T E C H N I S C H E U N I V E R S I T Ä T B E R L I N
Fakultät IV Institut für Techn. Informatik und Mikroelektronik
Fachgebiet PDV und Robotik Prof. Dr.-Ing. Dr. h.c. Günter Hommel

Sensor-Datenerfassungssystem für eine Exoskelett-Hand

Studienarbeit

Christian Richter <uni@ch-r.de>

Berlin, 13. Oktober 2004

Hochschullehrer: Prof. Dr.-Ing. Dr. h.c. Günter Hommel
Betreuer: Dipl.-Ing. Andreas Wege

Inhaltsverzeichnis

Zusammenfassung

Für viele Bereiche der Medizin - insbesondere für die Handchirurgie - ist es vorteilhaft, die genauen Positionen und Bewegungen der Finger einer menschlichen Hand erfassen zu können. Dies ist möglich, indem man die Beschleunigung und die Winkelgeschwindigkeit jedes einzelnen Fingergliedes, des Handrückens und des Unterarms bestimmt.

Ziel der folgenden Arbeit ist es, ein System zu entwerfen und aufzubauen, welches diese Größen direkt an der Hand mißt, sie digitalisiert und die Meßwerte in Echtzeit auf einen Rechner überträgt, wo sie zur weiteren Verarbeitung zur Verfügung stehen.

Aus dieser Aufgabe ergeben sich zwei Haupt-Problemstellungen: Zunächst müssen besonders kleine und leichte Sensorschaltungen entwickelt werden, die auf den einzelnen Fingergliedern befestigt werden können. Außerdem wird ein Datenerfassungssystem (DAQ-System) mit entsprechend vielen Eingängen benötigt, das über eine Standardschnittstelle direkt mit dem PC verbunden ist.

Beide Komponenten – sowohl die Sensoren als auch das DAQ-System – müssen dabei an die besonderen Gegebenheiten der menschlichen Hand angepaßt werden. Das betrifft neben der geringen Größe aller Komponenten vor allem die Aufteilung des Gerätes in mehrere Module, die sich leicht kaskadieren und auf dem Handrücken verteilen lassen.

Der letztendlich verwirklichte Entwurf erfüllt diese Anforderungen. Das DAQ-System wurde darüber hinaus so flexibel gehalten, daß auch viele andere Einsatzmöglichkeiten denkbar sind. Entstanden ist ein universell verwendbares, portables Meßsystem, das bis zu 128 analoge Signale gleichzeitig mit einer Auflösung von 12 Bit verarbeitet. Es kann überall dort eingesetzt werden, wo sehr viele Signale simultan erfaßt werden müssen und wo es außerdem auf besonders kleine und leichte Komponenten ankommt.

Kapitel 1

Die Sensoren

Um die Position der Hand bzw. der einzelnen Finger zu errechnen, muß ihre Bewegung und ihre Orientierung im Raum ermittelt werden. Dies geschieht über Beschleunigungssensoren – auch als Inertialsensoren bezeichnet – und Gyroskope.

Die Inertialsensoren liefern den Wert der auf sie ausgeübten Beschleunigung für zwei Achsen, wobei natürlich auch die, durch die Gravitation verursachte, Erdbeschleunigung $a = g \approx 9,81 \frac{m}{s^2}$ mit berücksichtigt wird. Die Gyroskope dagegen geben die Größe der momentanen Winkelgeschwindigkeit ω_0 an, mit der der Sensor gerade rotiert.

Da zur Vermessung von einzelnen Fingergliedern besonders kleine und leichte Sensoren benötigt werden, bieten sich sogenannte MEMS[1] an. Sie integrieren die für die Sensorik nötige Mechanik und die Elektronik zur Signalverarbeitung direkt auf einem Chip und erlauben es so, mit einem Minimum an äußerer Beschaltung und daher mit sehr wenig Platz auszukommen. Bei anderen Sensoren, wie z.B. der ENC-Serie von MURATA [Mur ENC] müssen sämtliche Filter und Signalverstärker extern realisiert werden, was den Platzbedarf und nicht zuletzt auch den Schaltungsaufwand in die Höhe treibt.

Die Firma ANALOG DEVICES [AD] bietet unter der Bezeichnung **ADXL210E** bzw. **ADXRS300** sowohl Inertialsensoren, als auch Gyroskope in MEMS-Technik an. Beide sind in den folgenden zwei Abschnitten näher beschrieben.

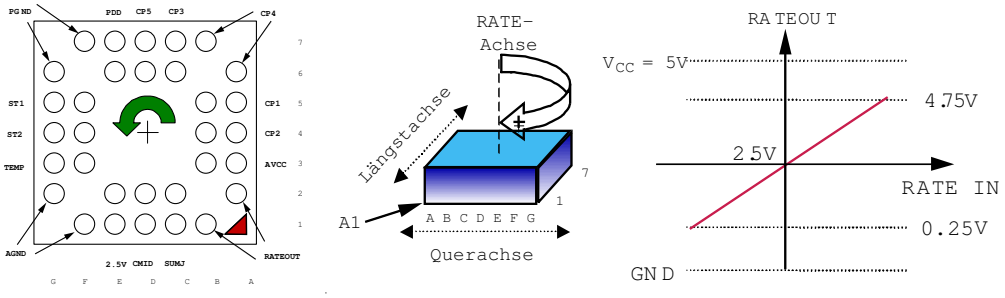
1.1 Gyroskop (ADXRS300)

Der Gyroskop-Sensor liegt im CSPBGA-Package[2] mit 32 Anschlüssen (0,6mm Pitch) vor und mißt aufgrund dieser extrem kleinen Bauform nur $5x5mm^2$, bei einer Gesamthöhe von 3,65mm. Er kann mit einer Versorgungsspannung zwischen 4,75V und 5,25V betrieben werden und nimmt im normalen Betrieb 6mA, maximal jedoch 8mA auf.

Der Sensor mißt die aktuelle Winkelgeschwindigkeit, mit der sich der Chip gerade dreht und gibt diese als analogen Spannungswert am Pin RATEOUT aus. Bei einer angenommenen Versorgungsspannung V_{CC} von 5V liegt dieser im Bereich zwischen 0,25V und 4,75V. In Ruhe ($\omega = 0$) sind es ungefähr 2,5V. Eine Rotation im Uhrzeigersinn um die z-Achse (senkrecht nach unten, bei Draufsicht auf den Chip) bewirkt eine positive Spannungsänderung, die proportional zur momentanen Winkelgeschwindigkeit ist. Die Abbildung 1.1b veranschaulicht den Zusammenhang.

[1] **M**icro-**E**lectro-**M**echanical **S**ystems

[2] CSPBGA: **C**hip **S**cale **P**ackage **B**all **G**rid **A**rray; verkleinertes BGA-Format bei dem die Außenfläche des Bauelementes maximal 120% der Chipfläche betragen darf



(a) Pinbelegung

(b) Ausgangssignal (RATEOUT)

Abbildung 1.1: Gyroskop ADXRS300 (entlehnt aus [AD Gyro, Seite 4 u. 7])

Da sich der Chip im Betrieb nicht unerheblich erwärmt und und der Meßwert auch von der Temperatur abhängig ist[3], wird außerdem noch kontinuierlich die Chiptemperatur gemessen und auf einem zweiten Ausgangs-Pin (TEMP) als analoges Signal zur Verfügung gestellt. Bei Zimmertemperatur (25 °C) beträgt dessen Wert 2,5V. Bei steigender Temperatur erhöht sich die Spannung um jeweils $8,4\frac{mV}{K}$.

Die genaue Pinbelegung des Bausteins ist in Abbildung 1.1a dargestellt und die wichtigsten Kennwerte sind noch einmal in Tabelle 1.1 zusammengefaßt.

Zu beachten ist noch, daß der Sensor niemals Querbeschleunigungen von über $2000g \approx 19600\frac{m}{s^2}$ ausgesetzt werden sollte, da sonst die empfindliche Mikro-Mechanik Schaden nimmt. Stürze oder das Aufschlagen auf harte Oberflächen sollte man also vermeiden.

Ausführlichere Angaben zum ADXRS300 sind in [AD Gyro] zu finden.

1.1.1 Temperaturverhalten

Die Ausgangsspannung des Gyroskops ist prinzipbedingt von der Temperatur abhängig. Um eine Vorstellung davon zu bekommen, wie stark diese Abhängigkeit ausgeprägt ist, wurde der in Ruhe ($\omega = 0$) befindliche Sensor mit einem Peltier-Element erwärmt und das RATE-Signal bei verschiedenen Temperaturen aufgezeichnet. Der im Gyroskop integrierte Temperatursensor ist hierbei besonders nützlich, da man ohne weitere externe Meßgeräte gleich die Temperatur des Dies mit aufnehmen kann.

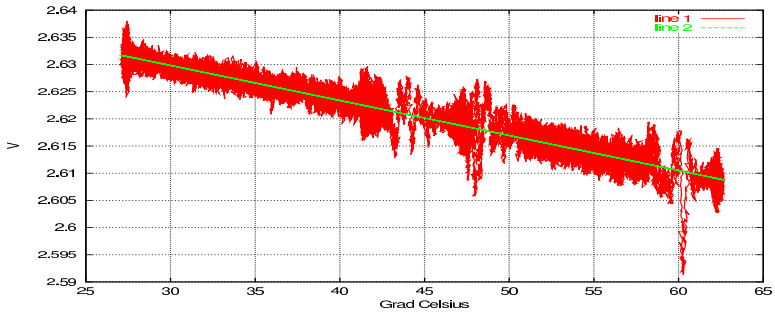


Abbildung 1.2: RATE-Signal (bei $\omega = 0$) über Temperatur

[3] näheres dazu im Abschnitt 1.1.1

Die Meßwerte wurden über eine DAQ-Karte mit 16 Bit AD-Umsetzern aufgenommen und es wurden kurze, gut abgeschirmte Signalleitungen verwendet. Daher können bei der Messung eingestreute Störungen in diesem Fall vernachlässigt werden. Das aufgenommene Rauschen stammt tatsächlich von den Sensoren und kann als Anhaltspunkt für deren Grundrauschen dienen.

Trägt man die gemessene RATE-Ausgangsspannung über der aus dem TEMP-Signal errechneten Temperatur auf, ergibt sich der in Abbildung 1.2 rot dargestellte Verlauf. Deutlich ist hier sowohl das Grundrauschen von etwa 5mV als auch die Temperaturdrift erkennbar. Die Spannung sinkt zu höheren Temperaturen hin immer mehr ab. Ein Fit dieses Verlaufs auf eine lineare Funktion (grün) ergibt eine Steigung von $-0,644\,\frac{mV}{K}$. Bei einer Erwärmung des Chips um 20K, die im Betrieb durchaus nicht ungewöhnlich ist, ändert sich das RATE-Signal also bereits um 12,9mV.

Der Einfluß der Temperatur auf die Ausgangsspannung des Gyroskops ist also nicht vernachlässigbar. Das TEMP-Signal muß mit aufgenommen und bei der späteren Umrechnung der RATE-Spannung in Winkelgeschwindigkeiten berücksichtigt werden.

1.2 Beschleunigungssensor (ADXL210E)

Wie bereits ausgeführt, mißt der Inertialsensor (Abbildung 1.3) die auf ihn wirkende Beschleunigung in zwei Achsen. Er besitzt daher **2 Ausgänge**, einen für die X- und einen für die Y-Achse. Die Meßwerte können in zweierlei Form entgegengenommen werden. Einerseits als PWM-Signale[4] (Pins X_{OUT} und Y_{OUT}) und andererseits als analoge Signale (Pins X_{FILT} und Y_{FILT}).

(a) Pinbelegung (b) von Oben (c) von Unten

Abbildung 1.3: Beschleunigungssensor ADXL210E

Da das Gyroskop ebenfalls analoge Signale liefert (näheres dazu in Abschnitt 1.1) ist es sinnvoll, auch für den Beschleunigungssensor die analogen Ausgänge zu benutzen. Dies hat zum einen den Vorteil, daß alle Meßsignale gemeinsam auf die gleiche Weise verarbeitet werden können, und es vermeidet gleichzeitig Störungen der Analogwerte durch die relativ steilen PWM-Flanken, welche sonst in unmittelbarer Nähe vorbeigeführt würden. Testmessungen haben außerdem gezeigt, daß bei Benutzung der PWM-Ausgabe hin und wieder unerklärliche „Ausreißer“ die Meßergebnisse verfälschen.

Ausführliche Angaben zum ADXL210E sind in [AD Acc] zu finden. Die wichtigsten Kenndaten *beider* Sensoren sind noch einmal in Tabelle 1.1 zusammengefaßt.

[4] PWM: **P**ulse **W**idth **M**odulation (**P**uls**w**eiten-**M**odulation)

Tabelle 1.1: Wichtige Kenndaten des ADXRS300 und des ADXL210E (bei $U_B = 5V$)

	Gyroskop (ADXRS300)		**Inertialsensor**
	Gyroskop (RATEOUT)	Temperatursensor (TEMP)	X_{FILT} / Y_{FILT}
Betriebsspannung U_B	4,75V ... 5,25V		3,0V...5,25V
typ. Stromaufnahme	6,0 mA		0,6 mA
max. Stromaufnahme	8,0 mA		1,0 mA
Meßbereich	$\pm 300\frac{^\circ}{s}$	$-40°C...+85°C$	$\pm 10g$
Ausgangsspannung	$u_{RATE} = 0,25V...4,75V$	$u_{TEMP}(T) = 2,16V...3,21V$	$u_{X/Y_{FILT}} \approx 1,5V...3,5V$
Signalnullpunkt	$u(\omega = 0) \approx 2,5V$	$u(T = 25°C) \approx 2,5V$	$u(a = 0g) \approx 2,5V$
Skalierungsfaktor	$5\frac{mV}{\frac{^\circ}{s}}$	$8,4\frac{mV}{K}$	$100\frac{mV}{g}$

1.2.1 Temperaturverhalten

Auch beim Inertialsensor muß die Abhängigkeit der Ausgangssignale von der Temperatur geprüft werden. Dazu wird der Sensor, wie in Abschnitt 1.1.1 beschrieben, mit einem Peltier-Element von 25°C auf 75°C erwärmt und währenddessen die Signalspannung aufgenommen. Die Verläufe für beide Achsen sind in Abbildung 1.4 dargestellt.

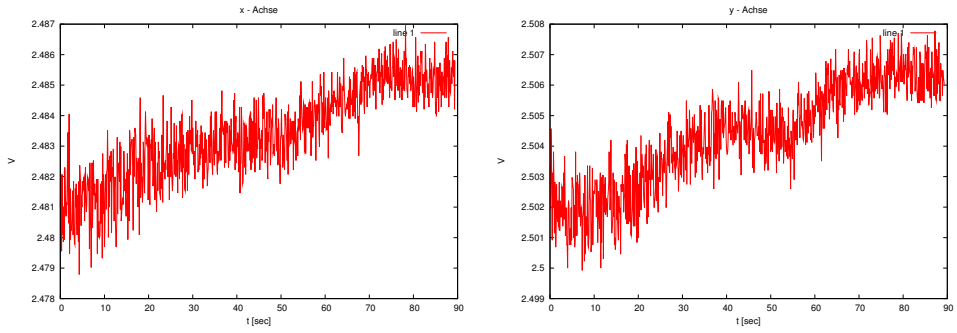


Abbildung 1.4: Ausgangssignal über Zeit (Sensor waagerecht)

Wie beim Gyroskop ist hier wieder ein gewisses Grundrauschen des Signals zu erkennen. Es fällt jedoch wesentlich geringer aus. Die Rauschamplitude beträgt etwa 3mV.

Der Sensor liegt bei dieser Messung waagerecht auf dem Tisch und wird nicht bewegt. Die Beschleunigung sollte also an beiden Achsen Null sein, was (vom Rauschen abgesehen) einer konstanten Ausgangsspannung von 2,5V entspräche. Es ist aber deutlich zu sehen, daß das Signal im Laufe der Zeit immer weiter ansteigt, der Sensor wird ja erwärmt. Trägt man die Spannung nicht über der Zeit, sondern über der Temperatur auf ist dieser Effekt gut zu erkennen.

Leider ist beim ADXL210E kein Temperatursensor integriert. Da auch nur ein relativ simples Thermometer zur Verfügung stand, konnte nur ungefähr alle 10 Sekunden per Hand eine Temperaturmessung vorgenommen werden. Dementsprechend grob sehen die Kurven in Abbildung 1.5 aus. Der Anstieg der Signalspannung ist aber trotzdem gut zu erkennen. Sie erhöht sich bei den 50K Temperaturdifferenz um durchschnittlich 4,7mA, was einer Steigung von $0,09\frac{mV}{K}$ entspricht.

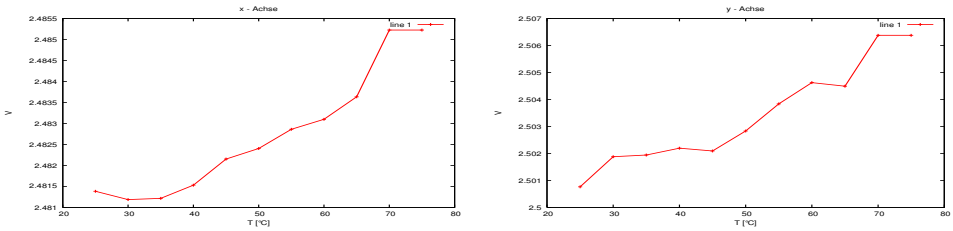


Abbildung 1.5: Ausgangssignal über Temperatur (Sensor waagerecht)

1.3 Integration auf einer Platine

Für erste Tests der beiden Sensoren standen vom Hersteller vertriebene Evaluation-Boards [AD AccEB, AD GyroEB] zur Verfügung. Sie sind gut geeignet, um relativ einfach und schnell Erfahrungen mit den Sensoren zu sammeln. Für die tatsächliche Montage auf der Hand sind diese Platinen allerdings bei weitem zu groß (und außerdem zu teuer). Aus diesem Grund ist es nötig, das Gyroskop und den Inertialsensor gemeinsam auf eine – möglichst kleine – Platine zu integrieren.

1.3.1 Schaltungsentwurf

Die Beschaltung der Sensoren (dargestellt in Abbildung 1.6) lehnt sich stark an die in den Datenblättern [AD Acc, AD Gyro] gemachten Vorschläge an. Die eigentliche Anpassung der Schaltung besteht darin, die Kapazitäten und Widerstände den Vorgaben entsprechend zu dimensionieren.

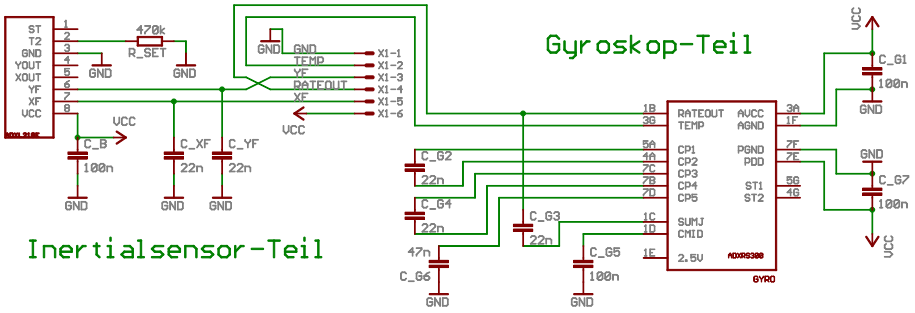


Abbildung 1.6: Schaltplan der Sensorplatine

Das bedeutet in erster Linie, die Sensoren auf die gewünschte Bandbreite zu begrenzen. Diese sollte dabei so hoch wie nötig, jedoch gleichzeitig so niedrig wie möglich gewählt werden, da die Rauschleistung P_n am Ausgang der Sensoren proportional mit der Bandbreite ΔB steigt. Das heißt, daß der Effektivwert der Rauschspannung $U_{n_{rms}}$ wegen

$$U_{n_{rms}} = \sqrt{\overline{u_n^2(t)}} = \sqrt{\frac{1}{T}\int_T u_n^2(t)dt} = \sqrt{P_n} = \sqrt{N_0 \cdot \Delta B} \tag{1.1}$$

mit der Quadratwurzel der Bandbreite zunimmt.[5] Für den Rausch-Effektivwert des Inertialsensors ist im Datenblatt der Ausdruck

$$a_{n_{rms}} = 200\frac{\mu g}{\sqrt{Hz}} \cdot \sqrt{\Delta B \cdot 1{,}6} \tag{1.2}$$

[5] Der Einfachheit halber wird hier von *weißem Rauschen* ausgegangen. Laut [AD Acc] entspricht das auch der Störsignalcharakteristik des Inertialsensors.

zu finden. Es ist dort also nicht der Effektivwert der Spannung, sondern der Beschleunigung $a_{n_{rms}}$ angegeben. Als Einheit wird $\mu g = 9,81 \cdot 10^{-6} \frac{m}{s^2}$ verwendet.

Die Einstellung der gewünschten Bandbreite wird bei beiden Sensoren über einen externen Kondensator vorgenommen. Dieser bildet zusammen mit einem internen Widerstand einen einfachen Tiefpaß erster Ordnung. Die 3dB-Grenzfrequenz ergibt sich also zu

$$f_{3dB} = \frac{1}{2\pi \cdot R_{int} \cdot C_{ext}} \tag{1.3}$$

Beim Inertialsensor sind die Kondensatoren C_{XF} und C_{YF} für die Bandbreite der X- bzw. Y-Achse verantwortlich. R_{int} hat einen Wert von $32k\Omega$. Mit Gleichung 1.2 und 1.3 ergeben sich also für verschiedene Grenzfrequenzen die in Tabelle 1.2 aufgeführten Kapazitäten und Rausch-Effektivwerte. Beim Gyroskop wird die Bandbreite mit C_{G3} variiert. R_{int} beträgt $180k\Omega$.

Tabelle 1.2: Bandbreiteneinstellung der Sensoren

Inertialsensor ($R_{int} = 32k\Omega$)			**Gyroskop** ($R_{int} = 180k\Omega$)	
f_{3dB} [Hz]	$a_{n_{eff}}$ [mg]	C_{XF} bzw. C_{YF} [nF]	f_{3dB} [Hz]	C_{G3} [nF]
10	0,8	497,4	10	88,4
50	1,8	99,5 => 100	20	44,2 => 47
100	2,5	49,7 => 47	30	29,5 => 33
200	*3,6*	*24,8 => 22*	*40*	*22,1 => 22*
300	4,4	16,6 => 15	50	17,6 => 15
500	5,7	9,9 => 10	100	8,8 => 10

Diese Bandbreiten lassen sich natürlich nicht ganz exakt einstellen. Nicht alle der berechneten Kapazitäten sind mit genau diesen Werten auch auf dem Markt verfügbar[6]. Ein Parallelschalten mehrerer Kondensatoren ist aus Platzgründen nicht möglich. Daher müssen die Werte teilweise gerundet werden, was zu etwas anderen Grenzfrequenzen führt. Hinzu kommt eine Toleranz derartiger Kondensatoren von mindestens $\pm 5\%$. Nicht zu vergessen ist auch, daß die Dämpfung bei einem einfachen RC-Glied nur 20dB pro Dekade beträgt, das Signal also nur relativ langsam zu höheren Frequenzen hin abklingt. Bei einer späteren AD-Umsetzung kann das zu Aliasing-Fehlern führen. Die Samplingfrequenz sollte dort also deutlich über der Nyquistgrenze liegen.

In vorher durchgeführten Tests mit den Evaluation-Boards haben sich **200Hz** als Grenzfrequenz für den Inertialsensor und **40Hz** für das Gyroskop bewährt. Die 40Hz lassen sich mit einem C_{G3} von 22nF noch recht genau treffen. Beim Inertialsensor führen die 2,8nF Abweichung von dem für $C_{XF/YF}$ berechneten Wert zu einer etwas verschobenen Grenzfrequenz von 226Hz.

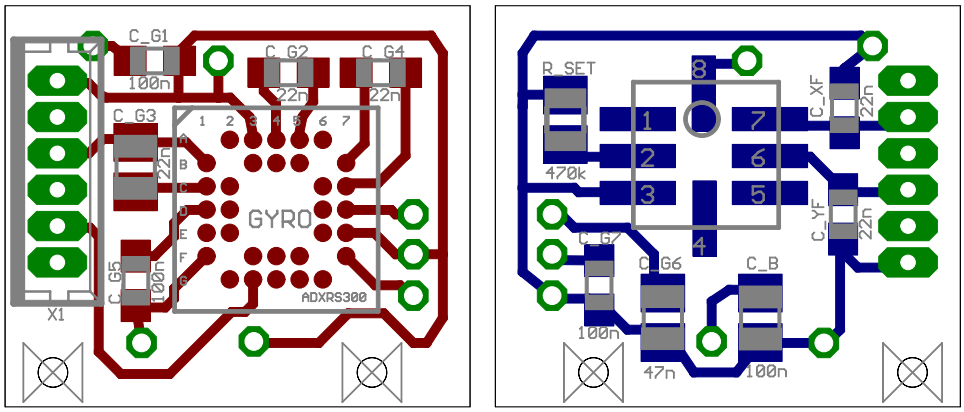
Neben der Grenzfrequenz für die analogen Signalausgänge, bietet der Inertialsensor außerdem die Möglichkeit, die Frequenz des PWM-Signals für die digitalen Ausgänge, die sogenannte *Duty Cycle Period,* über den Widerstand R_{set} zu justieren. Diese werden zwar hier nicht benutzt, trotzdem muß R_{set} vorhanden sein. Für ausschließlich analoge Operation soll er laut Datenblatt einen Wert um $500k\Omega$ erhalten.

Die restlichen Kondensatoren der Schaltung sind entweder Pufferkondensatoren zum Glätten der Eingangsspannung und Stabilisierung der Versorgung bei plötzlichen Leistungsspitzen ($C_B = C_{G1} = C_{G5} = C_{G7} = 100nF$) oder für die interne Funktion (charge pump) des Gyroskops notwendig ($C_{G2} = C_{G4} = 22nF$; $C_{G6} = 47nF$).

[6] Industriell ist *jeder* Wert zu bekommen. Bei Abnahme von weniger als 10.000 Stück hat man jedoch meist nur die Wahl zwischen 10 / 15 / 22 / 33 / 47 und 100 nF.

1.3.2 Platinendesign

Die eigentliche Herausforderung ist es nun, die beiden Sensoren – nebst äußerer Beschaltung – zusammen auf einer möglichst kleinen Platine unterzubringen. Das ist notwendig, da später auf jedem Fingerglied eine solche Platine sitzen soll (sieht Bild 1.13) und der Platz dort durch die Vorgaben der menschlichen Anatomie arg begrenzt ist.



(a) Bestückung Top-Layer

(b) Bestückung Bottom-Layer

Abbildung 1.7: Layout der Sensorplatine

Aufgrund dieser engen Vorgaben war es nötig, fast ausschließlich auf SMD-Bauteile der 0603er Bauform zurückzugreifen und die Platine doppelseitig zu bestücken, was den Entwurf des Platinen-Layouts (Abbildung 1.7) und die Fertigung etwas verkomplizierte. Im Endergebnis wurde es dadurch jedoch möglich, eine Kantenlänge von nur 16mm x 14mm zu erreichen. Die in Abbildung 1.8 gezeigten Darstellungen der fertig aufgebauten Sensorplatine sind also stark vergrößert.

(a) Oberseite (Gyroskop)

(b) Unterseite (Inertialsensor)

Abbildung 1.8: Fertig aufgebaute Sensorplatine

Die Sensorsignale werden durch einen kleinen, sechspoligen Stecker im 1,25mm-Raster von der Platine abgenommen. Durch diesen erfolgt auch die Versorgung der Schaltung mit Betriebsspannung. Die Signalbelegung der Steckkontakte und die Leiterbahnführung der Platine kann aus Abbildung A.1 auf Seite 64 im Anhang entnommen werden. Da aus Platzgründen kein Verpolungsschutz vorgesehen werden konnte, ist es besonders wichtig, den Stecker richtig herum aufzustecken. Das Vertauschen von VCC und GND führt zur sofortigen Zerstörung beider Sensoren!

1.3.3 Bestückung und Montage

Die Bestückung einer Platine ist normalerweise Routinearbeit und damit nicht weiter der Rede wert. Da aber in diesem Fall aufgrund der gewählten Bauelemente einige Schwierigkeiten zu überwinden waren und sich auch die sichere Montage der fertigen Platinen auf der Hand nicht ganz einfach gestaltete, sind jedoch ein paar Worte angebracht.

Bestückung

Kern des Problems ist der Gyroskopsensor, welcher leider nur in einem CSPBGA Package (siehe Abschnitt 1.1) verfügbar ist. Bei dieser Bauform sind die Anschlüsse unterhalb des Bauelementes als Matrix von kleinen Lötbällen (Bild 1.1a) herausgeführt. Dadurch ist der Sensor zwar extrem kompakt, allerdings per Hand nicht mehr zu bestücken.

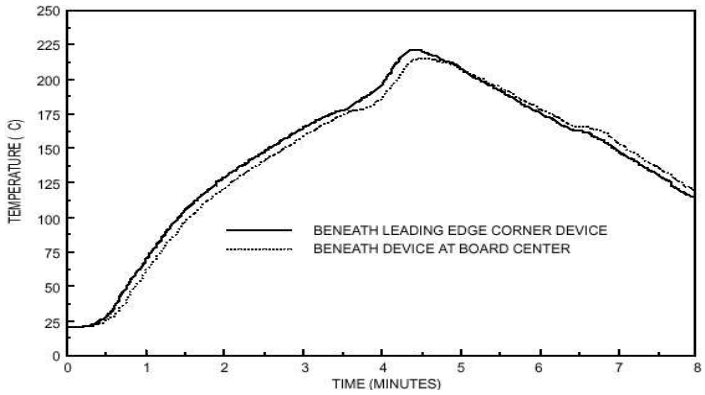


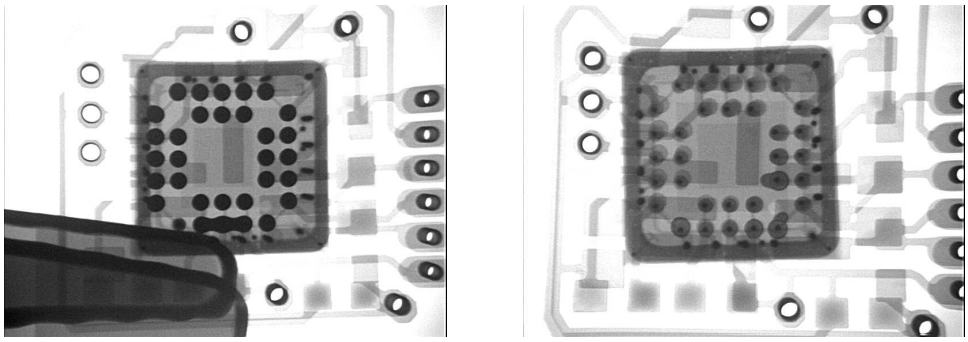
Abbildung 1.9: Standard BGA-Lötprofil Quelle: [Motorola]

BGA-Elemente müssen in einem sogenannten Reflow-Prozess aufgebracht werden. Das bedeutet, daß sie zunächst unter einem Mikroskop mit einem Microplacer exakt positioniert werden und anschließend durch eine Reflow-Straße laufen, wo sie mehrere Zonen unterschiedlicher Temperatur durchlaufen. Durch dieses genau spezifizierte Temperaturprofil (Abbildung 1.9) wird gewährleistet, daß sich das Lot der „Balls“ sicher mit den darunterliegenden Kontakten verbindet. Durch die dabei auftretenden Adhäsionskräfte werden kleine Ungenauigkeiten bei der Platzierung ausgeglichen und das Bauteil zentriert sich von selbst.

