

Entwicklung eines Schwingungsmesssystems für ein Radar

Sebastian Beyer

19. September 2012

Inhaltsverzeichnis

1 Abstrakt	3
2 Motivation	4
3 Beschleunigungssensoren	5
3.1 Piezoresistive Sensoren	7
3.2 Kapazitative Sensoren	7
4 Entwicklung des Schwingungsmesssystems: Prototyp (BMA180)	11
4.1 Bosch BMA180	11
4.2 Arduino	11
4.3 Schnittstellen	12
4.3.1 I ² C	12
4.3.2 Serieller Port	13
4.4 Aufbau und Schaltung	14
4.4.1 Testaufbau	14
4.4.2 Fester Aufbau	15
4.5 Software	18
4.5.1 Arduino	18
4.5.2 Restitution	24
4.5.3 PC Logger	24
5 Berechnung der Wegwerte aus den Beschleunigungsdaten	25
5.1 Numerische Integration	25
5.2 Problematik des Integrierens Noisebehafteter Daten	25
6 Experimente in Waakirchen	26
7 Ergebnisse der Messungen in Waakirchen	28
7.1 Messungen an der Radarantenne	28
7.2 Messungen am Autoklavenaufbau	30
8 Entwicklung des Schwingungsmesssystems: ASC 5511LN-002	31
8.1 ASC 5511LN-002	31
8.2 Diamond-MM-16-AT PC/104 Analog I/O Module	33
8.2.1 Interrupts	34
8.2.2 FIFO Speicher	35
8.3 Software	36
9 Experimente am Geomatikum	44
10 Verbesserungen	44

1 Abstrakt

2 Motivation

3 Beschleunigungssensoren

Beschleunigungssensoren messen die Beschleunigung, die auf ein System wirkt, indem sie die auf eine Testmasse wirkenden Kräfte bestimmen. Nach Newton Die auf einen Körper wirkende Beschleunigung setzt sich aus der Gravitation und Trägheitskräften zusammen. Isoliert man die Trägheitskräfte, so lässt sich daraus die Translationsbeschleunigung bestimmen. Mithilfe dieser und dem zweiten newtonschen Gesetz lässt sich die Positionsänderung des Systems berechnen:

$$F = m \cdot a \quad (1)$$

Es gibt verschiedene Prinzipien, mit denen Beschleunigung gemessen werden kann. Am gebräuchlichsten ist das der Federwaage.

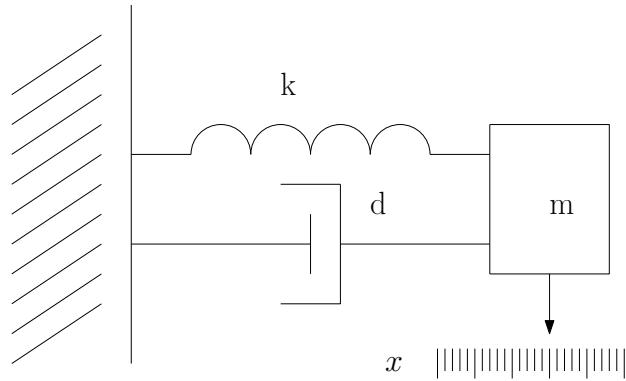


Abb. 1: Masse-Feder-System zur Beschleunigungsmessung. k , d , m und x bezeichnen Federkonstante, Dämpfung, Masse und Auslenkung. nach (Klingbeil, 2006).

Eine Masse m (auch seismische Masse genannt) ist über eine Feder der Federkonstanten k mit einem festen Bezugspunkt verbunden. Die Auslenkung x ist proportional zur auf die Masse wirkenden Beschleunigung a .

$$a = \frac{k}{m} \cdot x \quad (2)$$

Zu beachten ist die Ausrichtung des Sensors: a entspricht immer der Projektion der Beschleunigung auf die Auslenkungsrichtung von m .

