine für die A/D-Wandlung

Der Name für diese Interruptroutine sollte ebenfalls geändert werden.

```
SIGNAL(SIG_ADC)
```

```
//SIGNAL(SIG_ADC)
ISR(ADC_vect)
```

Hier ein paar Anweisungen, deren Adressen zwar stimmen, aber die aufgrund der Compiler-Fehlermeldungen ersetzt werden.

```
//TXBuf[i+1] = inp(ADCL);
TXBuf[i+1] = ADCL;
//TXBuf[i] = inp(ADCH);
TXBuf[i] = ADCH;
```

Wie oben erwähnt, werden für die Adressierung der A/D-Kanäle beim ATmega1280 fünf Bits verwendet. Die Tabelle 2.2 ist aus der Tabelle 26-4 aus [Atm07a] abgeleitet. Sie stellt für jeden Kanal das Bitmuster dar, welches in den Bits MUX0 bis MUX5 eingetragen werden muss. Die letzte Spalte beinhaltet den dezimalen Wert, der direkt in ADMUX geschrieben werden kann, um die Bits MUX0 bis MUX4 richtig einzustellen. Wie in der Tabelle zu sehen ist, muss das Bit MUX5 für die Kanäle 0–7 auf 0 und für die Kanäle 8–15 auf 1 gesetzt werden. Die drei Bits MUX0, MUX1 und MUX2 werden binär bis sieben hochgezählt und starten dann wieder von 0.

In dieser 6-Kanal Version werden die Kanäle 10–15 verwendet. Der Wert in ADMUX für den Kanal 10 muss aber 2 betragen und nicht 10. Daher kann die folgende Anweisung nicht mehr verwendet werden.

```
//Select the next channel.
//outb(ADMUX, channel_order[CurrentCh]);
//Select the next channel.
ADMUX = channel_order[CurrentCh];
```

Das Bit MUX5 im Register ADCSRB wird in der Main-Funktion auf 1 gesetzt und braucht hier nicht geändert zu werden. Für eine korrekte Auswahl der Kanäle wird von der Kanalnummer, die in *channel_order[]* steht, acht subtrahiert.

```
//Select the next channel.
//The right value for ADMUX e.g. for Channel 10 is 2
//and not 10, so 8 need to be sutracted
ADMUX = (channel_order[CurrentCh] - 8);
```

Das Gleiche gilt auch für die folgenden Anweisung.

Tabelle 2.2.: Bitmuster für die A/D-Wandler Kanalauswahl mit den Bits MUX0-MUX5 [Atm07a]

| Kanal | MUX5 | MUX4 | MUX3 | MUX2 | MUX1 | MUX0 | ADMUX Dezimal |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 | 2 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 | 3 |
| 4 | 0 | 0 | 0 | 1 | 0 | 0 | 4 |
| 5 | 0 | 0 | 0 | 1 | 0 | 1 | 5 |
| 6 | 0 | 0 | 0 | 1 | 1 | 0 | 6 |
| 7 | 0 | 0 | 0 | 1 | 1 | 1 | 7 |
| 8 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 10 | 1 | 0 | 0 | 0 | 1 | 0 | 2 |
| 11 | 1 | 0 | 0 | 0 | 1 | 1 | 3 |
| 12 | 1 | 0 | 0 | 1 | 0 | 0 | 4 |
| 13 | 1 | 0 | 0 | 1 | 0 | 1 | 5 |
| 14 | 1 | 0 | 0 | 1 | 1 | 0 | 6 |
| 15 | 1 | 0 | 0 | 1 | 1 | 1 | 7 |

```
//Prepare next conversion, on channel 0.
//outb(ADMUX, channel_order[0]);
```

Da hier wie in der Main-Funktion der Kanal 10 ausgewählt wird, kann die Anweisung aus der Main-Funktion verwendet werden.

```
//Prepare next conversion, on channel 10.
ADMUX |= (1 << MUX1);
ADMUX &=
     ~((1 << MUX0) | (1 << MUX2) | (1 << MUX3) | (1 << MUX4));
ADCSRB |= (1 << MUX5);
```

Auch hier muss die Adresse des ADC Control and Status Registers von ADCSR auf ADCSRA geändert werden:

```
cbi(ADCSR, ADIE);
```

```
//cbi(ADCSRA, ADIE);
ADCSRA &= ~(1 << ADIE);
```

Die Adressen der USART Register und Bits müssen wieder mit der Nummer des USART vervollständigt werden.

```
outb(UDR, TXBuf[0]);
sbi(UCSRB, UDRIE);
```

```
//outb(UDR0, TXBuf[0]);
UDR0 = TXBuf[0];
//sbi(UCSR0B, UDRIE0);
UCSR0B |= (1 << UDRIE0);
```

#### Interruptroutine für die Datenübertragung

Der Name der dritten Interruptroutine wird analog zu den anderen geändert.

```
SIGNAL(SIG_UART_DATA)
```

```
//SIGNAL(SIG_UART_DATA)
ISR(USART0_UDRE_vect)
```

Hier sind die Adressen der USART Register und Bits anzupassen sowie die Anweisungen zu ersetzen.
Alt:

```
outb(UDR, TXBuf[TXIndex]);   //Send next byte
//Disable SIG_UART_DATA interrupts.
cbi(UCSRB, UDRIE);
```

Neu:

```
//outb(UDR0, TXBuf[TXIndex]);   //Send next byte
UDR0 = TXBuf[TXIndex];          //Send next byte
//Disable SIG_UART_DATA interrupts.
//cbi(UCSR0B, UDRIE0);
//Disable SIG_UART_DATA interrupts.
UCSR0B &= ~(1 << UDRIE0);
```

### 2.6.2. Firmware für 16-Kanal-EEG mit ATmega1280

In diesem Abschnitt werden nur noch die Änderungen an der Firmware dargestellt, die nötig sind, um die vorhergehende 6-Kanal-Firmware für den ATmega1280 für 16 Kanäle zu erweitern. Die Änderungen sind wieder in der Reihenfolge des Programmablaufs angegeben. Die Kommentar-Zeilen mit den alten Anweisungen werden in dieser Version der Firmware gelöscht und nicht weiter behandelt.

#### Definitionen

Folgende Definitionen müssen geändert werden:

```
#define NUMCHANNELS 6
#define HEADERLEN 4
#define PACKETLEN (NUMCHANNELS * 2 + HEADERLEN + 1)
char const channel_order[]= { 10, 13, 11, 14, 12, 15 };
```

Die Anzahl der zu verarbeitenden Kanäle wird der Konstanten *NUMCHANNELS* zugewiesen. Die sechs muss also durch eine 16 ersetzt werden.

Die Länge des Paketheaders *HEADERLEN* kürzt sich durch das Weglassen der Angabe der Paketversion um ein Byte von vier auf drei.

Die Paketlänge *PACKETLEN* hat sich durch die Komprimierung der Daten verkürzt. In der Original-Firmware werden pro Kanal zwei Byte übertragen.[50] Das entspräche bei 16 Kanälen mit Header und Footer einer Paketlänge von 36 Byte. In dieser Version teilen sich zwei Kanäle drei Byte. Dadurch wird die Paketlänge auf 28 Byte gekürzt.