Es kursieren zwar im Internet verschiedene Anleitungen für manuelle Reflow-Bestückung (beispielsweise mit Hilfe eine Toasterofens, [Maxon]) jedoch ist dort eine korrekte Verbindung eher Glücksache[7]. Letztendlich kommt man um eine industrielle Bestückung (zumindest für die Gyroskope) nicht herum. Nachdem Anfragen bei mehreren Dienstleistern Preise zwischen 200 und 250 EUR für die Bestückung von zwei Sensorplatinen ergaben ist der Autor den Mitarbeitern des INSTITUT FÜR ZUVERLÄSSIGKEIT UND MIKROINTEGRATION (IZM) der Fraunhofer Gesellschaft besonders zu Dank verpflichtet, die sich netterweise bereit erklärten, das Aufbringen von vier Sensoren zu übernehmen. Da dort auch ein Röntgenmikroskop zur Verfügung stand, war es sogar möglich, den korrekten Sitz der aufgebrachten Sensoren zu überprüfen. In Abbildung 1.10a ist deutlich zu sehen, wie sich das Lot mehrerer Balls miteinander verbunden hat und somit Kurzschlüsse bildet. Abbildung 1.10b zeigt einen gelungenen Versuch.

Die Bestückung der restlichen Bauteile war danach problemlos in Handarbeit machbar.

[7] Bei einem Preis von ca. 50 EUR pro Sensor zzgl. der Platinenkosten wurde von solchen Experimenten abgesehen.



(a) mit Kurzschlüssen (b) korrekt bestückt

Abbildung 1.10: Röntgenbilder (48keV) des ADXRS300

Montage

Die fertigen Sensorplatinen müssen nun auf die einzelnen Fingerglieder montiert werden. Dies muß einerseits so geschehen, daß sich die Position der Sensoren zum Finger möglichst nicht verändert, andererseits soll das gesamte System aber schnell und unkompliziert anzulegen und wieder abzunehmen sein, auch bei Patienten mit sehr unterschiedlich geformten Händen. Die Befestigung muß also flexibel an verschiedene Größenverhältnisse anzupassen, und außerdem für den Patienten angenehm zu tragen sein. Die für einen möglichst exakten Sitz eigentlich nötigen Schrauben und Bolzen verbieten sich von selbst, so daß ein - wenn auch minimales - Verrutschen der Sensoren nicht völlig ausgeschlossen werden kann.

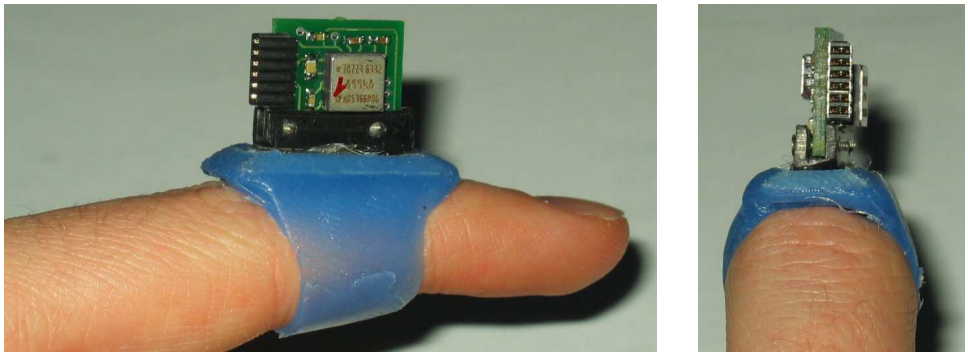


Abbildung 1.11: Fingerring mit aufmontierter Sensorplatine

Einen guten Kompromiss zwischen Tragekomfort und Positionsstabilität stellen Ringe dar, die auf die einzelnen Fingerglieder gesteckt werden. Diese sind aus einem gummiartigen Kunststoff gefertigt, der auch für Prothesen verwendet wird. An der Oberseite ist eine Aluminiumplatte eingelassen, auf der die Platine hochkant montiert wird (Abbildung 1.11).

Da die Ringe immer auf die Mitte der Fingerglieder gesteckt werden und die einzelnen Platinen nur über dünne Kabelstränge mit dem Rest des Systems verbunden sind, lassen sich unterschiedlich lange Glieder sehr gut kompensieren.

Eine Möglichkeit für zukünftige Erweiterungen wäre es, Ring und Platine über eine kleine Steckvorrichtung miteinander zu verbinden. Auf diese Weise könnte man je nach Patient

Ringe mit verschiedenen Durchmessern unter die Sensoren stecken und somit auch noch die unterschiedlich Dicke der Fingerglieder ausgleichen.

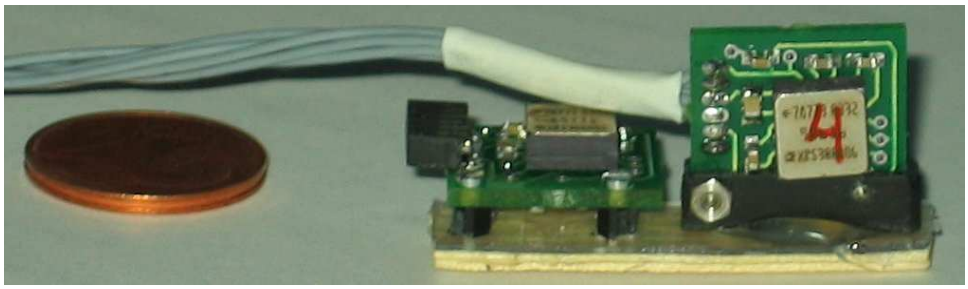


Abbildung 1.12: Zwei Sensorplatinen um 90° versetzt montiert

Auf den hinteren Fingergliedern (die dem Handteller am nächsten liegen) müssen Beschleunigung und Winkelgeschwindigkeit in zwei orthogonalen Ebenen gemessen werden (siehe Abschnitt 2.1). Es sind also zwei um 90° versetzte Platinen nötig (Abbildung 1.12).

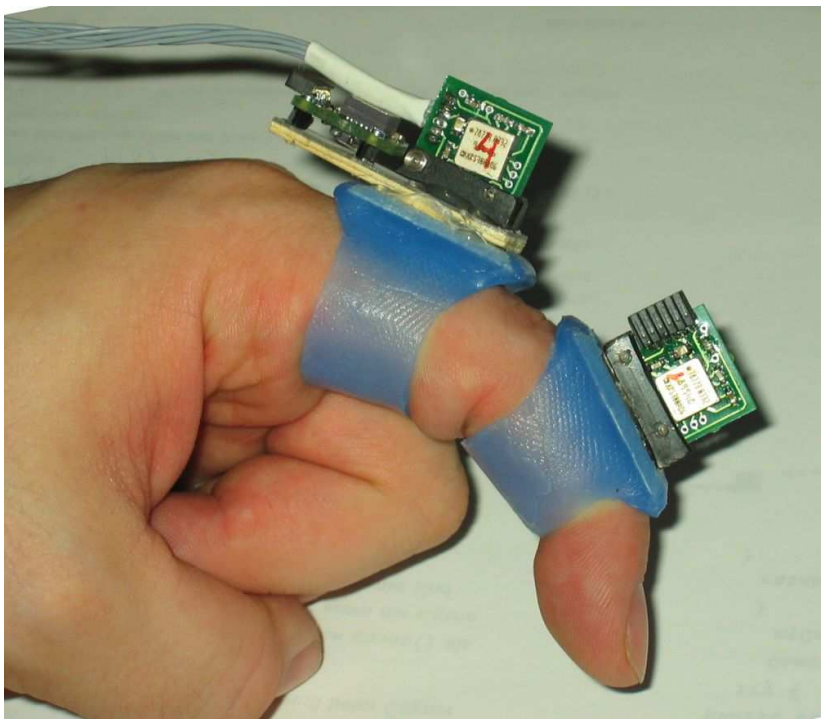


Abbildung 1.13: Sensorplatinen auf dem Finger

Abbildung 1.13 läßt bereits erahnen, wie das fertig aufgebaute Sensorsystem später aussehen wird.

Kapitel 2

Konzeption des Gesamtsystems

Im folgenden Kapitel wird der grundlegende Aufbau des gesamten Systems, bestehend aus Sensoren, DAQ-System[1] und Rechnerschnittstelle, Schritt für Schritt entwickelt. Dazu werden als Erstes die Mindestanforderungen spezifiziert und anschließend verschiedene Realisierungsmöglichkeiten auf ihre Vor- und Nachteile hin untersucht. In Abschnitt 2.3 wird schließlich die gewählte Lösung im Detail vorgestellt.

2.1 Anforderungen

2.1.1 Anordnung der Sensorplatinen

Die Sensorplatinen verteilen sich, wie in Abbildung 2.1 gezeigt, sowohl auf den einzelnen Fingern, als auch auf dem Handrücken.

Dabei ist **pro Fingerglied** eine Platine mit jeweils einem Gyroskop (rot) und einem Inertialsensor (blau) vorgesehen. Sie muß senkrecht (xy-Ebene) auf dem Finger stehen, so daß die x- und y-Komponenten des Inertialsensors die $\vec{e}_x$ und $\vec{e}_y$ Anteile der auf das Glied wirkenden Beschleunigung $\vec{a}$ messen. Die Winkelgeschwindigkeit $\vec{\omega}$, mit der es um die z-Achse rotiert, wird vom Gyroskop erfaßt. Eine mathematisch positive[2] Rotation um die z-Achse bedeutet ein negatives $\vec{\omega}$ und damit eine Ausgangsspannung $u_{rate} < 2,5V$, da das $\vec{\omega}$ des Gyroskops in $-\vec{e}_z$-Richtung definiert ist[3].

Auf dem **hinteren (größten) Fingerglied** müssen zwei Platinen orthogonal zueinander untergebracht werden. Die zusätzliche waagerechte Platine (yz-Ebene) ist notwendig, um das Abspreizen des Fingers erfassen zu können. Die Beschleunigung muß hier also sowohl in $\vec{e}_x$- und $\vec{e}_y$-Richtung (horizontaler Sensor) als auch in $\vec{e}_z$-Richtung (-x-Komponente des waagerechten Sensors) erfasst werden. Da der Einfachheit halber auf der waagerechten Platine der gleiche zweiachsiger Inertialsensor sitzt, wird die $\vec{e}_y$-Komponente der Beschleunigung doppelt gemessen. Diese Redundanz kann eventuell bei der späteren Auswertung der Daten für eine bessere Fehlerkorrektur genutzt werden.

Der gleiche Doppelsensor ist für den **Handrücken** nötig, der die selben Freiheitsgrade wie das hintere Fingerglied besitzt.

Der **Unterarm** kann sich (prinzipiell) völlig frei im Raum bewegen. Insbesondere ist eine Drehung nicht nur um die x- und z-, sondern auch um die y-Achse möglich. Um diese

[1] DAQ: **D**ata **Aq**uisition System; System zum Aufnehmen und Speichern von Daten
[2] entgegen dem Uhrzeigersinn
[3] siehe Abbildung 1.1 auf Seite 2

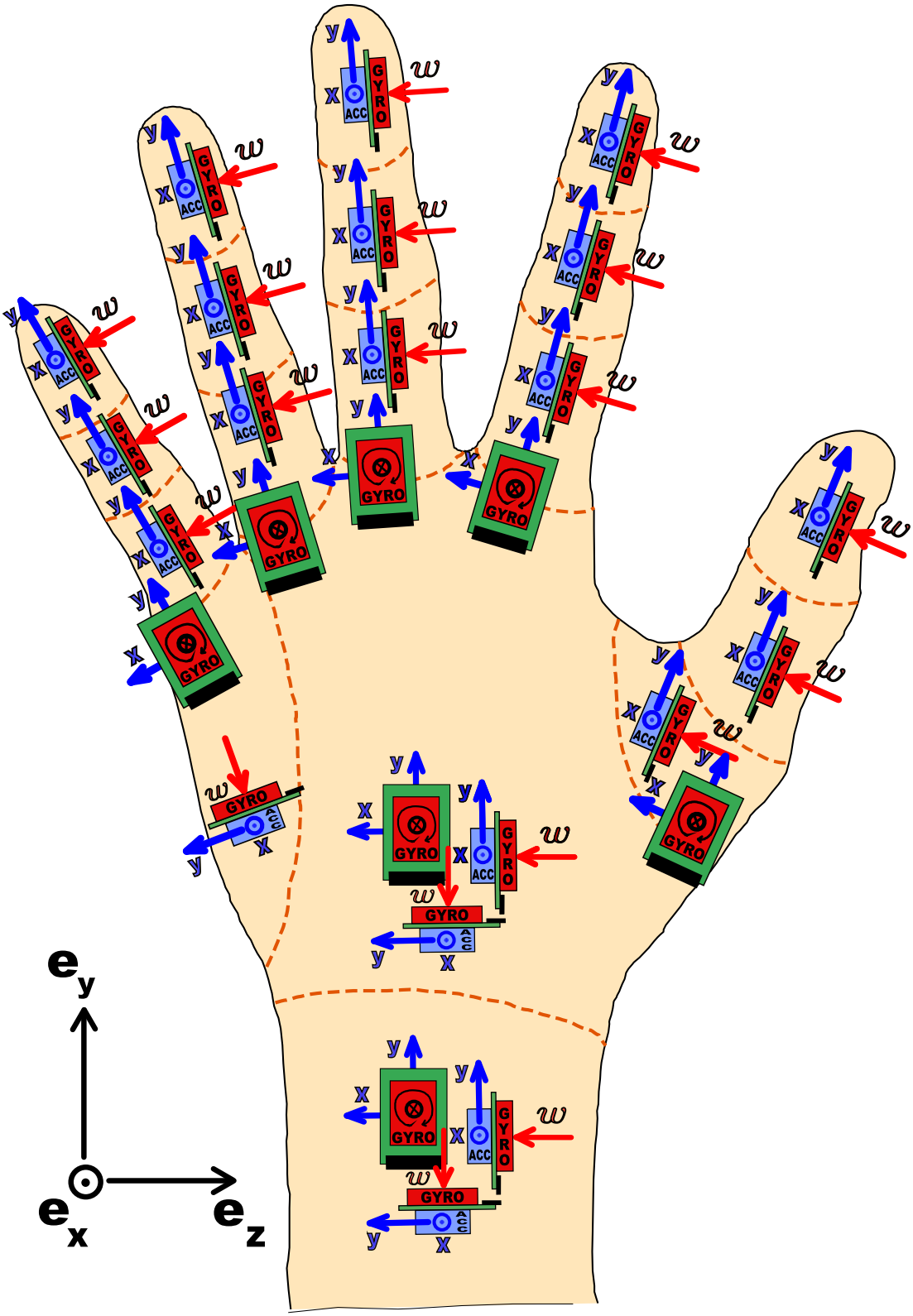


Abbildung 2.1: Anordnung der Sensorplatinen

durch das Gyroskop erfassen zu können, muß eine weitere senkrechte Sensorplatine in der xz-Ebene hinzugefügt werden, auch wenn dadurch die $\vec{e}_z$- und $\vec{e}_x$-Komponenten der Beschleunigung wiederum redundant erfaßt werden.

Zusätzlich werden wahrscheinlich auch noch zwei Platinen auf dem **linken Teil des Handrückens** nötig, um das seitliche „Abknicken“ dieses Teils zu erfassen.

2.1.2 Anzahl und Aufteilung der Kanäle

Aus der beschriebene Anordnung der Platinen läßt sich direkt die benötigte Anzahl der Kanäle herleiten. Die entstandene Aufteilung ist in Tabelle 2.1 dargestellt.

Tabelle 2.1: Benötigte Kanäle

Ort	**Platinen**	**Kanäle**		
		(komplett)	(nur 1 Temp / Zone)	(ohne redundante a-Werte)
vorderes Fingerglied	1	4	4	4
mittleres Fingerglied	1	4	4	4
hinteres Fingerglied	2	8	7	6
Handrücken	2	8	7	6
Handrücken linker Teil	2	8	7	6
Unterarm	3	12	10	7
GESAMT (1 Finger)	11	**44**	39	33
GESAMT (5 Finger)	27	**108**	99	89

Für den kompletten Ausbau des Systems werden also 108 Kanäle benötigt. Geht man von der Annahme aus, daß die Gyroskope auf den dicht beieinander liegende Platinen einer Zone (also die Cluster auf den hinteren Fingergliedern, dem Handrücken (links/Mitte) und dem Unterarm) eine ähnliche Temperatur haben, würde es genügen, nur einen Temperaturwert pro Zone auszuwerten. Es würden nur noch 99 Kanäle benötigt. Ignoriert man zusätzlich noch die redundant gemessenen Beschleunigungskomponenten der Inertialsensoren, ließe sich die Zahl der Kanäle sogar auf 89 drücken.

Diese Redundanzen sind jedoch nützlich, um später mögliche Meßfehler auszugleichen. Auch die Temperaturverteilung an den Sensoren ist noch keineswegs klar, so daß zunächst sicherheitshalber alle Temperaturmessungen berücksichtigt werden sollten. Bei der weiteren Dimensionierung des Systems wird also – nicht zuletzt um für später eventuell hinzukommende Sensoren noch Reserven zu haben – von mindestens **108 benötigten Kanälen** ausgegangen.

Da vorerst nur ein einziger Finger erfaßt werden soll, ist es notwendig, das System modular zu halten. Das macht es auch leichter, die einzelnen Komponenten auf der Hand zu verteilen. Es bietet sich an, die auf einem Finger liegenden Sensoren zusammenzufassen, was $4+4+8=16$ Kanäle pro Finger ergibt. Auf dem Handrücken sind vier Platinen mit jeweils vier Ausgängen nötig, was ebenfalls 16 Kanäle ergibt. Auch hier bietet es sich an, diese zu einem Modul zusammenzufassen. Die Sensoren auf dem Unterarm können ebenfalls zusammengefaßt werden, was $4 \cdot 3 = 12$ Kanäle ergibt.

Für den Aufbau eines Einfinger-Systems mit ansonsten kompletter Sensorik empfiehlt sich also folgende Aufteilung[4]:

[4] Aufgrund der sehr aufwändigen Fertigung der Sensorplatinen (siehe Abschnitt 1.3.3) konnte der im Rahmen dieser Arbeit tatsächlich erfolgte Aufbau leider nur mit 4 Platinen (für den Finger) und zwei orthogonalen Inertialsensoren (für den Handrücken) realisiert werden.

Finger: 4 Platinen => 16 Kanäle (mindestens 14)

Handrücken: 4 Platinen => 16 Kanäle (mindestens 12)

Unterarm: 3 Platinen => 12 Kanäle (mindestens 7)

2.1.3 Versorgung der Sensoren

Wie in Tabelle 1.1 auf Seite 4 aufgeführt, benötigt das Gyroskop eine Versorgungsspannung zwischen 4,75V und 5,25V, während der Beschleunigungssensor mit einer Spannung von 3V bis 5,25V arbeitet. Um eine gemeinsame Spannungsversorgung der Sensoren zu ermöglichen, sollten beide mit **5V** betrieben werden. Das hat zudem den Vorteil, daß der Signalnullpunkt für beide Sensoren bei 2,5V liegt, was die Auswertung erleichtert.

Bei dieser Spannung benötigt das Gyroskop maximal 8mA und der Inertialsensor maximal 1mA. Pro Sensorplatine werden also 9mA benötigt. Ein mit vier Platinen besetzter Finger verbraucht damit 36mA und bei einem voll ausgebauten System mit 27 Platinen müssen für die Sensoren 243mA, sicherheitshalber also **250mA** an Strom zur Verfügung gestellt werden.

2.1.4 Auflösung und Abtastfrequenz der A/D-Umsetzer

Die Bandbreite des Gyroskops ist bei den benutzten Evaluation-Boards auf 40Hz begrenzt, was sich in den bisher durchgeführten Experimenten als durchaus ausreichend herausgestellt hat. Das entspricht einer theoretischen Mindest-Abtastrate von **80 Samples/s**.

Die meßtechnische Bestimmung der Temperaturkurve des Gyroskops in Abschnitt 1.1.1 hat ergeben, das der Sensor in Ruhe ($\omega = 0$) bei einer Temperatur von 30 °C bereits mit 6mV rauscht. Die analoge Ausgangsspannung muß also nur auf 6mV genau quantisiert werden. Das ergibt nach

$$b = ld\left(\frac{U_{max} - U_{min}}{U_{LSB}}\right) = ld\left(\frac{5V - 0V}{6mV}\right) = 9,7Bit \tag{2.1}$$

eine benötigte Auflösung des AD-Umsetzers von mindestens **10 Bit**.

Mit der in Abschnitt 1.3.1 dargestellten Beschaltung des Beschleunigungssensors sind die beiden Ausgänge X_{FILT} und Y_{FILT} auf etwa 200Hz bandbegrenzt, so daß eine Samplingrate von 400 Samples/s pro Kanal theoretisch genügt. Aufgrund der dort besprochenen Ungenauigkeiten sind jedoch mindestens **500 Samples/s pro Kanal** zu empfehlen.

Die Ausgangsspannung bewegt sich bei sinnvollen Meßwerten zwischen 1V und 4V. Sicherheitshalber ist es aber ratsam, den ADC[5] auf den vollen Ausgangsbereich von **0V - 5V** auszulegen.

Die in Abschnitt 1.2.1 gemessene Temperaturkurve des Inertialsensors zeigt ein Rauschen von etwa 3mV. Nach Gleichung 2.1 ergibt das eine Mindestauflösung von 10,7 d.h. **11 Bit**.

Um eine einheitliche Behandlung aller Meßdaten zu ermöglichen, sollte man die AD-Umsetzer so auslegen, daß sie Meßdaten von Gyroskop und Beschleunigungssensor gleichermaßen verarbeiten können. Das führt zu folgenden Anforderungen an die ADCs:

Samplingrate: mindestens R_s =500 Samples/s

Auflösung: mindestens 11 Bit

Eingangsspannungsbereich: 0V - 5V

[5] Obwohl die Bauelemente im Deutschen **A**nalog-**D**igital **U**msetzer heißen, wird im Folgenden immer die in der Fachliteratur gebräuchlichere, englische Abkürzung ADC (**A**nalog **D**igital **C**onverter) verwendet.

2.2 Verschiedene Realisierungsmöglichkeiten

Es ergeben sich folgende Möglichkeiten, die ankommenden analogen Meßwerte zu digitalisieren und in den Rechner zu übertragen:

1. Übertragung der analogen Signale zum PC; Digitalisierung durch Meßkarten im Rechner
2. Digitalisierung der einzelnen Signale an Ort und Stelle; Übertragung der digitalen Datenströme zum Rechner
3. Digitalisierung der einzelnen Signale; Multiplexen der digitalen Datenströme und Übertragung dieses einen Stroms zum Rechner
4. Multiplexen der analogen Signale zu Gruppen; Digitalisierung durch einige, wenige ADCs; Übertragung dieser wenigen Datenströme zum Rechner

Die vollständig analoge Übertragung der Signale bis hin zum PC scheidet aus mehreren Gründen aus. Bei den sich ergebenden Leitungslängen von mehreren Metern wären umständliche Abschirmungen gegen äußere Einflüsse (HF-Einkopplung usw.) und gegen gegenseitige Störungen (Nebensprechen usw.) nötig. Beim voll ausgebauten System mit 108 Kanälen wären die entstehenden Kabelbündel kaum noch zu handhaben. Außerdem wären jede Menge teure Meßkarten vonnöten, um die Signale im Rechner zu digitalisieren. Bei den bisher verwendeten Karten von NATIONAL INSTRUMENTS mit 16 analogen Eingängen pro Karte wären sieben Karten im Gesamtwert von über 15.000 EUR vonnöten, die ja außerdem auch noch alle in einen einzigen Rechner eingebaut werden müßten.

Die getrennte Übertragung aller digitalisierter Signale bis zum Rechner scheidet aus ähnlichen Gründen aus. Zwar sind digitale Signale nicht so anfällig gegenüber Störungen, dafür werden durch die steilen Flanken viele hochfrequente Spektralanteile erzeugt, die in die Umgebung abstrahlen. Aus EMV-Gründen[6] müßten diese Kabel also auch abgeschirmt werden, was wiederum zu schwer handhabbaren Kabelbäumen führt. Außerdem werden weiterhin 108 Rechnereingänge gebraucht, auch wenn es sich jetzt um digitale Eingänge handelt und die dafür benötigten Karten preiswerter zu haben sind.

Eine weitere Möglichkeit wäre, jedes analogen Signale zunächst mit einem AD-Umsetzer zu digitalisieren, diese digitalen Datenströme dann zu multiplexen und anschließend über ein Kabel zum Rechner zu übertragen. Der Hauptnachteil dieser Lösung ist, daß für jeden Kanal ein eigener ADC benötigt wird. Da die Digitalisierung aus oben ausgeführten Gründen möglichst nahe an den Sensoren stattfinden sollte, müßten sämtliche Umsetzer (immerhin 108 beim voll ausgebauten System) nebst externer Beschaltung auf dem Handrücken bzw. den Sensorplatinen sitzen, was aus Platzgründen unmöglich ist. Außerdem sind die meisten ADCs mit den benötigten Samplingraten von 500 S/s extrem unterfordert.

Dieser enorme Hardwareaufwand läßt sich durch die in Punkt 4 aufgeführte Lösung weitestgehend vermeiden. Zusammenhängende Gruppen von Signalen (zum Beispiel alle Signale eines Fingers) werden durch Analog-Multiplexer zusammengefaßt und anschließend durch einen einzigen ADC digitalisiert. Die digitalen Datenströme aller Umsetzer werden dann zum Rechner übertragen. Auf diese Weise wird für jeden Finger nur ein einziger (statt 16) ADC benötigt. Bein einem Vollausbau des Systems müssen also nur $5 + 2 = 7$ ADCs auf der Hand untergebracht werden.

[6] EMV: **E**lektro**m**agnetische **V**erträglichkeit

2.3 Gewählte Lösung: A/D-Umsetzer mit mehreren Eingängen

Von den genannten Möglichkeiten kommt nur die Letzte wirklich in Frage. Der Hardwareaufwand hält sich in Grenzen, da jeder ADC eine ganze Gruppe von Signalen bearbeitet und die Verkabelung zum Rechner beschränkt sich auf eine einzige Datenleitung pro ADC.

Noch weiter vereinfachen läßt sich diese Lösung, wenn man anstatt einen Multiplexer mit nachgeschalteter Sample-And-Hold Stufe vor jedem A/D-Umsetzer zu platzieren, gleich einen ADC mit mehreren Eingängen benutzt, der sämtliche für das Multiplexen nötige Beschaltung bereits enthält. Das ist besonders vorteilhaft, da es mit jedem eingesparten Bauteil leichter wird, die ADCs – die ja so dicht wie möglich bei den Sensoren sitzen müssen – auf dem Handrücken unterzubringen.

Das letztendlich gewählte Konzept für das Gesamtsystem ist in Abbildung 2.2 dargestellt und soll im Folgenden kurz erläutert werden.

Die Sensorplatinen (grün) sind wie im Abschnitt 2.1.1 beschrieben auf der Hand verteilt. Die zu einer Gruppe (Finger, Handrücken, Unterarm) gehörigen Platinen sind alle gemeinsam an eine direkt auf dem Handrücken befindliche AD-Umsetzer-Platine (blau) angeschlossen. Dort werden die bis zu 16 analogen Signale der Sensorgruppe (blaugrün) digitalisiert und von dort werden die Sensoren auch mit Betriebsspannung versorgt. Der detaillierte Aufbau dieser ADC-Platine ist in Kapitel 3 beschrieben.

Durch die nur wenige Zentimeter von den Sensoren stattfindende Digitalisierung wird der Einfluß von Störungen wie z.B. Rauschen oder Nebensprechen auf ein Minimum reduziert. Als Signal- bzw. Versorgungsleitungen genügen daher dünne, ungeschirmte Drähte, was den Platzbedarf der Verkabelung deutlich reduziert.

Alle ADC-Platinen sind über einen seriellen Bus (rot) miteinander verbunden, über den die anfallenden Daten abtransportiert werden. Der Bus-Master ist ein Microcontroller (gelb). Er nimmt die Rohdaten entgegen, konvertiert sie in ein geeignetes Format und sendet sie dann über einen USB-Controller zum PC.

Im Gegensatz zur bisher skizzierten Lösung, bei der jeder ADC seine Daten über eine separate Verbindung zum Rechner sendet, hat dieser Ansatz mehrere Vorteile. Zunächst wird der Verkabelungsaufwand reduziert. Statt den sieben Datenleitungen (bei Vollausbau) zuzüglich Versorgungsspannung und Masse ist nur noch ein einziges USB-Kabel (grün) vonnöten. Außerdem kann das System durch die Verwendung einer derart weit verbreiteten Schnittstelle wie USB buchstäblich an jeden Rechner angeschlossen werden.

Die AD-Umsetzer müssen aus oben genannten Gründen direkt auf der Hand, in unmittelbarer Nähe der Sensoren untergebracht werden. Für den Microcontroller und den USB-Controller gilt das nicht. Sie sind gemeinsam auf einer externen Platine untergebracht. Diese unterliegt dadurch nicht den strikten Miniaturisierungsanforderungen, was Entwurf, Aufbau und Test erheblich vereinfacht. Details zu dieser Controller-Platine sind in Kapitel 4 zu finden.