Das System führt eine gedämpfte harmonische Schwingung aus. Einmal angeregt beginnt es zu schwingen, die Amplitude nimmt dann aber kontinuierlich ab. Die Bewegungsgleichung lautet: (vergleiche auch (Meschede, 2001))

$$m\ddot{x} + d\dot{x} + kx = 0 \quad (3)$$

Wobei m die Masse, d die Dämpfung und k die Federkonstante ist. Eine Lösung dieser Differentialgleichung lautet:

$$x(t) = x_0 e^{-\delta t} \sin(\omega_d t + \varphi_0) \quad \text{mit} \quad \delta = \frac{d}{2m} \quad (4)$$

Die Lösung ist aus zwei elementaren Funktionen zusammengesetzt, wobei die eine den periodischen Anteil ausmacht und die andere den dämpfenden Anteil. Der periodische Anteil lässt ich auch als $x_0 e^{i\omega_d t}$ schreiben, wobei

$$\omega_d = \sqrt{\omega_0^2 - \delta^2} \quad (5)$$

die gedämpfte Eigenfrequenz genannt wird und einen entscheidenden Einfluss auf das Schwingverhalten hat.

Bei kleinem δ ist die Dämpfung gering und es gilt

$$\omega_d \approx \omega_0 \quad (6)$$

Dies nennt man den *Schwingungsfall*

Wenn die Dämpfung jedoch so groß wird, dass

$$\delta = \omega_0 \quad (7)$$

gilt, wird $\omega_d = 0$ und das System kommt in kürzestmöglicher Zeit zur Ruhe. Man spricht vom *aperiodischen Grenzfall*. Dieses Verhalten ist bei Beschleunigungsaufnehmern erwünscht, weil man so die höchste Wiederholfrequenz von Messungen erreicht, ohne dass sich die Anregungen gegenseitig überlagern.

Wird δ noch größer, so wird der Term unter der Wurzel negativ und ω_d damit imaginär. Aus

$$e^{i\omega_d t} \quad (8)$$

wird

$$e^{-kt} \quad (9)$$

und bewirkt eine zusätzliche Dämpfung. Eine Schwingung existiert nicht mehr und das

System erreicht seine Ruhelage nach längerer Zeit als beim aperiodischen Grenzfall. Dieses als *Kriechfall* bezeichnete Verhalten ist in den meisten Fällen unerwünscht.

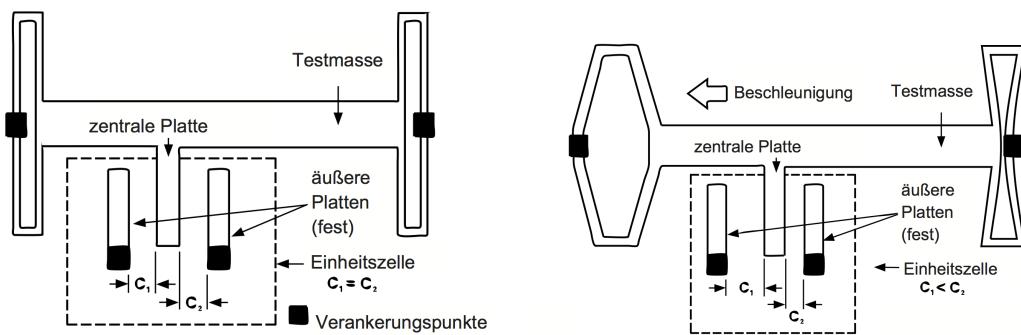
Mittlerweile sind Beschleunigungssensoren in den meisten Fällen als MEMS realisiert (**Micro Electro Mechanical Systems**). Sehr kleine mechanische Elemente (1-100 Mikrometer) werden zusammen mit elektronischen Schaltungen auf einen Siliziumwafer aufgebracht. Dabei werden Techniken aus der Fabrikation von integrierten Schaltkreisen (ICs) verwendet. So können komplizierte elektromechanische Systeme in winziger Größe und hoher Stückzahl hergestellt werden. Der geringe Preis ist maßgeblich dafür verantwortlich, dass die Sensoren in immer mehr Anwendungen integriert werden (Autos, Smartphones, Quadrokopter...).