Die Liste mit der Kanalreihenfolge *channel_order[]* muss um weitere Kanäle ergänzt werden. Die Reihenfolge entspricht diesmal nicht der Zuordnung zu den Analog-Platinen, da in der Testphase so die Kanäle besser identifiziert werden können.

```
#define NUMCHANNELS 16
#define HEADERLEN 3
#define PACKETLEN (((NUMCHANNELS/2)*3) + HEADERLEN + 1)
char const channel_order[]=
   { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
```

### Main-Funktion

Die Anweisung, der die Paketversion oder Protokollnummer in den Textpuffer schreibt, wird gelöscht. Um Zeit bei der Datenübertragung zu sparen, wird dieses Byte nicht mehr verwendet. Dadurch verschiebt sich der Inhalt

[50] vgl. [HPR03]

des Textpuffers um ein Byte nach vorne. Entsprechend müssen die Anweisungen, die *TXBuf[]* verwenden, angepasst werden.
6-Kanal:

```
TXBuf[2] = 2;              //Protocol version
TXBuf[3] = 0;              //Packet counter
```

16-Kanal:

```
TXBuf[2] = 0x00;           //Packet counter
```

In der Main-Funktion muss als erster Kanal für den A/D-Wandler der Kanal 0 ausgewählt werden.
6-Kanal:

```
//Select channel 10
ADMUX |= (1 << MUX1);
ADMUX &=
    ~((1 << MUX0) | (1 << MUX2) | (1 << MUX3) | (1 << MUX4));
ADCSRB |= (1 << MUX5);
```

16-Kanal:

```
//Select channel 0
ADMUX &= ~((1 << MUX0) | (1 << MUX1) | (1 << MUX2) |
                                    (1 << MUX3) | (1 << MUX4));
ADCSRB &= ~(1 << MUX5);
```

Die Datenübertragungsrate für den USART muss auf 115200 bps erhöht werden. Der Wert fünf für UBRR0 stammt aus der Tabelle 22-11 in [Atm07a].
6-Kanal:

```
UBRR0H = 0x00;             //Set speed to 57600 bps
UBRR0L = 0x0b;
```

16-Kanal:

```
UBRR0H = 0x00;             //Set speed to 115200 bps
```

```
UBRR0L = 0x05;
```

### Interruptroutine für die Abtastrate

In *SIGNAL(TIMER0_OVF_vect)* müssen folgende Anweisungen für die Anpassung an das neue Paketformat geändert werden.
6-Kanal:

```
// Increase packet counter (fourth byte in header)
TXBuf[3]++;
//Get state of switches on PA3..6,
//if any (last byte in packet).
TXBuf[2 * NUMCHANNELS + HEADERLEN] = (PINA >> 3) &0x0F;
```

16-Kanal:

```
// Increase packet counter (third byte in header)
TXBuf[2]++;
//Get state of switches on PA3..6,
//if any (last byte in packet).
TXBuf[((NUMCHANNELS/2)*3) + HEADERLEN] = (PINA >> 3) &0x0F;
```

### Interruptroutine für die A/D-Wandlung

Die Anweisungen für das Füllen des Textpuffers aus dem Speicher des A/D-Wandlers müssen für die Komprimierung geändert werden. Die Tabelle 2.3 zeigt den Inhalt des Textpuffers ohne Komprimierung. Dort ist deutlich zu sehen, dass pro Kanal sechs Bit frei bleiben. In Tabelle 2.4 ist der Inhalt des Textpuffers mit Komprimierung dargestellt. Die niedrigsten acht Bit des ersten Kanals, hier Kanal 0, werden in das vierte Byte des Textpuffers gespeichert. Danach werden die Bits acht und neun des ersten Kanals in Byte fünf gespeichert. Für den zweiten Kanal, hier Kanal 3, werden zuerst die niedrigsten acht Bit in das sechste Byte des Puffers geschrieben. Die Bits

acht und neun des zweiten Kanals werden um sechs Stellen nach links verschoben, mit dem Inhalt des fünften Byte Oder-Verknüpft und in das fünfte Byte des Puffers gespeichert. Die Kanalnummer ergibt sich aus der Kanalreihenfolge *channel_order[]* (siehe oben auf Seite 61).

6-Kanal ohne Komprimierung:

```
i = 2 * CurrentCh + HEADERLEN;
TXBuf[i+1] = ADCL;
TXBuf[i] = ADCH;
```

16-Kanal mit Komprimierung:

```
i = 2 * CurrentCh + HEADERLEN;
TXBuf[i] = ADCL;
    if ((CurrentCh % 2) == 0)
    {
        TXBuf[i-1] = ADCL;
        TXBuf[i] = ADCH;
    }
    else
    {
        TXBuf[i+1] = ADCL;
        TXBuf[i] |= (ADCH << 6);
        i += 3;
    }
```

Bei der Auswahl des nächsten Kanals muss gegenüber der 6-Kanal Version nun auch das MUX5 Bit berücksichtigt werden.
6-Kanal:

```
//Select the next channel.
//The right value for ADMUX
//e.g. for Channel 10 is 2 and not 10,
//so 8 need to be subtracted
ADMUX = (channel_order[CurrentCh] - 8);
```

Tabelle 2.3.: Anordnung der Daten im Textpuffer ohne Komprimierung

| Byte | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | Synchronisation (0xa5) | | | | | | | |
| 1 | Synchronisation (0x5a) | | | | | | | |
| 2 | Paketzähler | | | | | | | |
| 3 | - | - | - | - | - | - | Kanal 0 Bit 9 | Kanal 0 Bit 8 |
| 4 | Kanal 0 Bit 7 | Kanal 0 Bit 6 | Kanal 0 Bit 5 | Kanal 0 Bit 4 | Kanal 0 Bit 3 | Kanal 0 Bit 2 | Kanal 0 Bit 1 | Kanal 0 Bit 0 |
| 5 | - | - | - | - | - | - | Kanal 3 Bit 9 | Kanal 3 Bit 8 |
| 6 | Kanal 3 Bit 7 | Kanal 3 Bit 6 | Kanal 3 Bit 5 | Kanal 3 Bit 4 | Kanal 3 Bit 3 | Kanal 3 Bit 2 | Kanal 3 Bit 1 | Kanal 3 Bit 0 |
| 7 | - | - | - | - | - | - | Kanal 1 Bit 9 | Kanal 1 Bit 8 |
| 8 | Kanal 1 Bit 7 | Kanal 1 Bit 6 | Kanal 1 Bit 5 | Kanal 1 Bit 4 | Kanal 1 Bit 3 | Kanal 1 Bit 2 | Kanal 1 Bit 1 | Kanal 1 Bit 0 |
| 9 | - | - | - | - | - | - | Kanal 4 Bit 9 | Kanal 4 Bit 8 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Tabelle 2.4.: Anordnung der Daten im Textpuffer mit Komprimierung

| Byte | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | Synchronisation (0xa5) | | | | | | | |
| 1 | Synchronisation (0x5a) | | | | | | | |
| 2 | Paketzähler | | | | | | | |
| 3 | Kanal 0 Bit 7 | Kanal 0 Bit 6 | Kanal 0 Bit 5 | Kanal 0 Bit 4 | Kanal 0 Bit 3 | Kanal 0 Bit 2 | Kanal 0 Bit 1 | Kanal 0 Bit 0 |
| 4 | Kanal 3 Bit 9 | Kanal 3 Bit 8 | - | - | - | - | Kanal 0 Bit 9 | Kanal 0 Bit 8 |
| 5 | Kanal 3 Bit 7 | Kanal 3 Bit 6 | Kanal 3 Bit 5 | Kanal 3 Bit 4 | Kanal 3 Bit 3 | Kanal 3 Bit 2 | Kanal 3 Bit 1 | Kanal 3 Bit 0 |
| 6 | Kanal 1 Bit 7 | Kanal 1 Bit 6 | Kanal 1 Bit 5 | Kanal 1 Bit 4 | Kanal 1 Bit 3 | Kanal 1 Bit 2 | Kanal 1 Bit 1 | Kanal 1 Bit 0 |
| 7 | Kanal 4 Bit 9 | Kanal 4 Bit 8 | - | - | - | - | Kanal 1 Bit 9 | Kanal 1 Bit 8 |
| 8 | Kanal 4 Bit 7 | Kanal 4 Bit 6 | Kanal 4 Bit 5 | Kanal 4 Bit 4 | Kanal 4 Bit 3 | Kanal 4 Bit 2 | Kanal 4 Bit 1 | Kanal 4 Bit 0 |
| 9 | Kanal 2 Bit 7 | Kanal 2 Bit 6 | Kanal 2 Bit 5 | Kanal 2 Bit 4 | Kanal 2 Bit 3 | Kanal 2 Bit 2 | Kanal 2 Bit 1 | Kanal 2 Bit 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Es muss eine Unterscheidung zwischen den Kanälen 0–7 und 8–15 erfolgen. Wenn die Kanalnummer aus *channel_order[]* an der Stelle *CurrentCh* kleiner als acht ist, dann muss das MUX5 Bit in ADCSRB auf 0 gesetzt werden. Die anderen Bits, die in ADMUX stehen, können dann mit der Kanalnummer belegt werden. Ist das nicht der Fall, dann ist die Kanalnummer größer als sieben und das MUX5 Bit muss auf 1 gesetzt werden. Die Bits MUX0-MUX2 erhalten dann die Kanalnummer abzüglich acht. Das Bitmuster für die Kanaladressierung ist in Tabelle 2.2 auf Seite 58 dargestellt.
16-Kanal:

```
//Select the next channel.
if(channel_order[CurrentCh] < 8)
{
    ADCSRB &= ~(1 << MUX5);
    ADMUX = channel_order[CurrentCh];
}
else
{
    ADCSRB |= (1 << MUX5);
    ADMUX = (channel_order[CurrentCh] - 8);
}
```

Die Auswahl des ersten Kanals muss von Kanal zehn auf Kanal null geändert werden.
6-Kanal:

```
//Prepare next conversion, on channel 10.
ADMUX |= (1 << MUX1);
ADMUX &=
    ~((1 << MUX0) | (1 << MUX2) | (1 << MUX3) | (1 << MUX4));
ADCSRB |= (1 << MUX5);
```

16-Kanal:

```
//Prepare next conversion, on channel 0.
```

```
ADMUX &= ~((1 << MUX0) | (1 << MUX1) | (1 << MUX2)
                                    | (1 << MUX3) | (1 << MUX4));
ADCSRB &= ~(1 << MUX5);
```

## 2.7. Software

Als Software wird das Brainbay Programm verwendet, da es ein Open Source Programm ist. Das Programm, der Quellcode, eine ausführliche Anleitung für Änderungen am Programm und die notwendigen Bibliotheken können kostenlos auf der Brainbay-Internetseite[51] heruntergeladen werden.

Für die ersten Tests mit der 6-Kanal-Firmware braucht das Programm nicht geändert zu werden, da die Daten in der Paketversion 2 übertragen werden.

Die Funktion für die Paketversion 2 in der Datei *ob_eeg.cpp* dient als Vorlage für die 16-Kanal Version einer solchen Funktion. Die Implementierung der Dekomprimierung der Daten ist die wesentlichste Änderung an dem Programm. Auf die Änderungen wird nicht weiter eingegangen. Hier noch die Quellcodes für die Original-Funktion und für die 16-Kanal Version.

6-Kanal-Modular-EEG Paketversion 2:

```
void parse_byte_P2(unsigned char actbyte)
{
    switch (PACKET.readstate)
    {
          case 0: if (actbyte==165) PACKET.readstate++;
                  break;
          case 1: if (actbyte==90)  PACKET.readstate++;
                  else PACKET.readstate=0;
                  break;
```

[51] http://www.shifz.org/brainbay/

```
            case 2: PACKET.readstate++;
                    break;
            case 3: PACKET.number = actbyte;
                    // if (++PACKET.old_number != actbyte)
                    // GLOBAL.temp++;
                    // PACKET.old_number=actbyte;
                    PACKET.extract_pos=0;PACKET.readstate++;
                    break;
            case 4: if (PACKET.extract_pos < 12)
                    {    if ((PACKET.extract_pos & 1) == 0)
                            PACKET.buffer[PACKET.extract_pos>>1]
                                                    =actbyte*256;
                         else
                           PACKET.buffer[PACKET.extract_pos>>1]
                                                    +=actbyte;
                         PACKET.extract_pos++;
                    }
                    else
                    {  PACKET.switches= actbyte;
                       PACKET.readstate=0;
                       process_packets();
                    }
                    break;
            default: PACKET.readstate=0;
          }
}
```

16-Kanal Version mit Datenkomprimierung:

```
void parse_byte_16(unsigned char actbyte)
{
    int i,j;
    switch (PACKET.readstate)
    {
            case 0: if (actbyte==165) PACKET.readstate++;
                    break;
```

```
            case 1: if (actbyte==90)  PACKET.readstate++;
                    else PACKET.readstate=0;
                    break;
            case 2: PACKET.number = actbyte;
                    // if (++PACKET.old_number != actbyte)
                    //GLOBAL.temp++;
                    // PACKET.old_number=actbyte;
                    PACKET.extract_pos=0;PACKET.readstate++;
                    break;
            case 3: tmp_buffer[PACKET.extract_pos]=actbyte;
                    PACKET.extract_pos++;
                    if (PACKET.extract_pos==25)
                    {
                          for(i=0,j=0; j<24; i+=2,j+=3)
                          {
          PACKET.buffer[i] = ((unsigned short) tmp_buffer[j])
           | (((unsigned short) tmp_buffer[j+1]) & 0x03) << 8;
          PACKET.buffer[i+1] =
              (((unsigned short) tmp_buffer[j+1]) & 0xC0) << 2
                          | ((unsigned short) tmp_buffer[j+2]);
                          }
                          PACKET.switches = tmp_buffer[24];
                          PACKET.readstate=0;
                          process_packets();
                    }
                    break;
            default: PACKET.readstate=0;
          }
}
```

## 2.8. Tests

Folgende Tests wurden mit der neuen Hardware durchgeführt. Während der Lötarbeiten wurde der ordnungsgemäße Kontakt der Mikrocontroller-Pins

geprüft. Benachbarte Pins wurden auf Kurzschlüsse getestet.

Zum Testen der Funktionstüchtigkeit der Platine wurden diese Tests durchgeführt. In Brainbay wurden die Signale der Kanäle auf einem Oszilloskop ausgegeben. Als Quelle der Signale wurde der PWM-Ausgang mit jeweils einem Kanal verbunden und der Signalverlauf beobachtet.

Test 1: 6-Kanal-Firmware mit Original Brainbay Kanäle 10–15
Ergebnis: Normale Funktion

Test 2: 6-Kanal-Firmware mit geändertem Brainbay Kanäle 10–15
Ergebnis: Normale Funktion

Test 3: 6-Kanal-Firmware mit geändertem Brainbay Kanäle 0–5
Ergebnis: Fehler

Test 4: 6-Kanal-Firmware mit Original Brainbay Kanäle 0–5
Ergebnis: Gleiche Fehler
Kanal 0 normale Funktion
Kanal 1 normale Funktion
Ein Signal auf Kanal 2 wird in der Software auch auf Kanal 3 angezeigt. Wird das Signal auf Kanal 3 angeschlossen, ist in der Software nichts zu sehen.
Kanal 4 verringerte Amplitude
Kanal 5 verringerte Amplitude

Test 5: 6-Kanal-Firmware mit Original Brainbay Kanäle 0, 6, 7, 3, 4, 5 Ergebnis: Fehler
Kanal 6 verringerte Amplitude
Kanal 7 verringerte Amplitude und Signal auch auf Kanal 3.

Test 6: 6-Kanal-Firmware mit Original Brainbay Kanäle 8-13
Ergebnis: Normale Funktion

Vermutlich liegt ein Problem mit den Kondensatoren im Tiefpass der A/D-Eingänge vor.

Test 7: 16-Kanal-Firmware mit geändertem Brainbay Kanäle 0–16
Ergebnis: Gleiche Fehler bei den Kanälen wie bei den 6-Kanal Tests.

Test 8: 16-Kanal-Firmware mit geändertem Brainbay Kanäle 0–16, alle Kanäle angeschlossen
Ergebnis: Der Signalverlauf ist bei 16-Kanälen gleichmäßig.

Abbildung 2.14 zeigt einen Screenshot aus dem Brainbay-Programm bei gleichzeitiger Verwendung von 16 Kanälen.