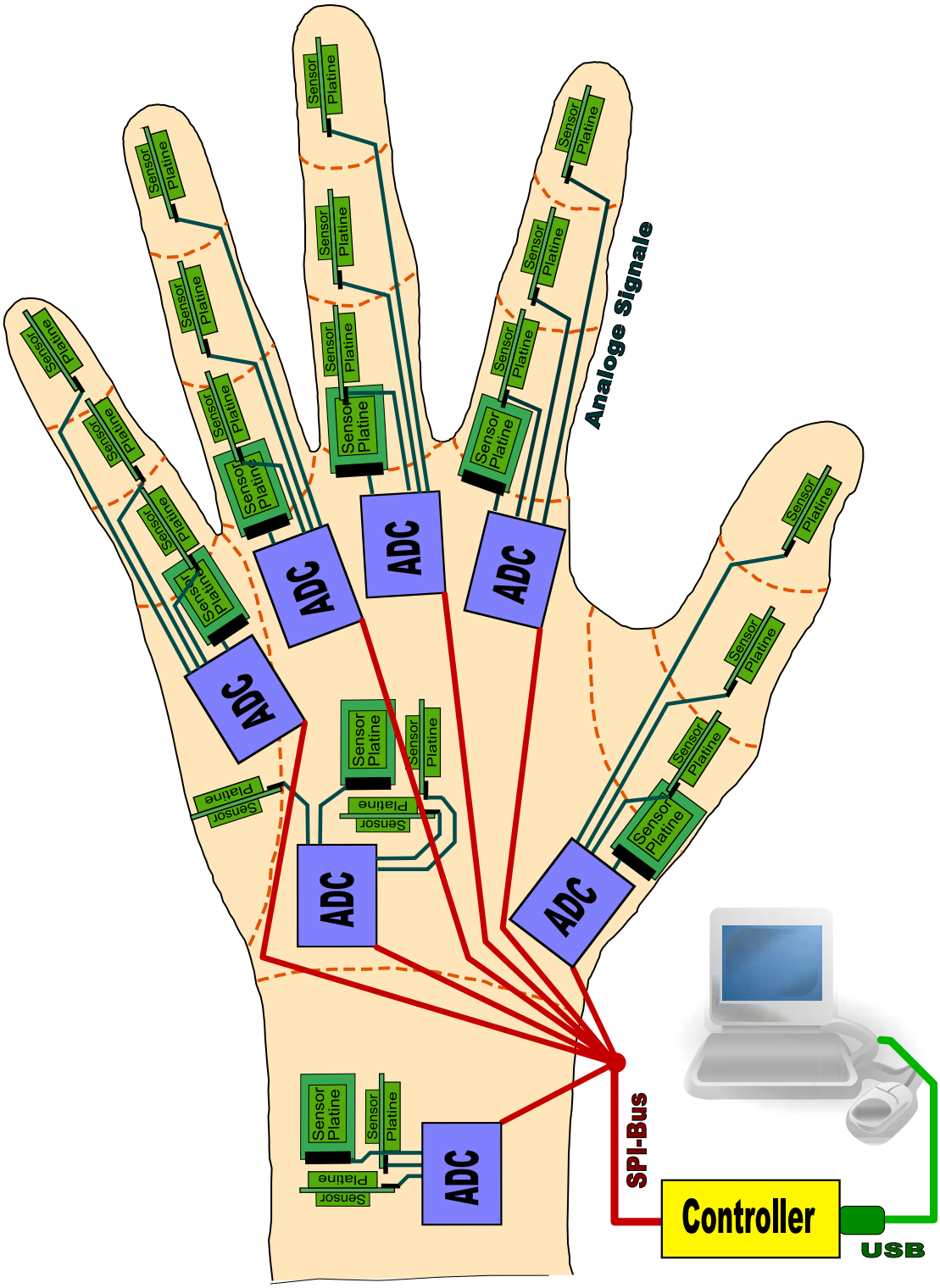


Abbildung 2.2: Blockschaltbild des Gesamtsystems

Kapitel 3

Die A/D-Umsetzer Platine

3.1 Aufgaben

Aus dem in Abschnitt 2.3 dargestellten Gesamtkonzept ergeben sich für die A/D-Umsetzer Platine mehrere Aufgaben.

Die Hauptaufgabe ist selbstverständlich, die analogen Sensorsignale mit der benötigten Genauigkeit (Abschnitt 2.1.4) zu digitalisieren und die Ergebnisse über ein serielles Bussystem an die Controller-Platine (Kapitel 4) zu senden. Jede Platine muß also 16 analoge Signaleingänge und einen Bus-Anschluß aufweisen.

Darüber hinaus muß die ADC-Platine aber auch die Sensoren mit Betriebsspannung versorgen. Natürlich könnte man diese auch einfach direkt von der Controller-Platine aus speisen. Da sich Störungen in der Versorgungsspannung aber direkt auf das Sensorsignal auswirken ist es besonders wichtig, eine möglichst konstante und rauschfreie Spannung zur Verfügung zu stellen. Versorgungsleitungen von der Controller-Platine zu jedem Sensor bedeuten nicht nur einen höheren Verkabelungsaufwand, über diese relativ langen Leitungen können außerdem vermehrt Störungen einkoppeln.

3.2 Auswahl des A/D-Umsetzers

Eine Recherche bei verschiedenen Herstellern [AD, INF, MAX, NSC, TI] förderte zwei Chips zu Tage, die den beschriebenen Anforderungen genügen. Das ist einmal der **AD7490** von ANALOG DEVICES und außerdem der **MAX1230** von MAXIM SEMICONDUCTORS.

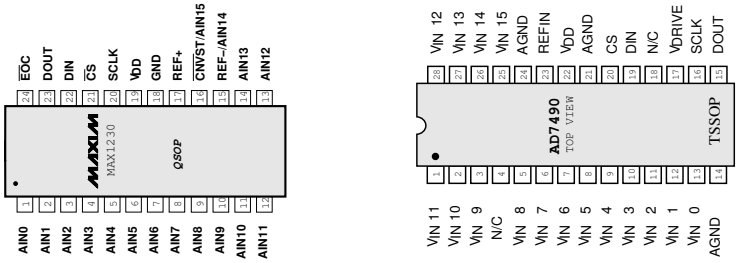


Abbildung 3.1: Pinbelegung beider ADCs (aus [MAX adc] und [AD adc])

Beide besitzen 16 Eingänge[1]. Das genügt, um alle Signale eines Fingers bzw. alle Signale von Handrücken und Unterarm zu digitalisieren. Für den zunächst anvisierten Ausbau mit einem Finger sind also nur zwei solcher Chips nötig. Sie bieten 12 Bit Auflösung und können mit einer einzelnen Betriebsspannung von $+5V$ betrieben werden. Die gewandelten Daten stellen beide am Ausgang über einen seriellen Bus (SPI/QSPI bzw. MICROWIRE) zur Verfügung.

Die 12 Bit Auflösung sind nötig, da das bei ADCs unvermeidliche Rauschen die letzten Bits zu einem gewissen Grade unzuverlässig macht und so einen Teil der möglichen Auflösung wieder verschlingt. Beim **MAX1230** berechnet sich die **E**ffective **N**umber **O**f **B**its (ENOB) laut [MAX adc, S.20] nach

$$ENOB = \frac{SINAD - 1.76}{6.02}$$

wobei *SINAD* das Signal/Rausch-Verhältnis inklusive Verzerrungen[2] darstellt und im Datenblatt bis zu einer Eingangssignalfrequenz von 10kHz mit nahezu konstant 70dB angegeben ist. Damit ergibt sich ein ENOB von $\frac{70dB-1,76}{6,02} = 11,33Bit$, was ausreichend ist.

Tabelle 3.1: Vergleich AD-Umsetzer

	AD7490		**MAX1230**	
Eingänge	16		16	
Auflösung	12		12	
Spannungsversorgung (V_{DD})	2,7V - 5,25V[3]		4,75V - 5,25V	
Ausgang	seriell		seriell	
max. Abtastrate	1MS/s		300kS/s	
Referenzspannung (V_{ref})	N/A	2,5V	4,096V	$0...V_{DD}+50mV$
Stromverbrauch	2,5mA		1,9mA	
Eingangsbandbreite (3dB)	8,2MHz		1MHz	
Eingangsspannung (V_{in})	$0...V_{ref}$ oder $0...2V_{ref}$		$0...V_{ref}$	

Wie aus der Tabelle 3.1 zu erkennen ist, erlaubt der **AD7490** eine schnellere Wandlung mit 1MS/s, wohingegen der **MAX1230** nur mit (immer noch sehr gut ausreichenden) 300kS/s sampeln kann, dafür aber auch mit weniger Strom auskommt. Beide Bausteine sind aber mehr als ausreichend schnell. Selbst bei einem Gesamtdurchsatz von 300kS/s bleiben bei voller Auslastung für jeden Kanal noch 18,75kS/s, was *deutlich* über den geforderten 0,5kS/s liegt. Die 3dB-Bandbreite liegt mit 1MHz bzw. 8,2MHz ebenfalls weit jenseits der benötigten 500Hz.

Leider ist für beide Bausteine eine externe Referenzspannungsquelle nötig. Der **MAX1230** stellt zwar intern eine Referenzspannung von 4,096V zur Verfügung, diese ist jedoch zu niedrig. Um den vorgegebenen Meßbereich von 0V bis 5V zu ermöglichen, wird eine Referenz von 5V benötigt. Der **AD7490** besitzt erst gar keine interne Quelle. Das Bereitstellen einer Referenzspannung ist jedoch relativ unproblematisch, da prinzipiell[4] eine einzige Quelle genügt, um alle 7 ADCs (Vollausbau) zu versorgen.

Preislich nehmen sich die beiden Bausteine nicht viel, sie sind jeweils mit etwas unter 6.000$ für 1000 Stück gelistet. Allerdings ist der **MAX1230** für das konkrete Vorhaben günstiger, da er einen eigenen Taktgenerator und einen Ausgangs-FIFO besitzt was ein flexibleres Timing bei der Ansteuerung erlaubt.

[1] single ended, 8 bei differentieller Ansteuerung

[2] SINAD: **si**gnal to **n**oise plus **a**dditional **d**istorsion

[4] Aus Gründen der Störsicherheit und wegen der einfacheren Verkabelung wird später trotzdem für jeden ADC eine extra Referenz verwendet.

3.3 Schaltungsentwurf

3.3.1 Beschaltung des ADC

Der im vorherigen Abschnitt ausgewählte MAX1230 benötigt eine Versorgungsspannung zwischen 4,75V und 5,25V an Pin 19 (VDD). Von der Controller-Platine werden etwa 5,1V über die Buchse X1 (Tabelle 3.3) zur Verfügung gestellt, die direkt dafür verwendet werden. Zur Stabilisierung der Spannung bei plötzlichen Lastwechseln ist zusätzlich ein Pufferkondensator ($C_{adc} = 100nF$) gegen Masse gelegt. Darüber hinaus ist nur ein Minimum an äußerer Beschaltung notwendig. In erster Linie müssen die benötigten Signaleingänge herausgeführt werden.

Eingänge

Konfiguriert man den Chip für „single ended“-Messungen (näheres zur Konfiguration siehe Abschnitt 4.5.2), stehen 16 analoge Eingänge (AIN0. . . AIN15) zur Verfügung. Immer vier dieser Eingänge werden auf jeweils eine der Buchsen S1 bis S4 geführt. An jede dieser Eingangsbuchsen wird später eine Sensorplatine angeschlossen.

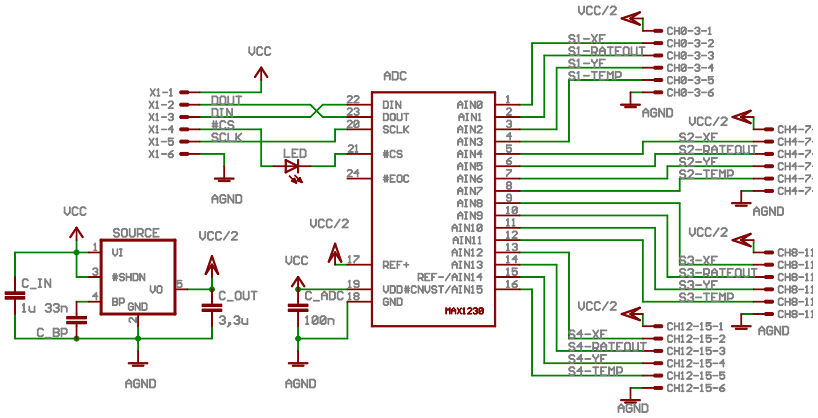


Abbildung 3.2: Schaltplan der ADC-Platine

Um die Übersicht zu erleichtern, wurde für sämtliche Sensor-Verbindungskabel ein einheitliches Farbschema verwendet. Details dazu und zur Pinbelegung sind in Tabelle 3.2 zu finden.

Tabelle 3.2: Signalbelegung der ADC-Eingangsbuchsen (S1. . . S4)

Pin	Signalname	Beschreibung	Farbe
1	VCC/2	konstante 5V-Versorgungsspannung für die Sensoren	rosa
2	XF	Eingang für das X_{FILT} Signal ($\sim a_x$) des Inertialsensors	weiß
3	RATEOUT	Eingang für das RATEOUT Signal ($\sim \omega$) des Gyroskops	gelb
4	YF	Eingang für das Y_{FILT} Signal ($\sim a_y$) des Inertialsensors	grau
5	TEMP	Eingang für das TEMP-Signal ($\sim T$) des Gyroskops	grün
6	GND	Masse	braun

Es ist wichtig zu erwähnen, daß der Eingang AIN14 nur zur Verfügung steht, wenn „single-ended“, also mit einer gemeinsamen Masse für alle Signale gemessen wird. Betreibt man

den ADC im differentiellen Modus wird Pin 15 als Eingang für die negative Referenzspannung REF- verwendet. Ebenso ist AIN15 nur dann verfügbar, wenn die *Conversion-Start* Leitung (CNVST) nicht verwendet wird (näheres dazu im Abschnitt 4.5.2 auf Seite 43).

Ausgang

Sind die Signale erst einmal digitalisiert, werden sie – wie bereits in Abschnitt 2.3 erwähnt – über einen seriellen Bus zum μC auf der Controller-Platine geführt. Konkret handelt es sich beim MAX1230 um den SPI-Bus[5] von MOTOROLA.

Auf die Funktionsweise und die Ansteuerung dieser Bus-Schnittstelle wird noch genauer im Abschnitt 4.3.1 eingegangen, wenn es um die Controller-Platine geht. Details sind unter anderem auch in [SPI] zu finden. An dieser Stelle soll erst einmal genügen, daß für das SPI-Interface die vier Leitungen *Data-Out* (DOUT), *Data-In* (DIN), *Chip-Select* (CS)[6] und *System-Clock* (SCLK) benötigt werden. Diese werden auf die ADC-Ausgangsbuchse X1 geführt, deren Pinbelegung und Farbcodierung in Tabelle 3.3 dargestellt ist.

Tabelle 3.3: Signalbelegung der ADC-Ausgangsbuchse (X1)

Pin	Signalname	Beschreibung	Farbe
1	VCC	unstabilisierte 5,1V-Versorgungsspannung	rosa
2	DOUT	SPI-Datenleitung vom ADC zum Microcontroller	weiß
3	DIN	SPI-Datenleitung vom Microcontroller zum ADC	gelb
4	CS	Auswahlleitung für den ADC	grau
5	SCLK	SPI-Systemtakt	grün
6	GND	Masse	braun

End-Of-Conversion

An Pin 24 kann das *End-Of-Conversion* Signal (EOC) abgegriffen werden. Es ist High, während ein Sample-Vorgang läuft und wechselt auf Low, sobald der ADC fertig ist.

Es wäre sehr vorteilhaft, diese Information im Microcontroller zur Verfügung zu haben. Unglücklicherweise würde das bedeuten, neben den vier SPI-Leitungen plus Versorgungsspannung und Masse noch eine weitere Leitung zu jedem ADC zu führen. Da dies aus Platzgründen jedoch nur schwer möglich ist, muß darauf leider verzichtet werden. Stattdessen wird – etwas weniger elegant – die für einen Sample-Vorgang benötigte (konstante!) Zeit per Hand bestimmt und eine entsprechende Wartezeit in das μC-Programm eingebaut (siehe Abschnitt 4.5.4).

Referenzspannung

Wie bereits in Abschnitt 3.2 angemerkt, besitzt der MAX1230 zwar eine interne Referenzspannungsquelle, diese ist allerdings fest auf 4,096V (1mV/Quantisierungsstufe) eingestellt und damit für den benötigten Eingangsspannungsbereich von 0...5V ungeeignet.

Es ist jedoch möglich, an Pin 17 (REF+) eine extern erzeugte Referenzspannung zwischen 1V und $V_{DD} + 50mV = 5,15V$ anzulegen. Wichtig ist dabei, daß ein Wert von $V_{DD} + 300mV = 5,4V$ niemals überschritten wird, da dies zur Zerstörung des Bausteins führt.

[5] **SPI**: **S**erial **P**eriphereal **I**nterface

[6] Die LED im CS-Signalweg ist unnötig und geht auf einen Designfehler der Schaltung zurück. Im aufgebauten Gerät ist sie jedoch vorhanden. Daher wurde sie (um Verwirrungen zu vermeiden) auch hier beibehalten.

3.3.2 Spannungsstabilisierung

Die Erzeugung dieser 5V-Referenzspannung ist der zweite Aspekt der Schaltung. Sie dient einerseits dem ADC als Referenz und andererseits den Sensoren als Versorgung.

Über Pin 1 (VCC) der X1-Buchse erhält die Schaltung 5,1V Versorgungsspannung von der Controller-Platine. Diese ist aber nicht weiter stabilisiert und kann aufgrund der langen Leitungen zwischen ADC- und Controller-Platine, sowie durch den dort befindlichen Microcontroller mit Störungen überlagert sein. Um sie zu stabilisieren, wird ein Low-Drop Spannungsregler von MAXIM, ein **MAX8877** eingesetzt.

Ausgangsspannung U_{out}	5V $\pm 1,4\%$
max. Ausgangsstrom $I_{out_{max}}$	150mA
Voltage-Drop U_{drop}	55mV (bei $I_{out} = 50mA$)
eigener Stromverbrauch I_{sup}	100μA

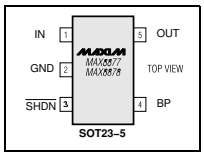


Abbildung 3.3: MAX8877 Low-Drop Spannungsregler Quelle: [MAX reg]

Dieser Regler (Details in Abbildung 3.3) liefert eine konstante 5V-Spannung und kann mit bis zu 150mA belastet werden. Das genügt für die Versorgung der vier Sensorplatinen mit insgesamt 36mA (siehe Abschnitt 2.1.3) völlig und läßt sogar noch genug Reserven für den Fall, daß später noch andere Sensoren mit höherer Stromaufnahme angeschlossen werden sollen.

Neben der 5V-Version (MAX887_EUK50-T) ist dieser Baustein auch für zahlreiche weitere Spannungen zwischen 1,5V und 5V erhältlich. Sollten also einmal Sensoren mit anderen Ansprüchen an Versorgungs- und Referenzspannung angeschlossen werden, so läßt sich die ADC-Platine durch simples Austauschen des Reglers auf eine andere Spannung umrüsten.

Im Gegensatz zu normalen Reglern benötigt ein Low-Drop Regler eine minimale Eingangsspannung, die nur wenig über der gewünschten Ausgangsspannung liegen muß. Beim MAX8877 sind das für Ausgangsströme um 50mA nur 55mV, bei normalen Reglern meist mehrere Volt. Daher ist es problemlos möglich, aus den von der Controller-Platine gelieferten 5,1V stabile 5,0V zu erzeugen.

Die Kapazitäten C_{in} und C_{out} dienen als Pufferkondensatoren für die Kompensation von hohen Lasttransienten und filtern gleichzeitig einen Teil des Rauschens aus der Eingangsspannung (VCC). Für den hier relevanten Betriebsbereich ($U_{out} \geq 2,5V$ und $I_{out} \leq 150mA$) ist im Datenblatt [MAX reg] ein Wert von 1μF für beide Kondensatoren empfohlen. Durch die Erhöhung von C_{out} auf 3,3μF verringert sich das Rauschen zusätzlich.

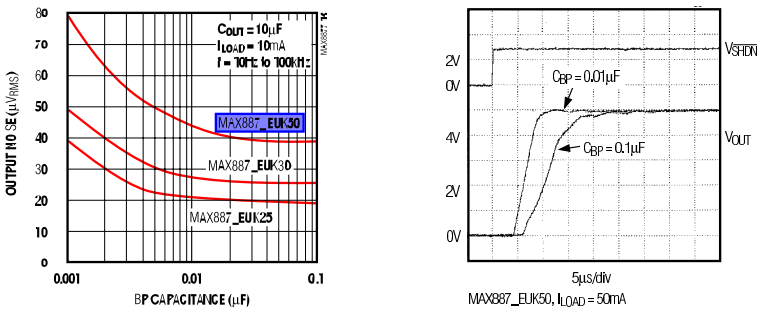


Abbildung 3.4: Einfluß von C_{BP} auf Rauschamplitude und Startzeit (aus [MAX reg])

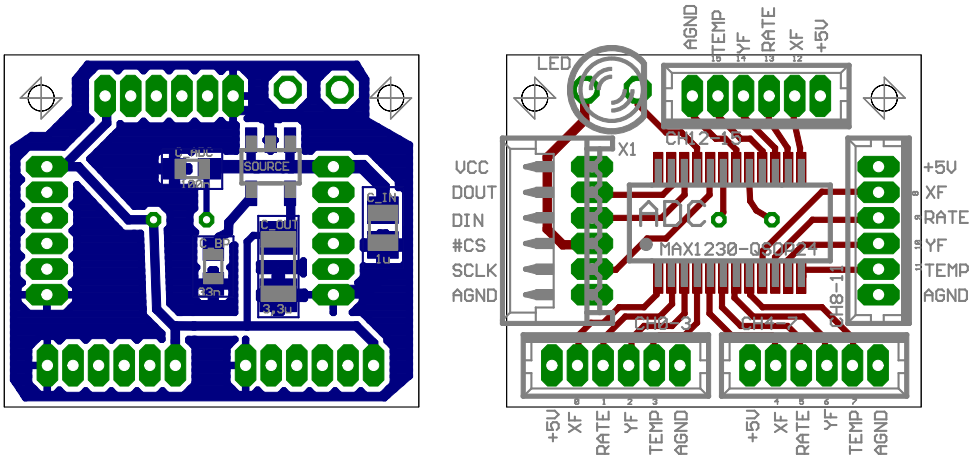
Auch C_{BP} dient dazu, eine rauscharme Ausgangsspannung sicherzustellen. Empfohlen wird ein Wert von 10nF. Höhere Werte bedeuten zwar weniger Rauschen am Ausgang, gleichzeitig dauert es beim aber Einschalten länger, bis die gewünschte Spannung U_{out} erreicht ist. Bei $C_{BP} = 10nF$ sind das etwa 10μs, während sich diese Startzeit bei $C_{BP} = 100nF$ nahezu verdoppelt (Abbildung 3.4 rechts). Da das Timing bei dieser Applikation jedoch relativ unkritisch ist wurde für C_{BP} ein Wert von 33nF gewählt. Größere Kapazitäten sind nicht sinnvoll, da sie nicht weiter zur Rauschminderung beitragen (Abbildung 3.4 links) und nur die Startzeit erhöhen.

Die stabilisierte Spannung (VCC/2) wird den Sensoren an Pin 1 der Ausgangsbuchsen S1 bis S4 zur Verfügung gestellt. Da der MAX8877 über einen integrierten Verpolungsschutz verfügt, sind diese dadurch auch gleich gegen Beschädigung durch Verdrehen des Steckers an X1 geschützt. Ein Verpolen an S1 bis S4 führt allerdings *trotzdem zur Zerstörung der Sensoren*!

Um es noch einmal zusammenzufassen: Die ADC-Platine arbeitet mit zwei verschiedenen Spannungen. Die erste (VCC) wird über X1 von der Controller-Platine geliefert und versorgt den AD-Umsetzer (Pin 19, VDD). Aus VCC wird durch den Regler die stabilisierte und gefilterte Spannung VCC/2. Diese dient als Referenzspannung für den ADC (Pin 17, REF+) und als Versorgungsspannung für die Sensorplatinen (Pin 1 an S1 bis S4).

3.4 Platinenlayout

Auch beim Layout der ADC-Platine, dargestellt in Abbildung 3.5, liegt besonderes Augenmerk auf einer möglichst platzsparenden Unterbringung der einzelnen Bauelemente. Insbesondere die fünf Buchsen für Eingangs- und Ausgangssignale beanspruchen einen Großteil der Platinenfläche. Es werden daher – wie bei den Sensorplatinen – einreihige Pin-Buchsen im 1,25mm-Raster benutzt. Sie sind zusammen mit dem ADC auf der Platinenoberseite (rot) untergebracht.



(a) Bestückung Bottom-Layer

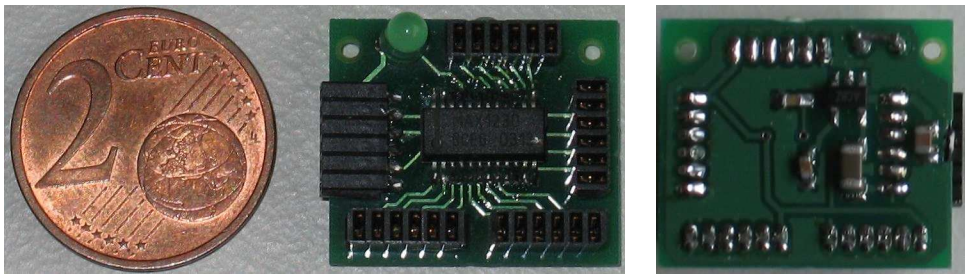
(b) Bestückung Top-Layer

Abbildung 3.5: Layout der ADC-Platine

Die vier Eingangsbuchsen S1...S4 stehen senkrecht, um die Leitungen in möglichst weitem Bogen zu den Sensorplatinen führen zu können. Das vermeidet ein zu starkes Knicken

der Kabel, was bei andauernder Bewegung der Finger zu Kabelbrüchen führen könnte. Außerdem werden so die Kräfte, die durch die Steifheit der Kabel auf die Sensorplatinen wirken, minimiert. Die Ausgangsbuchse X1 dagegen liegt waagerecht, um die Kabel von der flach auf dem Handrücken liegenden Platine möglichst direkt zum Unterarm führen zu können.

Generell ist anzumerken, daß die immer noch relativ großen Steckverbindungen den begrenzenden Faktor bei einer weiteren Verkleinerung der Platinen darstellen. Sowohl die Sensor- als auch (ganz besonders) die ADC-Platinen ließen sich mit kleineren Buchsen auf einen Bruchteil ihrer bisherigen Größe schrumpfen. Leider ist es per Hand schon jetzt sehr schwierig die zu den Buchsen gehörenden Stecker an die Kabel anzulöten. Eine Möglichkeit das Gerät später eventuell weiterzuentwickeln wäre daher, konsequent SMD-Buchsen und im Crimp-Verfahren kontaktierte Stecker zu benutzen.



(a) Oberseite (AD-Umsetzer) (b) Unterseite (U-Regler)

Abbildung 3.6: Fertig aufgebaute ADC-Platine

Der Spannungsregler ist nebst äußerer Beschaltung auf der Unterseite (blau) untergebracht. Hier findet auch der Pufferkondensator des A/D-Umsetzers C_{adc} Platz. Da das SOT23-Gehäuse[7] des MAX8877 nur 0,9mm hoch ist [MAX reg, Seite 9], kann die Platine trotzdem problemlos auf dem Handrücken befestigt werden.

Die Unterseite ist als Massefläche ausgeführt. Das vermindert den Einfluß von Störungen auf die analogen Signale und verbessert die Kühlung des Spannungsreglers. Der führt die entstehende Wärme nämlich über Pin 2 (GND) ab.

[7] SOT: **S**mall **O**utline **T**ransistor; SMD-Bauform mit 2 Anschlüssen auf der einen und 3 auf der anderen Längstseite (siehe Abbildung 3.3 rechts)

Kapitel 4

Die Controller-Platine

Die Controller-Platine stellt das Herz des Gerätes dar. Hier laufen alle Fäden zusammen. Im folgenden Kapitel wird detailliert auf Aufgaben und Implementierung eingegangen. Zur Implementierung gehört natürlich einerseits die Schaltung an sich (Abschnitt 4.3 und 4.4), vor allem aber auch das Steuerprogramm des Microcontrollers, das in Abschnitt 4.5 besprochen wird.

4.1 Aufgaben

4.1.1 Konfiguration und Ansteuerung der ADCs

Die A/D-Umsetzer der im vorherigen Kapitel beschriebenen ADC-Platine sind sehr vielseitig einsetzbare Bausteine. Sie bieten differentielle und nicht differentielle Messung. Im differentiellen Modus hat man wiederum die Wahl zwischen unipolarem ($0...U_{ref}$) und bipolarem ($-\frac{U_{ref}}{2}...+\frac{U_{ref}}{2}$) Betrieb. Der Chip kennt verschiedenen Arten der Ansteuerung und es ist sogar möglich, die aktuelle Die-Temperatur mit auszugeben oder zunächst mehrere Samples in Hardware zu mitteln und erst dieses Ergebnis als Meßwert zu übertragen.

All diese Funktionen werden durch vier 8 Bit breite Register im ADC konfiguriert, die nach jedem Start neu gesetzt werden müssen. Dies geschieht durch das Senden mehrerer Befehlsbytes über die SPI-Schnittstelle (Details siehe Abschnitt 4.5.2) und ist Aufgabe des Microcontrollers.

Sind die ADCs dann konfiguriert, folgt das Sampeln der einzelnen Kanäle. In dem hier gewählten Ansteuerungsmodus funktioniert das folgendermaßen: Der Microcontroller schickt per SPI ein Befehlsbyte an den ADC. Darin ist beschrieben, welche Kanäle abgetastet werden sollen. Der ADC macht sich an die Arbeit und schickt nach Beendigung der Operation die gewünschten Ergebnisse an den μC zurück. Dieses Verfahren ist im Detail in Abschnitt 4.5.4 beschrieben.

4.1.2 Kommunikation mit dem PC

Sind die Abtastwerte des ADCs auf die oben beschriebene Art und Weise beim Microcontroller angekommen, müssen sie dort weiterverarbeitet, und schließlich mit Hilfe des USB-Controllers zum PC geschickt werden.

Weiterverarbeitung heißt dabei in erster Linie, die empfangenen Werte so zu verpacken, daß sie über die USB-Schnittstelle übertragen werden können.

Man muß sich vor Augen halten, daß die USB-Übertragung immer byteweise geschieht. Die Samples sind jedoch 12Bit breit, benötigen also zwei Byte (High und Low) zur Übertragung. Hinzu kommt, daß ein ADC nicht nur einen, sondern bis zu 16 Werte bei jedem Abtastvorgang liefert. Von diesen ADCs sind wiederum mehrere an den Controller angeschlossen.

Es ergibt sich ein dreistufiges Multiplexing: Jeweils ein High- und ein Low-Byte bilden ein Kanal-Sample. Bis zu 16 dieser Kanal-Samples bilden einen ADC-Datensatz und bis zu 8 ADC-Datensätze bilden einen kompletten Datensatz, der die Werte aller Kanäle für einen bestimmten Abtastzeitpunkt $t_k \quad k \in \mathbb{N}$ enthält.

Es muß also ein Übertragungsprotokoll implementiert werden, daß auch bei eventuellen Verlusten einzelner Bytes durch Pufferüberläufe o.ä. die Synchronisation nicht verliert und es gleichzeitig ermöglicht, dem PC eventuelle Fehlerzustände des Gerätes zu signalisieren. Dieses Protokoll ist in Abschnitt 4.5.4 auf Seite 47 beschrieben.

4.1.3 Power Management

Die Controller-Platine muß die ADC-Platinen und damit indirekt auch sämtliche Sensoren des Systems mit Strom versorgen. Dieser kann aus dem USB-Port entnommen werden, dabei sind jedoch einige Einschränkungen zu beachten [USB, Abs. 7.2.3]. So darf jedes Gerät direkt nach dem Einstecken den Bus nur mit maximal 100mA belasten. Erst nach der korrekten Anmeldung des USB-Controllers am Host-System dürfen dann bis zu 500mA[1] entnommen werden. Geht der Host in den sogenannten USB-Suspend Modus, muß die Stromaufnahme ebenfalls reduziert werden. Da allein die Sensorplatinen eines voll ausgebauten Systems ungefähr 250mA benötigen (siehe Abschnitt 2.1.3) und für die AD-Umsetzer nochmals etwa 9mA hinzu kommen, ist das durchaus relevant. Selbst beim Einfinger-System wird die 100mA-Grenze schon knapp überschritten[2].