Die Beschleunigungsmessung erfolgt also über die Messung der Auslenkung einer Testmasse. Dazu haben sich zwei Verfahren durchgesetzt, die ich im Folgenden kurz erläutern möchte.

3.1 Piezoresistive Sensoren

Piezoresistive Sensoren machen sich den piezoelektrischen Effekt zunutze. In der Feder der Testmasse befinden sich Piezoelemente, welche sich bei Auslenkung verformen und damit ihren Widerstand ändern. Silizium ist ein geeignetes Material, da es sehr empfindlich und linear reagiert und gleichzeitig gut mit der MEMS Technik kombinierbar ist (Kanda, 1991).

3.2 Kapazitative Sensoren



(a) Sensor in Ruhe, Abstand der Kondensatorplatten ist gleich, $C_1 = C_2$ (b) Sensor während einer externen Beschleunigung, $C_1 < C_2$

Abb. 2: Vereinfachtes Diagramm des ADXL05 (ADXL05, 1996)

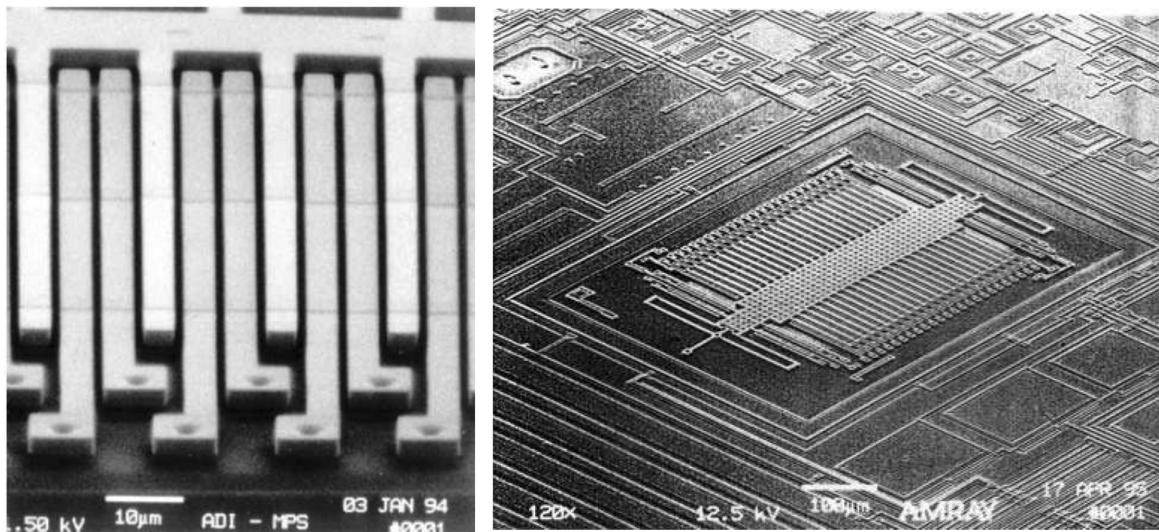


Abb. 3: ADXL05 unter dem Elektronenmikroskop, Beispiel für einen kapazitiven MEMS Beschleunigungssensor mit 46 Einzelzellen (Klingbeil, 2006).

Die Auslenkung lässt sich auch über eine Kapazitätsmessung bestimmen, wenn man je eine Elektrode an der Testmasse und am fixen Referenzpunkt anbringt (Sherman et al., 1992). Eine mögliche Realisierung ist in Abbildung 2 a,b zu sehen: Es handelt sich um einen sogenannten Differentialkondensatorsensor (Schmidt, 2002).