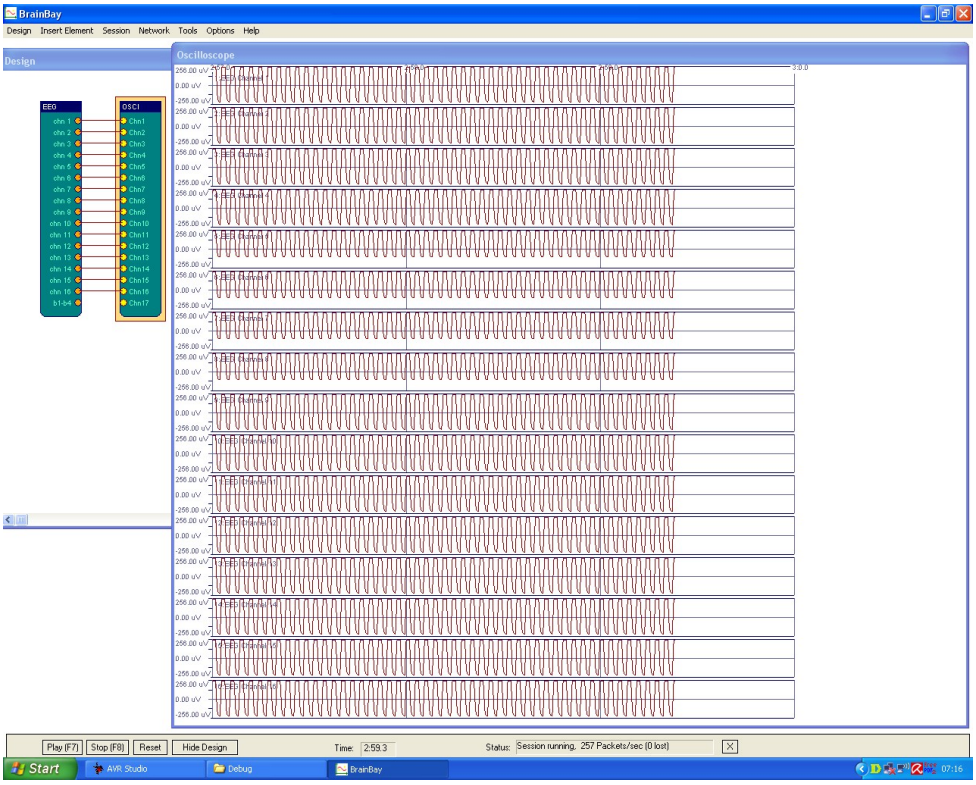


Abbildung 2.14.: Screenshot aus Brainbay mit 16 Kanälen

# Kapitel 3.

# Abschluss

## 3.1. Fazit

In dieser Diplomarbeit ist eine Platine entstanden, mit der sich zeigen lässt, dass die Zielsetzung und die theoretischen Überlegungen umsetzbar sind. Die Platine enthält noch kleine Fehler, für deren Beseitigung die Zeit nicht mehr ausgereicht hat.

## 3.2. Ausblick

Es gibt noch einige interessante Aufgaben, die sich an diese Arbeit anschließen lassen. So könnten weitere EEG-Programme für die Verwendung von 16-Kanälen erweitert werden. Im OpenEEG-Projekt wurde auf die Verwendung eines Programmierprogrammes zum Einspielen der Firmware auf den Mikrocontroller hingewiesen. Dieses Programm heißt *SP12* und hat keine Konfiguration um damit einen ATmega1280 zu programmieren. Der Vorteil von so einem Programm ist, dass mit ein paar Bauteilen ein eigenes Programmierkabel hergestellt werden kann. Eine Konfiguration für SP12 für den ATmega1280 herzustellen würde die Verwendbarkeit dieser Arbeit im OpenEEG-Projekt vergrößern.

# Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, den 27.03.2008

## Erklärung

Mir ist bekannt, dass nach § 156 StGB bzw. § 163 StGB eine falsche Versicherung an Eides Statt bzw. eine fahrlässige falsche Versicherung an Eides Statt mit Freiheitsstrafe bis zu drei Jahren bzw. bis zu einem Jahr oder mit Geldstrafe bestraft werden kann.

Dortmund, den 27.03.2008

# Literaturverzeichnis


[Atm05] Atmel Corporation: *AVR Instruction Set*, November 2005. http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf, besucht: 25.03.2008.

[Atm07a] Atmel Corporation: *ATmega640/1280/1281/2560/2561*, 1 Auflage, August 2007. http://www.atmel.com/dyn/resources/prod_documents/doc2549.pdf, besucht: 10.03.2008.

[Atm07b] Atmel Corporation: *ATmega8*, s Auflage, August 2007. http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf, besucht: 25.02.2008.

[AVR07] *AVR Libc*, 1.6.1 Auflage, Dezember 2007. http://www.nongnu.org/avr-libc/user-manual/index.html, besucht: 21.03.2008.

[Bro07a] *Der Brockhaus multimedial 2007 premium*, 2007, ISBN 3-411-06548-6. Artikel: Elektroenzephalogramm.

[Bro07b] *Der Brockhaus multimedial 2007 premium*, 2007, ISBN 3-411-06548-6. Artikel: C, Programmierung.

[Cre] *Namensnennung-Weitergabe unter gleichen Bedingungen 2.0*. http://creativecommons.org/licenses/by-sa/2.0/deed.de, besucht: 26.03.2008.

[Dem05] John N. Demos: *Getting Started with Neurofeedback*. W. W. Norton & Company, 2005, ISBN 0-393-70450-5.

[Fli01] Thomas Flik: *Mikroprozessortechnik*. Springer-Verlag Berlin Heidelberg New York, 6. Auflage, 2001, ISBN 3-540-42042-8.

[GNP+03] Dan Griffiths, Nelo, Jim Peters, Andreas Robinson, Jack Spaar und Yaniv Vilnai: *The ModularEEG Design*, 2002,2003. http://openeeg.sourceforge.net/doc/modeeg/modeeg_design.html, besucht: 26.03.2008.

[Hol07] Bastian Holtermann: *Modular-EEG und aktive Elektroden aus dem OpenEEG-Projekt*. Projektarbeit, Fachhochschule Dortmund, Fachbereich Informatik, August 2007.

[HPR03] Joerg Hansmann, Jim Peters und Andreas Robinson: *ModularEEG firmware for one-way transmission, v0.5.4-p2*, 2002,2003. http://downloads.sourceforge.net/openeeg/ModularEEG-firmware-v1.0.0.zip?modtime=1061769600&big_mirror=0, besucht: 10.03.2008.

[Lic] *Default site-wide license*. http://openeeg.sourceforge.net/doc/license.html, besucht: 26.03.2008.

[Mic02] Marek Michalkiewicz: *avr/iom8.h*, 2002. Datei aus WinAVR 20071221.

[MWW07] Marek Michalkiewicz, Joerg Wunsch und Eric Weddington: *avr/io.h*, 2007. Datei aus WinAVR 20071221.

[Pro] *Welcome to the OpenEEG project*. http://openeeg.sourceforge.net/doc/, besucht: 26.03.2008.

[Stu04] Peter Stuhlmüller: *A/D-Wandler in Embedded Systemen*. Franzis Verlag GmbH, 2004, ISBN 3-7723-5300-2.

[Wol06] Jürgen Wolf: *C von A bis Z*. Galileo Press GmbH, 2. Auflage, 2006, ISBN 3-89842-643-2. http://www.galileocomputing.de/openbook/c_von_a_bis_z/, besucht: 21.03.2008.

# Anhang A.

# Original Firmware für ATmega8

Hier folgt die Firmware Version 0.5.4-p2 aus dem OpenEEG-Projekt. (Quelle: [HPR03]) Zur besseren Lesbarkeit wurden lediglich ein paar Zeilenumbrüche eingefügt. Der Code an sich wurde nicht verändert.

```
/*
 * ModularEEG firmware for one-way transmission, v0.5.4-p2
 * Copyright (c) 2002-2003,
 * Joerg Hansmann, Jim Peters, Andreas Robinson
 * License: GNU General Public License (GPL) v2
 * Compiles with AVR-GCC v3.3.
 *
 * Note: -p2 in the version number means this firmware is for
 * packet version 2.
 */

//////////////////////////////////////////////////////////////

/*
////////// Packet Format Version 2 ////////////

// 17-byte packets are transmitted from the ModularEEG
// at 256Hz, using 1 start bit, 8 data bits, 1 stop bit,
// no parity, 57600 bits per second.
```

```
// Minimial transmission speed is
// 256Hz * sizeof(modeeg_packet) * 10 = 43520 bps.

struct modeeg_packet
{
        uint8_t         sync0;          // = 0xa5
        uint8_t         sync1;          // = 0x5a
        uint8_t         version;        // = 2
        // packet counter. Increases by 1 each packet.
        uint8_t         count;
        // 10-bit sample (= 0 - 1023) in big endian
        // (Motorola) format.
        uint16_t        data[6];
        // State of PD5 to PD2, in bits 3 to 0.
        uint8_t         switches;
};

// Note that data is transmitted in big-endian format.
// By this measure together with the unique pattern in sync0
// and sync1 it is guaranteed, that re-sync
// (i.e after disconnecting the data line) is always safe.

// At the moment communication direction is only from Atmel-
// processor to PC. The hardware however supports full duplex
// communication. This feature will be used in later firmware
// releases to support the PWM-output and LED-Goggles.

*/

////////////////////////////////////////////////////////////



/*
 * Program flow:
 *
```

```
 * When 256Hz timer expires: goto SIGNAL(SIG_OVERFLOW0)
 * SIGNAL(SIG_OVERFLOW0) enables the ADC
 *
 *
 * Repeat for each channel in the ADC:
 * Sampling starts. When it completes: goto SIGNAL(SIG_ADC)
 * SIGNAL(SIG_ADC) reads the sample and restarts the ADC.
 *
 * SIGNAL(SIG_ADC) writes first byte to UART data register
 * (UDR) which starts the transmission over the serial port.
 *
 * Repeat for each byte in packet:
 * When transmission begins and UDR empties:
 * goto SIGNAL(SIG_UART_DATA)
 *
 * Start over from beginning.
 */

#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#define NUMCHANNELS 6
#define HEADERLEN 4
#define PACKETLEN (NUMCHANNELS * 2 + HEADERLEN + 1)

#define SAMPFREQ 256
#define TIMER0VAL 256 - ((7372800 / 256) / SAMPFREQ)

char const channel_order[]= { 0, 3, 1, 4, 2, 5 };

/** The transmission packet */

volatile uint8_t TXBuf[PACKETLEN];
```

```
/** Next byte to read or write in the transmission packet. */

volatile uint8_t TXIndex;

/** Current channel being sampled. */

volatile uint8_t CurrentCh;

/** Sampling timer (timer 0) interrupt handler */

SIGNAL(SIG_OVERFLOW0)
{
    outb(TCNT0, TIMER0VAL); //Reset timer to get correct
                            //sampling frequency.

    CurrentCh = 0;

    // Write header and footer:

    // Increase packet counter (fourth byte in header)

    TXBuf[3]++;

    //Get state of switches on PD2..5,
    //if any (last byte in packet).

    TXBuf[2 * NUMCHANNELS + HEADERLEN] =
                                    (inp(PIND) >> 2) &0x0F;

    cbi(UCSRB, UDRIE);      //Ensure UART IRQ's are disabled.
    sbi(ADCSR, ADIF);       //Reset any pending ADC interrupts
    sbi(ADCSR, ADIE);       //Enable ADC interrupts.

    //The ADC will start sampling automatically as soon
    //as sleep is executed in the main-loop.
}
```

```
/** AD-conversion-complete interrupt handler. */
SIGNAL(SIG_ADC)
{
    volatile uint8_t i;

    i = 2 * CurrentCh + HEADERLEN;

    TXBuf[i+1] = inp(ADCL);
    TXBuf[i] = inp(ADCH);

    CurrentCh++;

    if (CurrentCh < NUMCHANNELS)
    {
        //Select the next channel.
        outb(ADMUX, channel_order[CurrentCh]);
        //The next sampling is started automatically.
    }
    else
    {
        //Prepare next conversion, on channel 0.
        outb(ADMUX, channel_order[0]);

        // Disable ADC interrupts to prevent further calls to
        // SIG_ADC.

        cbi(ADCSR, ADIE);

        // Hand over to SIG_UART_DATA, by starting
        // the UART transfer and enabling UDR IRQ's.

        outb(UDR, TXBuf[0]);
        sbi(UCSRB, UDRIE);
        TXIndex = 1;
    }
}
```

```
/*** UART data transmission register-empty
                                        interrupt handler ***/
SIGNAL(SIG_UART_DATA)
{
    outb(UDR, TXBuf[TXIndex]);   //Send next byte
    TXIndex++;
    //See if we're done with this packet
    if (TXIndex == PACKETLEN)
    {
        //Disable SIG_UART_DATA interrupts.
        cbi(UCSRB, UDRIE);
        //Next interrupt will be a SIG_OVERFLOW0.
    }
}

/** Initialize PWM output (PB1 = 14Hz square wave signal) */

void pwm_init(void)
{
   // Set timer/counter 1 to use 10-bit PWM mode.
   // The counter counts from zero to 1023 and then back down
   // again. Each time the counter value equals the value
   // of OCR1(A), the output pin is toggled.
   // The counter speed is set in TCCR1B,
   // to clk / 256 = 28800Hz.
   // Effective frequency is then clk / 256 / 2046 = 14 Hz

#if defined (__AVR_ATmega8__)

    outb(OCR1AH,2);       // Set OCR1A = 512
    outb(OCR1AL,0);
    // Set 10-bit PWM mode
    outb(TCCR1A, BV(COM1A1) + BV(WGM11) + BV(WGM10));
    // Start and let run at clk / 256 Hz.
    outb(TCCR1B, (1 << CS12));
```

```
#else // __AVR_AT90S4434__

    outb(OCR1H,2);        // Set OCR1 = 512
    outb(OCR1L,0);
    // Set 10-bit PWM mode
    outb(TCCR1A, BV(COM11) + BV(PWM11) + BV(PWM10));
    // Start and let run at clk / 256 Hz.
    outb(TCCR1B, (1 << CS12));

#endif
}

int main( void )
{
    //Write packet header and footer

    TXBuf[0] = 0xa5;          //Sync 0
    TXBuf[1] = 0x5a;          //Sync 1
    TXBuf[2] = 2;             //Protocol version
    TXBuf[3] = 0;             //Packet counter

    //Set up the ports.

    outb(DDRD,  0xc2);
    outb(DDRB,  0x07);
    outb(PORTD, 0xff);
    outb(PORTB, 0xff);

    //Select sleep mode = idle.

#if defined (__AVR_ATmega8__)

    outb(MCUCR,(inp(MCUCR) | BV(SE)) &
                          (~BV(SM0) | ~BV(SM1) | ~BV(SM2)));
```

```
#else // __AVR_AT90S4434__

    outb(MCUCR,(inp(MCUCR) | BV(SE)) & (~BV(SM)));

#endif

    //Initialize the ADC

    // Timings for sampling of one 10-bit AD-value:
    //
    // prescaler > ((XTAL / 200kHz) = 36.8 =>
    // prescaler = 64 (ADPS2 = 1, ADPS1 = 1, ADPS0 = 0)
    // ADCYCLE = XTAL / prescaler = 115200Hz or 8.68 us/cycle
    // 14 (single conversion) cycles = 121.5 us
        // (8230 samples/sec)
    // 26 (1st conversion) cycles = 225.69 us

    outb(ADMUX, 0);          //Select channel 0

    //Prescaler = 64, free running mode = off, interrupts off.

    outb(ADCSR, BV(ADPS2) | BV(ADPS1));
    sbi(ADCSR, ADIF);        //Reset any pending ADC interrupts
    sbi(ADCSR, ADEN);        //Enable the ADC

    //Initialize the UART

#if defined (__AVR_ATmega8__)

    outb(UBRRH,0);               //Set speed to 57600 bps
    outb(UBRRL,7);
    outb(UCSRA, 0);
    outb(UCSRC, BV(URSEL) | BV(UCSZ1) | BV(UCSZ0));
    outb(UCSRB, BV(TXEN));
```

```
#else // __AVR_AT90S4434__

    outb(UBRR, 7);                  //Set speed to 57600 bps
    outb(UCSRB, BV(TXEN));

#endif

    //Initialize timer 0

    outb(TCNT0, 0);                 //Clear it.
    //Start it. Frequency = clk / 256
    outb(TCCR0, 4);
    outb(TIMSK, BV(TOIE0));         //Enable the interrupts.

    //Initialize PWM (optional)

    pwm_init();

    sei();

    //Now, we wait. This is an event-driven program,
    //so nothing much happens here.

    while (1)
    {
        __asm__ __volatile__ ("sleep");
    }
}
```

# Anhang B.

# 6-Kanal-EEG Firmware für ATmega1280

Dies ist die Firmware für den ATmega1280 für den ersten Test der Platine mit 6-Kanal A/D-Wandlung und der unveränderten Software, auf die im OpenEEG-Projekt verlinkt wird.

```
/*
 * 6-Channel ModularEEG firmware for one-way transmission,
 * v0.1-p2 for ATmega1280 (6-Ch EEG) at 11059,2 kHz
 * Modified by: Bastian Holtermann
 * Date: March 2008
 * License: GNU General Public License (GPL) v3
 * Compiles with AVR-GCC v4.2.2
 * This is a modified Version of:
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * ModularEEG firmware for one-way transmission, v0.5.4-p2
 * Copyright (c) 2002-2003,
 * Joerg Hansmann, Jim Peters, Andreas Robinson
 * License: GNU General Public License (GPL) v2
 * Compiles with AVR-GCC v3.3.
 *
 * Note: -p2 in the version number means this firmware
 * is for packet version 2.
```

```
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 */

//////////////////////////////////////////////////////////////

/*

///////// Packet Format Version 2 ///////////

// 17-byte packets are transmitted from the ModularEEG at
// 256Hz, using 1 start bit, 8 data bits, 1 stop bit,
// no parity, 57600 bits per second.

// Minimial transmission speed is
// 256Hz * sizeof(modeeg_packet) * 10 = 43520 bps.

struct modeeg_packet
{
    uint8_t             sync0;    // = 0xa5
    uint8_t             sync1;    // = 0x5a
    uint8_t             version; // = 2
    // packet counter. Increases by 1 each packet.
    uint8_t             count;
    // 10-bit sample (= 0 - 1023)
        // in big endian (Motorola) format.
    uint16_t    data[6];
    // State of PA6 to PA3, in bits 3 to 0.
    uint8_t             switches;
};

// Note that data is transmitted in big-endian format.
// By this measure together with the unique pattern in
// sync0 and sync1 it is guaranteed, that re-sync
// (i.e after disconnecting the data line) is always safe.

// At the moment communication direction is only from
```

```
// Atmel-processor to PC. The hardware however supports
// full duplex communication. This feature will be used
// in later firmware releases to support the PWM-output
// and LED-Goggles.

*/

////////////////////////////////////////////////////////////

/*
 * Program flow:
 *
 * When 256Hz timer expires: goto ISR(TIMER0_OVF_vect)
 * ISR(TIMER0_OVF_vect) enables the ADC
 *
 * Repeat for each channel in the ADC:
 * Sampling starts. When it completes: goto ISR(ADC_vect)
 * ISR(ADC_vect) reads the sample and restarts the ADC.
 *
 * ISR(ADC_vect) writes first byte to UART 0 data register
 * (UDR0) which starts the transmission over the serial port.
 *
 * Repeat for each byte in packet:
 * When transmission begins and UDR empties:
 * goto ISR(USART0_UDRE_vect)
 *
 * Start over from beginning.
 */

#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>
//#include <avr/signal.h>

#define NUMCHANNELS 6
#define HEADERLEN 4
```

```
#define PACKETLEN (NUMCHANNELS * 2 + HEADERLEN + 1)

#define SAMPFREQ 256
#define TIMER0VAL 256 - ((11059200 / 256) / SAMPFREQ)

char const channel_order[]= { 10, 13, 11, 14, 12, 15 };

/** The transmission packet */

volatile uint8_t TXBuf[PACKETLEN];

/** Next byte to read or write in the transmission packet. */

volatile uint8_t TXIndex;

/** Current channel being sampled. */

volatile uint8_t CurrentCh;

/** Sampling timer (timer 0) interrupt handler */

//SIGNAL(SIG_OVERFLOW0)
ISR(TIMER0_OVF_vect)
{
    //Reset timer to get correct sampling frequency.
    //outb(TCNT0, TIMER0VAL);
    TCNT0 = TIMER0VAL;

    CurrentCh = 0;

    // Write header and footer:

    // Increase packet counter (fourth byte in header)

    TXBuf[3]++;
```

```
    //Get state of switches on PA3..6,
    //if any (last byte in packet).

    //TXBuf[2 * NUMCHANNELS + HEADERLEN] =
                                        //(inp(PINA) >> 3) &0x0F;
    TXBuf[2 * NUMCHANNELS + HEADERLEN] = (PINA >> 3) &0x0F;

    //Ensure UART IRQ's are disabled.
    //cbi(UCSR0B, UDRIE0);
    UCSR0B &= ~(1 << UDRIE0);
    //Reset any pending ADC interrupts
    //sbi(ADCSRA, ADIF);
    ADCSRA |= (1 << ADIF);
    //Enable ADC interrupts.
    //sbi(ADCSRA, ADIE);
    ADCSRA |= (1 << ADIE);

    //The ADC will start sampling automatically as soon
    //as sleep is executed in the main-loop.
}

/** AD-conversion-complete interrupt handler. */

//SIGNAL(SIG_ADC)
ISR(ADC_vect)
{
    volatile uint8_t i;

    i = 2 * CurrentCh + HEADERLEN;

    //TXBuf[i+1] = inp(ADCL);
    TXBuf[i+1] = ADCL;
    //TXBuf[i] = inp(ADCH);
    TXBuf[i] = ADCH;

    CurrentCh++;
```

```
    if (CurrentCh < NUMCHANNELS)
    {
        //Select the next channel.
        //The right value for ADMUX
        //e.g. for Channel 10 is 2 and not 10,
        //so 8 need to be subtracted
        ADMUX = (channel_order[CurrentCh] - 8);

        //The next sampling is started automatically.
    }
    else
    {
        //Prepare next conversion, on channel 10.
        //outb(ADMUX, channel_order[0]);
        ADMUX |= (1 << MUX1);
            ADMUX &= ~((1 << MUX0) | (1 << MUX2)
                        | (1 << MUX3) | (1 << MUX4));
        ADCSRB |= (1 << MUX5);

        //Disable ADC interrupts to prevent
        //further calls to SIG_ADC.
        //cbi(ADCSRA, ADIE);
        ADCSRA &= ~(1 << ADIE);

        // Hand over to SIG_UART_DATA, by starting
        // the UART transfer and enabling UDR IRQ's.

        //outb(UDR0, TXBuf[0]);
        UDR0 = TXBuf[0];
        //sbi(UCSR0B, UDRIE0);
        UCSR0B |= (1 << UDRIE0);
        TXIndex = 1;
    }
}
```

```
/*** UART data transmission register-empty
                                        interrupt handler ***/

//SIGNAL(SIG_UART_DATA)
ISR(USART0_UDRE_vect)
{
    //Send next byte
    //outb(UDR0, TXBuf[TXIndex]);
    UDR0 = TXBuf[TXIndex];
    TXIndex++;
    //See if we're done with this packet
    if (TXIndex == PACKETLEN)
    {
        //Disable SIG_UART_DATA interrupts.
        //cbi(UCSR0B, UDRIE0);
        UCSR0B &= ~(1 << UDRIE0);
        //Next interrupt will be a SIG_OVERFLOW0.
    }
}

/** Initialize PWM output (PB1 = 14Hz square wave signal) */

void pwm_init(void)
{
    // Set timer/counter 1 to use 10-bit PWM mode.
    // The counter counts from zero to 1023 and then back
    // down again. Each time the counter value equals the
    // value of OCR1(A), the output pin is toggled.
    // The counter speed is set in TCCR1B,
    // to clk / 256 = 28800Hz.
    // Effective frequency is then
    // clk / 256 / 2046 = 21,11 Hz

    // Set OCR1A = 512
    //outb(OCR1AH,2);
    OCR1AH = 0x02;
```

```
    //outb(OCR1AL,0);
    OCR1AL = 0x00;
    // Set 10-bit PWM mode
    //outb(TCCR1A, BV(COM1A1) + BV(WGM11) + BV(WGM10));
    TCCR1A |= (1 << COM1A1) | (1 << WGM11) | (1 << WGM10);
    // Start and let run at clk / 256 Hz.
    //outb(TCCR1B, (1 << CS12));
    TCCR1B |= (1 << CS12);

}

int main( void )
{
    //Write packet header and footer

    TXBuf[0] = 0xa5;          //Sync 0
    TXBuf[1] = 0x5a;          //Sync 1
    TXBuf[2] = 2;             //Protocol version
    TXBuf[3] = 0;             //Packet counter

    //Set up the ports.

    //outb(DDRA,   0x00);
    DDRA = 0x00;
    //outb(DDRB,   0x60);
    DDRB = 0x60;
    //outb(DDRC,   0x00);
    DDRC = 0x00;
    //outb(DDRD,   0x70);
    DDRD = 0x70;
    //outb(DDRE,   0x02);
    DDRE = 0x02;
    //outb(DDRG,   0x00);
    DDRG = 0x00;
    //outb(DDRH,   0x00);
    DDRH = 0x00;
```

```
    //outb(DDRJ,   0x00);
    DDRJ = 0x00;
    //outb(DDRL,   0x00);
    DDRL = 0x00;
    //outb(PORTA,  0xff);
    PORTA = 0xff;
    //outb(PORTB,  0xff);
    PORTB = 0xff;
    //outb(PORTC,  0xff);
    PORTC = 0xff;
    //outb(PORTD,  0xff);
    PORTD = 0xff;
    //outb(PORTE,  0xff);
    PORTE = 0xff;
    //outb(PORTG,  0xff);
    PORTG = 0xff;
    //outb(PORTH,  0xff);
    PORTH = 0xff;
    //outb(PORTJ,  0xff);
    PORTJ = 0xff;
    //outb(PORTL,  0xff);
    PORTL = 0xff;


    //Select sleep mode = idle.

    //outb(SMCR,(inp(SMCR) | BV(SE)) & (~BV(SM0)
                                    // | ~BV(SM1) | ~BV(SM2)));
    SMCR |= (1 << SE);
    SMCR &= ~((1 << SM0) | (1 << SM1) | (1 << SM2));

    //Initialize the ADC

    // Timings for sampling of one 10-bit AD-value:
    //
    // prescaler > ((XTAL / 200kHz) = 55.296 =>
```

```
// prescaler = 64 (ADPS2 = 1, ADPS1 = 1, ADPS0 = 0)
// ADCYCLE = XTAL / prescaler
//         = 172800Hz or 5.787 us/cycle
// 14 (single conversion) cycles
//                    = 81,01 us (12344 samples/sec)
// 26 (1st conversion) cycles = 150.46 us

//Select channel 10
//outb(ADMUX, (inp(ADMUX) | BV(MUX1))
// & (~BV(MUX0) | ~BV(MUX2) | ~BV(MUX3) | ~BV(MUX4));
ADMUX |= (1 << MUX1);
ADMUX &= ~((1 << MUX0) | (1 << MUX2) | (1 << MUX3)
                                      | (1 << MUX4));
//outb(ADCSRB,(inp(ADCSRB) | BV(MUX5)));
ADCSRB |= (1 << MUX5);

//Prescaler = 64, free running mode = off, interrupts off.

//outb(ADCSRA, BV(ADPS2) | BV(ADPS1));
ADCSRA |= (1 << ADPS2) | (1 << ADPS1);
//Reset any pending ADC interrupts
//sbi(ADCSRA, ADIF);
ADCSRA |= (1 << ADIF);
//Enable the ADC
//sbi(ADCSRA, ADEN);
ADCSRA |= (1 << ADEN);

//Initialize the UART

//Set speed to 57600 bps
//outb(UBRR0H,0x00);
UBRR0H = 0x00;
//outb(UBRR0L,0x0b);
UBRR0L = 0x0b;
//outb(UCSR0A, 0);
UCSR0A = 0x00;
```

```
    //outb(UCSR0C, BV(UCSZ01) | BV(UCSZ00));
    UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);
    //outb(UCSR0B, BV(TXEN0));
    UCSR0B |= (1 << TXEN0);


    //Initialize timer 0

    //Clear it.
    //outb(TCNT0, 0);
    TCNT0 = 0x00;
    //Start it. Frequency = clk / 256
    //outb(TCCR0B, 4);
    TCCR0B = (1 << CS02);
    //Enable the interrupts.
    //outb(TIMSK0, BV(TOIE0));
    TIMSK0 |= (1 << TOIE0);

    //Initialize PWM (optional)

    pwm_init();

    sei();

    //Now, we wait. This is an event-driven program,
    //so nothing much happens here.

    while (1)
    {
        __asm__ __volatile__ ("sleep");
    }
}
```

# Anhang C.

# 16-Kanal Firmware für ATmega1280

Hier ist die Firmware für 16-Kanal A/D-Wandlung mit dem ATmega1280.

```
/*
 * 16-Channel ModularEEG firmware for one-way transmission,
 * v0.1-p2 for ATmega1280 (16-Ch EEG) at 11059,2 kHz
 * Modified by: Bastian Holtermann
 * Date: March 2008
 * License: GNU General Public License (GPL) v3
 * Compiles with AVR-GCC v4.2.2
 * This is a modified Version of:
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * ModularEEG firmware for one-way transmission, v0.5.4-p2
 * Copyright (c) 2002-2003,
 * Joerg Hansmann, Jim Peters, Andreas Robinson
 * License: GNU General Public License (GPL) v2
 * Compiles with AVR-GCC v3.3.
 *
 * Note: -p2 in the version number means this firmware is
 * for packet version 2.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 */
```

```
//////////////////////////////////////////////////////

/*

////////// Packet Format  ///////////

// 28-byte packets are transmitted from the ModularEEG
// at 256Hz, using 1 start bit, 8 data bits, 1 stop bit,
// no parity, 115200 bits per second.

// Minimial transmission speed is
// 256Hz * sizeof(modeeg_packet) * 10 = 71680 bps.

struct modeeg_packet
{
    uint8_t     sync0;     // = 0xa5
    uint8_t     sync1;     // = 0x5a
    // packet counter. Increases by 1 each packet.
    uint8_t     count;
    // 10-bit sample (= 0 - 1023)
    uint16_t     data[16];
    // State of PA6 to PA3, in bits 3 to 0.
    uint8_t     switches;
};


// By this measure together with the unique pattern in sync0
// and sync1 it is guaranteed, that re-sync
// (i.e after disconnecting the data line) is always safe.

// At the moment communication direction is only from
// Atmel-processor to PC. The hardware however supports full
// duplex communication. This feature will be used in later
// firmware releases to support the PWM-output and
// LED-Goggles.
```

```
*/

//////////////////////////////////////////////////////////////

/*
 * Program flow:
 *
 * When 256Hz timer expires: goto ISR(TIMER0_OVF_vect)
 * ISR(TIMER0_OVF_vect) enables the ADC
 *
 * Repeat for each channel in the ADC:
 * Sampling starts. When it completes: goto ISR(ADC_vect)
 * ISR(ADC_vect) reads the sample and restarts the ADC.
 *
 * ISR(ADC_vect) writes first byte to UART 0 data register
 * (UDR0) which starts the transmission over the serial port.
 *
 * Repeat for each byte in packet:
 * When transmission begins and UDR empties:
 * goto ISR(USART0_UDRE_vect)
 *
 * Start over from beginning.
 */

#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>

#define NUMCHANNELS 16
#define HEADERLEN 3
//2 Channels share 3 Bytes in Packet + 3 Bytes HEADERLEN
                                    //+ 1 Byte for the footer
#define PACKETLEN (((NUMCHANNELS/2)*3) + HEADERLEN + 1)

#define SAMPFREQ 256
```

```
#define TIMER0VAL 256 - ((11059200 / 256) / SAMPFREQ)

char const channel_order[]= { 0, 1, 2, 3, 4, 5, 6, 7, 8,
                              9, 10, 11, 12, 13, 14, 15 };

/** The transmission packet */
volatile uint8_t TXBuf[PACKETLEN];

/** Next byte to read or write in the transmission packet. */
volatile uint8_t TXIndex;

/** Current channel being sampled. */
volatile uint8_t CurrentCh;

volatile uint8_t i;


/** Sampling timer (timer 0) interrupt handler */
ISR(TIMER0_OVF_vect)
{
    //Reset timer to get correct sampling frequency.
    TCNT0 = TIMER0VAL;

    //Reset channelcounter
    CurrentCh = 0;
    i = 4;

    // Increase packet counter (third byte in header)
    TXBuf[2]++;

    //Get state of switches on PA3..6,
    //if any (last byte in packet).
    TXBuf[((NUMCHANNELS/2)*3) + HEADERLEN] =
                                          (PINA >> 3) &0x0F;

    //Ensure UART IRQ's are disabled.
```

```
    UCSR0B &= ~(1 << UDRIE0);
    //Reset any pending ADC interrupts
    ADCSRA |= (1 << ADIF);
    //Enable ADC interrupts.
    ADCSRA |= (1 << ADIE);

    //The ADC will start sampling automatically as soon
    //as sleep is executed in the main-loop.
}


/** AD-conversion-complete interrupt handler. */
ISR(ADC_vect)
{

    if ((CurrentCh % 2) == 0)
    {
        TXBuf[i-1] = ADCL;
        TXBuf[i] = ADCH;
    }
    else
    {
        TXBuf[i+1] = ADCL;
        TXBuf[i] |= (ADCH << 6);
        i += 3;
    }

    //Increase channel counter
    CurrentCh++;

    if (CurrentCh < NUMCHANNELS)
    {
        //Select the next channel.
        //If the channel number (not the counter)
        //is lower than 8, the MUX5 bit has to be cleared
        if(channel_order[CurrentCh] < 8)
```

```
        {
            //Clear the MUX5 bit for channels 0-7
            ADCSRB &= ~(1 << MUX5);
            //Set the MUX0-4 bits to
            //the right value for channel 0-7
            ADMUX = channel_order[CurrentCh];
        }
        //channel number is higher than 7
        else
        {
            //Set the MUX5 bit for channels 8-15
            ADCSRB |= (1 << MUX5);
            //Set the MUX0-4 bits to
            //the right value for channel 0-7
            ADMUX = (channel_order[CurrentCh] - 8);
        }
        //The next sampling is started automatically.
    }
    else
    {
        //Prepare next conversion, on channel 0.
        ADMUX &= ~((1 << MUX0) | (1 << MUX1) | (1 << MUX2)
                                | (1 << MUX3) | (1 << MUX4));
        ADCSRB &= ~(1 << MUX5);

        // Disable ADC interrupts to prevent
        //further calls to SIG_ADC.
        ADCSRA &= ~(1 << ADIE);

        // Hand over to SIG_UART_DATA, by starting
        // the UART transfer and enabling UDR IRQ's.

        UDR0 = TXBuf[0];
        UCSR0B |= (1 << UDRIE0);
        TXIndex = 1;
    }
```

```
}

/*** UART data transmission register-empty
                                        interrupt handler ***/
ISR(USART0_UDRE_vect)
{
    UDR0 = TXBuf[TXIndex]; //Send next byte
    TXIndex++;
    //See if we're done with this packet
    if (TXIndex == PACKETLEN)
    {
        //Disable SIG_UART_DATA interrupts.
        UCSR0B &= ~(1 << UDRIE0);
        //Next interrupt will be a SIG_OVERFLOW0.
    }
}

/** Initialize PWM output (PB1 = 14Hz square wave signal) */
void pwm_init(void)
{
    // Set timer/counter 1 to use 10-bit PWM mode.
    // The counter counts from zero to 1023 and
        // then back down again. Each time the counter
        // value equals the value of OCR1(A),
        // the output pin is toggled.
    // The counter speed is set in TCCR1B,
        // to clk / 256 = 28800Hz.
    // Effective frequency is then
        // clk / 256 / 2046 = 21,11 Hz

    OCR1AH = 0x02;        // Set OCR1A = 512
    OCR1AL = 0x00;
    // Set 10-bit PWM mode
    TCCR1A |= (1 << COM1A1) | (1 << WGM11) | (1 << WGM10);
    // Start and let run at clk / 256 Hz.
    TCCR1B |= (1 << CS12);
```

```
}

int main( void )
{
    //Write packet header
    TXBuf[0] = 0xa5;          //Sync 0
    TXBuf[1] = 0x5a;          //Sync 1
    TXBuf[2] = 0x00;          //Packet counter

    //Set up the ports.
    //Write a 1 for pin is output, 0 for pin is input
    DDRA = 0x00;
    DDRB = 0x60;
    DDRC = 0x00;
    DDRD = 0x70;
    DDRE = 0x02;
    DDRG = 0x00;
    DDRH = 0x00;
    DDRJ = 0x00;
    DDRL = 0x00;
    //Activate pull-up resistor for all pins that are inputs
    PORTA = 0xff;
    PORTB = 0xff;
    PORTC = 0xff;
    PORTD = 0xff;
    PORTE = 0xff;
    PORTG = 0xff;
    PORTH = 0xff;
    PORTJ = 0xff;
    PORTL = 0xff;


    //Select sleep mode = idle.
    SMCR |= (1 << SE);
    SMCR &= ~((1 << SM0) | (1 << SM1) | (1 << SM2));
```

```
//Initialize the ADC

// Timings for sampling of one 10-bit AD-value:
//
// prescaler > ((XTAL / 200kHz) = 55.296 =>
// prescaler = 64 (ADPS2 = 1, ADPS1 = 1, ADPS0 = 0)
// ADCYCLE = XTAL / prescaler =
// 172800Hz or 5.787 us/cycle
// 14 (single conversion) cycles =
// 81,01 us (12344 samples/sec)
// 26 (1st conversion) cycles = 150.46 us

//Select channel 0
ADMUX &= ~((1 << MUX0) | (1 << MUX1) | (1 << MUX2)
                       | (1 << MUX3) | (1 << MUX4));
ADCSRB &= ~(1 << MUX5);

//Prescaler = 64, free running mode = off, interrupts off.
ADCSRA |= (1 << ADPS2) | (1 << ADPS1);
//Reset any pending ADC interrupts
ADCSRA |= (1 << ADIF);
//Enable the ADC
ADCSRA |= (1 << ADEN);

//Initialize the USART
UBRR0H = 0x00;      //Set speed to 115200 bps
UBRR0L = 0x05;
UCSR0A = 0x00;
UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);
UCSR0B |= (1 << TXEN0);


//Initialize timer 0

TCNT0 = 0x00;        //Clear it.
```

```
    //Start it. Frequency = clk / 256
    TCCR0B = (1 << CS02);
    TIMSK0 |= (1 << TOIE0); //Enable the interrupts.

    //Initialize PWM (optional)
    pwm_init();

    //Enables interrupts by setting the global interrupt mask.
    sei();

    //Now, we wait. This is an event-driven program,
    //so nothing much happens here.

    while (1)
    {
        __asm__ __volatile__ ("sleep");
    }
}
```