Es ist also nötig, die externen Komponenten (Sensoren, AD-Umsetzer) gegebenenfalls abschalten zu können, um dem Verbrauch des Gesamtsystems zu senken. Dieses „Power-Management“ wird von der Controller-Platine übernommen und ist in Abschnitt 4.3.3 näher beschrieben.

Für den Fall das später einmal leistungshungrigere Sensoren angeschlossen werden sollen, bietet die Platine außerdem die Möglichkeit einer externen Versorgung, bei der die Einschränkungen des USB-Busses nicht gelten.

4.2 Auswahl der Komponenten

USB-Controller

Wie bereits angedeutet, soll die Übertragung der Daten zum PC über die USB-Schnittstelle erfolgen. Da das USB-Protokoll äußerst komplex und wegen der hohen Busfrequenzen nicht in einem generischen μC zu implementieren ist, muß ein darauf spezialisierter Baustein, ein USB-Controller, verwendet werden.

Die Anforderungen an die zu übertragende Datenmenge sind dabei recht gering. Jedes Sample ist 12Bit groß. Bedingt durch einige Zusatzinformationen und die schon erwähnte byteweise USB-Übertragung ergeben sich 16Bit, also 2Byte pro Sample. Pro ADC kommt schließlich noch ein weiteres Byte Protokolloverhead (siehe Abschnitt 4.5.4 auf Seite 47) hinzu.

[1] beim Anschluß an einen USB-Hub ohne eigene Stromversorgung (bus powered hub) maximal 100mA

[2] Genaue Angaben über den Stromverbrauch aller Komponenten sind in Tabelle 4.3 auf Seite 34 zu finden.

Die zu transportierende Datenrate R berechnet sich also

$$R = f_s \cdot (8 + 16 n_{ch})\, n_{adc} \tag{4.1}$$

Mit einer Abtastfrequenz f_s von 500S/s ergibt sich für das voll ausgebaute Gerät (7 angeschlossenen ADCs (n_{adc}) und 16 Kanäle (n_{ch}) pro ADC) eine Bruttodatenrate von $R = 924 \frac{kBit}{s}$ bzw. $112,79 \frac{KB}{s}$. Das sollte mit jedem Controller problemlos realisierbar sein.

Die schottische Firma FUTURE TECHNOLOGY DEVICES [FT] stellt einen äußerst leistungsfähigen und einfach zu handhabenden USB-Controller mit einer maximalen Übertragungsrate von $1 \frac{MB}{s}$ bereit. Der **FT245BM** (Datenblatt siehe [FT usb]) ist speziell auf die Anbindung von Microcontrollern an den PC ausgelegt. Allerdings ist er nur im 6 x 7 mm^2 großen, 32poligen LQFP Format[3] verfügbar und daher schlecht zu handhaben. Zudem ist noch einiges an äußerer Beschaltung vonnöten. Da die Controller-Platine jedoch in einem externen Gehäuse untergebracht wird, steht ausnahmsweise genügend Platz zur Verfügung. Um den genannten Schwierigkeiten aus dem Weg zu gehen wird daher ein fertiges Evaluation-Board auf Basis des FT245BM verwendet.

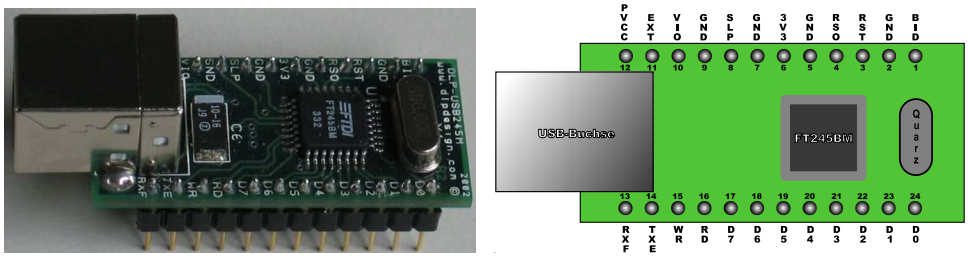


Abbildung 4.1: Evaluation-Board DLP-245M

Dieses Evaluation-Board, das **DLP-USB245M**, wird von der Firma DLP DESIGN [DLP] angeboten. Es ist komplett auf der Fläche eines 24poligen DIL-Gehäuses[4] untergebracht (siehe Abbildung 4.1) und läßt sich genau wie ein solches benutzen. Das macht es insbesondere einfach, den USB-Controller testweise auf einer Experimentierplatine (Abbildung 4.2) zu betreiben. Sämtliche notwendige Beschaltung [DLP evb, S. 15] ist bereits integriert und ein USB-Kabel läßt sich über die USB-B Buchse direkt anschließen.

Microcontroller

Der Microcontroller muß in der Lage sein, sowohl die A/D-Umsetzer [MAX adc] als auch den USB Controller [DLP evb] anzusteuern. Die wichtigste Anforderung ist also, daß ausreichend viele Anschlüsse für Steuerleitungen (IO-Pins) vorhanden sind. Eine Aufstellung dazu findet sich in Tabelle 4.1.

Tabelle 4.1: Benötigte μC-Pins

Pins	SPI-Interface	Pins	USB-Interface
3	SPI-Leitungen (MOSI, MISO, SCLK)	8	Datenleitungen (8 Bit parallel)
8	Chip-Select Leitungen ADC-Auswahl	8	Steuerleitungen
11	**Gesamt**	**16**	**Gesamt**

[3] LQFP: **L**ow **Q**uadrat **F**lat **P**ack; quadratische SMD-Bauform, nur 1,4mm dick
[4] DIL: **D**ual **In**line Package; Standard-Chip-Format mit seitlichen Anschlußreihen im 2,5mm-Raster

Hier wird allerdings von acht angeschlossenen A/D-Umsetzern ausgegangen, was die Fähigkeiten des Gerätes sogar noch über die 108 mindestens benötigten Kanäle hinaus erweitert (siehe unten). Durch den Anschluß von acht ADCs mit je 16 Kanälen sind nunmehr insgesamt maximal 128 Kanäle möglich.

Da das Steuerprogramm nicht allzu umfangreich ist, muß nicht unbedingt sehr viel Programmspeicher (Flash-EEPROM[5]) zur Verfügung stehen. Auch Merkmale wie besonders geringer Energieverbrauch und ein ausgefeiltes Power-Management (z.B. durch flexibles Absenken der Taktfrequenz in mehreren Stufen) spielen nur eine untergeordnete Rolle. Der Rest der Schaltung benötigt soviel Strom, daß es nicht ins Gewicht fällt, ob der μC nun 10mA oder 20mA verbraucht.

Die oben skizzierten Anforderungen werden durch die ATmega-Serie von ATMEL [ATM] gut erfüllt. Diese Controller sind verhältnismäßig einfach zu handhaben und außerdem in vielen unterschiedlich leistungsfähigen Ausführungen verfügbar. Sollte sich herausstellen, daß eine gewünschte Funktion mit dem gewählten Modell nicht zu realisieren ist, kann man mit minimalen Änderungen des Codes auf ein schnelleres oder größeres Modell umsteigen oder umgekehrt, im Nachhinein eine kleinere bzw. langsamere Version verwenden um Platinenfläche bzw. Strom zu sparen.

Um die Handhabung und vor allem die Fehlersuche zu erleichtern, ist ein Controller im DIL-Gehäuse von Vorteil. Das größte Modell, das noch in dieser Gehäuseform verfügbar ist, ist der **ATmega32** [ATM uC], dessen für diese Applikation relevante Eckdaten in Tabelle 4.2 zusammengefaßt sind.

Tabelle 4.2: Eckdaten des ATmega32

Architektur	8Bit RISC	**Betriebsspannung**	4,5…5,5V
max. Taktfrequenz	16MHz	**Verbrauch (typisch)**	12mA (8MHz)
FLASH-Speicher	32KByte	**Schnittstellen**	I^2C, SPI, USART
SRAM (intern)	2 KByte	**IO-Pins**	32 (4 Ports)
Timer	1 x 16Bit, 2 x 8Bit	**ext. IQR-Eingänge**	3

Besonders interessant ist die bereits hardwaremäßig vorhandene SPI-Schnittstelle. Das bedeutet, daß das Busprotokoll nicht extra durch die μC-Software implementiert werden muß - das „manuelle“ Schalten der Datenleitungen entfällt. Hierdurch wird die Anbindung der AD-Umsetzer stark vereinfacht und die CPU des Controllers entlastet.

Die 32 generischen IO-Pins des ATmega32 sind in vier sogenannten *Ports* mit je acht Ein- bzw. Ausgängen organisiert. Für jeden Port existiert ein extra Register, das sogenanntes *Data Register*. Die Leitungen des Ports werden gesetzt, indem ein Byte in das Register geschrieben wird.

Diese registerbasierte Ansteuerung der Pins ist auch der Grund dafür, warum nun – abweichend von der ursprünglichen Spezifikation – der Anschluß von maximal acht ADCs vorgesehen ist. Jeder ADC benötigt eine eigene Chip-Select Leitung (siehe Abschnitt 4.3.1), über die er ausgewählt wird. Da durch einen Registerzugriff sowieso immer alle Leitungen eines Ports auf einmal modifiziert werden, liegt es nahe, diese acht Leitungen auch komplett für die Chip-Select Signale zu nutzen.

Wie aus Tabelle 4.1 zu entnehmen ist, werden für die Steuerung des SPI-Busses insgesamt 11 und für die Übergabe der Daten an den USB-Controller 16 IO-Pins benötigt. Die 32 IO-Leitungen bieten also sogar noch fünf Pins Reserve für spätere Erweiterungen.

[5] EEPROM: **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory; ein elektrisch wiederbeschreibbarer, nichtflüchtiger Halbleiterspeicher

4.3 Schaltungsentwurf

Die Schaltung der Controller-Platine wurde nicht „am Stück“ entworfen und gebaut, sondern sie ist nach und nach auf einer Experimentierplatine (Abbildung 4.2) entstanden.

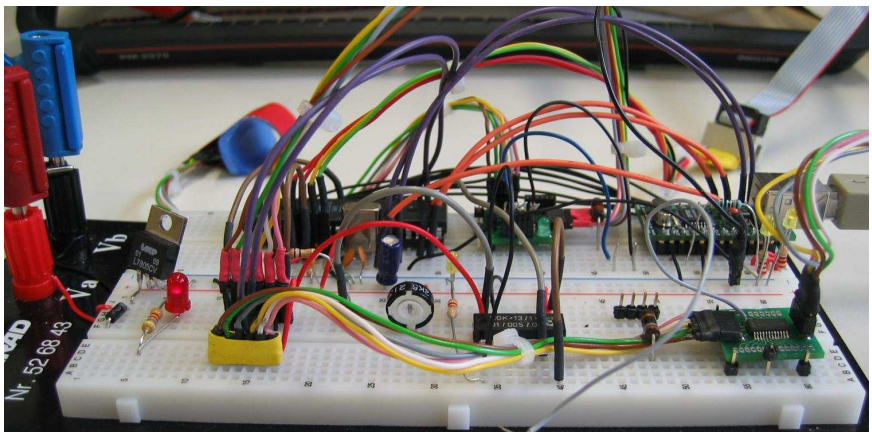


Abbildung 4.2: Controller, aufgebaut auf einer Experimentierplatine

In dem nun folgenden Abschnitt wird dieses schrittweise Herangehen jedoch abgekürzt und die Schaltung gleich in der endgültigen Fassung vorgestellt. Dabei geht es zunächst um die Anbindung der ADC-Platinen über die im ATmega32 integrierte SPI-Schnittstelle und um die Kommunikation mit dem FT245B USB-Controller. Schließlich wird noch die Versorgung aller Komponenten und das damit verbundene Power-Management beschrieben.

4.3.1 SPI-Anbindung

Grundsätzliche Funktionsweise des SPI-Busses

Der SPI-Bus [SPI] ist nach dem klassischen Master-Slave Konzept aufgebaut. In der einfachsten Konfiguration ist ein Gerät der Master und das andere der Slave (Abbildung 4.3).

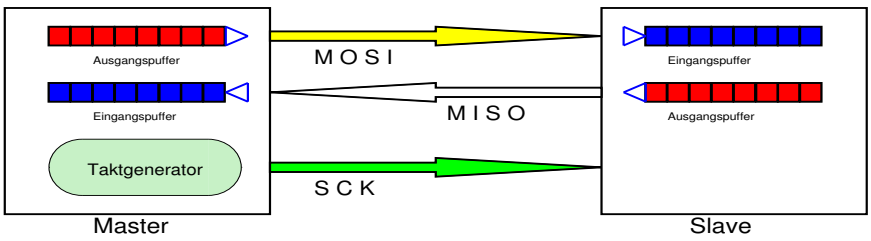


Abbildung 4.3: Einfachste SPI-Konfiguration

Beide besitzen einen Sende- und einen Empfangspuffer von je einem Byte Größe. Der Ausgang des Masters ist mit dem Eingang des Slaves durch die MOSI-Leitung[6] verbunden. Umgekehrt führt die MISO-Leitung[7] vom Slave-Ausgang zum Master-Eingang. Zusätzlich sind beide Geräte noch an eine gemeinsame Taktleitung (SCLK) angeschlossen.

[6] MOSI: **M**aster **O**ut / **S**lave **I**n
[7] MISO: **M**aster **I**n / **S**lave **O**ut

Die Datenübertragung kann man sich wie eine Art Tausch vorstellen. Der Master beginnt, indem er einen Takt auf die SCLK-Leitung legt. Er sendet das Byte in seinem Ausgangspuffer über die MOSI-Leitung in den Eingangspuffer des Slaves, während dieser *gleichzeitig* das Byte aus seinem Ausgangspuffer über die MISO-Leitung zum Master schickt. Mit jeder Flanke des Clock-Signals wird ein Bit übertragen. An Ende des Zykluses befindet sind das Byte aus dem Slave-Ausgangspuffer im Master-Eingangspuffer und umgekehrt.

Sollen mehrere unabhängige Slaves an einem Bus betrieben werden, so wird für jedes eine zusätzliche Chip-Select-Leitung (CS) benötigt. Über die CS-Leitung wählt der Master das Gerät, mit dem er Daten austauschen möchte. Alle anderen Geräte versetzen ihre Ausgangstreiber in einen hochohmigen Tri-State Zustand und verhalten sich passiv.

Konkrete Bus-Konfiguration

In unserem konkreten Fall kommuniziert der Microcontroller (Master) mit bis zu acht AD-Umsetzern (Slaves), wie in Abbildung 4.4 gezeigt. Jeder ADC hat seine eigene CS-Leitung (CS0...7). Darüber hinaus teilen sich alle Geräte die selben MOSI-, MISO- und SCLK-Leitungen, auch wenn in den Datenblättern des ADCs der MISO-Pin mit DOUT und der MOSI-Pin mit DIN bezeichnet ist.

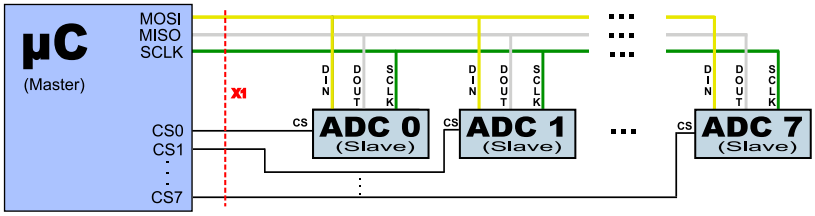


Abbildung 4.4: SPI-Konfiguration des Gerätes

Diese Konfiguration ermöglicht es, alle externen ADCs mit nur 11 digitalen Signalleitungen anzubinden. Die acht CS-Leitungen (CS0...7) sind, wie im Schaltplan auf Seite 31 zu erkennen ist, am Port C des Microcontrollers angeschlossen. Sie werden zusammen mit den drei SPI-Signalen, sowie mit Masse und der Sensor-Speisespannung VCC_SENS (siehe Abschnitt 4.5.6) auf die Buchse X1 geführt. Diese entspricht der rot gestrichelten Linie in Abbildung 4.4 und kennzeichnet die Grenze der Controller-Platine. Ab hier verläuft der SPI-Bus extern und führt direkt zu den einzelnen ADC-Platinen.

In Versuchen mit der Experimentierplatine hat sich gezeigt, daß die Ausgangstreiber am DOUT-Pin des MAX1230 nicht ausreichen, um die MISO-Leitung auf einen TTL-konformen High-Pegel[8] anzuheben. Sie muß daher zusätzlich mit einem hochohmigen Pullup-Widerstand (R_{MISO}) auf 5V gezogen werden. R_{MISO} ist dafür allerdings nicht direkt an die normale Versorgungsspannung VCC/2 angeschlossen sondern liegt an VCC_SENS, der Sensor-Speisespannung. Das hängt mit dem Abschalten der externen Komponenten zusammen und wird in Abschnitt 4.5.6 genau erläutert.

Die SPI-Signale sind recht hochfrequent. Der ATmega32 – getaktet mit 16MHz – ermöglicht Frequenzen von bis zu 8MHz und das SPI-Interface des MAX1230 käme theoretisch sogar mit 10MHz zurecht. Daher ist es entscheidend, daß die Signalflanken möglichst steil und damit einfach zu detektieren sind. Andernfalls kann es zu Übertragungsfehlern auf dem Bus kommen. Das gilt insbesondere, da die Signale über ein externes Kabel zu den AD-Umsetzern geführt werden. Aus diesem Grund wird auch die SCLK Leitung durch einen Pulldown-Widerstand (R_{SCK}) auf Masse gezogen. Messungen mit dem Oszilloskop haben

[8] TTL-High: $U > 2V$; TTL-Low: $U < 0,8V$

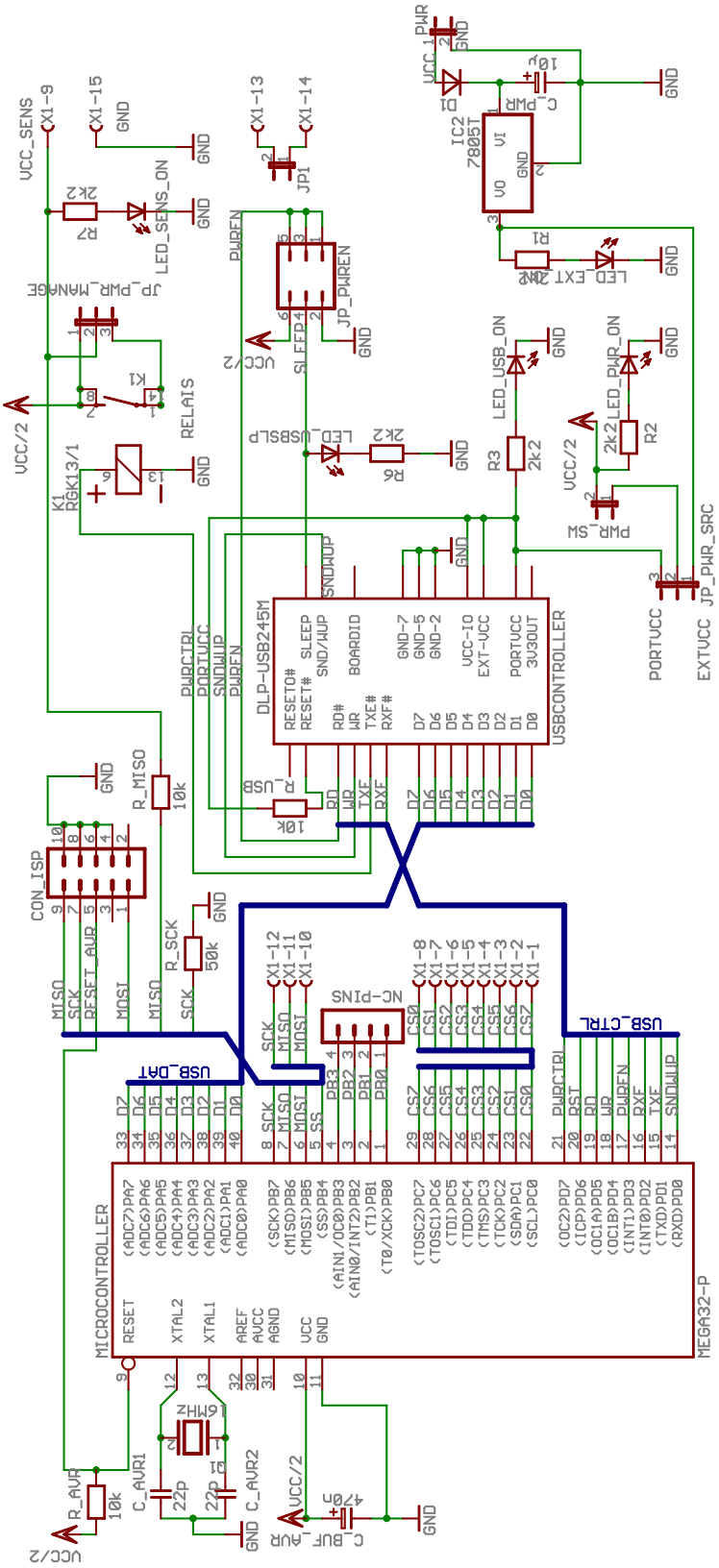


Abbildung 4.5: Schaltplan der Controller-Platine

ergeben, daß dies die Signalqualität deutlich verbessert. Für die MOSI-Leitung ist dagegen kein Widerstand vorgesehen. Hier war keine meßbare Verbesserung zu erkennen.

ISP-Schnittstelle

Nicht nur die Kommunikation mit den ADCs läuft über den SPI-Bus. Auch die Programmierung des Microcontrollers selbst, das sogenannte 'flashen', wird darüber abgewickelt. Zu diesem Zweck werden die drei SPI-Signale zusätzlich noch auf einen 10poligen Stecker (CON_ISP) geführt. Auch die `RESET`-Leitung des Microcontrollers ist dort angeschlossen, so daß dieser nach dem Flashen neu gestartet werden kann. Ein Pullup-Widerstand (R_{AVR}) verhindert, daß die Leitung im Betrieb auf Low geht und dadurch versehentlich ein Reset ausgelöst wird. Die Anschlußbelegung ist dem Schaltplan auf Seite 31 zu entnehmen.

An diesen sogenannten ISP-Stecker[9] kann direkt ein handelsübliches Programmiergerät angeschlossen werden. Dadurch ist es möglich, das Steuerprogramm des Controllers zu ändern, ohne ihn aus seinem Sockel zu entfernen. Auf diese Weise wird das Testen und die spätere Erweiterung des Gerätes stark vereinfacht. Zu beachten ist allerdings, daß dabei der Stecker an X1 abgezogen werden muß, da sonst die ADCs den Programmiervorgang stören. Näheres zum Thema Flashen ist am Anfang von Abschnitt 4.5 zu finden.

4.3.2 USB-Anbindung

Der FT245BM hat ein paralleles Interface für die Anbindung an einen Microcontroller. Jedes der acht Bits in einem Datenbyte wird über eine eigene Leitung übertragen. Diese acht USB-Datenleitungen (`D0...7`) bilden den mit USB_DAT bezeichneten Bus (blau) und sind an Port A des ATmega32 (Pin 33...40) angeschlossen.

Werden über USB_DAT Daten zum USB-Controller gesendet, so landen diese dort zunächst in einem Ausgangs-FIFO. Dort werden sie gesammelt und schließlich über die USB-Schnittstelle zum Rechner geschickt. Sendet der Rechner, so werden die empfangenen Daten wiederum von einem Eingangs-FIFO entgegengenommen und können anschließend über die USB_DAT-Leitungen ausgelesen werden. Um das Lesen und Schreiben der Daten zu steuern, werden vier FIFO-Kontrolleitungen benutzt:

WR übernimmt das momentan auf USB_DAT anliegende Datenbyte und schreibt es in den Ausgangspuffer, sobald die Leitung von High auf Low geht

TXE zeigt an, ob der USB-Controller im Moment bereit zum übernehmen / senden von Daten ist. (*Low:* Ja, *High:* Nein)

RD liest ein Byte aus dem Eingangs-FIFO und legt dessen Wert auf USB_DAT, sobald die Leitung von High auf Low wechselt

RXF zeigt an, ob zur Zeit Daten im Eingangs-FIFO liegen. (*Low:* Ja, *High:* Nein)

Diese Leitungen sind an Port D (Pin 18, 15, 19, 16) des Microcontrollers angeschlossen. Die RXF-Leitung wurde dabei mit Absicht auf PD2 (Pin 16) gelegt, da dieser Pin gleichzeitig auch als Eingang für externe Interruptsignale verwendet werden kann. Es ist also möglich, einen Interrupt auszulösen, sobald Daten vom PC empfangen werden. Diese Funktionalität wird zwar in der momentanen Implementierung nicht verwendet, sie kann aber als Grundlage für interessante Erweiterungen dienen. Mehr dazu in Abschnitt 6.2.

[9] ISP = **I**n **S**ystem **P**rogramming

Neben diesen vier Kontrolleitungen stellt der FT245BM noch die SNDWUP-Leitung zur Verfügung. Diese dient im normalen Modus (kein USB-Suspend) dazu, den Inhalt des Ausgangspuffers an den Rechner zu senden. Normalerweise bestimmt der USB-Controller selbst den optimalen Zeitpunkt dafür. Durch kurzes low setzen der Leitung kann jedoch für spezielle Anwendungen ein sofortiges Senden aller verbliebenen Daten erzwungen werden.

Dies sollte allerdings erst geschehen, wenn sich genügend Daten im FIFO angesammelt haben. Da USB ein paketorientiertes Übertragungsverfahren ist, würde sonst das Verhältnis von Nutzdaten zum Protokolloverhead zu sehr abnehmen, was zu Einbrüchen in der Übertragungsrate führt. In den am Experimentieraufbau durchgeführten Messungen hat die Automatik jedenfalls durchweg bessere Übertragungsraten erzielt als ein „per Hand“ bestimmter Sendezeitpunkt. Für eventuelle spätere Anwendungen ist die SNDWUP-Leitung trotzdem am Port D des Microcontrollers (Pin 14) angeschlossen.

4.3.3 Power-Management

Normale Versorgung

Es gibt zwei Möglichkeiten, die Controller-Platine und damit auch sämtliche daran angeschlossene AD-Umsetzer und Sensoren mit Strom zu versorgen: **Extern** über ein separat angeschlossenes Netzteil oder direkt über den **USB-Bus**.

Die externe Variante ist vorgesehen, falls das gesamte Gerät (zum Beispiel durch den Anschluß leistungshungrigerer Sensoren) irgendwann einmal mehr als 500mA benötigen sollte. Am externen Power-Stecker (PWR) kann eine unstabilisierte Spannung $\geq 6,2V$ angelegt werden. Diese wird dann von einem Low-Drop Spannungsregler (IC2) auf stabile 5V geregelt (siehe Schaltplanausschnitt in Abbildung 4.6). Das hier verwendete Modell **LM2940** [NSC reg] liefert einen Strom von bis zu 1A. Es ist eingangsseitig mit einem 22μF-Kondensator (C_{PWR}) gepuffert um eine stabile Spannungsversorgung sicherzustellen. Die Diode D1am Eingang des Reglers sorgt für einen verpolungssicheren Anschluß. Sobald eine externe Quelle angeschlossen ist, leuchtet die rote LED (LED_EXT_ON), unabhängig davon, ob das Gerät eingeschaltet ist oder nicht.

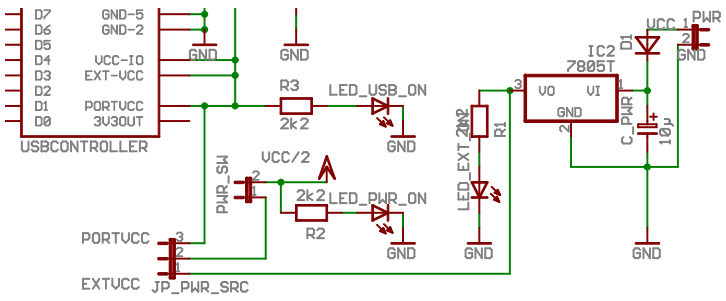


Abbildung 4.6: Ausschnitt: Duale Spannungsversorgung der Controller-Platine

Eleganter ist es jedoch, den Strom direkt aus dem USB-Port zu entnehmen. Das USB-Evaluation Board stellt die 5V Versorgungsspannung des Busses am EXT-VCC Pin zur Verfügung. Hier gilt das bereits mehrfach beschriebene Limit von 500mA bzw. 100mA.

Sowohl die externe Spannung vom Ausgang des LM2940, als auch die USB-Spannung vom EXT-VCC Pin des DLP-USB245M werden beide auf den Jumper JP_PWR_SRC geführt. Durch ihn wird festgelegt, welche der beiden Spannungen benutzt werden soll. Verbindet man die Pins 2 und 1, wird das Gerät extern versorgt. Eine Verbindung von 2 und 3 aktiviert die Versorgung über den USB-Port.

Am PWR_SW Stecker kann ein Hauptschalter für das Gerät angeschlossen werden. Sind beide Kontakte verbunden, ist das Gerät eingeschaltet. Die LED_PWR_ON leuchtet und alle Komponenten (auch die externen!) sind mit Strom versorgt.

Hierbei ist es wichtig zu erwähnen, daß der USB-Controller selbst dann mit Strom aus dem USB-Port versorgt wird, wenn der Hauptschalter aus ist oder eine externe Quelle gewählt wurde. Die EXT-VCC Leitung geht direkt in die Versorgungseingänge (PORTVCC und VCC-IO) des Controllers. Dieser ist also immer aktiv, sobald ein USB-Kabel eingesteckt ist! Auf diese Weise wird ein ständiges An- und Abmelden des Gerätes am PC vermieden.

Suspend-Modus

Um die Stromaufnahme des Gerätes während der Anmeldephase und den USB-Suspend Perioden unter 100mA zu halten müssen zwei Voraussetzungen gegeben sein. Erstens muß der Microcontroller Kenntnis von diesen Vorgängen erhalten und zweitens muß er in der Lage sein, die externen Komponenten, welche allein über 75% des Stromverbrauchs ausmachen (siehe Tabelle 4.3), sicher abzuschalten.

Tabelle 4.3: Stromverbrauch des Gesamtsystems

ext. Komponenten	Normal	Suspend	**int. Komponenten**	Normal	Suspend
Sensor-Platinen	27 x 9mA	0	USB-Controller	25mA	100μA
ADC-Platinen	8 x 1,5mA	0	Microcontroller	24mA	10mA
			LEDs	3 x 2mA	3 x 2mA
			Reed-Relais	15 mA	0
Gesamt	255mA	0	**Gesamt**	70mA	16,1mA

Der USB-Controller stellt für diese Zwecke eine spezielle Leitung (PWREN) zur Verfügung. Beim Anschluß an den USB-Bus ist sie zunächst High. Nachdem der Controller sich erfolgreich angemeldet hat, geht sie auf Low, was normalen Betrieb bedeutet. Versetzt der PC den Bus in den Suspend-Zustand, wird PWREN wieder High. Gleichzeitig geht der USB-Chip selbst in einen Stromsparmodus, in dem er nur noch 100μA benötigt.