Die seismische Masse befindet sich zwischen zwei fest verankerten Platten, die zusammen zwei Kondensatoren C_1 und C_2 bilden (Abb. 2(a)). Eine auftretende Beschleunigung führt also zu einer Auslenkung der Mittelelektrode des Kondensatorenpaars (seismische Masse) um die Länge $\pm x$ und damit zu einer symmetrischen Kapazitätsänderung um $\pm\Delta C$ (Abb. 2(b)). Dieser Aufbau ist eine sogenannte Einzelzelle. In einem Sensor befinden sich viele Einzelzellen, um die Kapazitätsvariation und damit die Sensibilität zu erhöhen (Abb. 3)

Um diese Variation in ein elektrisches Ausgangssignal umzusetzen, wird eine sogenannte Brückenschaltung verwendet (Abb. 4).

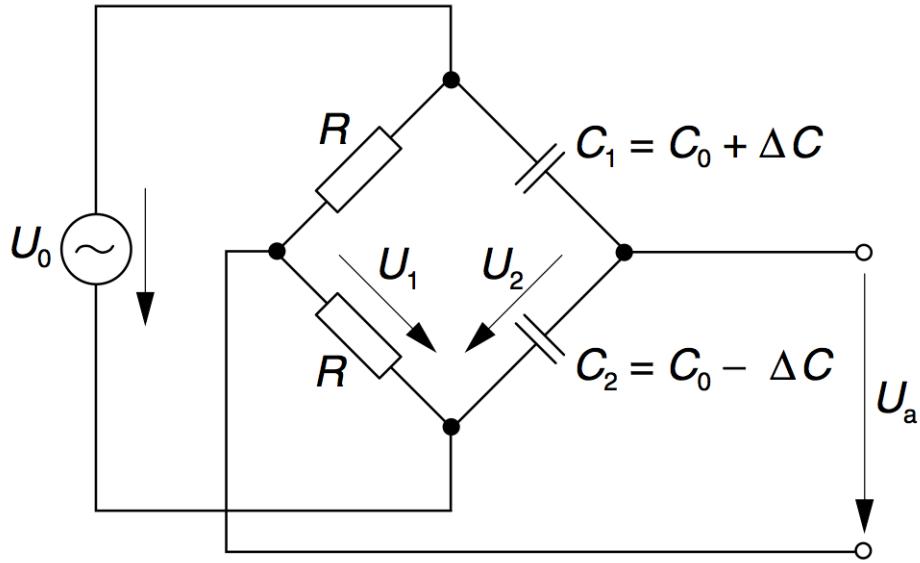


Abb. 4: Brückenschaltung mit Differentialkondensator (Die mit U_a verbundenen Elektroden von C_1 und C_2 bilden eine gemeinsame Platte) (Schmidt, 2002)

Die folgende Herleitung ist aus (Schmidt, 2002) entnommen:

Wird eine Wechselspannung an die Brücke angelegt, so ergibt sich nach der Spannungssteilerregel für den Ausgang U_a :

$$U_a = U_1 - U_2 = U_0 \frac{R}{2R} - U_0 \frac{\frac{1}{j\omega C_2}}{\frac{1}{j\omega C_2} + \frac{1}{j\omega C_1}} \quad (10)$$

Durch Kürzen ergibt sich:

$$U_a = \frac{U_0}{2} - U_0 \frac{\frac{1}{C_2}}{\frac{1}{C_2} + \frac{1}{C_1}} = U_0 \left(\frac{1}{2} - \frac{C_1}{C_2 + C_1} \right) = \frac{U_0}{2} \left(\frac{C_2 - C_1}{C_2 + C_1} \right) \quad (11)$$

Mit $C_1 = C_0 + \Delta C = \epsilon_0 \cdot \epsilon_r \cdot A / (d_0 - x)$ und $C_2 = C_0 - \Delta C = \epsilon_0 \cdot \epsilon_r \cdot A / (d_0 + x)$ erhält man eine lineare Abhängigkeit der Ausgangsspannung U_a von der Auslenkung x :

$$U_a = -U_0 \frac{x}{2d_0} \quad (12)$$

Diese Spannung lässt sich nun digitalisieren und auslesen. In vielen MEMS Bausteinen ist bereits ein integrierter Analog Digital Wandler eingebaut, sodass die Messwerte direkt digital abrufbar sind.

4 Entwicklung des Schwingungsmesssystems: Prototyp (BMA180)

Um abschätzen zu können wie groß die auftretenden Beschleunigungen an der Radarantenne sind, habe ich damit begonnen einen günstigen und einfach zu verwendenden Prototypen zu entwickeln. Letzteres ist vor Allem wichtig, um entscheiden zu können, für welchen Messbereich und welche Frequenzen der eigentliche Sensor ausgelegt sein muss.

4.1 Bosch BMA180

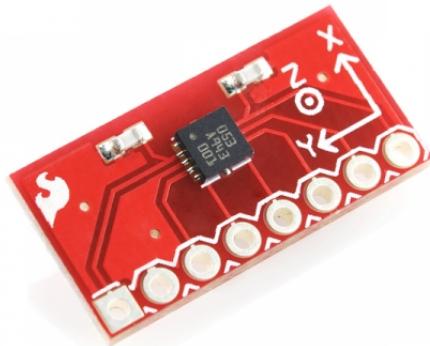


Abb. 5: BMA180 Breakoutboard von Sparkfun

Die Wahl fiel auf einen digitalen, dreiachsigen MEMS Beschleunigungssensor *Bosch BMA180*. Er verfügt über einen eingebauten 14-bit Analog-Digital Wandler und sieben per Software verstellbaren Messbereiche von ± 1 bis $\pm 16\text{g}$. Die Kommunikation kann über SPI (Serial Peripheral Interface) oder I²C (Inter-Integrated Circuit) erfolgen.

Ich nutze ein Breakoutboard von Sparkfun (Abb. 5), welches die IC Pins (SMD Technik) mit praktische Lötösen verbindet. Außerdem sind bereits zwei spannungsstabilisierende Kondensatoren mit auf der Platine verbaut.

4.2 Arduino

Arduino ist eine auf ATmega Mikroprozessoren basierende Open-Source Entwicklungsplattform zur Verarbeitung von analogen und digitalen Signalen. Die Programmierung kann über eine eigene Entwicklungsumgebung in einer an *Processing*¹ angelehnten Sprache erfolgen, die im Prinzip ein vereinfachtes C/C++ darstellt.

¹processing.org

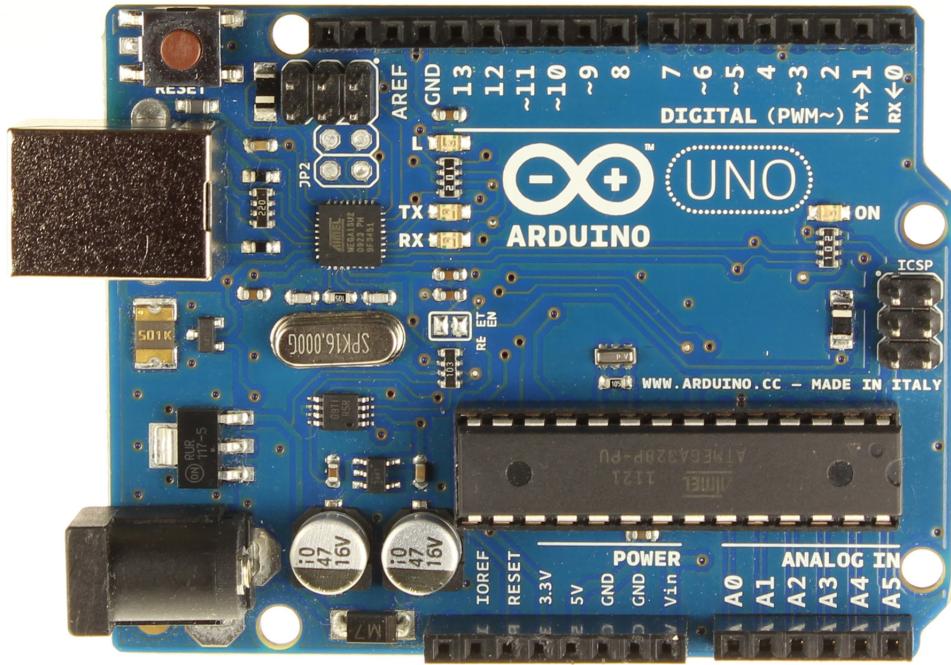


Abb. 6: Arduino UNO

Die Plattform ist auf Prototyping und Experimente ausgelegt. Es ist bereits ein Bootloader vorinstalliert, so kann die Programmierung direkt über die serielle Schnittstelle erfolgen. Die Boards machen die meisten Pins des ATmegas für eigene Schaltungen verfügbar, in den gängigen Boards sind das 14 Pins, die frei als Ein- oder Ausgänge genutzt werden können. Die Stromversorgung kann über USB oder eine externe 5V Quelle erfolgen. Als Kommunikationsinterfaces werden SPI, ICSP (In-Circuit Serial Programming) und I²C angeboten.

Der ATmega arbeitet mit 16 MHz und hat einen geringen Energieverbrauch.

4.3 Schnittstellen

4.3.1 I²C

Zur Kommunikation zwischen Beschleunigungssensor und Arduino habe ich den I²C Bus (NXP, 2012) gewählt. Dabei handelt es sich um einen von Phillips entwickelten seriellen Datenbus, der ursprünglich entwickelt wurde, um Chips in Fernsehgeräten steuern zu können. Inzwischen ist das Patent ausgelaufen und er wird in vielen Hardwareprojekten verwendet, da er sehr einfach zu verstehen und zu verwenden ist.

Ich benutze die Arduino Wire Library (Arduino, 2012), welche die gesamte Kommunikation über I²C steuert und einfache Funktionsaufrufe, wie zum Beispiel Senden und Empfangen, zur Verfügung stellt.

4.3.2 Serieller Port

Zum einfachen Anschluss des Arduinos an einen PC oder Datenlogger wird eine serielle Schnittstelle nach RS232 verwendet. Auf dem Arduino befindet sich ein ATmega16U2, welcher die seriellen Signale in USB umwandelt und dafür sorgt, dass der Arduino am PC als virtueller COM-Port erscheint.

4.4 Aufbau und Schaltung

4.4.1 Testaufbau

Um die korrekte Verschaltung zu überprüfen und die Software für das Auslesen der Daten zu entwickeln, habe ich zunächst auf dem Breadboard gearbeitet. Der dazu verwendete Schaltplan ist in Abbildung 8 zu sehen.

Da der Sensor sowohl im I²C- sowie im SPI-Modus betrieben werden kann, ist es notwendig, den Modus bereits bei der Verschaltung einzustellen. Über den CS Pin (Pin 4) ist dies möglich. Schaltet man ihn auf High (3.3V), so benutzt der Chip I²C, verbindet man ihn mit GND, so wird ISP verwendet. Je nach Modus haben die Pins unterschiedliche Funktionen. In Tabelle 1 sind die verschiedenen Konfigurationen aufgeführt.

Pin 6 legt im von mir benutzten I²C Modus die Adresse des Chips fest. Ist dieser auf Low (GND), so ist die Adresse 0x40.

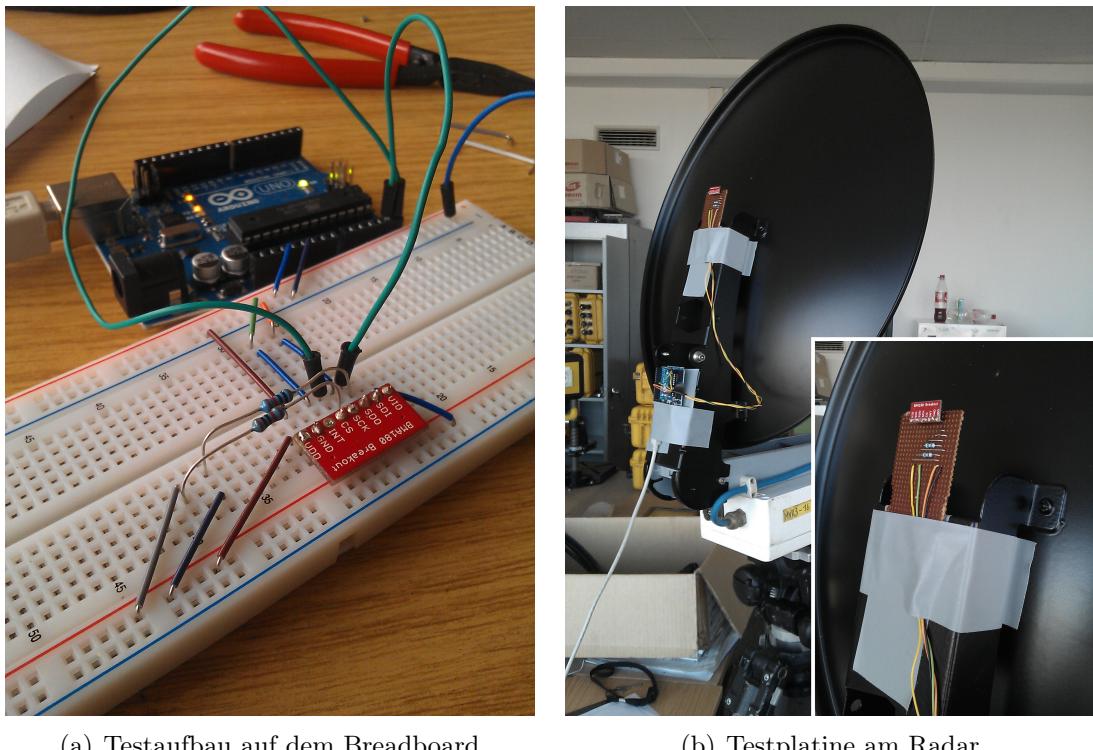
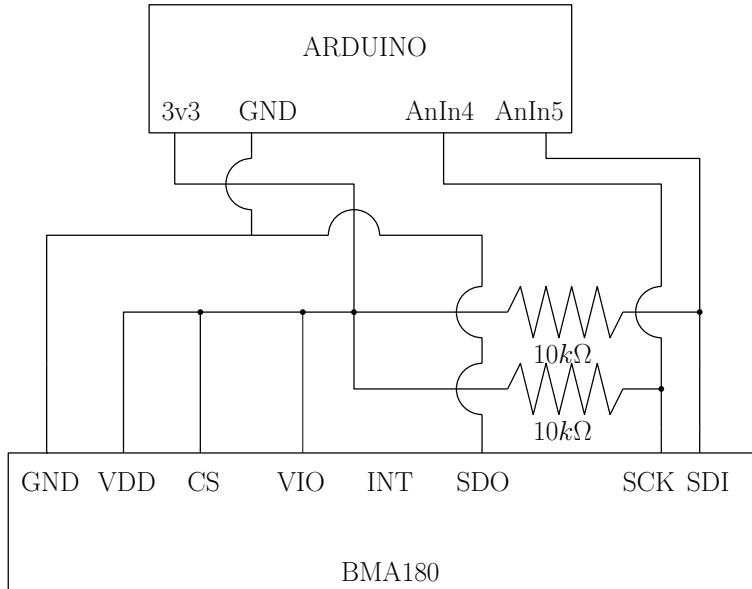


Abb. 7: Testaufbau des Sensors

Pin	SPI mode	I2C mode
7	SDI input	SDA birectional (!)
6	SDO output	ADDR adress bit, input
5	SCLK input	SCL input
4	CSB chip select, input	I2C mode select, input

Tab. 1: BMA180 Pinbelegung für SPI und I²C Modes (Sensortec, 2009)**Abb. 8:** Schaltplan Testaufbau. Der Interrupt ist nicht verbunden, VDD und CS sind auf 3.3V geschaltet, GND und SDO auf die Masse des Arduinos gezogen, die I²C Datenleitungen SCK und SDI sind mit den Arduinopins A4 und A5 verbunden, wobei zusätzlich 10kