BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN

University of Applied Sciences

2012

Projekt zur Vorbereitung der Bachelorarbeit:

Raw - Auslesen eines LCD

Student: Enrico Nussbaum
Betreuer: Prof. Dr. Peter Gober
Abgabe des Projekts: 01.03.2012

Inhaltsverzeichnis

1. Motivation

Das Projekt entstand aus der Idee heraus, Entwicklungszeit zu sparen und bereits vorhandene Hardware für eigene Zwecke zu nutzen.

Im Modellbaubereich wird oft spezielles Zubehör benötigt, welches entweder zu überzogenen Preisen oder gar nicht käuflich zu erwerben ist. Daher ist der Modellbauer auch bei seinen Messinstrumenten häufig auf Eigenbau angewiesen.

Ein Schubmessstand stellt ein solches Messgerät dar. Mit diesem Messgerät ist es möglich, die Antriebskraft von Motoren zu messen, Kurven zu plotten und die Daten für spätere Weiterverarbeitung zu speichern.
Mit diesen Daten können dann Optimierungen am Fluggerät, am Motor oder an der Steuerungssoftware vorgenommen werden. Schubmessstände werden in erster Linie im Raketenmodellbau verwendet, aber auch für senkrechtstartende Fluggeräte, welche ihre komplette Auftriebskraft aus dem Motor beziehen, ist ein solches Messgerät interessant.

Die vorliegende Arbeit behandelt die Messdatenerfassung, in welcher mit einem Mikrocontroller das Ansteuerungsschema eines LCD abgegriffen und die Messdaten zur Kontrolle optisch ausgegeben werden.

Diese Arbeit kann im Allgemeinen an jedes Messgerät adaptiert werden, deren Messdaten man extern weiterverarbeiten möchte, obwohl diese nur per LCD zur Verfügung gestellt werden.

2. Messprinzip

Da der zu messende Schub nichts anderes als eine Kraft ist, sollte eine gewöhnliche Digitalwaage modifiziert werden um die Gewichtsdaten, welche proportional zum Schub sind, weiterverarbeiten zu können. Der senkrecht nach unten beschleunigte Luftstrom bei Senkrechtstartern sorgt für den nötigen Auftrieb. Die dabei auftretende Schubkraft wird über eine Vorrichtung, die dem Prinzip einer Balkenwaage ähnelt, umgelenkt und mit der Digitalwaage gemessen, wie in Abbildung 1 erkennbar.

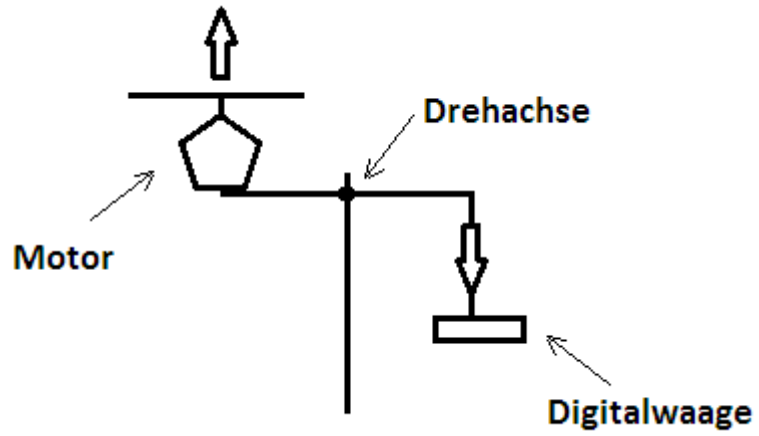


Abbildung 1: Schema Schubmessstand

Mit diesem Prinzip können sofort quantitative Vergleiche zwischen verschiedenen Konfigurationen geschlossen werden, da die systematischen Fehler, hervorgerufen durch die zusätzliche Masse des Messaufbaus, bei allen Konfigurationen gleich bleibend sind.

Da die Waage eine Refreshrate von 3 Hz hat, können z.B. Schwingungen nur bis zu einer Frequenz von 1,5 Hz zuverlässig entdeckt werden. Dies ist insbesondere für Raketenmotoren unzureichend. Diese Erkenntnis kam allerdings erst, nachdem das Projekt schon so weit fortgeschritten war, dass ein Abbruch den bisher aufgebrachten Aufwand nicht rechtfertigte, zumal ein solcher Aufbau (ein Display auzulesen) auch bei anderen Gelegenheiten Anwendung finden kann.

3. Ansteuerung eines LCD

3.1. Multiplexen allgemein

Ein Liqud Crystal Display besteht aus von einander unabhängigen Segmenten, welche Flüssigkristalle enthalten. Diese können bei Anlegen einer Spannung durch das resultierende elektrische Feld ihre Polarisationsrichtung ändern und somit polarisiertes Licht hindurchlassen oder sperren. Dadurch erscheint ein solches Segment transparent oder schwarz.

Aus den Segmenten werden dann Ziffern, Buchstaben oder Symbole gebildet. Standarddisplays besitzen mehrere 7-Segment-Anzeigen, Punkte und informative Segmente wie zum Beispiel eine Batterieanzeige o.ä.
Um zur Ansteuerung eines jeden Segments nicht je 2 Anschlüsse nach außen führen zu müssen, bedient man sich der Multiplextechnik. Dabei sind einzelne Segmente in einer Art Matrix horizontal und vertikal verbunden und werden nicht gleichzeitig, sondern nacheinander mit einer Frequenz angesteuert, die uns Menschen glauben lässt, alle Segmente „leuchten“ gleichzeitig.

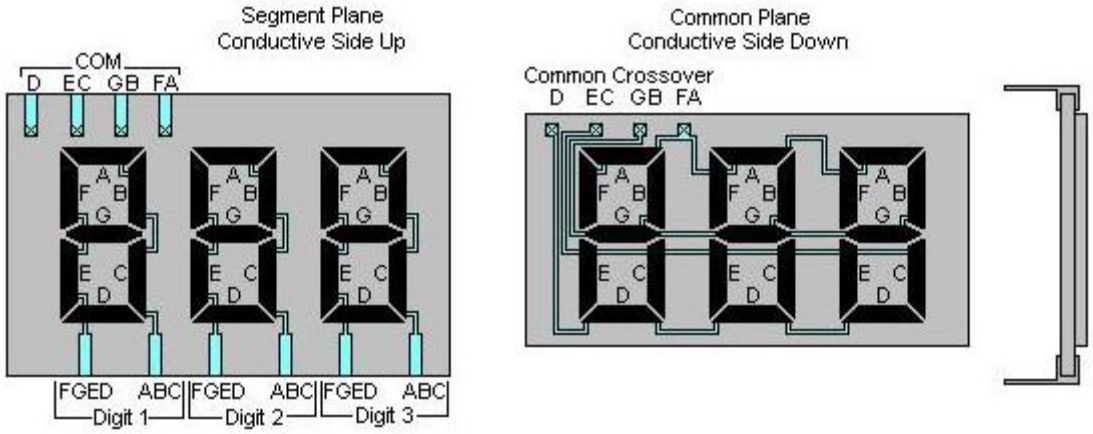


Abbildung 2: Beispiel einer Ansteuerung

In Abbildung 2 erkennt man die Bezeichnungen der Segmente A-G für jede Zifferneinheit sowie die gemeinsamen Anschlüsse für jeweils alle D, EC, GB und FA, auch Common genannt, und je zwei Anschlüsse pro Digit, mit denen die Segmente ABC oder FGED angesteuert werden. Zum Multiplexen werden nun nacheinander die Commons aktiviert und dazu passend die restlichen gemeinsamen Anschlüsse der Segmente angesteuert. Im Beispieldisplay werden nicht wie üblich die Ziffern nacheinander angesteuert, sondern (fängt man bei Common D an) Teile

aller Ziffern gleichzeitig, beginnend mit den jeweils untersten Segmenten (D), gefolgt E und C bis zuletzt F und A.

Will man nun die Zahl 001 anzeigen, so wird im ersten Multiplexzyklus Common D gleichzeitig mit FGED von Digit 1 und 2 aktiviert (dunkelblaue Markierung). Im darauffolgenden Zyklus wird Common EC zusammen mit FGED und ABC (Digit 1 und 2), sowie ABC (Digit 3) aktiviert usw.:

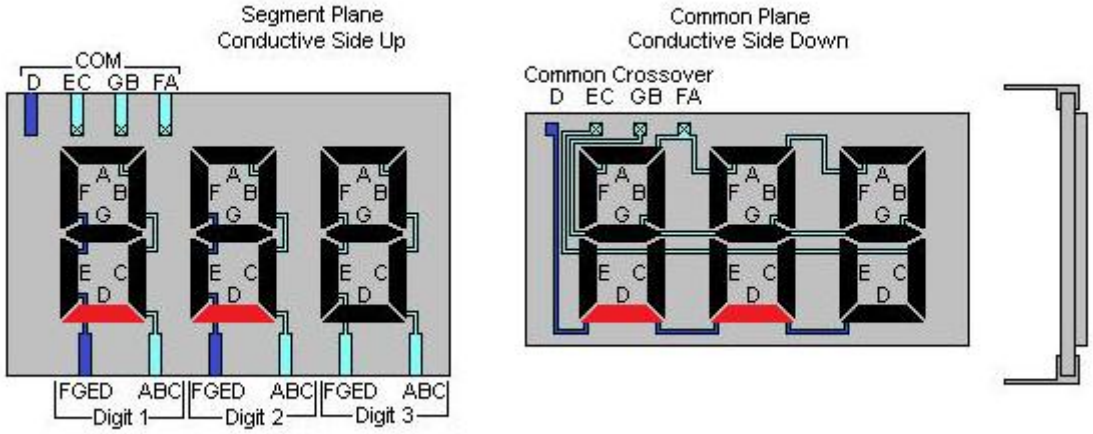


Abbildung 3: Erster Zyklus

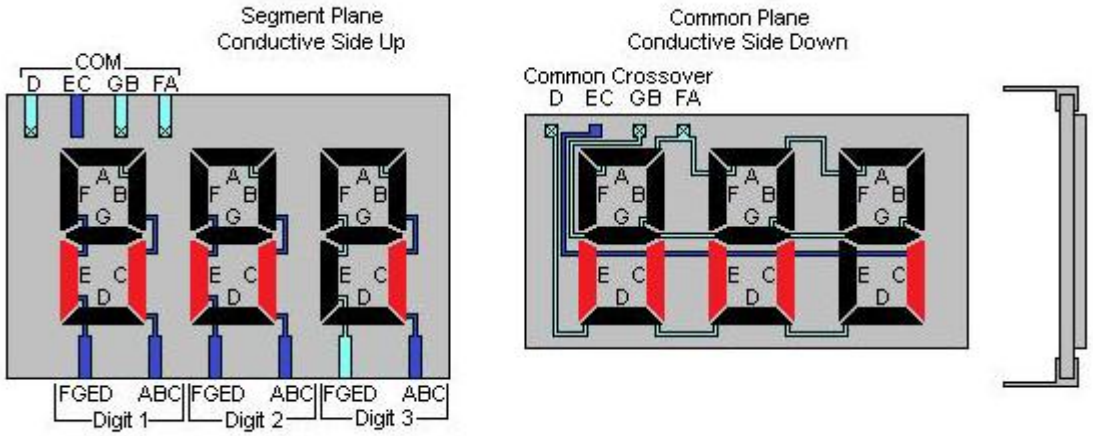


Abbildung 4: Zweiter Zyklus

Dabei bilden die Commons einen Pol der Spannungsquelle und die Datenanschlüsse (Digit 1 - 3) den anderen Pol.

3.2. Anschlussbelegung des verwendeten LCD

Da zum verwendeten Display keine Datenblätter zur Verfügung standen, musste durch Probieren eine geeignete Rechteckspannung gefunden werden, die durch Anlegen an je ein Common und einen Daten-Kanal ein einzelnes Segment aktivierte. Somit konnte das Belegungsschema des Displays in Erfahrung gebracht werden:

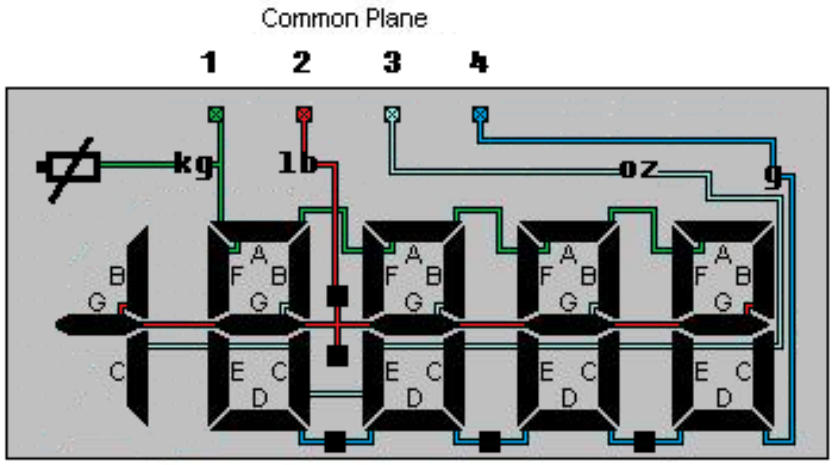


Abbildung 5: Verdrahtungsschema der Commons (Kanäle 1-4)

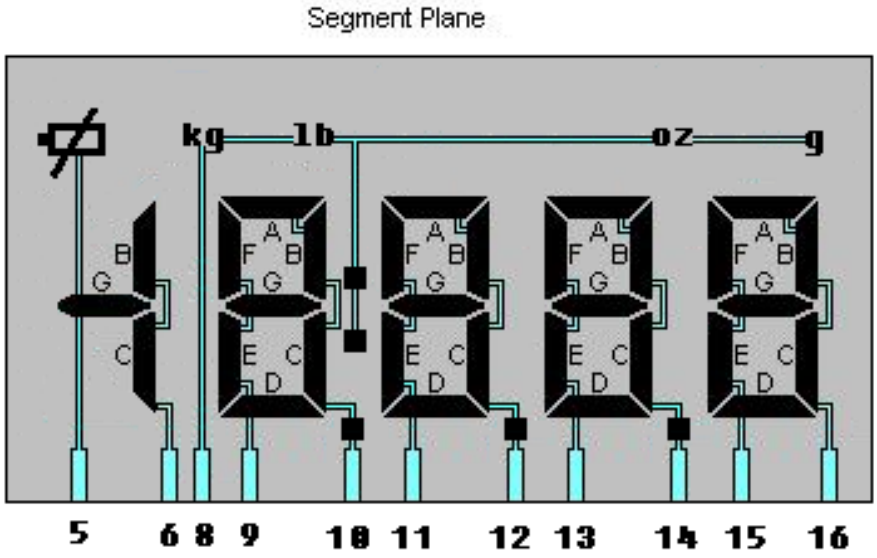


Abbildung 6: Verdrahtungsschema der Datenkanäle 5-16 (Kanal 7 ohne Funktion)

Nach diesen Vorüberlegungen war es nun möglich, die Anforderungen an die Hardware zu spezifizieren.

3.3. Multiplexen beim LCD

Ein wichtiges Merkmal bei der Ansteuerung von LCD's ist die Ansteuerung mit Wechselspannung, da es sonst zu irreparablen Schäden der Segmente kommen kann. Um den Kontrast der Anzeige zu optimieren, wird diese mit komplexen Rechtecksignalen angesteuert, welche derart gestaltet sind, dass aktive Segmente pro Zyklus einen Spannungsmittelwert (U_{RMS}) erhalten, der über derjenigen Schwelle liegt, die das Segment schwarz erscheinen lässt, während andere Segmente einen Spannungsmittelwert erhalten, der unterhalb der Schwellenspannung liegt und sie somit transparent bleiben.
Die jeweiligen Leitungen für Common und Data werden dabei so codiert, dass die Differenzspannung zwischen beiden entweder groß oder klein ist, um in der Summe eine U_{RMS} zu erhalten, die ein Segment sichtbar oder unsichtbar (transparent) erscheinen lässt.
Durch Messungen am verwendeten Display wurde festgestellt, dass es 4 Commons und 12 Data-Leitungen besitzt. Im Folgenden wird undifferenziert und vereinfacht von Kanälen gesprochen. Ein kompletter Multiplex-Zyklus dauert
13,3 ms (1/75 s) und ist in 8 Zeitschlitze unterteilt. Die Spannungen der Kanäle können dabei die Werte 0 – 3 Volt in 1 Volt-Schritten annehmen.
Nachfolgend ist beispielhaft für die Anzeige „0g" ein kompletter Zyklus aller 16 Kanäle dargestellt:

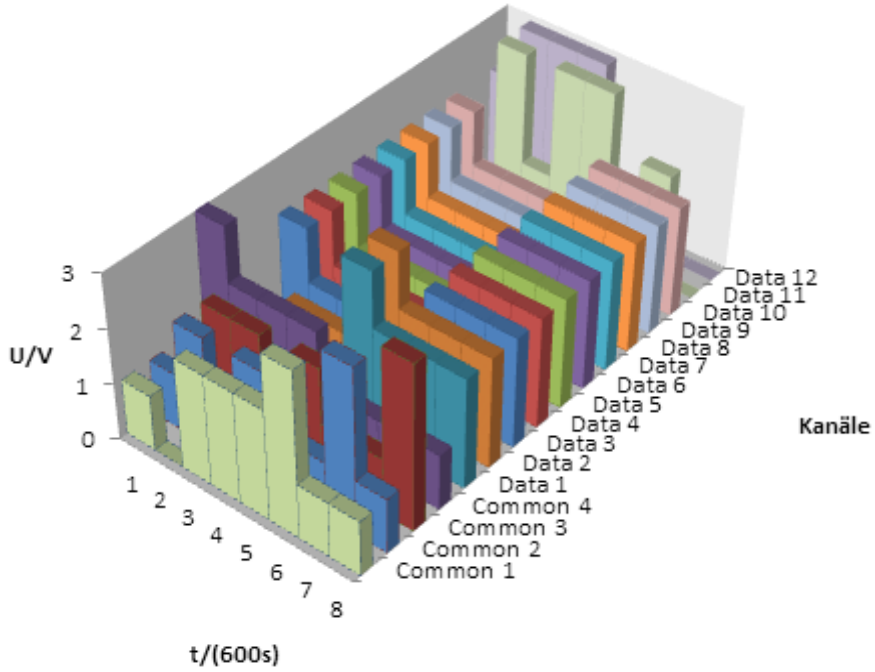


Abbildung 7: Rahmendiagramm eines kompletten Zyklus

Im Diagramm ist eine Besonderheit gegenüber dem „einfachen“ Multiplexen zu erkennen: Es lässt sich nicht ohne weiteres feststellen, welches Common gerade das Aktive ist.
Es scheint hingegen so zu sein, dass jedes Segment zu jeder Zeit mit einer Spannung angesteuert wird und somit dann, wenn z.B. die Segmente zu Common D (Abb. 2) angesteuert werden sollen, auch die Segmente der restlichen Commons einen Spannungswert erhalten. Dies wirft folgende Fragen auf: Beeinflussen sich dadurch die Kanäle gegenseitig? Warum sind nur die Segmente aktiv, die auch aktiv sein sollen?

Wie bereits erwähnt, werden die Segmente nicht mit Gleichspannung angesteuert, sondern mit Wechselspannung. Entscheidend darüber, ob ein Segment aktiv ist oder nicht, ist der Spannungsmittelwert eines gesamten Zyklus!
Durch geschickte Codierung des Ansteuerungsschemas ist es somit tatsächlich möglich, alle Segmente gleichzeitig anzusteuern. Wenn also nur ein Segment seinen Zustand ändern soll, müssen mehrere Kanäle ihre Codierung ändern.
Erläuterungen zu beispielhaften Codierungsschemen können im Datenblatt zum PCF85162 der Firma NXP nachgelesen werden.

4. Hardware

4.1. Vorüberlegungen

Grundlegend muss die Hardware Folgendes leisten:

Es müssen 16 Kanäle innerhalb einer bestimmten Zeit mit einem ADU ausgelesen und verarbeitet werden, damit die angezeigten Messwerte der Digitalwaage auch als Zahlenwert im Controller vorliegen.

Hierfür wurden zuerst die absoluten Grenzen bestimmt:

Das Display wird mit 75 Zyklen pro Sekunde angesteuert. Ein Zyklus besteht aus 8 Zeitschlitzen, also dauert ein Zeitschlitz 1,66 ms. Während dieser Zeit bleiben die Spannungswerte der Kanäle konstant und es müssen alle 16 Kanäle abgetastet werden.
Somit darf der Analog-Digital-Umsetzer für eine Wandlung eine maximale

Umsetzzeit von $\frac{1\ s}{75\ Zyklen * 8\ Zeitschlitze * 16\ Kanäle} = \mathbf{104\ \mu s}$ benötigen.

Geht man davon aus, dass Pufferzeiten für z.B. Flankenwechsel auf den Signalleitungen eingeplant werden müssen, verringert sich die maximale Umsetzzeit noch weiter.

Die Bittiefe des ADU spielt eine untergeordnete Rolle, da lediglich die Werte 0, 1, 2 und 3 Volt erkannt werden müssen. Es würde also ein **2 Bit-ADU** ausreichend sein.

Der ADU bzw. der Controller oder die Schaltung muss in der Lage sein,
16 Kanäle abtasten zu können.

4.2. Festlegung der Komponenten

Da die gängigen 8 bzw. 16-Bit-Mikrocontroller nur 8 ADU-Eingänge haben und diese intern auf nur einen einzigen ADU multiplext werden, musste eine Lösung außerhalb des Controllers gefunden werden.

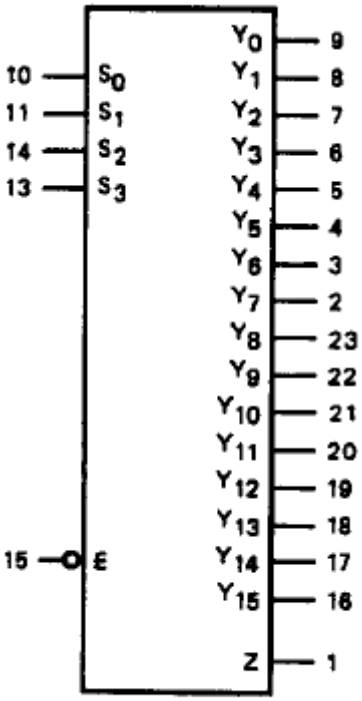


Abbildung 8: Logic Symbol des 4067

Die Wahl fiel auf den 74HC4067, ein 16 Kanal Analog Multiplexer. Er besitzt 16 Ports Y_N, welche durch binäre Vorwahl auf einen Port Z (oder umgekehrt) geschalten werden können.
Das Propagation Delay beträgt 20 ns bei 5V U_B. Die turn-on/off-time beträgt laut Datenblatt ca. 70 ns.

Eingangs wurde festgestellt, dass pro Abtastzyklus 104 µs zur Verfügung stehen. Die Verzögerungszeiten von <0,1µs bedürfen daher keiner besonderen Berücksichtigung.

Als Controller wurde ein ATmega16 von Atmel gewählt. Dieser kann mit 16 MHz betrieben werden, besitzt unter anderem 4 Ports mit je 8 Kanälen, 8 ADC Eingänge, eine JTAG-Schnittstelle, eine I2C-Schnittstelle, mehrere Timer/Counter sowie Interruptquellen.

Dies ist für die vorliegende Anwendung ausreichend, ist weder unter- noch überdimensioniert und lässt genügend Spielraum für spätere Erweiterungen.

5. Funktionsbeschreibung / Softwarebeschreibung

5.1. Überblick

Das Programm läuft nach einer Initialisierung der Timer und anderer Komponenten in 2 Phasen ab:

Phase 1 – Abtastphase:
Hier werden alle 16 Kanäle je 8 mal abgetastet, da ein Zyklus der Waage aus 8 Zeitschlitzen besteht.
Die Abtastphase findet nebenläufig statt, während die Hauptschleife auf das Ende der Abtastphase wartet.

Phase 2 – Berechnung:
Es werden aus den abgetasteten Spannungswerten die korrespondierenden Anzeigewerte berechnet.

5.2. Feinstruktur des Programms

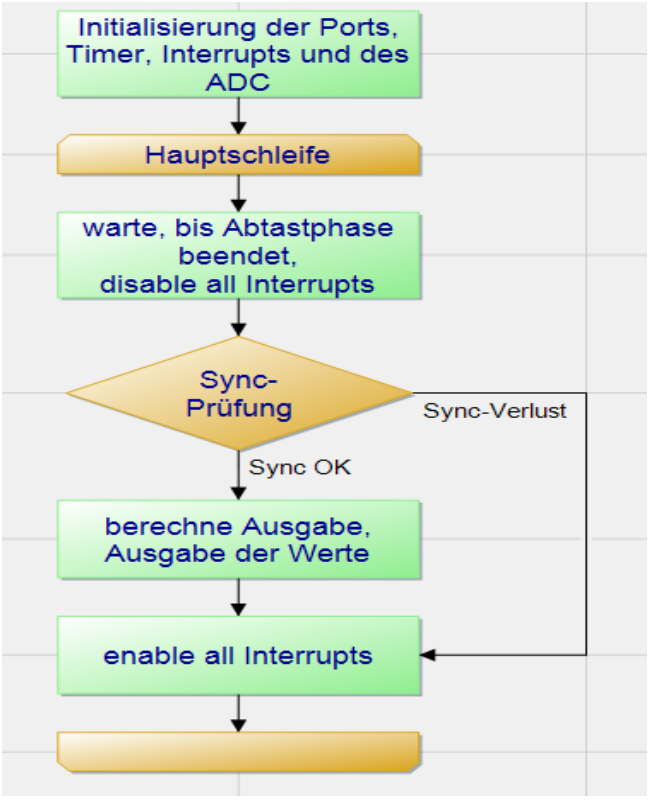


Abbildung 9: Hauptprogramm

Im Programm werden zuerst die Ports, die Timers, der ADC und die Interrupts initialisiert.

In der folgenden Hauptschleife werden nacheinander das Ende der Abtastphase abgewartet, alle Interrupts gesperrt, die Synchronisation geprüft und die Ausgabewerte mit Hilfe der Voltmatrix berechnet und ausgegeben.

Da für diese Berechnung der Inhalt der Voltmatrix unverändert sein muss, werden während der Berechnungsphase alle Interrupts deaktiviert (die Voltmatrix wird in Phase 2 von nebenläufigen Prozessen verändert).

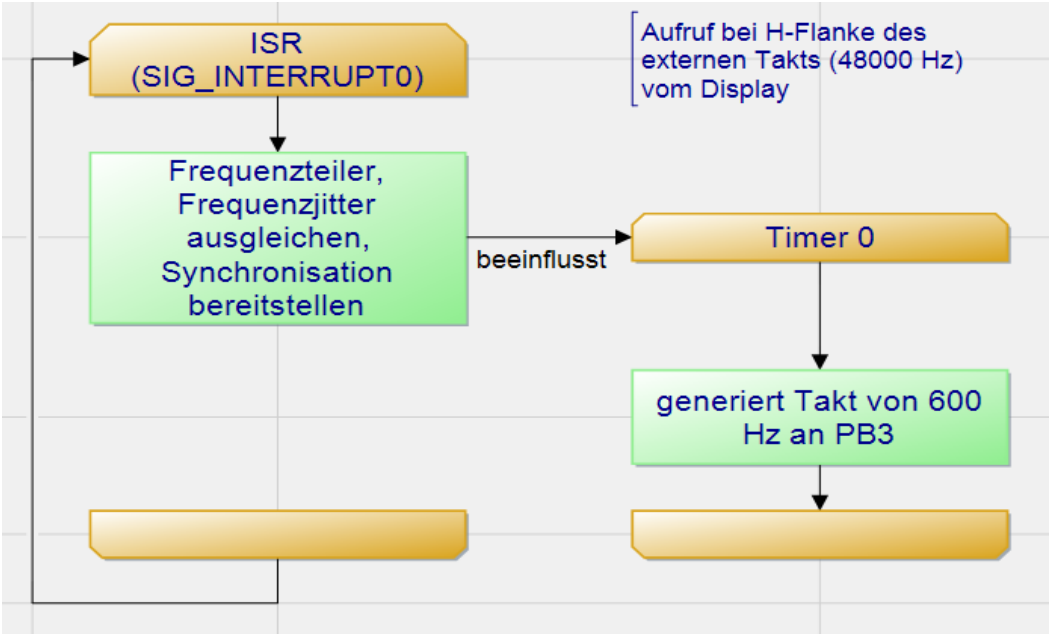


Abbildung 10: Takt und Synchronisation

Nebenläufig wird mit Timer 0 und Interrupt 0 die Synchronisation sichergestellt und der Takt für das Abtasten bereitgestellt. Nähere Beschreibungen dazu findet man im nächsten Abschnitt.

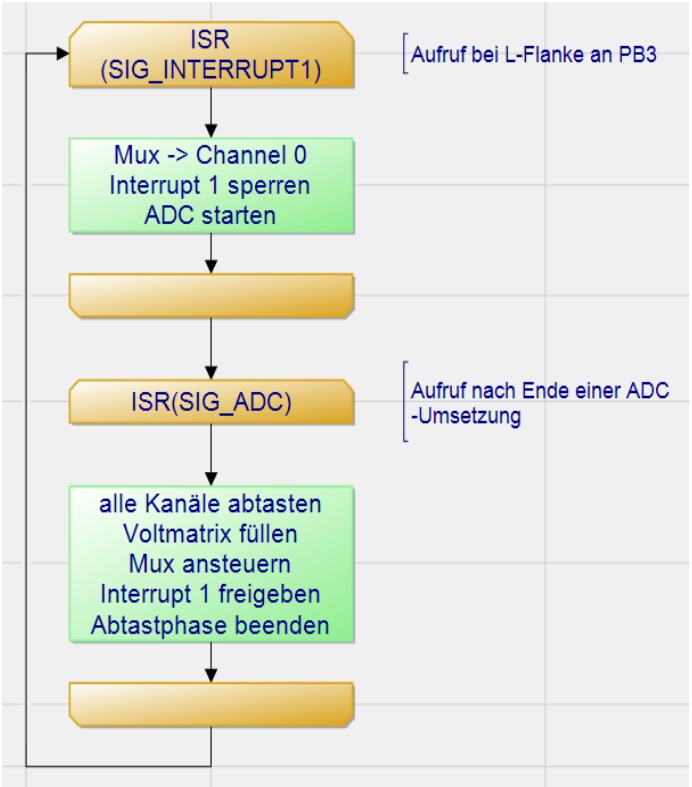


Abbildung 11: Abtasten

Ebenso nebenläufig findet das Abtasten statt. Der an PB3 bereitgestellte Takt startet den Abtastzyklus, in welchem zuerst der externe Multiplexer angesteuert wird und einen Kanal vom Display auf den ADC0 des Mikrocontroller legt.

Während der Abtastphase wird die Voltmatrix mit den entsprechenden Werten gefüllt. Sobald alle Kanäle in allen 8 Zeitschlitzen abgetastet wurden, wird die Abtastphase beendet (eine Variable wird gesetzt) und die Berechnungsphase kann beginnen.

5.2.1. Synchronisation

Die ständige Synchronisation der Abtastungen mit dem Takt des Displays sind essenziell für die Funktion des Programms.
Um den Wert der Displayanzeige zu erhalten, muss immer je ein Zyklus komplett erfasst werden, erst dann lässt sich der Inhalt des Displays berechnen. Für einen vollständigen Zyklus (t = 13,3 ms) müssen 144 Abtastungen vorgenommen werden (16+1 Kanäle mal 8 Zeitschlitze). Diese Abtastungen können jedoch nicht willkürlich vorgenommen werden, sondern müssen immer zum richtigen Zeitpunkt geschehen. Der Abtasttakt und die Synchronisation müssen sicherstellen, dass innerhalb eines Zeitschlitzes alle 16+1 Kanäle abgetastet werden und die zusammengehörigen Abtastungen nicht über einen Zeitschlitz „hinausragen“.

Sichergestellt wird die Synchronisation folgendermaßen: an Port PD2 liegt ein ca. 48 kHz Takt, abgegriffen vom Display, an. Geteilt durch 80 ergeben die 600 Hz, was der Länge eines Zyklus entspricht. Durch Auszählen der Flanken kann so sicher erkannt werden, wann die Zeitdauer eines Zyklus abgelaufen ist. Nun muss aber noch der Beginn der Zeitschlitze erkannt werden. Es gibt dabei keine feste Reihenfolge der Zeitschlitze im gesamten Datenstrom, wichtig ist nur, dass 8 aufeinander folgende Zeitschlitze erfasst werden. Diese stellen dann einen Zyklus dar.
Ein weiterer 600 Hz Takt wird im Mikrocontroller mit Timer 0 generiert, die fallenden Flanken lösen den Abtastzyklus aus. Beim Einschalten des Mikrocontrollers sind dieser Takt und die Zeitschlitze willkürlich gegeneinander verschoben und es kann passieren, dass ein Teil der 16+1 Abtastungen in einem Zeitschlitz, die restlichen Abtastungen im nächsten Zeitschlitz erfolgen. Überprüft wird dies, indem bei allen 8 Zeitschlitzen der 17. ausgelesene Kanal (an 17. Stelle wird einfach noch einmal Kanal 1 ausgelesen) mit dem ersten verglichen wird, diese müssen identisch sein. Sind sie es nicht, liegt eindeutig ein Synchronisationsverlust vor. Ausgeglichen wird dieser, indem die Phase des Taktes an PD3 um 180° verschoben wird. Somit liegt die Abtastphase nun nicht mehr zwischen zwei Zeitschlitzen, sondern ziemlich genau mittig in einem Zeitschlitz, siehe Abbildung 14.
Da der vom MC generierte 600 Hz Takt nicht exakt dem Takt des Displays entspricht, wird der entstehende Jitter ständig korrigiert (Quelltextbeschreibung S. 25 oben).

Um während der Entwicklung die Abtastvorgänge sichtbar zu machen, wurde ein Portpin des ATmega so angesteuert, dass er während einer ADC-Abtastung High war. So konnten mit dem Oszilloskop die Abtastvorgänge den Displaysignalen

gegenübergestellt und genau nachvollzogen werden, wann die Abtastungen stattfinden.

Die nachfolgenden Abbildungen verdeutlichen den zeitlichen Zusammenhang zwischen Abtastungen, Takt und den Zeitschlitzen.

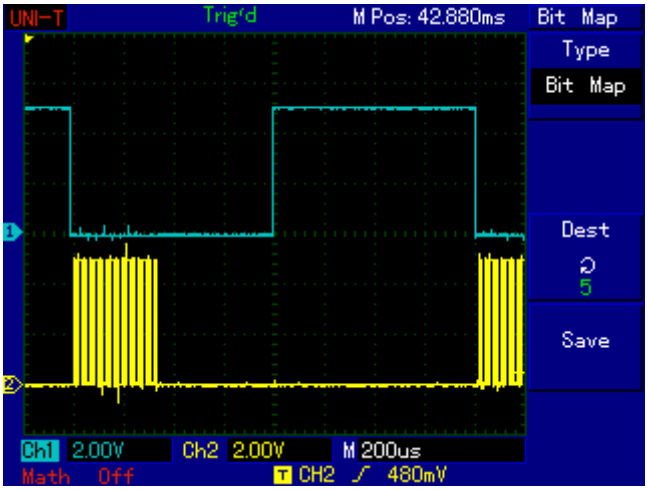


Abbildung 12: Abtastvorgänge

Kanal 1 (blau) in Abbildung 12 zeigt den vom Timer 0 generierten Takt.

Kanal 2 (gelb) zeigt 16 aufeinander-folgende Spitzen, diese stellen die 16 Abtastungen dar. Man erkennt außerdem, dass der Beginn der Abtastungen mit der fallenden Flanke des Taktes initiiert wird.

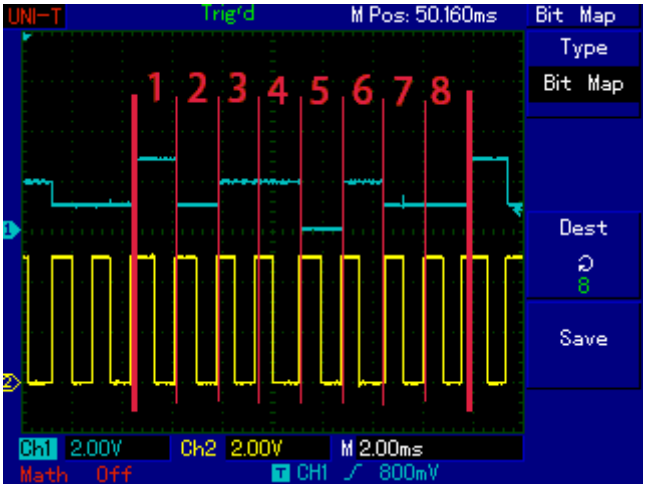


Abbildung 13: Takt synchron zum Displaykanal

Kanal 1 (blau) in nebenstehender Abbildung zeigt einen Datenkanal des Displays. Zu sehen ist ein Zyklus, bestehend aus 8 Zeit-schlitzen mit verschiedenen Span-nungswerten. Danach wieder-holt sich das Spannungsmuster.

Kanal 2 (gelb) zeigt den von Timer 0 generierten und von Interrupt 0 manipulierten Takt. Die fallende Flanke liegt fast mittig in einem Zeitschlitz.

Die rechte Abbildung zeigt einen Überblick und einen Ausschnitt aus einem Zyklus:

Blau dargestellt ist ein Kanal des Displays, gelb dargestellt sind die Abtastungen.

Zu erkennen ist, dass alle 16+1 Abtastungen in einem Zeitschlitz stattfinden und somit Synchronisation vorliegt.

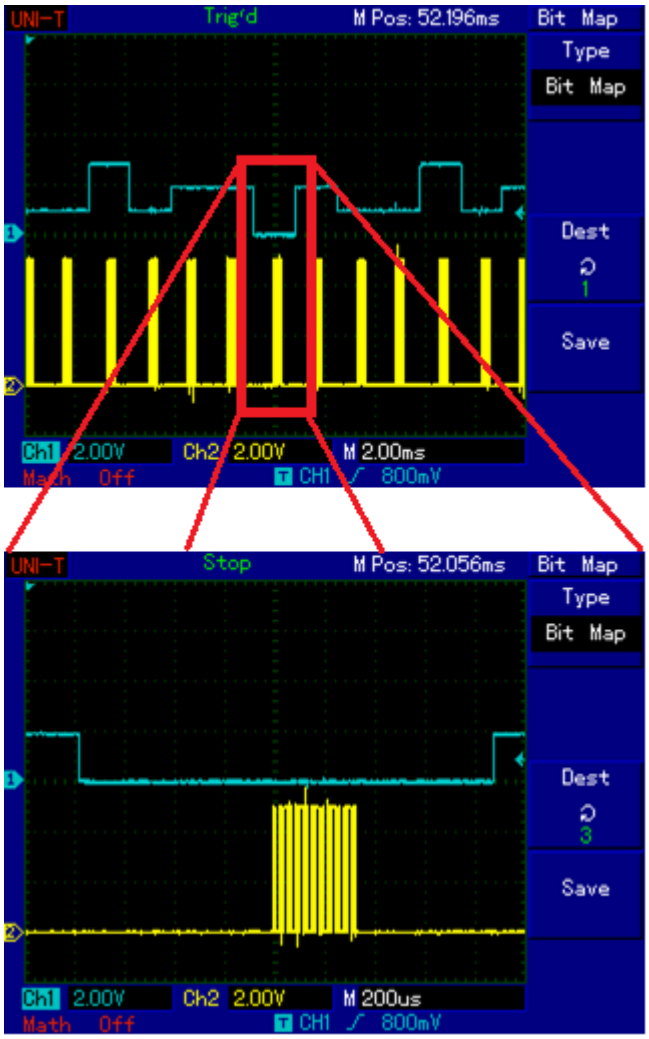


Abbildung 14: Abtastungen im Zeitschlitz

5.2.2. Kanal-Multiplexer

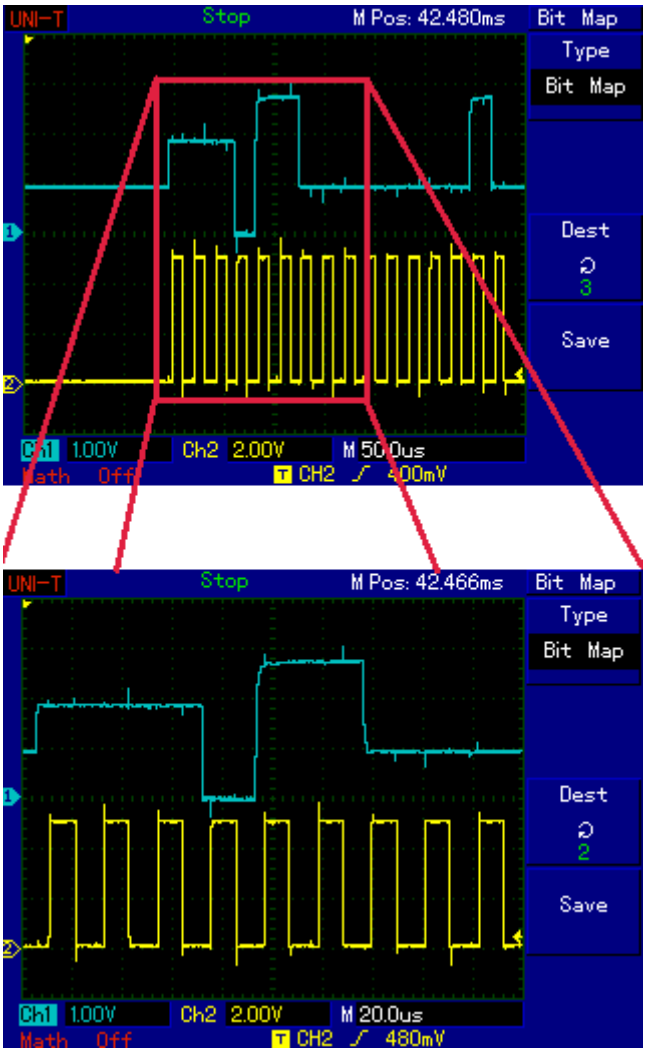


Abbildung 15: Multiplexdiagramm

Die Abbildung zeigt in blau den Ausgang des externen Multiplexers, welcher gleichzeitig der ADC0 – Eingang des Mikrocontrollers ist. Man sieht, dass die Kanäle gewechselt werden und während der Abtastungen in gelb die Signale konstant bleiben.

Vergleicht man die Spannungswerte des blauen Kanals von nebenstehender Abbildung mit Abbildung 5 so erkennt man, dass gerade Zeitschlitz 5 abgetastet wird:
2,2,2,0,3,3,1,1,1,1,1,1,1,1,3,1

5.2.3. Phase 2 - Berechnung

In der Berechnungsphase werden alle Interrupts gesperrt und es findet die Syncprüfung anhand der vorher gesammelten Daten statt. Fällt sie negativ aus, wird die Berechnungsphase sofort beendet und in der nächsten Abtastphase der Takt angepasst.

Fällt die Syncprüfung positiv aus, werden aus den Spannungswerten der Voltmatrix die Segmentzustände berechnet. Dafür wird in einer Schleife aus den Werten aller Commons und Dataleitungen über alle 8 Zeitschlitze eines Zykluses die Differenzspannung zwischen beiden Leitungen berechnet. Die Beträge dieser Spannunswerte werden über einen Zyklus addiert (ähnlich der Berechnung einer U_{RMS}). Als Ergebnis kommen nur die Werte 8 und 12 Volt vor. Diese sind direkt mit dem Segmentzustand gleichzusetzen: 8 Volt steht für ein transparentes Segment, 12 Volt bedeutet, das Segment ist sichtbar. Aus den Segmentzuständen werden letztendlich die Ziffernwerte berechnet und für die weitere Verarbeitung zur Verfügung gestellt.

Nachfolgend ist ein Foto vom Testaufbau mit binärer LED-Ausgabe des abgetasteten Wertes „19“ abgebildet.

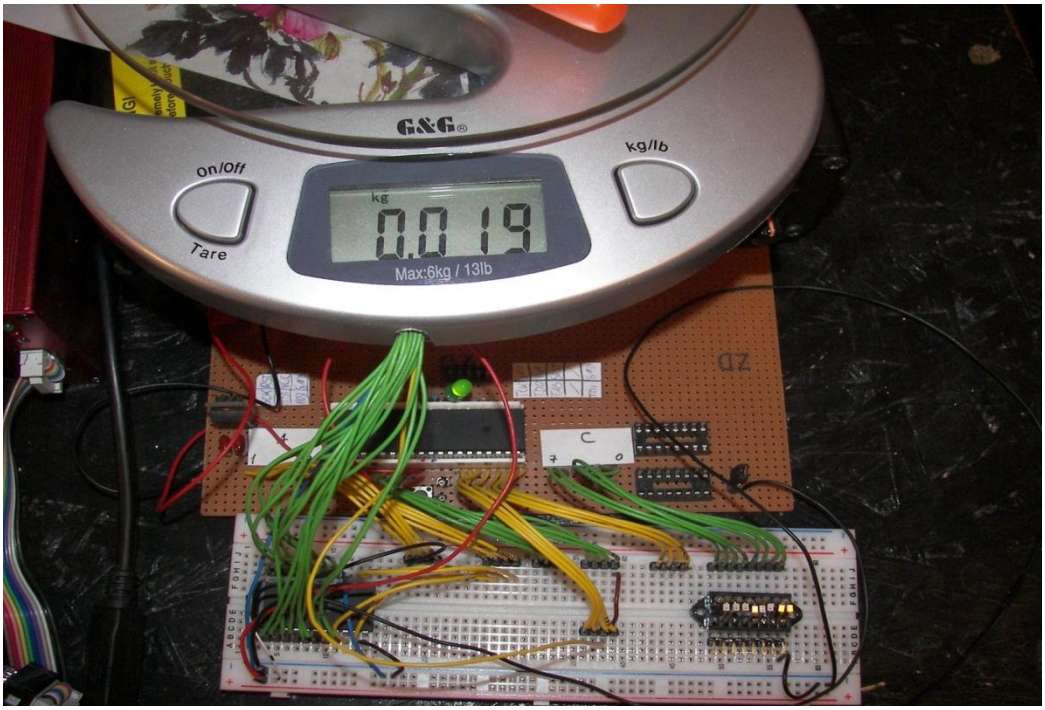
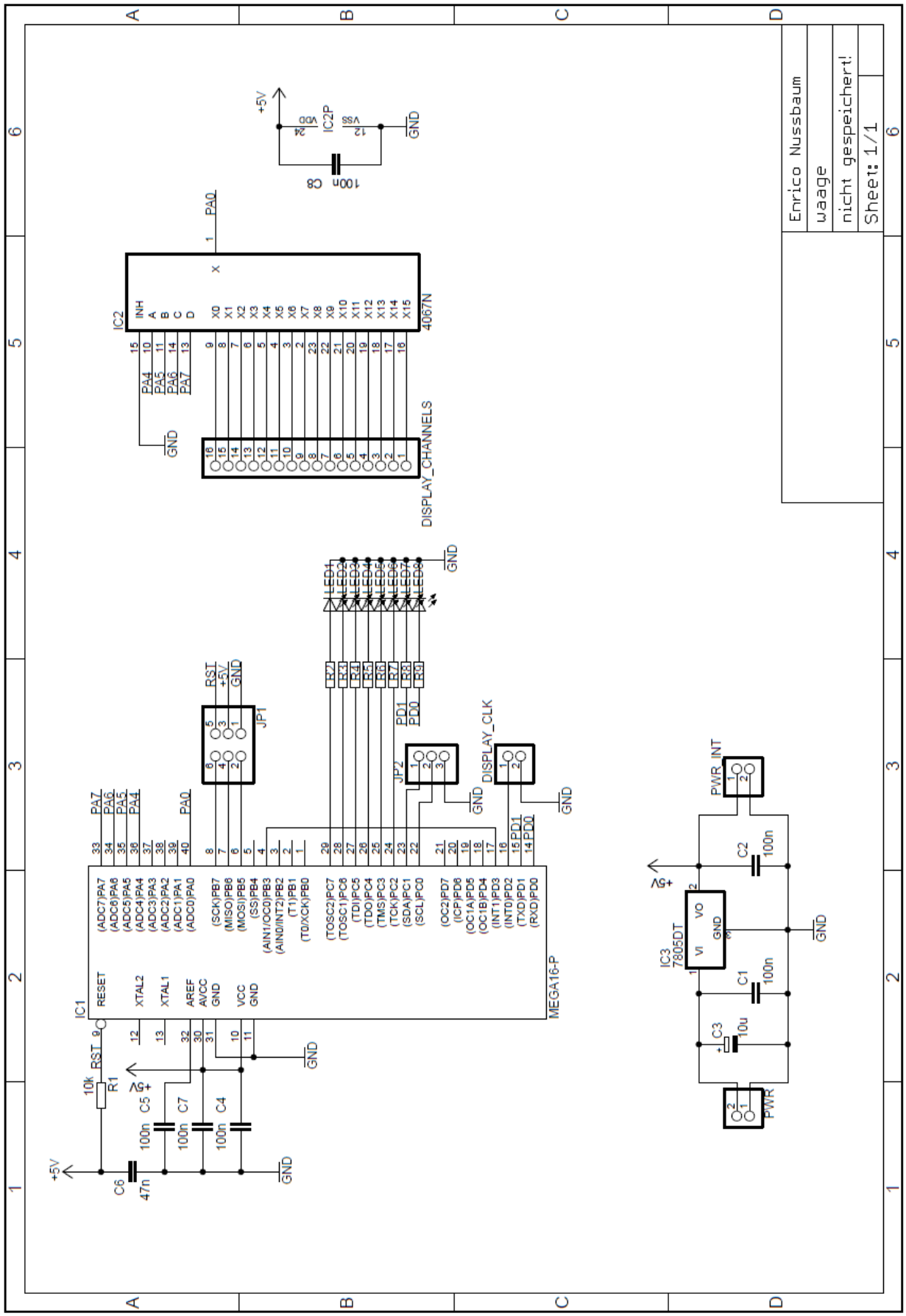
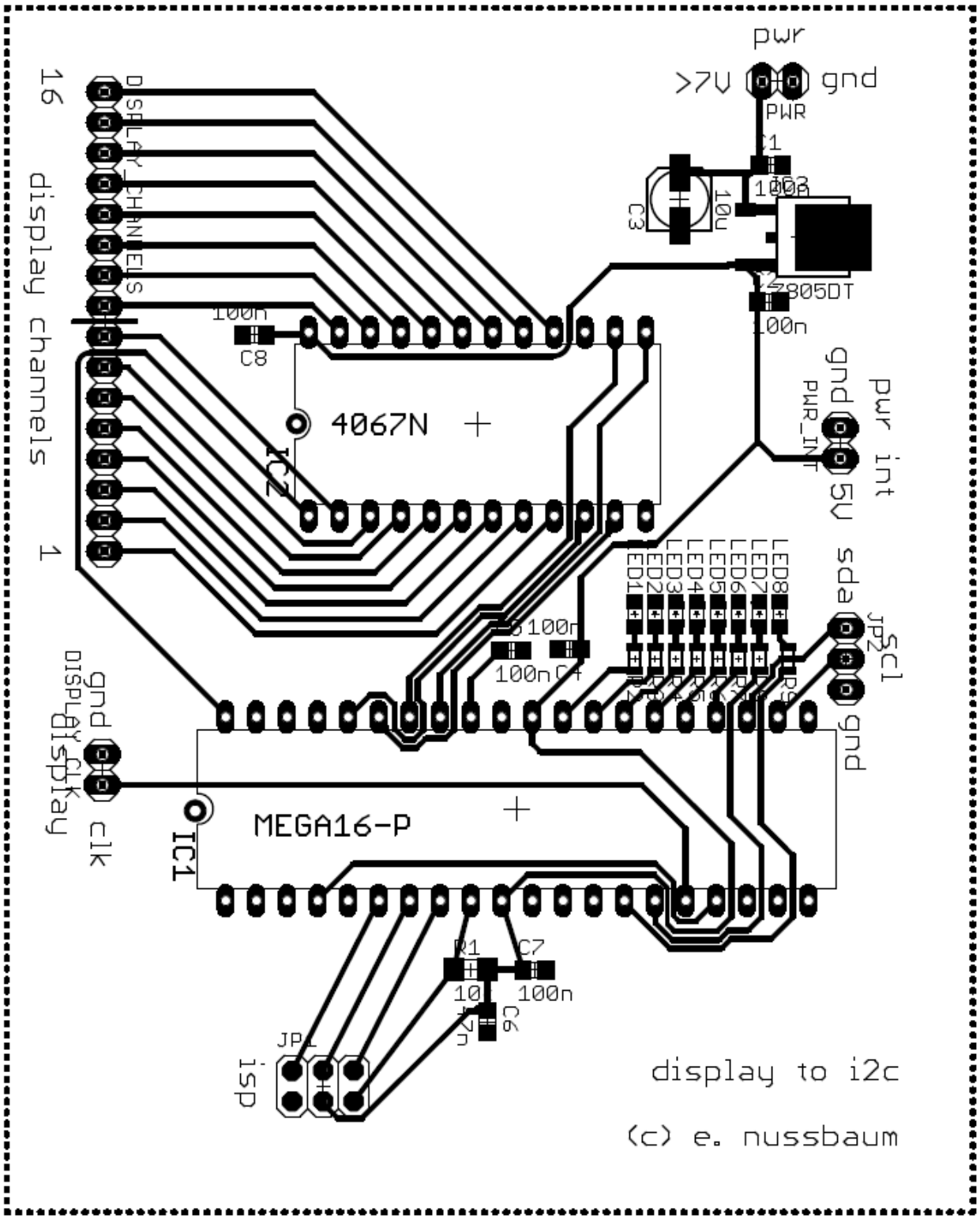


Abbildung 16: Testaufbau

6. Schaltplan



7. Layout (ohne Groundplane)



8. Quelltext

```
/*
 * main.c
 *
 * Created: 22.09.2011 00:04:07
 *  Author: Enrico Nussbaum
 */

// timing infos:
// 75 mal pro sekunde wird voltmatrix neu beschrieben
// display aktualisiert wert 3 mal pro sekunde
// dh trotz dauernder aktualisierungen ist mind ca 25 mal in folge (40 ms)
// der inhalt der voltmatrix identisch
//

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


// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
// ------------------------ define globals ------------------------------
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

// wird in adc-routine (sig_interrupt1) genutzt, enthält informationen
// zum aktuell gewandelten kanal
int channel = 0;

// temp-werte für spannungen der 16 + 1 kanäle
int Channel[17] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// adc zeitschlitz 1 bis 8 (eine refreshpariode besteht aus 8 zeitschlitzen.
// alle 8 werden benötigt um segmentzustände zu berechnen)
int zeitschlitz = 1;
```

```
// array für voltwerte der kanäle
// 8 zeitschlitze
// 16 + 1 kanäle (letzter dient syncprüfung)
// enthält werte 0, 1, 2, 3 (volt)
/*  INFO:
    - werte entsprechen displayanzeige "0g"
    - jeweils letzter wert ist gleich dem ersten und dient der syncüberprüfung:
        bei syncronität müssen erster und letzter gleiche werte liefern
    - wurde nur aus testzwecken bereits mit werten gefüllt
    - vorgegebene werte werden mit dem ersten abtastzyklus überschrieben
*/
int VoltMatrix[8][17]={ {1, 1, 1, 3, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 1},
                        {0, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 0},
                        {2, 0, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 2},
                        {2, 2, 0, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 2},
                        {2, 2, 2, 0, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2},
                        {3, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 3},
                        {1, 3, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 1},
                        {1, 1, 3, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 1}
};

// enthält segmentzustände des Displays
// 4 commons (zeilen)
// 12 segmentanschlüsse (spalten)
// 0 = off
// 1 = on
short int SegmentMatrix[4][12]={    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

// überlaufwert für timer0 [für synchronen abtasttakt =>
// timerOV = f_clock / (2*teiler*gewünschte frequenz) - 1]
// wurde durch nachträgliches probieren auf besten wert angepasst
short unsigned int timerOV = 0b01101101;

// zählvariable für int0
short int TaktTeiler = 0;

// variable zum synchronisieren des abtasttaktes
unsigned short int lostSync = 0;

// phasenmarker
unsigned short int abtastPhase = 1;
```

```
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
// --------------------------- define functions ----------------------------
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

// ---------------------- rechne adc-wert (pin A0) in volt um --------------

// adc 8 bit auflösung -> 255 entspricht 5V, 51 entspricht 1V, alles zw 25 und 76
// wird als 1V erkannt.
int AdcToVolt(int chx){
    int volt;
    if (chx<=25){volt=0;}
    else if ((chx > 25) && (chx <= 76)){volt=1;}
    else if ((chx > 76) && (chx <= 127)){volt=2;}
    else if ((chx > 127) && (chx <= 178)){volt=3;}
    else if ((chx > 178) && (chx <= 229)){volt=4;}
    else if (chx > 229){volt=5;}
    return volt;
}

// ------------------------------------------------------------------------------
// int PD2, hier wird mit timer 0 generiertes signal auf signal vom display synchronisiert
// wird bei jeder steigenden flanke an PD2 aufgerufen
// ------------------------------------------------------------------------------
ISR(SIG_INTERRUPT0){

    // der abgegriffene takt des displays ist 80 mal schneller als eine refresh-periode
    // des displays, der interrupt muss also ca 80 mal durchlaufen werden, um synchron
    // zu einer periode des displaytaktes zu sein (bei leerer else{} sinds genau 80 mal..)
    if (TaktTeiler<79)  {
        TaktTeiler++;
    }

    // wenn syncverlust festgestellt: verschiebe die phase des abtasttaktes um 180°,
    // danach liegen abtastzeitpunkte mittig in einer Refresh-Periode
    else{
        if (lostSync == 1){

            if(TCNT0 < (timerOV>>1)){       // wenn TCNT0 noch nicht bis zur Hälfte des
                TCNT0=TCNT0+(timerOV>>1);   // Maximalwerts (timerOV) gezählt hat, addiere
            }                               // die hälfte des maximalwerts zu TCNT0
                                            // (entspricht Phasenverschiebung des
                                            // Abtastbursts um 180 grad)

            else {
                TCNT0=TCNT0-(timerOV>>1);   // ansonsten ziehe die hälfte des maximalwerts ab
            }
            lostSync = 0;                   // syncmarker zurücksetzen
            TaktTeiler = -4;                // taktteiler zurücksetzen: da 80 takte an PD2
                                            // genau 8 perioden vom abtasttakt entsprechen,
                                            // nun aber die phase des abtasttaktes um 180 grad
                                            // verschoben wurde, was 5 taktflanken an
                                            // PD2 entspricht, muss der taktteiler entsprechend
                                            // angepasst werden.
                                            // warum -4 statt -5? aus dem gleichen grund, warum
                                            // die schleife nur bis 79 statt bis 80 zählt:
                                            // die schleife ist nicht leer und verbraucht somit
                                            // rechenzeit -> 1 takt
        }
```

```
        else {                                  // kein syncverlust -> ausgleich des frequenzjitters
            TCNT0 = (timerOV-2);                // bei interrupt wird timer0counter auf kurz vor
                                                // wechsel gesetzt (im ggsatz zu nullsetzen werde
                                                // dadurch doppelte periodenlängen verhindert)
            TaktTeiler = 0;
        }
    }
}

// ------------------------------------------------------------------------------
// int bei fallender flanke an PD3, hier wird auslesen der kanäle mittels adc initiiert
// wird nur zum start eines durchgangs (zeitschlitzes) aufgerufen
// ------------------------------------------------------------------------------
ISR(SIG_INTERRUPT1){
    abtastPhase = 1;        // beginne neue Abtastphase
    PORTA &= 0b00001111;    // wähle kanal 0 am mux
    GICR &= ~(1 << INT1);   // sperre INT1 (starte adc-routine)
    ADCSRA |= (1 << ADIE);  // adc interrupt enable
    ADCSRA |= (1 << ADSC);  // starte adc
}


// ------------------------------------------------------------------------------------------
// int bei beendigung adc-wandlung
// ------------------------------------------------------------------------------------------
ISR(SIG_ADC){

    Channel[channel] = (int)(ADCL);     // erst L dann H auslesen, L wird verworfen
    Channel[channel] = (int)(ADCH);     // speichere adc wert in kanalspeicher

    unsigned char port = PORTA;         // ADC kanäle einlesen (interessant ist nur ADC0,
                                        // er enthält den zu wandelnden wert)

    if (channel<15){                    // kanalwahl über mux

        // setze mux ansteuerbits
        // nibbles tauschen: auf high nibble liegen die 4 ansteuerbits des mux
        // durch verschieben auf low-nibble, plus 1 und zurückschieben kann nächster
        // mux-kanal gewählt werden
        port = (port >> 4);
        port++;
        port = (port << 4);
        PORTA = port;

        // stoße nächste wandlung an
        channel++;              // setze nächsten zu wandelnden kanal
        ADCSRA |= (1 << ADSC);  // starte adc
    }

    else if(channel == 15){     // wenn letzter Kanal eingelesen wurde
        PORTA &= 0b00001111;    // setze kanal = 0 (für syncprüfung)
        channel++;
        ADCSRA |= (1 << ADSC);  // starte adc
    }
```

```
    else {                      // wenn alle kanäle + sync information eingelesen wurden

        // konvertiere adc-werte von 16 + 1 kanälen in spannungen und speichere in matrix
        // (ca 75 mal pro sekunde)
        for (int kanal = 0; kanal < 17; kanal++){
            VoltMatrix[zeitschlitz-1][kanal] = AdcToVolt(Channel[kanal]);
        }

        // setze zahl des nächsten zeitschlitzes
        if (zeitschlitz >= 8){      // wenn alle 8 zeitschlitze abgetastet wurden
            zeitschlitz = 1;        // setze durchgang zurück
        }
        else zeitschlitz++;         // setze nächsten durchgang (zeitschlitz)

        channel = 0;                // setze nächsten zu bearbeitenden kanal zurück
        ADCSRA &= ~(1 << ADIE);     // adc interrupt disable
        GICR |= (1 << INT1);        // gib INT1 frei (starte adc-routine)
        abtastPhase = 0;            // ENDE abtastphase, alle 8 zeitschlitze wurden abgetastet
                                    // freigabe der berechnungen
    }
}

// ------------------------------------------------------------------------------
// berechnet segmentzustände (an, aus) aus den informationen der VoltMatrix und
// speichert sie in die SegmentMatrix
// ------------------------------------------------------------------------------
void getSegments(void){

    int Summe = 0;      // hilfsvariable zum berechnen der segmentzustände ein/aus

    // rechne einzel-volt-werte in segment-zustände um
    for (int common = 0; common < 4; common++){          // common, an stelle 0-3 stehen
                                                         // die common-volt-werte im array

        for (int segment = 4; segment < 16;segment++){   // "segment", an stelle 4-15 stehen
                                                         // die segment-volt-werte im array

            // berechne die summe der differenzspannungen pro segment aus den
            // zeitschlitzen 1 bis 8
            for (int pass = 1; pass < 9; pass++){
                Summe += fabs(VoltMatrix[pass-1][common]-VoltMatrix[pass-1][segment]);
            }

            // speichere segmentzustände
            if (Summe == 8) SegmentMatrix[common][segment-4] = 0;    // 8 => "segment aus"
            if (Summe == 12) SegmentMatrix[common][segment-4] = 1;   // 12 => "segment an"
            Summe = 0;
        }
    }
}
```

```
// ------------------------------------------------------------------------------------
// berechnet aus den zuständen einer 7 segment-anzeige die resultierende ziffer
// und gibt sie zurück
// ------------------------------------------------------------------------------------

int getNumber(  short int a, short int b, short int c, short int d,
                short int e, short int f, short int g){
    short int erg = 0;
    if ((a+b+c+d+e+f+g)==0){erg = 0;}                   // alle segmente aus
    else if ((a+b+c+d+e+f+g)==7){erg = 8;}              // alle segmente an
    else if (((a+b+c+e+f+g)==6)&&(d==0)){erg = 0;}      // alle segmente an außer d
    else if (((a+b+d+e+g)==0)&&(c+f==2)){erg = 1;}      // alle segmente aus außer c, f
    else if (((a+c+d+e+g)==5)&&(b+f==0)){erg = 2;}      // alle segmente an außer b, f
    else if (((a+b+d+f+g)==5)&&(c+e==0)){erg = 5;}      // alle segmente an außer c, e
    else if (((a+c+d+f+g)==5)&&(b+e==0)){erg = 3;}      // alle segmente an außer b, e
    else if (((b+c+d+f)==4)&&(a+e+g==0)){erg = 4;}      // alle segmente an außer a, e, g
    else if (((a+b+d+e+f+g)==6)&&(c==0)){erg = 6;}      // alle segmente an außer c
    else if (((a+c+f)==3)&&(b+d+e+g==0)){erg = 7;}      // alle segmente an außer b, d, e, g
    else if (((a+b+c+d+f+g)==6)&&(e==0)){erg = 9;}      // alle segmente an außer e
    return erg;
}
int main(void){

    // xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    // ---------------------------- initialisierung -------------------------------
    // xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

    // ------------------------------------------------------------------------------
    // ************************ variablen ***************************************
    // ------------------------------------------------------------------------------

    short int einer = 0;
    short int zehner = 0;
    short int hunderter = 0;
    short int tausender = 0;
    int ergebnis = 0;


    // ------------------------------------------------------------------------------
    // ************************** ports *****************************************
    // ------------------------------------------------------------------------------
    UCSRB &= !(1<<RXEN);        // uart explizit ausschalten, da es sonst zu problemen
    UCSRB &= !(1<<TXEN);        // mit dem gemeinsam genutzten port kommt

    DDRB |= (1 << PB3);         // timer0 frequenzgenerator
    DDRC = 0b11111111;          // PC2-PC7 Ausgabe des abgetasteten wertes
    DDRD |= (1 << PD0);         // PD0 PD1 Ausgabe des abgetasteten wertes
    DDRD |= (1 << PD1);
    DDRA = 0b11110000;          // PA4 bis PA7 ist ausgang für ansteuerung mux
    DDRD &= ~(1 << PD3);        // PD3 = Eingang (adc-sync-spannung als trigger), löst int1 aus
```

```
// ------------------------------------------------------------------------
// ************************ timer0 ***************************************
// für syncronen rechtecktakt (abtasttakt)
// ------------------------------------------------------------------------

OCR0 = timerOV;                 // OCR0
TCCR0 = 0b00011011;             // 00011xxx: OC0 (PB3) Rechteck 1:1 frequenz variabel
                                // durch OCR0: timer zählt von 0 aufwärts und
                                // ändert beim erreichen von OCR0 den zustand von OC0 (PB3)
                                // xxxxx011: teiler = systemtakt / 64

// ------------------------------------------------------------------------
// ************************* adc *****************************************
// ------------------------------------------------------------------------
ADMUX |= (1 << REFS0)|(1 << ADLAR);              // nutze interne Uref (5V),
                                                 // linksbündig speichern
ADCSRA &= ~(    (1 << ADPS2)|                    // 000 -> adctakt = systemtakt/2
                (1 << ADPS1)|
                (1 << ADPS0));
ADCSRA |= (1 << ADEN);                           // adc wandler ein


// ------------------------------------------------------------------------
// ********************** interrupt0 *****************************************
// ruft ISR(SIG_INTERRUPT0) auf:
// modifiziert timer0 counter und passt somit den von timer0 generierten takt
//(abtasttakt) an displaytakt an
// ------------------------------------------------------------------------
MCUCR |= (1 << ISC01);      // 11: löse interrupt0 bei h-flanke an PD2 aus
MCUCR |= (1 << ISC00);
GICR |= (1 << INT0);        // gib INT0 frei (sync clock to display)


// ------------------------------------------------------------------------
// ********************** interrupt1 *****************************************
// stößt adc umwandlung synchron an
// ------------------------------------------------------------------------

MCUCR |= (1 << ISC11);      // ISC1=10: löse interrupt1 bei fallender flanke an PD3 aus
MCUCR &= ~(1 << ISC10);     // (ca aller 13ms, fallende flanke ist mittig jedes "durchgangs")

GICR |= (1 << INT1);        // gib INT1 frei (starte adc-routine)
sei();                      // alle interrupts frei


// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
// ------------------------ betriebsschleife ---------------------------------
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
while(1){


    // ------- auslesen der displaykanäle und synchronisation ------------------

    while (abtastPhase==1){}                // warte, bis abtastphase beendet
```

```
    // ++++++++++++++ syncprüfung ++++++++++++++++++++++
        // es wird überprüft, ob alle abtastwerte innerhalb eines refreshzyklus stattfanden

        cli();      // disable all int, voltmatix darf während syncprüfung nicht verändert werden

        // wenn erster vom letzten durchgang abweicht: dann syncverlust
        // 1. bis 16. durchgang ist in zeitschlitz a, 17. durchgang in zeitschlitz a+1
        if ((VoltMatrix[0][0]!=VoltMatrix[0][16])||
            (VoltMatrix[1][0]!=VoltMatrix[1][16])||
            (VoltMatrix[2][0]!=VoltMatrix[2][16])||
            (VoltMatrix[3][0]!=VoltMatrix[3][16])||
            (VoltMatrix[4][0]!=VoltMatrix[4][16])||
            (VoltMatrix[5][0]!=VoltMatrix[5][16])||
            (VoltMatrix[6][0]!=VoltMatrix[6][16])||
            (VoltMatrix[7][0]!=VoltMatrix[7][16]))  {
            lostSync = 1;   // globale sync-variable setzen, wird in ISR(SIG_INTERRUPT0) benutzt
            sei();          // starte ohne berechnungen sofort eine neue abtastphase
        }
    // +++++++++++++++++ ende syncprüfung ++++++++++++++

        else{

            // rechne volt in segmentzustände um
            getSegments();

            // ----- segmentzustände in reale dezimalwerte (gewicht) zurückwandeln --------
            // berechne dezimalwerte und hilfswerte aus segmentzuständen
            // die parameter
            einer = getNumber(  SegmentMatrix[0][11], SegmentMatrix[0][10],
                                SegmentMatrix[1][11], SegmentMatrix[1][10],
                                SegmentMatrix[2][10], SegmentMatrix[2][11],
                                SegmentMatrix[3][10]);
            zehner = getNumber( SegmentMatrix[0][9], SegmentMatrix[0][8],
                                SegmentMatrix[1][9], SegmentMatrix[1][8],
                                SegmentMatrix[2][8], SegmentMatrix[2][9],
                                SegmentMatrix[3][8]);
            hunderter = getNumber(  SegmentMatrix[0][7], SegmentMatrix[0][6],
                                    SegmentMatrix[1][7], SegmentMatrix[1][6],
                                    SegmentMatrix[2][6], SegmentMatrix[2][7],
                                    SegmentMatrix[3][6]);
            tausender = getNumber(  SegmentMatrix[0][5], SegmentMatrix[0][4],
                                    SegmentMatrix[1][5], SegmentMatrix[1][4],
                                    SegmentMatrix[2][4], SegmentMatrix[2][5],
                                    SegmentMatrix[3][4]);
            ergebnis = tausender*1000+hunderter*100+zehner*10+einer;


            // ---------------- werte ausgeben ---------------------------------------------

            PORTC = (ergebnis & 0b11111100);        // ausgabe bereitstellen
            PORTD = (ergebnis & 0b00000011);

            sei();  // gib ints wieder frei (wurden in ISR(SIG_ADC) disabled) und
                    // beginne neue abtastphase
        }
    }
}
```

9. Abbildungsverzeichnis

10. Quellenverzeichnis

10.1. Internetquellen

http://www.nxp.com/documents/data_sheet/PCF85162.pdf

www.mikrocontroller.net/ [verschiedene Seiten]

10.2. Literaturverzeichnis

Günter Schmitt. Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie. *Oldenbourg Wissenschaftsverlag GMBH*, 2007

Prof. Dr. Karlheinz Blankenbach. Vorlesungsbegleitendes Skript / FH Pforzheim *Blankenbach / Elektronische Displays / 08.11.01*