Durch den Anschluß der PWREN-Leitung an den Microcontroller kann sich dieser über den aktuellen Bus-Zustand informieren und entsprechende Maßnahmen ergreifen. Um maximale Flexibilität zu gewährleisten, wurde sie auf Pin 17 (Port D) gelegt. Dabei handelt es sich um den zweiten Interrupt-Eingang (INT1) des Microcontrollers. Ein „USB-Suspend“ kann also gegebenenfalls auch einen Interrupt auslösen, der dann die Stromsparmaßnahmen einleitet. In dieser Implementierung wurde jedoch aus verschiedenen Gründen (siehe Abschnitt 4.5.6) nur mit einfachem Polling gearbeitet.

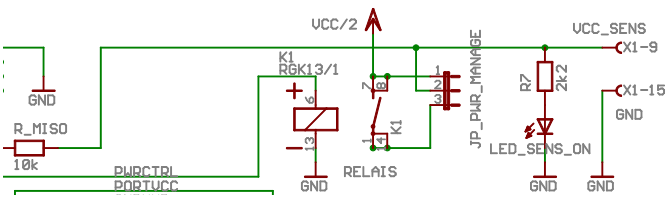


Abbildung 4.7: Ausschnitt: Externe Spannungsabschaltung

Zum Abschalten der externen Komponenten steht dem Microcontroller die PWRCTRL-Leitung am Port D (Pin 21) zur Verfügung. Sie steuert ein Reed-Relais, welches die Sensor-

Speisespannung VCC_SENS an Buchse X1 ein- bzw ausschaltet (siehe Schaltplanausschnitt 4.7). Das Relais benötigt zum Durchschalten (VCC_SENS ein) nur etwa 15mA und kann daher problemlos direkt vom Ausgangstreiber des ATmega32 angesteuert werden [ATM uC, S. 285]. Im USB-Suspend Modus ist es offen (VCC_SENS aus) und verbraucht daher selbst keinen Strom. Mit dem Jumper JP_PWR_MANAGE kann das Relais bei Bedarf wahlweise umgangen und die Sensor-Speisespannung fest auf VCC/2 gelegt werden (Position 1-2). Das Power-Management ist damit deaktiviert.

Beim Abschalten der AD-Umsetzer ist unbedingt zu beachten, daß *sämtliche* SPI-Leitungen vor dem Abschalten auf Low-Pegel liegen! Andernfalls würden die Potentiale an den ADC-Eingängen über der angelegten Betriebsspannung – nämlich Null – liegen, was laut [MAX adc, S. 2] zur Beschädigung der Bausteine führen kann. Die SCLK- und MOSI-Leitungen werden softwareseitig auf Low gesetzt (siehe Abschnitt 4.5.6). Bei der MISO-Leitung wird das dadurch erreicht, daß der schon beschriebene Pullup-Widerstand R_{MISO} nicht direkt an VCC/2 sondern an VCC_SENS angeschlossen ist.

Insgesamt ist es durch die beschriebenen Maßnahmen möglich, den Stromverbrauch des Gesamtsystems von ungefähr 325mA um über 95% auf etwa 16mA zu senken. Durch den Verzicht auf Kontroll-LEDs und die Benutzung des Power-Down Modus am Microcontroller [ATM uC, S. 31] wäre es bei Bedarf auch problemlos möglich, den Verbrauch auf unter 200μA zu drücken.

4.4 Platinenlayout

Die Controller-Schaltung ist auf einer halben EURO-Platine (100 x 80 mm^2) untergebracht, und kann daher bei Bedarf leicht in ein handelsübliches Gehäuse eingebaut werden. Hierfür sind fünf Befestigungslöcher vorgesehen. Das relativ große Format ermöglicht es außerdem, das Design auf problemlose Fertigung und eine einfache Bestückung hin zu optimieren. Abbildung 4.8) zeigt die fertig aufgebaute und angeschlossene Controller-Platine.

Abbildung 4.8: Fertig aufgebaute Controller-Platine

Auf der linken Seite ist der USB-Controller mit angeschlossenem Verbindungskabel zu erkennen. Darunter liegt der Hauptschalter und die Buchse für die externe Spannungsversorgung. Sie wird über den blauen Jumper direkt neben dem Schalter aktiviert. Rechts sitzt der Microcontroller und die SPI-Anschlußbuchse X1. Es wird eine 15polige D-SUB Buchse mit der in Abbildung 4.9 gezeigten Anschlußbelegung verwendet.

Da im Moment nur 13 Leitungen benötigt werden, sind die Anschlüsse 13 und 14 nicht belegt. Für eventuelle, spätere Erweiterungen sind beide über den Stecker JP1 direkt neben

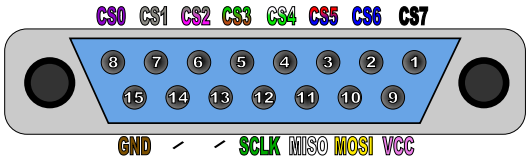


Abbildung 4.9: Pinbelegung der SPI-Buchse (Draufsicht)

der Buchse zugänglich. Ebenso sind die vier unbenutzten Anschlüsse von Port B (Pin 1 bis 4) auf den mit *Port B* beschrifteten Stecker herausgeführt.

Der 16MHz Quarz links neben dem Microcontroller ist austauschbar und kann bei Bedarf durch eine langsamere Variante ersetzt werden, um die Taktfrequenz des Controllers herabzusetzen und so Strom zu sparen. An den *AVR-ISP* Stecker darunter kann ein Programmiergerät angeschlossen werden, um das Steuerprogramm zu ändern. Hierbei ist unbedingt auf die korrekte Polung zu achten. Pin 1 des Steckers ist daher besonders gekennzeichnet.

Die zusätzlich angebrachten Kontroll-LEDs erlauben es, den Zustand des Gerätes zu überwachen. Ist die Platine am USB-Bus angeschlossen, leuchtet die rote, mit USB-ON beschriftete LED. Wird das Gerät eingeschaltet, geht zusätzlich die PWR_ON LED an. Sollte eine externe Spannungsquelle angeschlossen sein, ist das an der EXT_PWR LED zu erkennen und ob die externen Komponenten gerade mit Strom versorgt werden (normaler Modus), zeigt die grüne SENS_ON LED. Ist das Gerät nicht korrekt am Bus angemeldet oder befindet es sich im USB-Suspend Modus, wird dies durch die gelbe USB-SLEEP LED signalisiert.

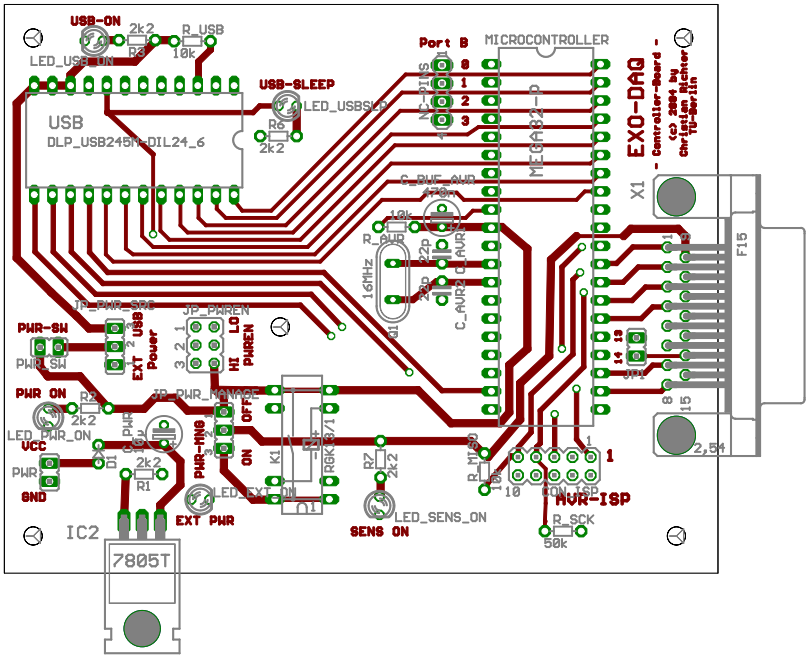


Abbildung 4.10: Layout der Controllerplatine (Bestückungsseite)

Das komplette Platinenlayout ist im Anhang auf Seite 69 zu finden. Abbildung 4.10 zeigt lediglich die Bestückungsseite.

4.5 Microcontroller-Steuerungsprogramm

Die im vorhergehenden Abschnitt beschriebene Schaltung legt den Grundstock dafür, daß der Microcontroller alle Vorgänge im Gerät erfassen und beeinflussen kann. Sämtliche vorhandenen USB- und ADC-Leitungen – auch solche, die im Moment noch gar nicht genutzt werden[10] – sind mit dem Controller verbunden. Was jedoch konkret passiert, wird einzig und allein durch dessen Steuerprogramm bestimmt. Da dies einfach ersetzt werden kann (siehe Abschnitt 4.5.1), läßt sich das Gerät leicht an geänderte Aufgabenstellungen anpassen und ist flexibel erweiterbar. Einige Vorschläge hierzu sind in Kapitel 6 zu finden.

Die grundlegende Struktur des Controller-Programms ist in Abbildung 4.11 dargestellt. Im Hauptprogramm werden zunächst sämtliche Komponenten des Systems initialisiert (Abschnitt 4.5.2). Anschließend läuft das Programm in einer Endlosschleife, in der lediglich geprüft wird, ob ein USB-Suspend auftritt. Ist das der Fall, werden die externen Komponenten herunter- und nach Beendigung dieses Zustands wieder hochgefahren (Abschnitt 4.5.6).

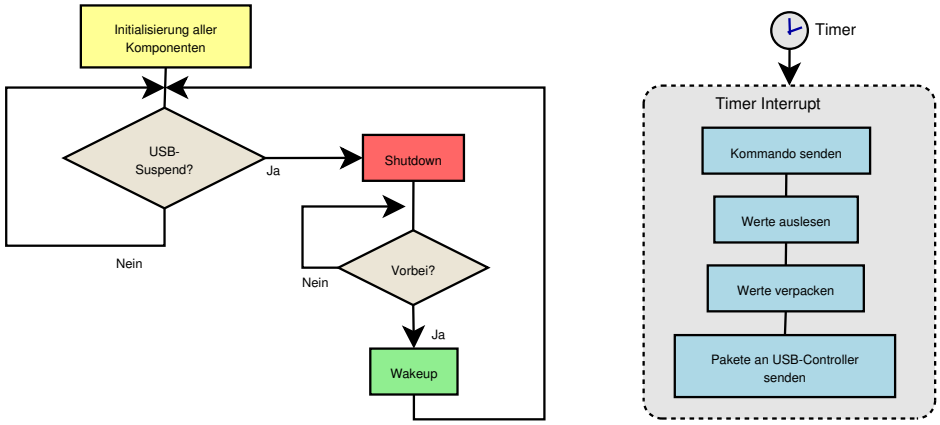


Abbildung 4.11: Flußdiagramm des μC-Steuerprogramms

Die eigentliche Arbeit, also das Entgegennehmen und Verpacken der Werte, sowie das Überstellen der fertigen Datenpakete an den USB-Controller findet in einer Interrupt-Routine statt (Abschnitt 4.5.4). Diese wird durch einen Timer in einstellbaren Abständen regelmäßig aufgerufen. Dadurch ist gewährleistet, daß die exakten Abtastzeitpunkte eingehalten werden und kein Jitter[11] auftritt.

Das Senden der Pakete an den PC und das Abholen der Meßwerte durch die Applikation läuft asynchron dazu und ist nicht mehr zeitkritisch. Die Daten müssen lediglich so rechtzeitig entgegengenommen werden, daß der FIFO des USB-Controllers nicht überläuft. Sollte dies doch einmal geschehen, oder ein anderer Fehlerzustand eintreten, so werden spezielle Fehlerpakete zum PC gesendet, um die Applikation über die Probleme zu unterrichten (Abschnitt 4.5.5).

4.5.1 Die Programmierumgebung

Für die Programmierung eines Microcontrollers kommt zur Zeit aufgrund der starken Nähe zur Hardware und den begrenzten Ressourcen (besonders was Speicher angeht) eigentlich

[10] `RESET`-, `RXF`- und `SND/WUP`-Leitung des USB-Controllers
[11] Jitter: zittern, wackeln; zeitliche Schwankungen bei periodisch auftretenden Ereignissen

nur Assembler oder C als Programmiersprache in Frage. Um eine einfache Erweiterbarkeit sicherzustellen und das Programm gut wartbar zu halten, wurde das Steuerprogramm in ISO-C99 geschrieben. Gegenüber einer reinen Assembler-Lösung ergeben sich dadurch zwar leichte Performance-Einbußen, dafür findet die Programmierung auf einer (zumindest etwas) höheren Abstraktionsebene statt, was in diesem Fall den entscheidenden Ausschlag gab.

Die Atmel-Microcontroller haben den Vorteil, daß sämtliche Tools, die zu ihrer Programmierung nötig sind, unter einer freien Lizenz zur Verfügung stehen. Insbesondere sind das:

- der GCC-Compiler [GCC],
- die GNU Binutils [AVR utils], welche alle nötigen Programme zum Erstellen und Manipulieren von Object-Dateien bereitstellen,
- die AVR-LibC [AVR lib], eine auf AVR-Microcontroller angepaßte Version der GLibC,
- AVR-Dude [AVR dude], ein Programm zum Beschreiben des Controller-EEPROMs („flashen“).

Eine hervorragende Anleitung zum Aufbau einer kompletten Arbeitsumgebung auf Basis dieser Programme ist unter [AVR inst] zu finden. Vorausgesetzt wird eine normale Linux-Installation. Für den Fall das man gezwungen ist, mit dem Betriebssystem „MS Windows“ auszukommen, ist unter [WinAVR] ein selbst extrahierendes Archiv zu finden, daß auch dort eine Arbeitsumgebung auf Basis der genannten Programme zur Verfügung stellt.

In diesem Paket ist auch ein Beispiel-Makefile enthalten, das in einer angepaßten Version in diesem Projekt Verwendung findet. Dieses Makefile macht es möglich, mit einem simplen '`make program`' das Steuerprogramm zu kompilieren und zu linken, die dabei entstandene ELF-Datei in das Intel-HEX Format zu konvertieren und sie anschließend in den Flash-EEPROM des Microcontrollers zu schreiben.

Das dazu nötige Programmiergerät läßt sich schnell aus nur wenigen Bauteilen fertigen. Ein Beispiellayout ist unter [ISP] zu finden. Im Extremfall genügt es sogar, die SPI-Leitungen, sowie Reset und Masse über $1k\Omega$-Widerstände direkt mit dem Parallelport des Rechners zu verbinden [ISP einf]. Im Gegensatz zur erstgenannten Lösung ist dieses Vorgehen allerdings weder sehr komfortabel, noch besonders sicher und kann leicht zu einer Zerstörung des Ports führen.

4.5.2 Initialisierung aller Systemkomponenten

Nach dem Einschalten des Gerätes müssen zunächst einige Initialisierungen vorgenommen werden. So wird z.B. festgelegt, welchen Pegel die IO-Leitungen beim Programmstart besitzen und ob sie als Ausgang oder Eingang fungieren.

Für jeden Port *n* existieren zu diesem Zweck zwei Register, das *Data Register* (`PORTn`) und das *Data Direction Register* (`DDRn`). Durch das Schreiben eines Bytes in ein Data Register, werden alle zu diesem Port gehörenden Leitungen `Pn0`...`Pn7` auf die entsprechenden Pegel gesetzt. Die Anweisung

```
1 PORTA = 0xF0;   // 0xF0 = 0b11110000
```

beispielsweise setzt die oberen vier Leitungen von Port A (`PA7`...`PA4`) – das sind die Pins 33 bis 36 – auf Eins bzw. High. Die unteren vier Leitungen (`PA3`...`PA0`) werden auf Null bzw. Low gesetzt. Das sind die Pins 37 bis 40. Durch das Auslesen des Data Registers lassen sich die aktuellen Werte der entsprechenden Leitungen in Erfahrung bringen. Das

Data-Direction Register legt fest, welche Leitungen des Ports als Ausgänge fungieren, also vom μC selbst geschaltet werden, und welche als Eingang benutzt werden, um die Pegel von extern geschalteten Leitungen auszuwerten. Ein Null-Bit im DDR konfiguriert die dazugehörige Leitung als Eingang, ein Eins-Bit als Ausgang.

Um von der konkreten Beschaltung des μC zu abstrahieren, werden die Namen der Port-Register nicht direkt verwendet. Eine Reihe von `#define`-Anweisungen in `mcprog.h` ermöglicht es, beschreibende Namen zu verwenden, die erst vom Präprozessor durch die konkreten Register ersetzt werden.

```
#define PORT_CS PORTC       /* Port für Chip-Select Leitungen */
#define DDR_CS  DDRC        /* DDR für Chip-Select Leitungen  */

#define PORT_USBDAT PORTA   /* Port für USB Daten-Leitungen */
#define DDR_USBDAT  DDRA    /* DDR für USB Daten-Leitungen */

#define PORT_USBCTRL PORTD /* Port für USB Kontroll-Leitungen */
#define PIN_USBCTRL  PIND  /* Pin für USB Kontroll-Leitungen */
#define DDR_USBCTRL  DDRD  /* DDR für USB Kontroll-Leitungen */
```

Durch diese zusätzliche Abstraktionsebene kann das Programm jederzeit leicht an eine veränderte Beschaltung des Controllers angepaßt werden. Es genügt, die `#defines` zu ändern. Natürlich stehen nicht alle Funktionen an allen Ports zur Verfügung. So können beispielsweise die USB-Datenleitungen nicht einfach auf Port B verlegt werden, da sonst die SPI-Leitungen `PB5...PB7` blockiert wären.

Die Leitungen `0...7` an `PORT_USBCTRL` haben jeweils eine spezielle Bedeutung. Sie wurden daher durch folgende Anweisungen mit eigenen Namen versehen, um die Programmierung zu erleichtern:

```
#define USB_SND 0    /* (OUT) Snd/Wup-Pin */
#define USB_TXE 1    /* (IN)  Zustand des Sendepuffers */
#define USB_RXF 2    /* (IN)  Daten im Empfangspuffer */
#define USB_PWR 3    /* (IN)  USB-Bereit */
#define USB_WR  4    /* (OUT) WR-Pin */
#define USB_RD  5    /* (OUT) RD-Pin */
#define USB_RST 6    /* (OUT) Reset-Pin */
#define PWRCTRL 7    /* (OUT) Steuerung VCC_SENS */
```

Die Konfiguration findet zu Beginn der `main()`-Routine statt. Es werden durch den Aufruf von `cli()` zunächst einmal sämtliche Interrupts gesperrt, damit die Initialisierung nicht von äußeren Ereignissen unterbrochen werden kann. Danach werden nacheinander mehrere Funktionen aufgerufen, welche die einzelnen Subsysteme konfigurieren, bevor schließlich die Interrupts durch ein `sei()` wieder freigegeben werden. Auf die Konfiguration diese Subsysteme wird im Folgenden näher eingegangen.

USB-Subsystem

Die Funktion `usbInit()` wird zuerst ausgeführt und ist für das USB-System zuständig. Für die Kommunikation mit dem USB-Controller sind die acht Datenleitungen an `PORT_USBDAT` und sieben Steuerleitungen an `PORT_USBCTRL` nötig. Die achte Leitung (`PWRCTRL`) gehört eigentlich zum Power-Management (siehe Abschnitt 4.5.6), wird jedoch der Einfachheit halber zusammen mit den anderen USB-Leitungen initialisiert.

Die Richtungen und Grundzustände der beiden Ports werden durch die folgenden Zeilen festgelegt, welche in `usbInit()` zusammengefaßt sind. Das `_BV(n)`-Makro ist dabei nur

eine Abkürzung für `(1«n)` und ermöglicht durch verodern eine komfortable Konstruktion der benötigten Bitmasken.

```
1 DDR_USBDAT  = 0xff;
2 PORT_USBDAT = 0x00;
3 DDR_USBCTRL = _BV(USB_SND) | _BV(USB_WR) | _BV(USB_RD) |
4               _BV(USB_RST) | _BV(PWRCTRL);
5 PORT_USBCTRL= _BV(USB_SND) | _BV(USB_WR) | _BV(USB_RD) |
6               _BV(USB_RST);
7 while ( bit_is_set( PIN_USBCTRL, USB_PWR ) ) { U_WAIT(100); }
```

Der USB-Datenport soll im normalen Betrieb Informationen vom μC zum USB-Controller transportieren und ist deshalb als Ausgang (1) konfiguriert. Der Startzustand ist unwichtig, `0x00` ist eine sichere Wahl, da so alle Leitungen auf Low liegen und bei einem Fehler kein Strom fließen kann.

Die `SND`-, `WR`- und `RD`-Leitungen des USB-Kontrollports werden, wie in Abschnitt 4.3.2 beschrieben, für die Steuerung des USB-FIFOs verwendet und sind demnach Ausgänge. Da der Chip immer auf fallende Flanken triggert, müssen sie initial auf High liegen.

Die `RST`-Leitung ist ebenfalls ein Ausgang und wurde bisher nicht erwähnt. Durch sie kann ein Reset am USB-Controller und damit ein Löschen des kompletten FIFOs, sowie eine Neuanmeldung am Bus erzwungen werden. Momentan wird sie nicht benutzt, zukünftige Anwendungen sind jedoch denkbar. Um eine korrekte Funktion des USB-Controllers (kein Dauerreset) zu gewährleisten, muß sie das ganze Programm hindurch auf High liegen.

Die `TXE`-, `RXF`- und `PWR`-Pins sind als Eingänge geschaltet. `TXE` und `RXF` geben über den Zustand des FIFOS Auskunft und `PWR` signalisiert durch Wechsel von High auf Low, wann der USB-Controller korrekt am Bus angemeldet ist. Die while-Schleife in der letzten Zeile sorgt dafür, daß die Funktion so lange blockiert, bis dies geschehen ist.

SPI-Subsystem

Nachdem `usbInit()` die Kontrolle wieder an `main()` zurückgegeben hat steht demnach fest, daß der Anmeldevorgang abgeschlossen ist und nun bis zu 500mA Strom aus dem Bus entnommen werden können. Als nächstes werden daher die externen Sensoren eingeschaltet, indem die `PWRCTRL`-Leitung auf High gelegt wird. Anschließend wird die `spiInit()`-Funktion aufgerufen, die das SPI-System initialisiert.

```
1 PORT_USBCTRL |= _BV(PWRCTRL);
2 spiInit();
```

Dort werden zunächst einmal die Datenrichtungen für die `SCK`- `MOSI`- und `MISO`-Leitungen gesetzt. Diese liegen beim ATmega32 fest auf Port B, so daß der direkte Registername verwendet werden kann.

```
1 DDRB = _BV(PB7) | _BV(PB5) | _BV(PB4);
```

Neben SCK (`PB7`) und MOSI (`PB5`) wird auch noch die eigentlich unbenutzte SS-Leitung (`PB4`) als Ausgang konfiguriert. Diese Leitung wird im Slave-Betrieb als Chip-Select Eingang benutzt. Da der μC aber als SPI-Master operiert, kann sie theoretisch als normaler IO-Pin benutzt werden. Dabei ist jedoch eine Einschränkung zu beachten: Falls der Pin als Eingang konfiguriert wird, muß er ständig durch einen Pullup-Widerstand auf Highpegel gehalten werden. Wird er auf Low gezogen, geht das SPI-System davon aus, daß irgendwo am Bus ein anderer Master existiert, der versucht den μC als Slave anzusprechen und wechselt folgerichtig in den Slave-Modus. Indem man SS als Ausgang schaltet, wird dieses Problem vermieden. Durch

```
1 DDRB &= ~_BV(PB6);
2 DDR_CS = 0xff ;
3 PORT_CS = 0xff;
```

wird der MISO-Pin (PB6) zum Eingang, um Daten von den DOUT-Ausgängen der A/D-Umsetzer entgegennehmen zu können. Außerdem werden die acht Chip-Select Leitungen CS0...7 als Ausgänge konfiguriert und gleichzeitig auf High (ADC nicht selektiert) gesetzt.

Nachdem nun die Richtungen der einzelnen Pins feststehen, muß jetzt der Rest des SPI-Systems konfiguriert werden. Dies geschieht über das SPI-Control Register SPCR [ATM uC, S. 134 ff.]. Hier kann man einstellen:

- ob beim Empfang eines Datenbytes ein Interrupt ausgelöst werden soll (SPI Interrupt Enable, SPIE),
- ob der Controller im Master- oder Slave-Betrieb operiert (Master-Slave Select, MSTR),
- in welcher Reihenfolge (LSB first / MSB first) die Bits auf den Bus gelegt werden (Data Order, DORD),
- ob die SCLK-Leitung im Ruhezustand Low oder High ist (Clock Polarity, CPOL),
- ob die Daten bei steigender oder Fallender SCLK-Flanke gelesen werden (Clock Phase, CPHA) und
- mit welcher Geschwindigkeit ($f/2$ bis $f/128$) der Bus operiert (SPI Clock Rate Select, SPR0/1).

Da leider kein verbindlicher Standard für den SPI-Bus existiert, müssen diese Parameter für jedes anzuschließende Peripheriegerät extra angepaßt werden. Die Zeile

```
1 SPCR = _BV(SPE) | _BV(MSTR);
```

stellt die für den MAX1230 passenden Parameter [MAX adc, S. 10] ein. SCLK ist Idle-Zustand Low und Daten werden mit steigender Flanke gelesen, wobei das MSB zuerst übertragen wird. Der ATmega32 ist Master und es wird kein SPI-Interrupt ausgelöst.

Die Geschwindigkeit (SCLK-Frequenz) wurde auf $f/4$, also 4MHz[12] festgelegt. Dieser Wert ist eher konservativ gewählt und stellt in jedem Fall eine stabile Datenübertragung sicher. Hier liegt jedoch eventuell noch Potential für eine Optimierung des Systems. Zwar sind die SPI-Leitungen recht lang, so daß hohe Busfrequenzen leicht auf Kosten der Flankensteilheit gehen, trotzdem kann die Frequenz möglicherweise noch etwas angehoben werden. Maximal möglich wären $f/2$, also 8MHz, die AD-Umsetzer erlauben sogar Busfrequenzen bis 10MHz. Getestet wurde die aktuell vorliegende Implementierung jedoch nur mit 4MHz.

Da in der gewählten Konfiguration ein eingetroffenes Datenbyte keinen Interrupt auslöst, muß der Abschluß einer Übertragung anders signalisiert werden. Dies geschieht über das SPI-Interrupt Flag (SPIF) im sogenannten SPI-Status Register (SPSR). Das Flag wird bei jedem Lesezugriff auf das Register wieder zurückgesetzt. Ein einfaches

```
1 status = SPSR;
```

am Ende von spiInit() sorgt also dafür, daß das Flag nach der Initialisierung des SPI-Systems zunächst ungesetzt ist.

[12] Der ATmega32 ist in dieser Schaltung mit den maximal möglichen 16MHz getaktet.

Timer

Der ATmega verfügt über zwei 8Bit- und einen 16Bit-Timer. Sie funktionieren im wesentlichen folgendermaßen:
Mit jedem Clock-Impuls wird der Timerwert um eins inkrementiert. Beim Erreichen eines vorgegebenen Wertes k oder wahlweise auch bei Überlauf des Timers wird ein Interrupt ausgelöst oder ein Flag gesetzt, welches dann ausgewertet werden kann. Außerdem ist es möglich, die Taktfrequenz mit einem programmierbaren Teiler (dem sog. *Prescaler*) auf $\frac{f}{p}$ herunterzuteilen, so daß der Timerwert nur jeden p-ten Clock-Impuls inkrementiert wird. Darüber hinaus existieren noch viele weitere Möglichkeiten, die Timer z.B. zum Erzeugen von PWM-Signalen zu verwenden oder externe Pulse damit zu zählen. Diese Fähigkeiten werden jedoch für das System nicht benötigt und hier daher nicht weiter ausgeführt.

Die Zeitspanne zwischen zwei Timer-Interrupts ergibt sich zu

$$\Delta t_k = \frac{p}{f} \cdot k \quad \forall k \in \{0 \ldots (2^n - 1)\} \subset \mathbb{N} \tag{4.2}$$

Bei einer Taktfrequenz f von 16MHz und einem Prescale-Faktor von $p = 8$ dauert ein Clock-Tick genau $\Delta t_1 = \frac{8}{16MHz} = 500ns$. Die Zeit läßt sich also beim 16Bit-Timer ($n = 16$) in 500ns Schritten zwischen $\Delta t_0 = 0$ und $\Delta t_{65535} = 32,7ms$ einstellen. Bei einem 8Bit-Timer ($n = 8$) liegt diese obere Grenze bei nur $\Delta t_{255} = 0,13ms$. Erhöht man den Prescale-Faktor p, lassen sich zwar größere Zeitbereiche überstreichen, das geht allerdings zu Lasten der möglichen Auflösung – die Schrittweite wird größer.

Aus diesem Grund wird für das System der 16Bit-Timer verwendet. Das damit erreichbare maximale Timerintervall von 32,77ms entspricht einer minimalen Abtastfrequenz von **30,5Hz**, was in der Praxis genügen sollte. Reicht das nicht aus, kann man immer noch den Prescaler von 8 auf 64 heraufsetzen, was eine Minimalfrequenz von 3,8Hz bei einer Schrittweite von 4μs entspricht.

Das Konfigurieren und Starten des 16Bit-Timers übernimmt die Funktion `timerInit()`.

```
1 void timerInit(unsigned int tics) {
2       TCCR1B = _BV(CS11) | _BV(WGM12);
3       OCR1A  = tics;
4       TCNT1  = 0;
5       TIMSK |= _BV(OCIE1A);
6 }
```

Die Einstellung ist wegen der oben erwähnten Vielseitigkeit sehr komplex und eine detaillierte Erklärung der einzelnen Register und ihrer Bedeutung würde den Rahmen dieses Kapitels sprengen. Stattdessen sei auf [ATM uC, S. 82 ff.] verwiesen. Hier soll nur das prinzipielle Vorgehen beschrieben werden.

Zunächst wird im *Timer-Counter-Control Register B* (`TCCR1B`) der oben gewählte Prescale-Faktor von 8 eingestellt (Zeile 2). Außerdem wird der Timer in einen Modus versetzt, in dem er bis zu einem im *Output-Compare Register A* (OCRA1) festgelegten Wert hochzählt und nach Erreichen des Wertes wieder auf Null zurückgesetzt wird. Dieser Maximalwert entspricht dem Parameter k aus Gleichung 4.2 und wird der Funktion als Parameter übergeben. Zeile 3 schreibt ihn in das entsprechende Register. Anschließend wird der, im Register *Timer Count* (`TCNT1`) gespeicherte, aktuelle Timerwert auf Null gesetzt (Zeile 4) und das Auslösen eines Timer-Interrupts beim Erreichen des Maximalwertes aktiviert (Zeile 5).

Der Timer ist nun gestartet. Da im Moment jedoch noch sämtliche Interrupts gesperrt sind, wird zunächst kein Timer-IRQ ausgelöst. Es werden noch keine Meßwerte aufgenommen.

A/D-Umsetzer

Nachdem sämtliche Initialisierungen innerhalb des Microcontrollers abgeschlossen sind, geht es nun an die Peripheriegeräte – die AD-Umsetzer. Wie bereits in Abschnitt 4.1.1 erwähnt, müssen die diese nach jedem Start konfiguriert werden. Dazu stehen vier Register zur Verfügung, die im Folgenden kurz beschrieben werden.

Im Setup-Register steht, welche *Referenzspannungsquelle* (intern/extern) der ADC verwendet und welcher *Clock-Mode* benutzt werden soll. Der Clock-Mode beschreibt, auf welche Weise der Abtastvorgang ausgelöst wird. Dies kann entweder durch ein Signal auf einer extra Conversion-Start Leitung (`CNVST`) geschehen, oder indem einfach ein Startbefehl (siehe Abschnitt 4.5.4) über das SPI-Interface gesendet wird. Außerdem ist es möglich, den Vorgang entweder extern über die `SCLK`-Leitung zu takten oder den internen Takt des MAX1230 zu benutzen.

Im Unipolar- bzw. Bipolar-Register können einzelne Kanalpaare für differentielle Messungen zusammengefaßt werden. Ein 1-Bit im Unipolar-Register aktiviert die differentielle Messung für ein Kanalpaar im unipolaren Modus ($0...U_{ref}$) , eine 1 an entsprechender Stelle im Bipolar-Register aktiviert den differentiellen, bipolaren Modus ($-\frac{U_{ref}}{2}...+\frac{U_{ref}}{2}$) für diese Kanalpaar.

Das Averaging-Register steuert, wie oft eine angeforderte Messung wiederholt werden soll, bevor der Mittelwert der dadurch erhaltenen Abtastwerte schließlich als Ergebnis zurückgesendet werden soll.

Die Register werden beschrieben, indem man Befehlsbytes mit einer bestimmten Präambel, welche das gewünschte Zielregister kennzeichnet, über den SPI-Bus zum ADC schickt. Das Setup- und das Averaging-Register sind auf diese Weise direkt erreichbar. Die Unipolar- und Bipolar-Register können nur indirekt über das Setup-Register erreicht werden. Die letzten beiden Bits dieses Registers bestimmen, ob ein *unmittelbar* nach dem Setup-Byte gesendetes Befehlsbyte ins Unipolar- oder Bipolar-Register geschrieben wird. Abbildung 4.12 veranschaulicht diesen Zusammenhang.

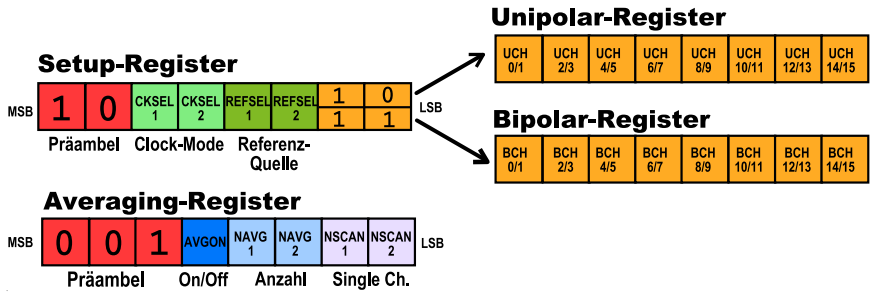


Abbildung 4.12: Registersatz des MAX1230

Da für das System im Moment keine differentielle Messung vorgesehen ist, bleiben sowohl Unipolar- als auch Bipolar-Register im Default-Zustand `0x00`. Auch eine Mittelung wird zur Zeit nicht durchgeführt, im Averaging-Register ist also nur Bit 5 (das dritte Bit der Präambel) gesetzt. Im Setup-Register müssen jedoch einige Einstellungen vorgenommen werden.

Zunächst muß der angeschlossene, externe MAX8877 als nicht-differentielle Referenzspannungsquelle ausgewählt werden (`REFSEL1/0=01`). Außerdem wird ein Clock-Mode eingestellt, in dem ein Abtastvorgang allein durch senden eines SPI-Befehls angestoßen

wird (CKSEL1/0=10). Das ist nötig, da eine separate CNVST-Leitung erstens einen der 16 Signaleingänge belegen, und zweitens die Verkabelung auf der Hand zusätzlich verkomplizieren würde. Der gesamte Vorgang wird in diesem Modus vom internen Takt des ADC gesteuert, da die SPI-Schnittstelle des Microcontrollers nur während der Datenübertragung ein Taktsignal an SCLK zur Verfügung stellt.

Die ermittelten Werte für die Konfigurationsregister sind als Präprozessorkonstanten in mcprog.h abgelegt und können daher jederzeit veränderten Gegebenheiten angepaßt werden. Besonders das Mitteln der Meßwerte (Averaging-Register) kann in einigen Anwendungsfällen nützlich sein und ist durch eine Änderung der Konstante ADC_AVR_REG schnell aktiviert. Dabei ist jedoch zu beachten, daß sich durch das mehrfache Abtasten die für eine Umsetzung benötigte Zeit (*conversion time*) vervielfacht.

Die Funktion adcInit() nimmt die Konstanten als Parameter entgegen, und schreibt sie in die Register des angegebenen ADCs. Da es sich dabei um eine normale SPI-Übertragung handelt, arbeitet sie im Prinzip genau wie die in Abschnitt 4.5.3 beschriebene toSPI()-Funktion. Der einzige Unterschied ist, daß die Werte für das Setup- und für das Uni- bzw. Bipolar-Register *direkt* nacheinander auf den Bus gelegt werden müssen, ohne das zwischendurch die CS-Leitung high wird, was normalerweise das Ende einer Übertragung signalisiert.

Da mehrere ADCs zu konfigurieren sind, wird adcInit() in einer Schleife aufgerufen. Das Makro in Zeile 1 veranlaßt den μC, etwa 1ms zu warten, bis die ADCs – deren Spannungsversorgung ja gerade erst eingeschaltet wurde – betriebsbereit sind.

```
1 U_WAIT(1000);
2 for (unsigned char adcNr=0; adcNr<8; adcNr++) {
3   if ( (activeADCs & _BV(adcNr)) != 0x00 ) {
4     adcInit(adcNr, ADC_SETUP_REG, ADC_UNIPOLAR_REG, ADC_AVRG_REG);
5   }
6 }
```

Da möglicherweise nicht alle acht ADCs am Bus angeschlossen sind, wird vorher in Zeile 3 geprüft, ob der entsprechende Chip tatsächlich als aktiv gekennzeichnet ist. Die Information darüber ist in zwei globalen Variablen abgelegt.

```
1 unsigned char channelADCs[] = { 15, 15, 0, 0, 0, 0, 0, 0 };
2 unsigned char activeADCs    = _BV(0) | _BV(1);
```

Der Array channelADCs[] hält für jeden ADC fest, bis zu welchem Kanal er belegt ist. Sind beispielsweise nur acht der 16 verfügbaren Eingänge genutzt[13], so werden auch nur die Kanäle 0 bis 7 gemessen und zum PC übertragen. Das spart Zeit und ermöglicht höhere Abtastraten, wenn die ADCs nicht voll bestückt sind.

Aus diesem Array könnte man auch ableiten, ob ein ADC angeschlossen ist und benutzt werden soll, oder nicht. Da diese Information aber nicht nur bei der Konfiguration eine Rolle spielt, sondern auch im Timer-Interrupt beim (zeitkritischen) Abtasten benötigt wird, existiert parallel zum Array noch der 8Bit-Wert activeADCs. Ein gesetztes Bit dort aktiviert den entsprechenden ADC, ein nicht gesetztes deaktiviert ihn.

[13] oder ist ein kleinerer ADC (beispielsweise ein MAX1226 mit nur acht Eingängen) angeschlossen

4.5.3 Grundlegende Ein- und Ausgabe

Nachdem das Gerät komplett initialisiert ist stellt sich die Frage, wie Daten über den SPI-Bus gesendet und empfangen werden können und wie der USB-Controller angesteuert werden muß.

Da der ATmega32 über eine Hardware-Implementierung des SPI-Protokolls verfügt, ist das Senden extrem einfach. Es muß lediglich die CS-Leitung für den gewünschten ADC `nr` auf Low gelegt (Zeile 2) und das zu übertragene Datenbyte `cmd` in das SPI-Data Register (`SPDR`) geschrieben werden (Zeile 3).

```
1 inline void toSPI(unsigned char nr, byte cmd) {
2    PORT_CS = ~_BV(nr);
3    SPDR = cmd;
4    while ( !( SPSR & (1<<SPIF) ) ) { ... }
5    PORT_CS |= _BV(nr);
6 }
```

Sobald die Übertragung erfolgreich beendet ist, wird das SPI-Interrupt Flag (`SPIF`) im Status-Register (`SPSR`) gesetzt. Ist dies geschehen (Zeile 4), kann die `CS`-Leitung des ADC wieder auf High gesetzt werden (Zeile 5). Das `SPIF`-Flag wird beim Auslesen automatisch auf Null zurückgesetzt.

Das Empfangen funktioniert genauso. Tatsächlich wird ja mit jedem gesendeten Byte auch gleichzeitig eines empfangen (vergleiche Abschnitt 4.3.1). Nach dem Aufruf von `toSPI()` steht dieses empfangene Byte im SPI-Data Register und kann von dort gelesen werden.

Für das Senden von Daten zum USB-Controller ist die Funktion `toUSB()` zuständig. Hier ein Ausschnitt:

```
1 while ( bit_is_set( PIN_USBCTRL, USB_TXE ) ) { ... }
2 PORT_USBDAT = data;
3 PORT_USBCTRL &= ~_BV(USB_WR);
4 while ( bit_is_set( PIN_USBCTRL, USB_TXE ) ) { ... }
5 PORT_USBCTRL |= _BV(USB_WR);
```

Zunächst wird geprüft, ob der Controller überhaupt bereit ist, Daten entgegenzunehmen. Solange die `TXE`-Leitung auf High liegt, ist entweder der Ausgangs-FIFO voll, oder es läuft gerade ein Datentransfer. Sobald sie auf Low wechselt (Zeile 1), kann die Übertragung starten. Das zu sendende Datenbyte `data` wird angelegt (Zeile 2) und die `WR`-Leitung wird von High auf Low umgeschaltet (Zeile 3). Diese fallende Flanke ist für den USB-Controller das Signal, daß die anliegenden Daten stabil sind und in den FIFO übernommen werden können. Während der Übernahme wechselt die `TXE`-Leitung auf High. Sobald sie wieder auf Low liegt (Zeile 4), ist der Transfer abgeschlossen und die `WR`-Leitung kann wieder zurück auf High gesetzt werden (Zeile 5).

Das Empfangen von Daten ist im Moment noch nicht implementiert. Prinzipiell funktioniert es folgendermaßen:

Die `RXF`-Leitung wird vom USB-Controller auf Low gesetzt, sobald Daten vom PC im Eingangs-FIFO verfügbar sind. `RXF` ist am Interrupt-Eingang INT0 angeschlossen, so daß dieser Wechsel dazu benutzt werden kann, eine Interrupt-Service Routine auszuführen, in der das anliegende Byte zum Microcontroller übertragen und dort gespeichert wird. Das Gerüst für die Funktion `fromUSB()`, welche das eingelesene Byte entgegennimmt, existiert bereits.

Daten vom PC zu empfangen ist in der aktuellen Implementierung nicht notwendig. Es existieren jedoch verschiedene Ansätze, das Gerät weiterzuentwickeln. Eine mögliche Anwendung für `fromUSB()` ist in Abschnitt 6.2 zu finden.

4.5.4 Timer-Interrupt

Die vom Timer regelmäßig aufgerufene Interrupt-Service Routine (ISR) ist der Kern des Microcontroller-Programms. Ihre grundlegende Struktur wurde bereits in Abbildung 4.11 auf Seite 37 dargestellt:
Als Erstes wird an alle A/D-Umsetzer ein Kommando gesendet, daß den Abtastvorgang eingeleitet. Nach dessen Abschluß liegen die Meßwerte aller angeforderten Kanäle im Ausgangs-FIFO des ADC. Von dort werden sie über den SPI-Bus ausgelesen, in Datenpakete verpackt und schließlich dem USB-Controller übergeben, der sich um den Transport der Pakete zum PC kümmert.

Senden des ADC-Kommandos

Das ADC-Kommando ist ein Befehlsbyte, das angibt, welche Kanäle abgetastet werden sollen. Es wird per SPI in das sogenannte *Conversion Register* des ADCs geschrieben und hat den in Abbildung 4.13 gezeigten Aufbau.

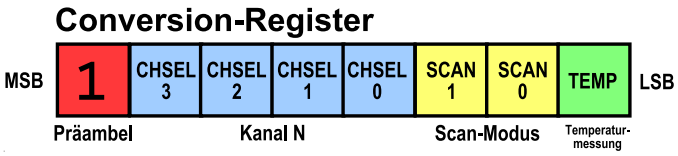


Abbildung 4.13: Conversion Register des AD-Umsetzers

Die 1 im MSB ist die Präambel für das Conversion Register. Die nächsten vier `CHSEL`-Bits spezifizieren einen Kanal $N \in 0...15$. Durch die nun folgenden zwei `SCAN`-Bits wird bestimmt, was dieser Kanal N bedeutet. Es kann

- vom niedrigsten Kanal (0) bis N gescannt werden (`SCAN1/0=00`)
- von N bis zum höchsten Kanal[14] gescannt werden (`SCAN1/0=01`)
- der Kanal N mehrmals abgetastet und gemittelt werden (`SCAN1/0=10`)
- der Kanal N ein einziges Mal abgetastet werden (`SCAN1/0=11`).

Für dieses Gerät wird der Modus 00 verwendet. Es werden also immer die untersten $N+1$ Kanäle abgetastet. Diese Annahme wird beim Auslesen des FIFOs benutzt, um die Anzahl der Ergebnisse zu ermitteln. Der Scan-Modus kann daher nicht einfach abgeändert werden, ohne daß weitergehende Änderungen im Code nötig sind!

Das LSB signalisiert, ob zusätzlich zu den Abtastwerten noch ein Temperaturwert aufgenommen werden soll (`TEMP=1`) oder nicht (`TEMP=0`). Diese Temperaturmessung benötigt mit 55μs mehr als 13 mal soviel Zeit wie das Abtasten eines Kanals (4,1μs) und ist daher im Interesse eine höheren Abtastrate deaktiviert.

Der ADC arbeitet im *Clock Mode* 10 [MAX adc, S. 17 ff]. Das bedeutet, daß ein Schreibzugriff auf dieses Register einen internen Taktgenerator startet und den Abtastvorgang für alle angeforderten Kanäle anstößt. Die Kommandos werden nacheinander an alle ADCs übertragen (Zeile 1). Auf diese Weise kann die Parallelität der Bausteine gut ausgenutzt werden. Während der zweite ADC seine Instruktionen erhält, kann der erste bereits mit der Arbeit beginnen.

[14] beim MAX1230 ist das Kanal 15

```
1 for (adcNr=0; adcNr<8; adcNr++) {
2    if ( (activeADCs & _BV(adcNr)) != 0x00 ) {
3      toSPI(adcNr, ADC_CNV_REG | (channelADCs[adcNr] << 3) );
4    }
5 }
```

Berücksichtigt werden dabei jedoch nur die als aktiv gekennzeichneten ADCs (Zeile 2). Die Übertragung selbst erledigt die `toSPI()`-Funktion (Zeile 3). Das übergebene Befehlsbyte setzt sich dabei aus der in `ADC_CNV_REG` definierten Bitmaske (Präambel, Scan-Modus 00, keine Temperaturmessung) und dem obersten Kanal *N* zusammen, der durch Bitverschiebung aus `channelADCs[]` gewonnen wird.

Paketformat

Bevor es im nächsten Abschnitt um das Auslesen der Ergebnisse aus dem ADC-FIFO und um das Verpacken und Absenden der gewonnenen Werte geht, soll noch kurz auf das dazu benutzte Paketformat eingegangen werden.

Wie bereits in Abschnitt 4.1.2 ausgeführt, ergibt sich der zum PC gesendete Datenstrom durch Multiplexen der einzelnen Kanäle. Ein einfaches Abzählen der Bytes genügt jedoch nicht, um diese den Kanälen zuzuordnen. Einzelne Datenbytes können durch Pufferüberläufe im USB-FIFO verloren gehen und nach einem Neustart des Microcontrollers stören eventuell im FIFO verbliebene Daten die Synchronisation. Auch Fehlermeldungen (siehe Abschnitt 4.5.5) oder eventuelle andere Nachrichten (Abschnitt 6.2) fügen zusätzliche Bytes in den Datenstrom ein.

Es ist also nicht möglich, die Daten einfach so weiterzuschicken, wie sie von den einzelnen ADCs geliefert werden. Stattdessen müssen Teile von Meßwerten (Datenpakete) und andere Daten wie z.B. Fehlermeldungen (Kontrollpakete) als solche gekennzeichnet werden, um eine Unterscheidung zu ermöglichen. Außerdem müssen in regelmäßigen Abständen Markierungen gesendet werden, so daß die Synchronisation spätestens nach *n* Werten zuverlässig wiederhergestellt werden kann. Der Datenstrom vom Microcontroller zum PC sieht dann wie in Abbildung 4.14 dargestellt aus.

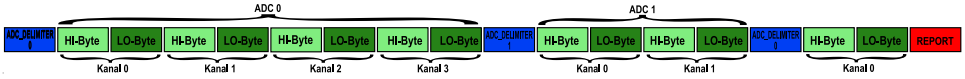


Abbildung 4.14: Datenstrom vom μC zum PC für zwei ADCs

Der 12Bit-Abtastwert eines Kanals wird in zwei Datenpakete, ein High-Byte (Hellgrün) und ein Low-Byte (Dunkelgrün) verpackt. Es wird immer zuerst das High-Byte und danach das Low-Byte übertragen. Vor den Daten eines jeden ADCs wird zusätzlich ein ADC_DELIMITER Paket (Blau) als Markierung eingefügt. Es enthält die Nummer des ADCs, dessen Meßwerte nun folgen. Auf diese Weise wird sichergestellt, daß die Datenübertragung spätestens beim nächsten ADC, also nach maximal 16 Abtastwerten wieder synchron ist. Die empfangende Applikation (Kapitel 5) muß noch nicht einmal wissen, wieviele Kanäle ein ADC momentan abtastet. Das ergibt sich automatisch aus den Delimiter-Paketen. Im obigen Beispiel benutzt der erste ADC vier und der zweite drei Kanäle.

Meldungen (Rot) sind – genau wie die ADC_DELIMITER – Kontrollpakete. Sie enthalten einen 6Bit breiten Fehlercode (0...63) und können zu jedem Zeitpunkt in den Datenstrom eingefügt werden. Ihre Bedeutung hängt stark vom Kontext ab. Durch den vorhergegangenen ADC_DELIMITER ist jedoch zumindest bekannt, auf welche ADC sich die Meldung bezieht.

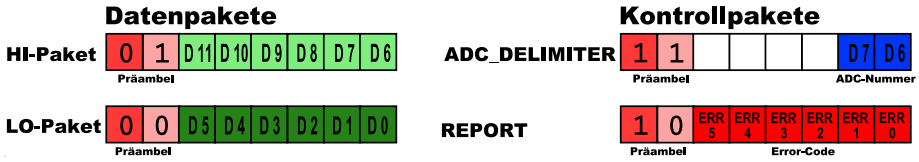


Abbildung 4.15: Aufbau der verschiedenen Pakete

Der genaue Aufbau der einzelnen Pakete ist aus Abbildung 4.15 zu entnehmen. Jedes Paket ist ein Byte lang. Das höchstwertige Bit entscheidet über den Pakettyp. Eine Null zeigt an, daß es sich um ein Datenpaket (High-Byte oder Low-Byte) handelt. Eine Eins kennzeichnet ein Kontrollpaket (ADC_DELIMITER oder REPORT). Das nächste Bit unterscheidet bei Datenpaketen zwischen HI bzw. LO und bei Kontrollpaketen zwischen ADC_DELIMITER und REPORT.

Auslesen und Versenden der Daten

Nachdem alle ADCs mit Umsetzen fertig sind, müssen die Ergebnisse aus den FIFOs ausgelesen werden. Laut [MAX adc, S. 18 fig. 6] geschieht das, indem die CS-Leitung des entsprechenden ADCs auf Low gesetzt wird. Mit jeder SCLK-Flanke wird ein Bit auf die DOUT-Leitung gelegt. Dies entspricht dem normalen SPI-Sende/Empfangszyklus. DIN muß dabei allerdings auf Low bleiben. Das wird erreicht, indem man einfach `0x00` sendet. Die folgende Codesequenz wird reihum für jeden ADC ausgeführt:

```
1  toUSB( ADC_DELIMIT_HEADER | adcNr );
2  for (chNr=0; chNr <= channelADCs[adcNr]; chNr++) {
3    toSPI(adcNr, 0x00); //Hi-Byte einlesen
4    result = SPDR;      // Hi-Byte aus SPI-Data-Register lesen
5    hiFrame = HI_HEADER | (result << 2); // Hi-Paket erstellen
6    toSPI(adcNr, 0x00); // Lo-Byte einlesen
7    result = SPDR;      // Lo-Byte aus SPI-Data-Register lesen
8    hiFrame |= (result & HI_MASK) >> 6; // Hi-Byte vervollständigen
9    toUSB(hiFrame);     // ... und auf USB-Bus schreiben
10   loFrame = LO_HEADER | (result & LO_MASK); // Lo-Byte erstellen
11   toUSB(loFrame);     // ... und auf USB-Bus schreiben
12 }
```

Als Erstes wird ein ADC_DELIMITER Paket mit der Nummer des aktuellen ADCs (`adcNr`) gesendet (Zeile 1). Anschließend werden für jeden Kanal zwei Ergebnis-Bytes aus dem ADC-FIFO gelesen (Zeilen 3 u. 6).

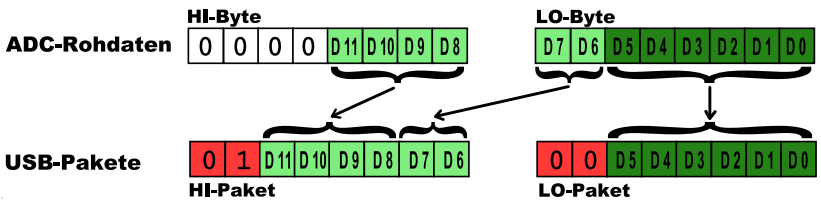


Abbildung 4.16: Konstruktion der HI- und LO-Pakete

Die vier niederwertigsten Bits des ersten Bytes (HI-Byte) und die zwei höchstwertigen des zweiten Bytes (LO-Byte) werden in ein HI-Paket gepackt (Zeilen 5 u. 8) und zum USB-Controller gesendet (Zeile 9). Die restlichen sechs Bits des LO-Bytes werden als LO-Paket

verpackt (Zeile 10) und ebenfalls gesendet (Zeile 11). Abbildung 4.16 veranschaulicht diesen Zusammenhang.

4.5.5 Fehlerbehandlung

Die bisherige Beschreibung des Microcontroller-Programms erfolgte unter der impliziten Annahme, daß alles erwartungsgemäß funktioniert. Tatsächlich gibt es aber mehrere mögliche Fehlerzustände, die den normalen Betrieb stören können. Folgende Fehler werden behandelt:

SPI-Timeout Eine SPI-Übertragung wird nicht innerhalb einer bestimmten Zeit abgeschlossen. Ursachen können eine Unterbrechung in der Bus-Leitung oder ein defekter ADC sein. Es ist auch möglich, daß ein AD-Umsetzer zwar im Programm aktiviert, jedoch nicht angeschlossen wurde.

USB-FIFO voll Der PC holt die Daten nicht bzw. nicht schnell genug vom USB-Controller ab. Als Folge davon läuft dessen Eingangs-FIFO voll und die Annahme weiterer Daten wird abgelehnt.

IRQ-Rate zu hoch Aus irgendwelchen Gründen (SPI-Timeout, voller USB-FIFO, zu hohe Abtastrate) kann die Interrupt-Service Routine des Timers nicht rechtzeitig beendet werden. Die Interrupts werden schneller ausgelöst, als sie bearbeitet werden können.

SPI-Timeout

Eine SPI-Übertragung ist immer dann erfolgreich abgeschlossen, wenn das `SPIF`-Bit im SPI-Service Register gesetzt ist, was durch eine while-Schleife in `toSPI()` geprüft wird (Abschnitt 4.5.3). Geschieht das nicht, würde das Programm theoretisch unendlich lange blockieren. Um dies zu verhindern, wird die Schleife nach `MAX_WAIT` Iterationen verlassen (Zeile 3) und ein ein REPORT-Paket mit dem Fehlercode `ERR_SPI_TIMEOUT` (`0x01`) an den PC gesendet.

```
1 while ( !( SPSR & (1<<SPIF) ) ) {
2     cnt++;
3     if (cnt > MAX_WAIT) {          // Timeout aufgetreten
4         report(ERR_SPI_TIMEOUT);   // Fehlerpaket senden
5         break;                     // Warten abbrechen
6     }
7 }
```

Die Funktion `report()` in Zeile 4 konstruiert das REPORT-Paket mit dem übergebenen Fehlercode und schickt es durch Aufruf von `toUSB()` an den USB-Controller. Die Applikation weiß durch das bereits vorher gesendete ADC_DELIMITER-Paket, bei welchem ADC der Fehler aufgetreten ist und kann entsprechend reagieren.

Voller USB-FIFO

Der USB-Controller signalisiert einen vollen Eingangs-FIFO, indem er die TXE-Leitung auf High setzt (Abschnitt 4.3.2). Soll währenddessen durch den Aufruf von `toUSB()` ein weiteres Byte dort abgelegt werden, muß die Funktion warten, bis wieder Platz im FIFO ist.

Das ist an sich noch nicht problematisch. Schwierig wird es erst, wenn der PC die Daten nicht schnell genug vom USB-Bus liest und bereits neue Bytes geschrieben werden müßten, ohne das `toUSB()` das vorherige Byte im FIFO unterbringen konnte. Es gibt zwei Möglichkeiten, mit dieser Situation umzugehen.

1. Das Programm arbeitet normal weiter, neu eintreffende Bytes werden jedoch verworfen, bis wieder Platz im FIFO ist.
2. Die `toUSB()`-Funktion blockiert so lange, bis wieder neue Daten geschrieben werden können.

Die zweite Möglichkeit ist hier aus mehreren Gründen zu bevorzugen. Der Microcontroller selbst speichert keine Meßwerte. Daher ist es sinnlos, die AD-Umsetzer weiter arbeiten zu lassen, obwohl die anfallenden Daten nicht weitergesendet werden können. Dies wäre sogar schädlich, da durch die auftretenden Paketverluste die Synchronisation verloren geht und Meßwerte nicht mehr korrekt ihren Kanälen zugeordnet werden könnten.

Blockiert das Programm jedoch, kann an exakt der gleichen Stelle wieder angesetzt werden, sobald der FIFO wieder frei ist. Die bis dahin noch nicht gesendeten Daten des gerade aktiven ADCs sind dann zwar veraltet, dies fällt jedoch nicht weiter ins Gewicht, da der PC offensichtlich ohnehin Probleme mit der Verarbeitung der Meßwerte hatte. Spätestens beim nächsten ADC (also nach maximal 16 Meßwerten) sind alle Daten wieder aktuell.

Zu hohe IRQ-Rate

Die Service-Routine des Timer-Interrupts ist nicht unterbrechbar. Wird während ihrer Abarbeitung ein weiterer Interrupt ausgelöst, so wird dieser unmittelbar nach dem Verlassen der ersten Routine behandelt. Das Gerät arbeitet also trotzdem korrekt, auch wenn die Service-Routinen nahtlos aufeinander folgen.

Trotzdem ist es wichtig, die Applikation darüber zu informieren, daß die Meßwerte nicht zu den gewünschten Abtastzeitpunkten sondern erst später aufgenommen wurden. Aus diesem Grund wird ein REPORT-Paket mit dem Fehlercode `0x02` (`ERR_IRQ_RATE_OVERFLOW`) gesendet. Ein solches Paket weist also nicht auf einen konkreten Fehler hin, sondern signalisiert lediglich, daß die Timer-ISR nicht rechtzeitig beendet werden konnte. Dies kann zum Beispiel bedeuten, daß die gewählte Abtastrate zu hoch ist oder das der PC die Daten nicht schnell genug abholt und es daher zu Verzögerungen durch einen vollen USB-FIFO kommt.

Um festzustellen, ob die IRQs rechtzeitig behandelt werden, wird in der Hauptschleife der `main()`-Funktion ein Flag gesetzt.

```
1 loopFlag = 1;
```

In der Service-Routine des Timers wird dieses Flag geprüft (Zeile 1) und anschließend wieder auf Null zurückgesetzt (Zeile 4).

```
1 if ( loopFlag == 0 ) {           // main() noch nicht gelaufen
2   report(ERR_IRQ_RATE_OVERFLOW); // Fehlerpaket senden
3 }
4 loopFlag = 0;                    // reset
```

Ist es beim nächsten Aufruf der ISR immer noch Null, so bedeutet das, daß die `main()`-Routine zwischen zwei Timer-Interrupts nicht ausgeführt werden konnte. Die Interrupts wurden zu schnell hintereinander ausgelöst. In diesem Fall wird ein REPORT-Paket versandt (Zeile 2).

4.5.6 Power-Management

Es wurde bereits in Abschnitt 4.3.3 beschrieben, wie die Energieversorgung der externen Komponenten durch Verändern der PWRCTRL-Leitung ein- bzw. ausgeschaltet wird und das durch Überwachen der PWREN-Leitung der aktuelle Zustand (normal oder suspend) festgestellt werden kann. In diesem Abschnitt soll es nun darum gehen, wie dieses Powermanagement im Microcontroller implementiert ist.

Für die Überwachung der PWREN-Leitung gibt es zwei Möglichkeiten. Entweder kann durch den Pegelwechsel ein IRQ ausgelöst werden, oder die Leitung wird regelmäßig abgefragt (Polling).

Die IRQ-Methode ist im vorliegenden Fall etwas ungünstig, da die Interrupt-Service Routine des Timers normalerweise nicht durch andere Interrupts unterbrochen werden kann. Das hieße, daß kein Wechsel in den Suspend-Modus stattfinden kann, während das Programm beispielsweise auf das Freiwerden des USB-FIFOs wartet.

Deklariert man die Timer-Routine als unterbrechbar, kann sie allerdings an *jeder* beliebigen Stelle unterbrochen werden, was wiederum nicht wünschenswert ist, da so die ADCs in einen undefinierten Zustand versetzt werden könnten. Eine Lösung wäre, die Timer-ISR nur an bestimmten Stellen als unterbrechbar zu kennzeichnen und ansonsten eine Schachtelung der Interrupts zu gestatten. Das ist jedoch unnötig komplex, so daß hier nur mit Polling gearbeitet wird. Da der Shutdown nicht zeitkritisch ist, reicht das vollkommen aus.

Die PWREN-Leitung wird regelmäßig in der Hauptschleife der main()-Routine abgefragt.

```
1 if ( bit_is_set( PIN_USBCTRL, USB_PWR ) ) {
2    shutdown();
3    while ( bit_is_set( PIN_USBCTRL, USB_PWR ) ) {
4       U_WAIT(100);
5    }
6    wakeup();
7 }
```

Sobald sie auf High wechselt (Zeile 1), wird die Funktion shutdown() ausgeführt. Diese sperrt zunächst einmal alle Interrupts, so daß keine weiteren Abtastvorgänge angestoßen werden. Als nächstes werden sämtliche SPI-Datenleitungen auf Low gesetzt, um eine Beschädigung der ADCs zu vermeiden. Erst dann wird die Spannungsversorgung für alle externen Komponenten abgeschaltet, und die Funktion beendet.

Danach wartet der Microcontroller, bis die PWREN-Leitung wieder auf Low wechselt (Zeile 3) und führt daraufhin die wakeup()-Routine aus. Diese schaltet die externe Spannungsversorgung wieder ein, reinitialisiert das SPI-System und sämtliche ADCs und gibt schließlich die Interrupts wieder frei, so daß erneut Meßwerte aufgenommen werden können.

Um sicherzustellen, daß auch dann in den Suspend-Modus gewechselt wird, wenn das Programm sich gerade in der toUSB()-Funktion befindet und auf Platz im USB-FIFO wartet, wird dort ebenfalls die PWREN-Leitung überprüft und gegebenenfalls die Funktion verlassen. Daraufhin wird die oben dargestellte Prüfroutine im Hauptprogramm durchlaufen und ein Shutdown eingeleitet.

Kapitel 5

Die Treibersoftware

Nachdem es in den bisherigen Kapiteln um den Aufbau der Sensoren und des DAQ-Systems ging, soll nun die PC-Seite näher beschrieben werden. Auf dem Rechner, der die Meßdaten entgegennehmen und verarbeiten bzw. visualisieren soll, wird ein Treiber benötigt, der die USB-Daten empfängt, dekodiert und für Anwendungsprogramme verfügbar macht.

5.1 Aufbau der Treiberarchitektur

Gerätetreiber sind unter Linux üblicherweise als Module realisiert, die zur Laufzeit in den Kernel des Systems eingebunden werden – sogenannte Kernelmodule. Für den hier benutzen USB-Controller wird von [FT] bereits ein Kernelmodul unter der GPL[1] zur Verfügung gestellt. Es stellt eine virtuelle, serielle Schnittstelle bereit, über die mit dem Controller kommuniziert werden kann. Nach dem Einbinden des Moduls durch `modprobe ft245` steht eine Device-Datei `/dev/ttyUSB`*x* zur Verfügung, auf die schreibend (Senden von Daten) und lesend (Empfangen von Daten) zugegriffen werden kann. Bei aktuellen Distributionen wie z.B. SUSE-Linux ab Version 9.0 ist das `ft245`-Modul bereits enthalten und wird beim Anstecken des Gerätes automatisch durch den Hotplug-Dämon eingebunden.

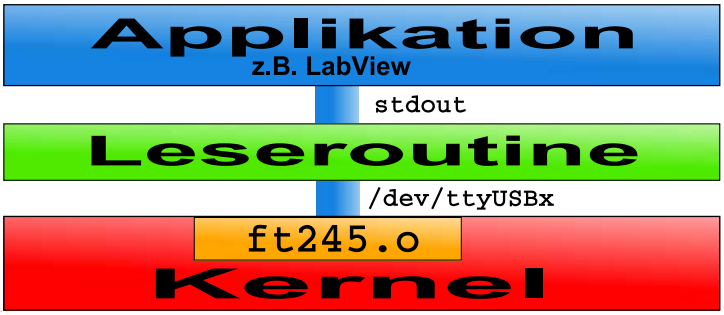


Abbildung 5.1: Die einzelnen Schichten der Treiberarchitektur

Über die serielle Device-Datei können nur die rohen Datenbytes vom USB-Bus ausgelesen werden. Das Übertragungsprotokoll, welches daraus die Meßwerte rekonstruiert, ist als Userspace-Programm realisiert und wird im nächsten Abschnitt genauer beschrieben.

[1] GPL: **G**eneral **P**ublic **L**icence; eine von RICHARD STALLMAN entwickelte Lizenz für freie Software, welche die Einsicht in den Programmquelltext, sowie dessen Modifikation und Weitergabe gestattet. [GPL]

Nachdem die Meßdatensätze durch dieses Programm – die Leseroutine – wieder zusammengesetzt worden sind, können sie über eine Pipe an jedes beliebige Programm weitergeleitet werden, welches dann die Verarbeitung und Visualisierung der Daten übernimmt. Als Beispiel wird in Abschnitt 5.3 die Implementierung einer Anbindung an das populäre LABVIEW System [LV] gezeigt. Der Zusammenhang zwischen den einzelnen Schichten der Architektur ist in Abbildung 5.1 dargestellt.

5.2 Die Leseroutine

Die Leseroutine hat die Aufgabe, den Rohdatenstrom aus `/dev/ttyUSB`*x* zu lesen, aus den HI- und LO-Bytes die Meßwerte zu extrahieren, zusammenzusetzen und schließlich den einzelnen Kanälen zuzuordnen. Anschließend werden die Meßwerte, die ja immer noch als Intergerwerte (Quantisierungsstufen) vorliegen, in reale Spannungen umgerechnet und in einem Array abgelegt.

Dieser Array, der alle zum aktuellen Abtastzeitpunkt gemessenen Spannungen enthält, wird anschließend über die Standardausgabe (`stdout`) ausgegeben. Dies geschieht entweder in binärer Form, so daß eine Applikation wie LABVIEW die Daten leicht weiterverarbeiten kann, oder als Textrepräsentation. Die textuelle Darstellung ermöglicht es, sich die Daten direkt anzusehen oder in eine Datei umzuleiten, und anschließend mit Programmen wie GNUPLOT grafisch darzustellen. Gesteuert wird die Art der Ausgabe durch die Optionen `-b` (binäre Ausgabe) bzw. `-t` (textuelle Darstellung).

5.2.1 Realisierung

Zunächst wird das Gerät `/dev/ttyUSB`*x* geöffnet. Das *x* steht dabei für die Nummer des USB-Gerätes. Das zuerst angeschlossene Gerät erhält die Nummer 0, das nächste die 1 und so weiter. Getreu der UNIX-Philosophie „Alles ist eine Datei.“, kann man das Gerät wie eine normale Datei öffnen. Anschließend werden mit `tcsetattr()` einige Attribute der seriellen Schnittstelle gesetzt. Einstellungen zur Baudrate, die bei realen seriellen Schnittstellen nötig sind, werden vom `ft245`-Kernelmodul jedoch ignoriert. Das Gerät arbeitet immer mit der maximal möglichen Geschwindigkeit.

Nachdem die Schnittstelle geöffnet und initialisiert ist, wird die Funktion `readValues()` aufgerufen, welche die vom USB-Controller gelieferten Rohdaten aus der Gerätedatei ausliest, verarbeitet und auf `stdout` wieder ausgibt.

Für das Zwischenspeichern der Meßwerte wird als erstes der oben bereits erwähnte Ergebnisarray vom Typ `float` deklariert. Dort ist für jeden der maximal acht ADCs (`MAX_ADC`) ein Block für 16 Meßwerte (`MAX_CH`) reserviert. Die Kanäle 0 bis 15 des ersten ADCs belegen die Elemente 0 bis 15 des Arrays, die Kanäle 0 bis 15 des zweiten ADCs dagegen die Elemente 16 bis 31, und so weiter. Auf diese Weise ist eine eindeutige Zuordnung von Meßwerten zu ihren Kanälen bzw. ADCs möglich. Nicht benutzte Kanäle werden einfach frei gelassen. Alle Elemente werden mit NaN[2] initialisiert:

```
1 float voltages[MAX_ADC*MAX_CH];
2 for (i=0; i < (MAX_ADC*MAX_CH); i++) {
3    voltages[i] = NAN; // ungültig bis zum Beweis des Gegenteils :)
4 }
```

Zunächst sind also alle Werte ungültig. Sie werden dann nach und nach mit empfangenen Meßwerten überschrieben. Sind nach Abschluß des Scan-Vorganges noch NaN-Werte im

[2] NaN: **n**ot **a** **n**umber; Spezieller Zahlenwert bei IEEE 754-konformen Fließkommazahlen

Array, bedeutet das, daß an dieser Stelle kein Meßwert zur Verfügung steht. Das kann zwei mögliche Ursachen haben. Entweder sind Pakete verloren gegangen, so daß Meßwerte fehlen, oder es wurde für diesen Kanal keine Messung durchgeführt, weil er nicht aktiviert oder der dazugehörige ADC gar nicht angeschlossen war.

Nach der Initialisierung von `voltages[]` tritt die Funktion in einer Endlosschleife ein. Hier werden Datenpakete über die geöffnete Schnittstelle eingelesen und nacheinander ausgewertet.

Besitzt ein Pakete einen REPORT_HEADER, handelt es sich um eine Fehlermeldung bzw. um eine Bestätigung. In diesem Fall werden entsprechende Meldungen ausgegeben, um der Applikation zu ermöglichen, auf die Fehler zu reagieren. Alle anderen Pakete werden an den in Abbildung 5.2 gezeigten endlichen Zustandsautomaten[3] weitergereicht.

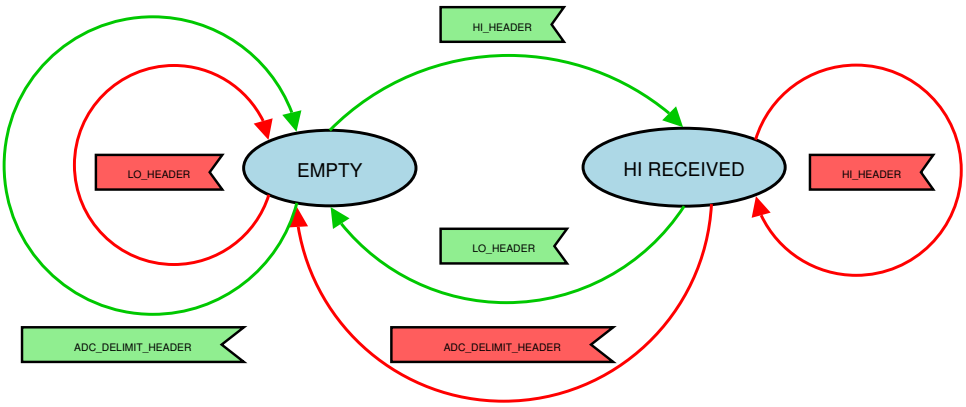


Abbildung 5.2: Zustandsmodell der Leseroutine

Die grünen Transitionen stellen dabei die normale Verarbeitung ohne Übertragungsfehler dar. Zunächst befindet sich das System im Zustand EMPTY. Wird ein Paket mit einem HI_HEADER (das High-Byte eines Meßwertes) empfangen, so werden die sechs enthaltenen Datenbits gespeichert und das System wechselt in den Zustand HI RECEIVED.
Jetzt wird ein Paket mit einem LO_HEADER (das Low-Byte eines Meßwertes) erwartet. Trifft dieses ein, wird der Meßwert komplettiert, in eine Spannung (Float) umgerechnet und anschließend an die richtige Stelle im Ergebnisarray eingetragen. Das System befindet sich nun wieder im Zustand EMPTY. Das High-Byte des nächsten Meßwertes kann empfangen werden.
Sind alle Kanalmeßwerte eines ADCs empfangen worden, folgt ein Paket mit dem ADC_-DELIMIT-Header. Es zeigt an durch die enthaltene ADC-Nummer an, zu welchem Umsetzer die nun folgenden Werte gehören. Die Leseroutine muß keinerlei Information darüber haben, in welcher Reihenfolge die ADCs abgefragt werden. Sie muß noch nicht einmal wissen, wieviele Kanäle pro ADC aktiv sind, solange diese in aufsteigender Reihenfolge und ohne Lücken angeordnet sind. Alle Meßwerte werden einfach nacheinander ab der ADC-Startposition (Kanal 0) in den Array geschrieben. Trifft ein ADC_DELIMITER-Paket ein, werden alle für den vorherigen ADC noch verbliebenen Kanäle übersprungen und die Aufzeichnung beginnt bei der Startposition des neuen ADCs.

Geht alles glatt, springt der Zustandsautomat immer zwischen den beiden Zuständen EMPTY und HI RECEIVED hin und her. Durch Paketverluste ist es jedoch möglich, daß Pakete in der falschen Reihenfolge d.h. zu unerwarteten Zeitpunkten eintreffen. Das sind die rot dargestellten Transitionen. Wird beispielsweise ein LO-Paket empfangen, obwohl sich das

[3] **F**inite **S**tate **M**aschine (FSM)

System ist noch im Zustand EMPTY befindet, wird davon ausgegangen, daß das dazugehörige HI verlorengegangen ist. Der Meßwert ist damit unvollständig und wird verworfen. Das entsprechende Feld in `voltages[]` bleibt leer (NaN).
Wird umgekehrt im Zustand HI RECEIVED ein weiteres HI empfangen, ist wohl das LO-Paket des letzten Meßwertes abhanden gekommen. Das gerade empfangene HI gehört also bereits zum nächsten Wert. Das System bleibt im Zustand HI RECEIVED und setzt lediglich den Kanalzähler um Eins weiter. Anschließend wird das Empfangene Paket normal verarbeitet. Sollte in diesem Zustand ein ADC_DELIMITER Paket eintreffen wird ebenfalls davon ausgegangen, daß das eigentlich erwartete LO abhanden gekommen ist. Sämtliche noch nicht empfangenen Meßwerte des alten ADCs werden übersprungen und das System beginnt wieder im Zustand EMPTY, bereit das High-Byte vom ersten Meßwert des neuen ADCs zu empfangen.

5.3 Anbindung an LabView

Um aus den empfangenen Meßwerten für Beschleunigung und Winkelgeschwindigkeit verwertbare Informationen über die Position der einzelnen Fingerglieder zu erhalten – was ja das eigentliche Ziel des Sensorsystems ist – müssen gelieferten Daten von Applikationen eingelesen und dort weiterverarbeitet werden können.

Als Applikation wurde hier, aufgrund der recht großen Verbreitung, das LABVIEW-System [LV] gewählt. Damit kann man durch Blockschaltbilder beliebige Programmabläufe festlegen und die Ergebnisse auf einer Art „virtuellen Instrumentenkonsole“ darstellen.

Das in diesem Abschnitt beschriebene Programm ist dabei jedoch nur als Beispiel gedacht, und soll die grundsätzliche Realisierbarkeit einer solchen LabView-Anbindung demonstrieren. Im Moment werden lediglich die Meßwerte der einzelnen Kanäle eingelesen und auf einer Art Oszilloskop grafisch dargestellt. Die weitergehende Verarbeitung der Daten ist nicht Thema dieser Arbeit und soll hier nicht weiter erläutert werden.

5.3.1 Realisierung

Jedes LabView-Programm besteht aus zwei Komponenten, dem Blockschaltbild (Abbildung 5.3), welches den Programmablauf vorgibt, und dem Instrumentenpanel (Abbildung 5.4), das die Anordnung der Bedienelemente und Anzeigen festlegt.

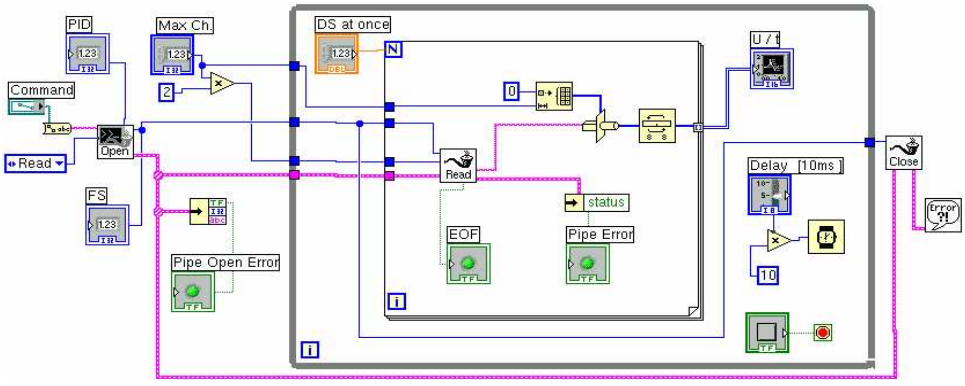


Abbildung 5.3: LabView Blockschaltbild

Zunächst wird durch das *Open*-Symbol eine Instanz der Leseroutine als separater Prozess gestartet. das Kommando für den Aufruf kann über das *Command*-Feld im Instrumentenpanel eingestellt werden. Der angehängte Kommandozeilen Parameter `-b` sorgt dafür, daß die Ausgabe der Routine binär erfolgt. Die Daten werden als Arrays von Short-Integer Werten (Spannung in Millivolt) nach `stdout` geschrieben. Jeder Meßwert belegt 2Byte, ein Array mit allen 128 Werten eines Abtastzeitpunktes ist demnach 256Byte groß.

Das *Reader*-Symbol liest die Daten aus der Standardausgabe der Leseroutine in das LABVIEW-Programm ein. Leider liegen sie anschließen nur als String vor und müssen erst wieder durch einen Cast-Operator in Short-Integer Arrays umgewandelt werden.

Um die Applikation zu beschleunigen, werden immer erst mehrere Datensätze (Arrays) eingelesen und formatiert, bevor diese dann im Ganzen an die Visualisierungskomponente (das *Graph*-Symbol) übergeben und von ihr dargestellt werden. Wieviele Datensätze auf einmal eingelesen werden sollen, kann man im Feld *DS at once* einstellen. In Tests hat sich ein Wert von 16 als optimal herausgestellt, wahrscheinlich weil der Linux-Kernel intern mit 4KB großen Datenblöcken arbeitet und 16 Arrays mit je 128 Meßwerten genau 4096Byte (4KB) ergeben.

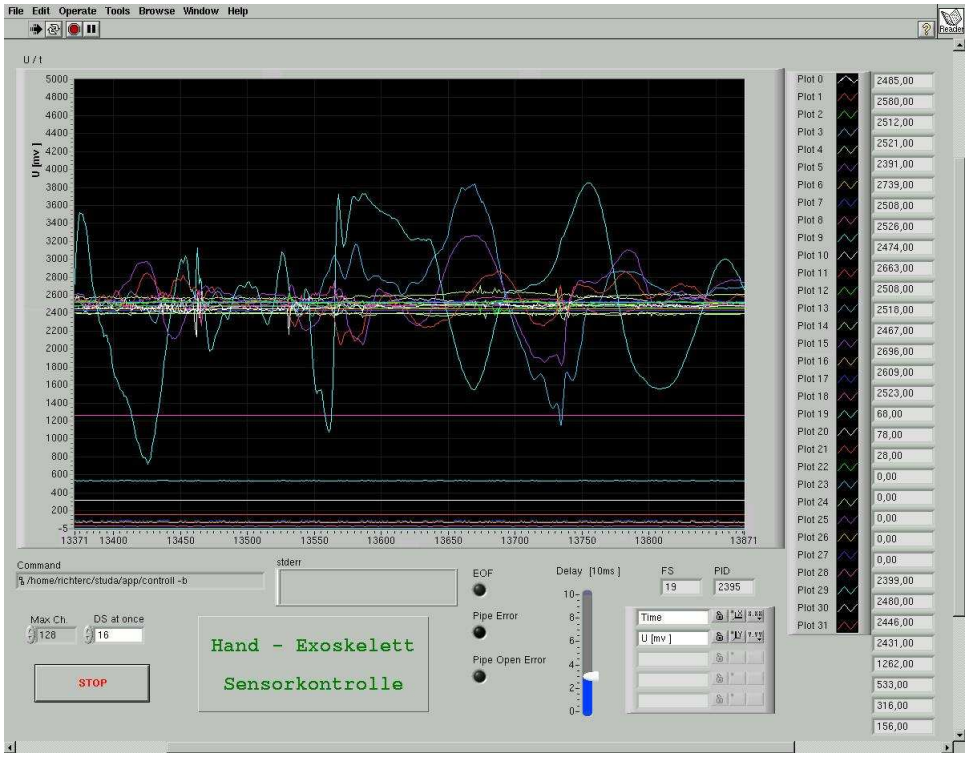


Abbildung 5.4: LabView Instrumentenpanel für die Visualisierung

Über den blauen Delay-Schieberegler läßt sich einstellen, wie lange die Applikation zwischen den Aktualisierungen der Graph-Komponente wartet. Je niedriger dieser Wert ist, desto geringer ist die Verzögerung, mit der die Daten dargestellt werden. Da das Neuzeichnen der Anzeige allerdings eine sehr rechenaufwändige Operation ist, steigt die Prozessorlast bei geringen Delay-Werten stark an. Auf dem benutzten Laborrechner – einem Pentium III mit 450MHz Taktfrequenz – lag die untere Grenze bei etwa 30ms.

Kapitel 6

Mögliche Erweiterungen

Das im Rahmen dieser Arbeit entstandene Sensor-Datenerfassungssystem ist auf einen bestimmten Zweck zugeschnitten – die Bewegungen der menschlichen Hand zu messen und die erfaßten Daten an einen Rechner zu übermitteln. Trotzdem ist der Teil des Systems, der die Meßwerte digitalisiert und transportiert – die DAQ-Komponente[1] – so flexibel gehalten, daß über diesen konkreten Einsatzzweck hinaus eine Vielzahl von anderen Anwendungen denkbar sind. Das Gerät kann überall dort eingesetzt werden, wo viele Signale simultan verarbeitet werden müssen und wo es gleichzeitig auf eine geringe Größe der Komponenten ankommt.

In diesem Kapitel sind nun verschiedene Ideen zur Verbesserung und Erweiterung des DAQ-Teils zusammengetragen. Für die ursprüngliche Kernaufgabe des Systems, die „*Sensor-Datenerfassung für eine Exoskelett-Hand*“, sind diese nicht unbedingt notwendig, sie erweitern das System jedoch um Fähigkeiten, die für andere Einsatzzwecke nützlich sein könnten.

6.1 Kaskadieren mehrerer Geräte

Das DAQ-System in seiner jetzigen Form kann bis zu 8 A/D-Umsetzer ansteuern und damit bis zu 128 Kanäle gleichzeitig digitalisieren. Genügt das nicht, kann man mehrere Geräte parallel einsetzen. Die meisten Rechner verfügen heutzutage über mehrere USB-Anschlüsse. Sollte das nicht der Fall sein, kann man USB-Hubs mit einer eigenen Stromversorgung benutzen, die an jedem Eingang die maximal möglichen 500mA zur Verfügung stellen.

Für einen derartigen Parallelbetrieb ist es allerdings nötig, die Abtastzeitpunkte der einzelnen Geräte exakt aufeinander abzustimmen. Eine solche Kaskade hätte dann einen Master, der den Takt wie bisher timergesteuert vorgibt, und mehrere Slaves, die den Takt des Masters zum Abtasten benutzen.

Die Realisierung gestaltet sich recht unkompliziert. In die Timer-ISR des Masters wird eine zusätzliche Instruktion eingefügt, die zu Beginn der Routine einen Impuls auf einen der freien IO-Pins (beispielsweise PB2) schickt. Dieser Pin ist mit dem noch freien IRQ-Eingang (INT2 auf PB2) aller Slave-Geräte verbunden. Sobald also der Master einen Abtastvorgang startet, wird bei allen Slaves ein externer Interrupt (INT2) ausgelöst. Sorgt man nun noch dafür, daß bei den Slaves die Abtastroutine nicht mehr durch den Timer-Interrupt, sondern durch den extern getriggerten INT2 ausgelöst wird, laufen alle Geräte synchron.

[1] gemeint sind damit die ADC-Platinen, die Controller-Platine, das Microcontrollerprogramm und die PC-seitigen Treiber

Wenn man einen weiteren, noch freien Pin (z.B. PB0) als Eingang konfiguriert, läßt sich die Master/Slave- Umschaltung direkt am Gerät durch einen Jumper vornehmen. Zieht man den Eingang über einen Widerstand z.B. auf +5V, so agiert das Gerät als Master. GND bedeutet dann: Slave.

6.2 Dynamische Konfiguration

Ein sehr interessantes, allerdings auch etwas aufwändigeres Feature ist die dynamische Konfiguration des Gerätes. Im Moment ist es noch so, daß jede Änderung der Beschaltung (Welche ADCs sind angeschlossen? Wieviele Kanäle werden an welchem ADC benutzt?) Änderungen direkt im Steuerprogramm nötig machen. Das wiederum erfordert eine Neuprogrammierung des Microcontrollers bei jeder Konfigurationsänderung.

Für die Exoskelett-Hand ist das kein Problem, da dort die Konfiguration ein einziges Mal eingestellt wird und sich danach nicht mehr ändert. Möchte man das System allerdings als universelles Meßgerät benutzen, ist diese Einschränkung sehr störend.

Die grundlegenden Voraussetzungen für eine Konfigurationsänderung im laufenden Betrieb sind schon gegeben.

- Der FT245BM ist prinzipiell in der Lage, Datenbytes nicht nur zu senden, sondern auch vom PC zu empfangen. Die Funktion `fromUSB()` existiert bereits – wenn auch nur als Rumpf.
- Die Daten über Kanalanzahl und Status der einzelnen ADCs sind nicht statisch als Präprozessorkonstanten, sondern dynamisch in Variablen (siehe Abschnitt 4.5.2 auf Seite 44) abgelegt und können so während des Programmablaufes modifiziert werden.
- Die REPORT-Pakete können einen beliebigen Fehlercode zwischen 0 und 63 mitschicken (siehe 4.5.4 auf Seite 47). Einer dieser Codes kann als Bestätigung für die vom PC empfangenen Konfigurationsdaten gesendet werden. Auf diese Weise wüßte die Applikation dort, ab wann die Meßdaten vom Gerät der eingestellten Konfiguration entsprechen.

Änderungen an der Hardware des Gerätes wären also nicht nötig. Allein ein erweitertes Microcontroller-Steuerungsprogramm und ein angepaßter Treiber würden genügen, um diese Funktionalität zu ergänzen und damit den Umgang mit dem DAQ-System sehr viel komfortabler zu gestalten.

6.3 Weitere „Peripheriegeräte“

Da es sich bei dem von der Controller-Platine benutzen Bus-System um einen normalen SPI-Bus handelt, lassen sich dort neben ADCs auch noch andere Geräte mit einer SPI-Schnittstelle anschließen. Das können beispielsweise Temperatursensoren wie der ADT-7301 von [AD] oder Digital/Analog-Umsetzer (MAX5234 von [MAX]) oder sogar LED-Displaytreiber, wie der MAX7221 sein.

Natürlich sind derartige Peripheriegeräte im Design nicht direkt vorgesehen, so daß einige Änderungen in der Steuersoftware nötig wären. Trotzdem ist es eine gute Möglichkeit, das System mit zusätzlichen Fähigkeiten auszustatten.

Literaturverzeichnis

[AD] ANALOG DEVICES INC., Homepage http://www.analog.com/

[AD Acc] ANALOG DEVICES, Datenblatt für ADXL210E
http://www.analog.com/UploadedFiles/Data_Sheets/473836157ADXL210E_0.pdf

[AD Gyro] ANALOG DEVICES, Datenblatt für ADXRS300
http://www.analog.com/UploadedFiles/Data_Sheets/29973351ADXRS300_a.pdf

[Mur ENC] MUTARA MANUFACTORING CO., Daten der ENC-Sensorserie
http://www.murata.com/ (Stichwort ENC)

[AD AccEB] ANALOG DEVICES, Datenblatt für ADXL210EB (Evaluation-Board)
http://www.analog.com/UploadedFiles/Data_Sheets/70885338ADXL202_10_b.pdf

[AD GyroEB] ANALOG DEVICES, Datenblatt für ADXRS300 (Evaluation-Board)
http://www.analog.com/UploadedFiles/Data_Sheets/732884779ADXRS300_b.pdf

[AD BGA] ANALOG DEVICES, Beschreibung des 32-lead CSPBGA-Packages
http://www.analog.com/UploadedFiles/Packages/501663966232134483BC32_adxrs.pdf

[Motorola] MOTOROLA INC., '*Plastic Ball Grid Array (PBGA)*', Dokument AN1231
http://www.analysistech.com/PDF_Files/PBGAExample.pdf

[Maxon] KENNETH MAXON, '*Have you seen my new soldering Iron?*', Anleitung zur manuellen Reflow-Bestückung
http://www.seattlerobotics.org/encoder/200006/oven_art.htm

[MAX] MAXIM DALLAS SEMICONDUCTORS, Homepage
http://www.maxim-ic.com/

[TI] TEXAS INSTRUMENTS, Homepage http://www.ti.com/

[INF] INFINEON, Homepage http://www.infineon.de/

[NSC] NATIONAL SEMICONDUCTORS, Homepage
http://www.national.com/

[AD adc] ANALOG DEVICES, Datenblatt für AD7490
http://www.analog.com/UploadedFiles/Data_Sheets/400753325AD7490_a.pdf

[MAX adc] MAXIM DALLAS SEMICONDUCTORS, Datenblatt für MAX1230
http://pdfserv.maxim-ic.com/en/ds/MAX1226-MAX1230.pdf

[MAX reg] MAXIM DALLAS SEMICONDUCTORS, Datenblatt für MAX8877
http://pdfserv.maxim-ic.com/en/ds/MAX8877-MAX8878.pdf

[USB] USB IMPLEMENTERS FORUM INC., '*Universal Serial Bus specification*', Revision 2.0
http://www.usb.org/developers/docs/usb_20.zip

[FT] FUTURE TECHNOLOGY DEVICES INTL. LIMITED, Homepage
http://www.ftdichip.com

[FT usb] FUTURE TECHNOLOGY DEVICES INTL. LIMITED, Datenblatt des FT245BM, Revision 1.4
http://www.ftdichip.com/Documents/ds245b14.pdf

[DLP] DLP-DESIGN, Homepage http://www.dlpdesign.com/

[DLP evb] DLP-DESIGN, Datenblatt des DLP-USB245M, Revision 1.2
http://www.dlpdesign.com/usb/dlp-usb245m12.pdf

[ATM] ATMEL CORP., Homepage http://www.atmel.com/

[ATM uC] ATMEL CORP., Datenblatt für ATmega32 (komplett)
http://www.atmel.com/dyn/resources/prod_documents/doc2503.pdf

[SPI] MARTIN SCHWERDTFEGER, '*SPI - Serial Peripheral Interface*'
http://www.mct.net/faq/spi.html

[NSC reg] NATIONAL SEMICONDUCTORS, Datenblatt des LM2940
http://www.national.com/ds.cgi/LM/LM2940.pdf

[GCC] GNU COMPILER COLLECTION, Homepage http://gcc.gnu.org/

[AVR utils] GNU BINUTILS, Homepage http://sources.redhat.com/binutils/

[AVR lib] AVR LIBC, Homepage http://www.nongnu.org/avr-libc/

[AVR dude] AVR-DUDE, Homepage http://www.nongnu.org/avrdude/

[AVR inst] AVR LIBC, '*Installing the GNU Tool Chain*' http://www.nongnu.org/avr-libc/user-manual/install_tools.html

[WinAVR] WINAVR-PROJEKT, Homepage http://winavr.sourceforge.net/

[ISP] Bauanleitung ISP-Dongle http://www.ch-r.de/projects/isp

[ISP einf] AVR-DUDE, '*The simplest possible programmer*', Anleitung für den Bau eines extrem einfachen ISP-Gerätes
http://www.bsdhome.com/avrdude/

[GPL] FREE SOFTWARE FOUNDATION, '*General Public License*'
http://www.gnu.org/copyleft/gpl.html

[LV] NATIONAL INSTRUMENTS, LabView
http://www.ni.com/german/labview

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

Anhang

A.1 Sensorplatine

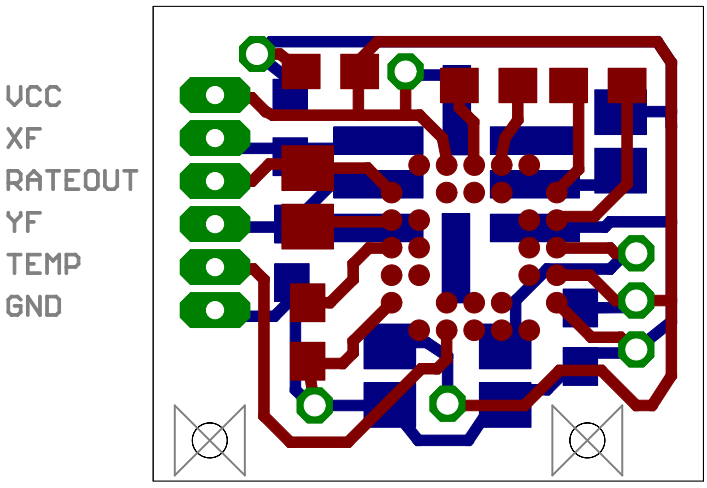
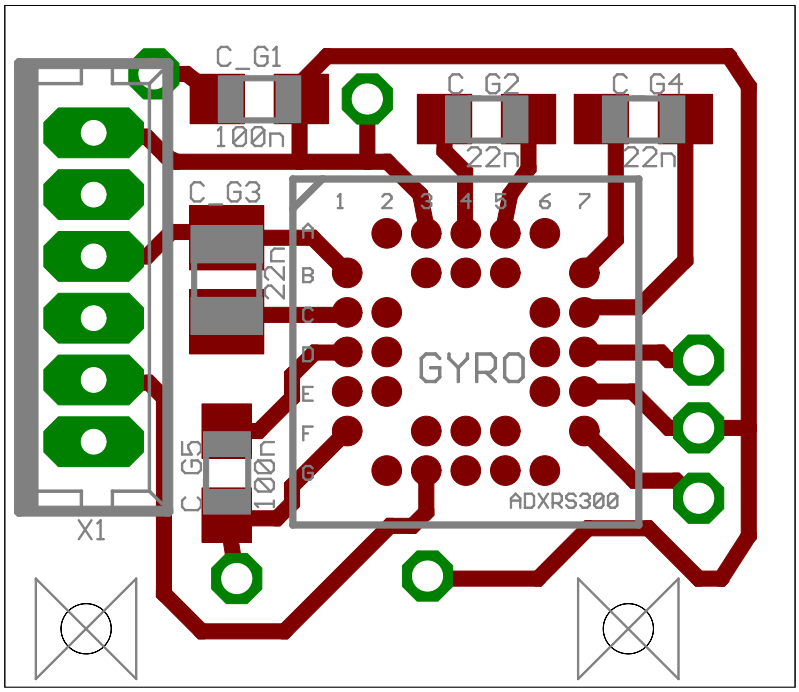


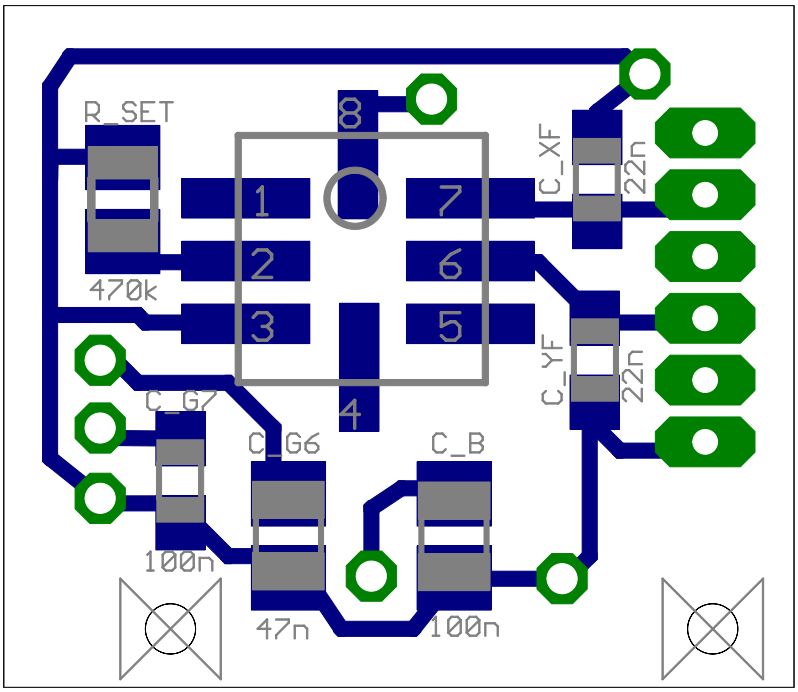
Abbildung A.1: Leiterbahnführung der Sensorplatine

Part	Value	Package	Library
ACCEL	ADXL210E-LCC8	LCC-8-IPC	adxl
C1-2	22n	C0603	rcl
C3-4	22n	C0603	rcl
C5	100n	C0805	rcl
C_G1	100n	C0603	rcl
C_G2	22n	C0603	rcl
C_G3	22n	C0603	rcl
C_G4	22n	C0603	rcl
C_G5	100n	C0603	rcl
C_G6	47n	C0805	rcl
C_G7	100n	C0603	rcl
GYRO	ADXRS300-CSPBGA32	CSPBGA32	adxr
R_SET	470k	R0805	rcl
X1		53047-06	con-molex

Abbildung A.2: Stückliste Sensor-Platine



(a) Bestückung Top-Layer



(b) Bestückung Bottom-Layer

Abbildung A.3: Layout der Sensorplatine

A.2 ADC-Platine

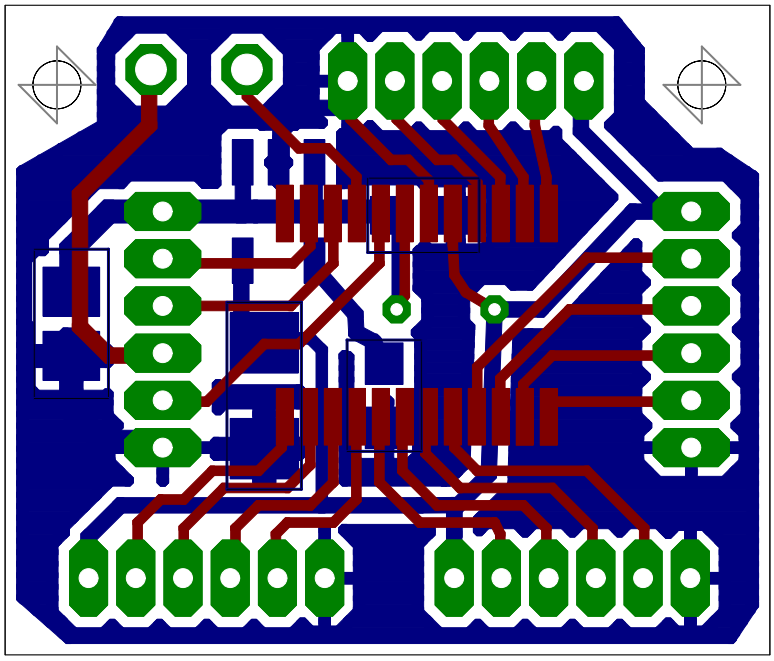
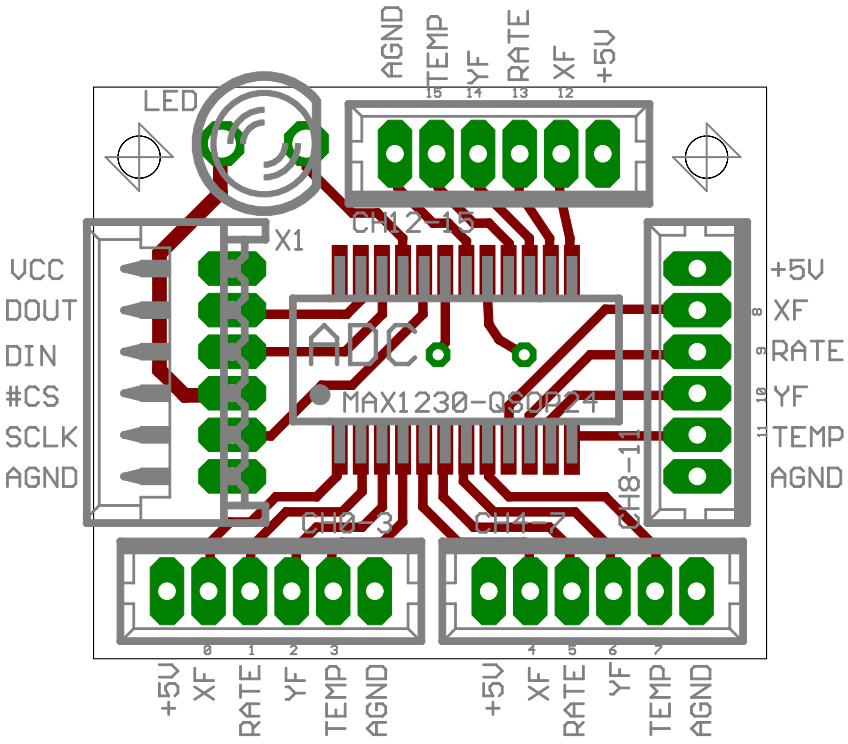


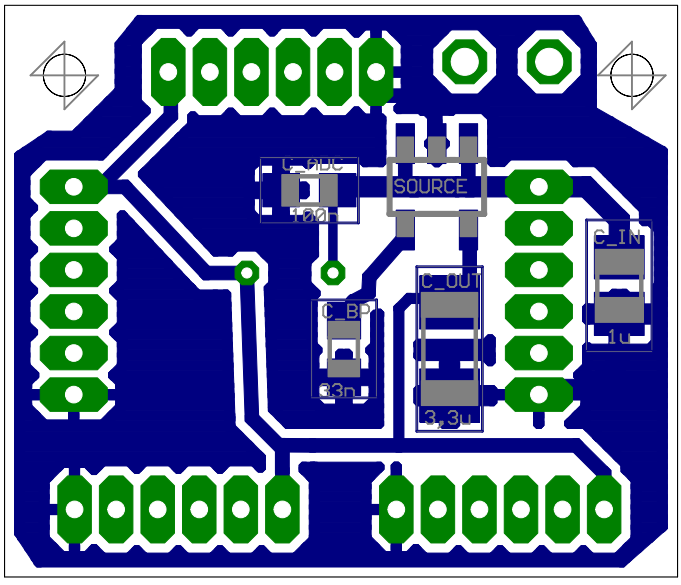
Abbildung A.4: Leiterbahnführung der ADC-Platine

Part	Value	Package	Library
ADC	MAX1230-QSOP24	QSOP24	max1230
CH0-3		53047-06	con-molex
CH4-7		53047-06	con-molex
CH8-11		53047-06	con-molex
CH12-15		53047-06	con-molex
C_ADC	100n	C0603	rcl
C_BP	33n	C0603	rcl
C_IN	1u	C0805	rcl
C_OUT	3,3u	C1206	rcl
LED		LED3MM	led
SOURCE		SOT23-5L	max8877
X1		53048-06	con-molex

Abbildung A.5: Stückliste ADC-Platine



(a) Bestückung Top-Layer



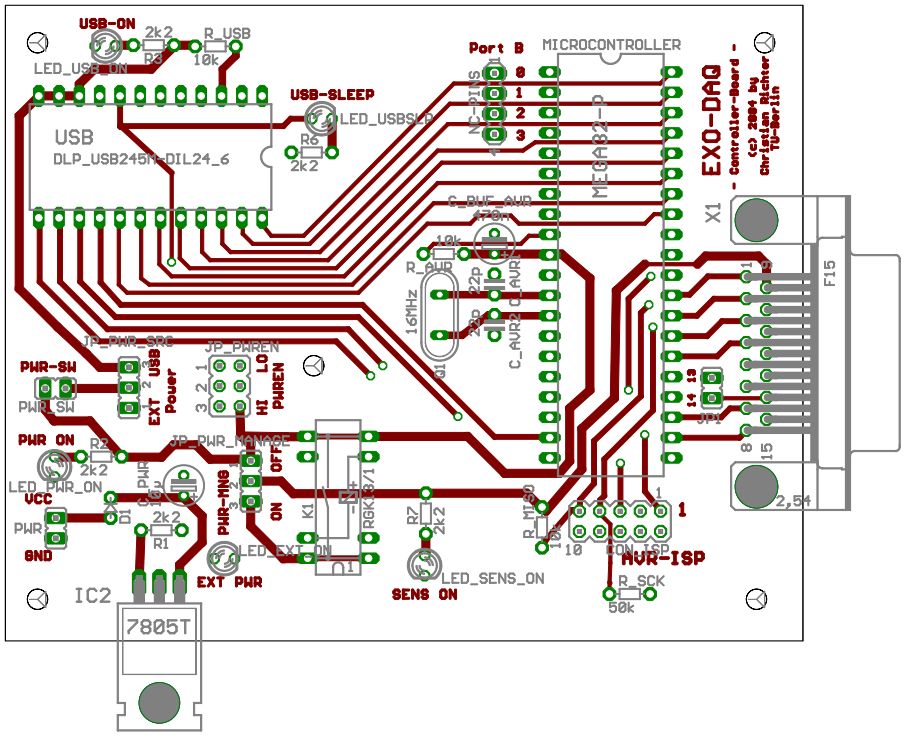
(b) Bestückung Bottom-Layer

Abbildung A.6: Layout ADC-Platine

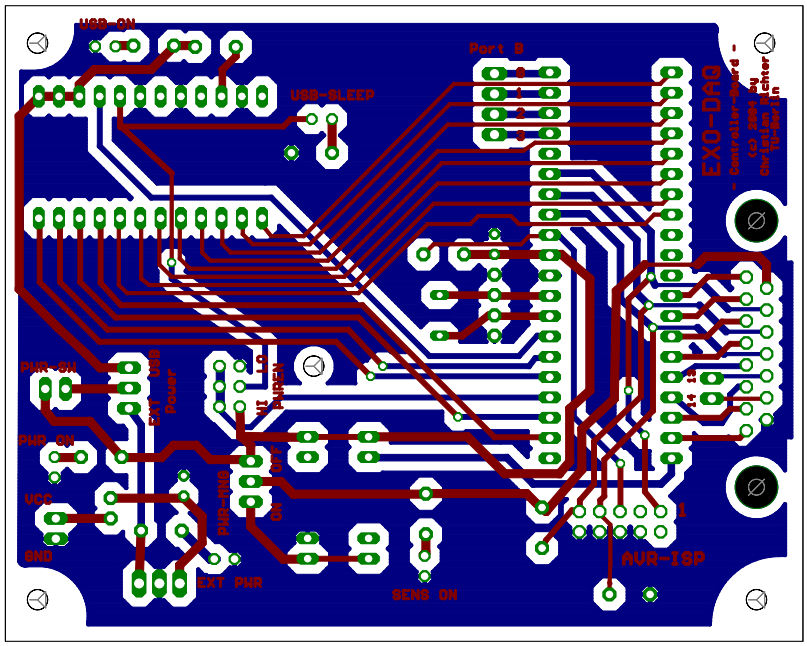
A.3 Controller-Platine

Part	Value	Package	Library
CON_ISP		MA05-2	con-lstb
C_AVR1	22p	C-2,5	discrete
C_AVR2	22p	C-2,5	discrete
C_BUF_AVR	470n	ES-2,5L	discrete
C_PWR	10μ	ES-2,5L	discrete
D1		D-2,5	discrete
IC2	7805T	TO220H	linear
JP1		JP1	jumper
JP_PWREN		JP3Q	jumper
JP_PWR_MANAGE		JP2	jumper
JP_PWR_SRC		JP2	jumper
K1	RGK13/1	DIL-14/8	relais
LED_EXT_ON		LED3MM	led
LED_PWR_ON		LED3MM	led
LED_SENS_ON		LED3MM	led
LED_USBSLP		LED3MM	led
LED_USB_ON		LED3MM	led
MICROCONTROLLER	MEGA32-P	DIL40	avr
NC-PINS		MA04-1	con-lstb
PWR		JP1	jumper
PWR_SW		JP1	jumper
Q1	16MHz	QS	special
R1	2k2	R-5	discrete
R2	2k2	R-5	discrete
R3	2k2	R-5	discrete
R6	2k2	R-5	discrete
R7	2k2	R-5	discrete
R_AVR	10k	R-5	discrete
R_MISO	10k	R-5	discrete
R_SCK	50k	R-5	discrete
R_USB	10k	R-5	discrete
USB	DLP_USB245M-DIL24_6	DIL24-6	dlp-us
X1		F15H	con-subd

Abbildung A.7: Stückliste Controller-Platine



(a) Bestücklungsseite (Top-Layer)



(b) Leiterbahnführung

Abbildung A.8: Layout der Controllerplatine

A.4 Microcontroller-Programm

```
/**
 * ------------------------------------------------
 *   EXOSENS - Microcontroller Steuerungsprogramm
 *
 * Programm fuer die Ansteuerung mehrerer MAX1230 und
 * die USB-Übetragung der empfangenen Daten zum PC
 * ------------------------------------------------
 *
 * mcprog.c   -   Hauptprogramm
 *
 * @author Christian Richter
 * @version 1.0
*/

#include "mcprog.h"  /* Projekt Headerdatei */


SIGNAL(SIG_OUTPUT_COMPARE1A) {
   unsigned char adcNr; // momentan angesprochener ADC (Nr. der CS-Leitung)
   unsigned char chNr;  // aktuell bearbeiteter Kanal
   byte result, hiFrame, loFrame;

   // ADCs Samplebefehl geben
   for (adcNr=0; adcNr<8; adcNr++) { // alle 8 möglichen ADCs durchgehen
      if ( (activeADCs & _BV(adcNr)) != 0x00 ) { // aktueller ADC aktiv?
         toSPI(adcNr, ADC_CNV_REG | (channelADCs[adcNr] << 3) ); //TEST, sonst wieder 0x82
      }
   }

   // Prüfen ob IRQs zu dicht aufeinander folgen
   if ( loopFlag == 0 ) { // Mainloop-Code noch nicht gelaufen => IRQ-Rate zu hoch
      report(ERR_IRQ_RATE_OVERFLOW);
   }
   loopFlag = 0;  // resetten

   // conversion abwarten
   for (int i=0;i<7200;i++); // kann bei mehreren ADCs noch irgendwie verringert werden !

   // Ergebnisse abfragen
   for (adcNr=0; adcNr<8; adcNr++) { // alle 8 möglichen ADCs durchgehen
      if ( (activeADCs & _BV(adcNr)) != 0x00 ) { // aktueller ADC aktiv?

         // ADC-Delimiter Frame senden
         toUSB( ADC_DELIMIT_HEADER | adcNr );

         // ADC-FIFO auslesen, Ergebnisse verpacken und senden
         for (chNr=0; chNr <= channelADCs[adcNr]; chNr++) {
            toSPI(adcNr, 0x00);                 // 0x00 => MOSI=Low und SCLK geht => Hi-Byte einlesen
            result = SPDR;                      // Hi-Byte aus SPI-Data-Register lesen
            hiFrame = HI_HEADER | (result << 2);// Hi-Frame erstellen: 01hhhh00 = 01000000 OR (0000hhhh << 2)
            toSPI(adcNr, 0x00);                 // 0x00 => MOSI=Low und SCLK geht => Lo-Byte einlesen
            result = SPDR;                      // Lo-Byte aus SPI-Data-Register lesen
            hiFrame |= (result & HI_MASK) >> 6; // Hi-Frame vervollständigen: 01hhhh00 |= (hllllll & 11000000) >> 6)
            toUSB(hiFrame);                     // ... und auf USB-Bus schreiben
            loFrame = LO_HEADER | (result & LO_MASK); // Lo-Frame erstellen: 00xxxxxx | (hllllll & 00111111)
            toUSB(loFrame);                     // ... und auf USB-Bus schreiben
         }
      }
   }
}


SIGNAL(SIG_INTERRUPT0) {

   // Für spätere Implementierung der "Dynamischen Konfiguration

}


void usbInit() {
   // USB Ports initialisieren
   DDR_USBDAT = 0xff;       // alle USB-Datenleitungen als Ausgang
   PORT_USBDAT = 0x00;      // und auf Low
   DDR_USBCTRL  = _BV(USB_SND) | _BV(USB_WR) | _BV(USB_RD) | _BV(USB_RST) | _BV(PWRCTRL); // SND, WR, RD, RST, PWRCTRL als Ausg
ang (1), Rest als Eingang (0)
   PORT_USBCTRL = _BV(USB_SND) | _BV(USB_WR) | _BV(USB_RD) | _BV(USB_RST); // ... und auf High (bis auf PWRCTRL), Rest Low
   while ( bit_is_set( PIN_USBCTRL, USB_PWR ) ) { // Warten bis Controller am Bus angemeldet (PWREN low)
      U_WAIT(100);
   }

}

void spiInit() {
   byte status;

   // SPI Ports initialisieren
   DDRB = _BV(PB7) | _BV(PB5) | _BV(PB4); // setze SCK, MOSI und SS als Ausgang (1)
                                          // (wenn SS nicht OUT wäre, müßte er HIGH liegen um Master-Betrieb nicht zu stören!!!
)
   DDRB &= ~_BV(PB6);                     // setze MISO als Eingang (0)
   PORTB = _BV(PB7);                      // SCK high
   DDR_CS = 0xff ;                        // setze alle CS-Leitungen als Ausgang
   PORT_CS = 0xff;                        // alle CS-Leitungen high

   // Hardware SPI initialisieren (SPI Controll-Register (SPCR))
   SPCR = _BV(SPE) | _BV(MSTR);  // SPIE = 0 : keine SPI-Interrupts auslösen
                                 // SPE  = 1 : Hardware SPI enabled (!!!)
                                 // DORD = 0 : MSB zuerst
```

Abbildung A.9: mcprog.c — Seite 1/3

```
                                // MSTR = 1 : AVR ist Master
                                // CPOL = 0 : SCLK 0->1 (SCLK low wenn idle)
                                // CPHA = 0 : Daten bei steigender SCLK-Flanke lesen
                                // SPR0/1=0/0 : SCLK = f/4 (f/2 mit SPSR.SPI2X=1)
   status = SPSR; // SPI-Statusbyte löschen
}


void timerInit(unsigned int tics) {
   TCCR1B = _BV(CS11) | _BV(WGM12); // Clock-Scaler: 16MHz / 8 = 2e6 Tics/s => alle 500ns ein Tic
   OCR1A  = tics;          // Compare-Wert A. Bei tics=4000: Alle 4000 Tics ein IRQ => alle 2ms => 500Hz
   TCNT1  = 0;             // Rücksetzen des Timers
   TIMSK |= _BV(OCIE1A);   // bei Compare-Match auf A IRQ auslösen
}


void adcInit(unsigned char nr, byte setupByte, byte diffselByte, byte avrgByte) {
   unsigned int cnt = 0;

   // Setup-Register und Diffsel-Register schreiben (direkt hintereinander!)
   PORT_CS = ~_BV(nr);              // CSnr Low (alle anderen CS High)=> Beginn der Übertragung
   SPDR = setupByte;                // Schreiben des ADC-Setup-Registers (Hier: Wert 0x66)
   while ( !( SPSR & (1<<SPIF) ) ){ // warten bis Übertragung komplett (SPSR.SPIF=1, reset autom.)
      cnt++;
      if (cnt > MAX_WAIT) {         // Timeout aufgetreten
         report(ERR_SPI_TIMEOUT);   // Fehlerpaket senden
         break;                     // Warten abbrechen
      }
   }
   SPDR = diffselByte;              // Schreiben des ADC-Unipolar bzw. -Bipolar-Registers (je nach setupByte)
   while ( !( SPSR & (1<<SPIF) ) ){ // warten bis Übertragung komplett (SPSR.SPIF=1, reset autom.)
      cnt++;
      if (cnt > MAX_WAIT) {         // Timeout aufgetreten
         report(ERR_SPI_TIMEOUT);   // Fehlerpaket senden
         break;                     // Warten abbrechen
      }
   }
   PORT_CS |= _BV(nr);              // CSx High => Ende der Übertragung

   // Averaging-Register schreiben
   PORT_CS = ~_BV(nr);              // CSnr Low (alle anderen CS High) => Beginn der Übertragung
   SPDR = avrgByte;                 // Schreiben des ADC-Averaging-Registers (Hier: Wert 0x30)
   while ( !( SPSR & (1<<SPIF) ) ){ // warten bis Übertragung komplett (SPSR.SPIF=1, reset autom.)
      cnt++;
      if (cnt > MAX_WAIT) {         // Timeout aufgetreten
         report(ERR_SPI_TIMEOUT);   // Fehlerpaket senden
         break;                     // Warten abbrechen
      }
   }
   PORT_CS |= _BV(nr);              // CSx High => Ende der Übertragung

}


inline void toSPI(unsigned char nr, byte cmd) {
   unsigned int cnt = 0;

   PORT_CS = ~_BV(nr);              // CSnr Low (alle anderen CS High)=> Beginn der Übertragung
   SPDR = cmd;                      // Schreiben der Daten (SPDR : SPI Data-Register)
   while ( !( SPSR & (1<<SPIF) ) ) {   // warten bis Übertragung komplett (SPSR.SPIF=1, reset autom.)
      cnt++;
      if (cnt > MAX_WAIT) {         // Timeout aufgetreten
         report(ERR_SPI_TIMEOUT);   // Fehlerpaket senden
         break;                     // Warten abbrechen
      }
   }
   PORT_CS |= _BV(nr);              // CSx High => Ende der Übertragung
}


inline void toUSB(byte data) {
   byte i;

   // warten bis USB_TXE Low => schreiben möglich
   while ( bit_is_set( PIN_USBCTRL, USB_TXE ) ) {
      // inzwischen USB_PWR pollen (falls Shutdown während USB-Blockierung)
      if ( bit_is_set( PIN_USBCTRL, USB_PWR ) ) {  // USB-Suspend
         return; // Sendeversuch abbrechen (einfach aus Fkt. raus, da bisher noch nichts passiert ist)
      }
   }

   // wenn ja, Daten anlegen
   PORT_USBDAT = data;
   PORT_USBCTRL &= ~_BV(USB_WR); // USB_WR Low

   // warten bis USB_TXE wieder Low => Datenübergabe beendet
   while ( bit_is_set( PIN_USBCTRL, USB_TXE ) ) {
      // inzwischen USB_PWR pollen (falls Shutdown während USB-Blockierung)
      if ( bit_is_set( PIN_USBCTRL, USB_PWR ) ) {  // USB-Suspend
         break;   //  warten abbrechen (d.h. US_WR einfach wieder high und raus aus der Fkt.)
      }
   }
   PORT_USBCTRL |= _BV(USB_WR);  // USB_WR High
}


inline byte fromUSB() {
   byte data;
```

Abbildung A.10: mcprog.c — Seite 2/3

```
    DDR_USBDAT = 0x00;             // alle USB-Datenleitungen auf Eingang
    PORT_USBCTRL &= ~_BV(USB_RD); // USB_RD Low
    U_NOP;   // etwas warten, bis valide Daten anliegen (mind. 20ns => 1µs ist ok)
    data = PORT_USBDAT;            // Daten einlesen
    PORT_USBCTRL |= _BV(USB_RD);  // USB_RD High
    DDR_USBDAT = 0xff;             // alle USB-Datenleitungen wieder auf Ausgang
    return data;
}


void report(unsigned char errCode) {
    errCode &= _BV(5) | _BV(4) | _BV(3) | _BV(2) | _BV(1) | _BV(0); // Error-Code ausschneiden (zur Sicherheit)
    errCode |= REPORT_HEADER;  // Fehler-Frame-Header davor setzen
    toUSB(errCode);            // an den PC senden
}


void shutdown() {
    // Interrupts aus (kein Samplevorgang darf mehr laufen!!!)
    cli();

    // Externe Komponenten stromlos schalten
    // (WICHTIG!!! Vorher unbedingt alle SPI- und CS-Leitungen low sonst ADCs=SCHROTT !!!!!!!!!)
    SPCR = 0x00;       // Hardware-SPI aus (jetzt ist Zugriff auf SCK und MISO/MOSI möglich)
    PORTB &= ~_BV(PB7) | _BV(PB5) | _BV(PB4); // SCK, MOSI und SS low
    PORT_CS = 0x00;                           // Alle CS-Leitungen Low
    PORT_USBCTRL &= ~_BV(PWRCTRL);            // VCC für ext. Komponenten aus (PWRCTRL=Low)
}


void wakeup() {
    // VCC für ext. Komponenten wiedr an (PWRCTRL=High)
    PORT_USBCTRL |= _BV(PWRCTRL);
    PORT_CS = 0xff;    // Alle CS-Leitungen wieder High
    spiInit();  // SPI-Bus neu starten

    // ADCs neu initialisieren
    U_WAIT(1000);  // 1ms auf ADC Wakeup warten (zur Sicherheit)
    for (unsigned char adcNr=0; adcNr<8; adcNr++) { // alle 8 möglichen ADCs durchgehen
        if ( (activeADCs & _BV(adcNr)) != 0x00 ) { // aktueller ADC aktiv?
            adcInit(adcNr, ADC_SETUP_REG, ADC_UNIPOLAR_REG, ADC_AVRG_REG); // ADC initialisieren
        }
    }

    // Interrupts (Timer) wieder aktivieren
    sei();
}


int main() {
    cli();                         // Interrupts aus (Initialisierung darf nicht unterbrochen werden)
    usbInit();                     // USB-System initialisieren
    PORT_USBCTRL |= _BV(PWRCTRL); // VCC für ext. Komponenten an (PWRCTRL = High)
    spiInit();                     // SPI-System initialisieren

    // Konfigurationsdaten empfangen
    // ... kann später implementiert werden! Jetzt erstmal Werte hardcodieren.

    // ADCs initialisieren
    U_WAIT(1000);  // 1ms auf ADC Wakeup warten (zur Sicherheit)
    for (unsigned char adcNr=0; adcNr<8; adcNr++) { // alle 8 möglichen ADCs durchgehen
        if ( (activeADCs & _BV(adcNr)) != 0x00 ) { // aktueller ADC aktiv?
            adcInit(adcNr, ADC_SETUP_REG, ADC_UNIPOLAR_REG, ADC_AVRG_REG); // ADC initialisieren
        }
    }

    timerInit(TIMER_INTERVAL); // Timer starten
    sei();   // Interrupts ein, ab jetzt läuft der Timer!


    for (;;) { // Endlosschleife,  Arbeit passiert im Interrupt

        // PWREN-Leitung prüfen (falls inzwischen USB-Suspend)
        if ( bit_is_set( PIN_USBCTRL, USB_PWR ) ) { // USB-Suspend
            shutdown(); // Shutdown der externen Komponenten
            while ( bit_is_set( PIN_USBCTRL, USB_PWR ) ) { U_WAIT(100); } // warten bis kein USB-Suspend mehr
            wakeup();   // Externe Komponenten wieder hochfahren
        }

        // Mainloop-Code als ausgeführt markieren
        loopFlag = 1;

    }

    return (0);
}
```

Abbildung A.11: mcprog.c — Seite 3/3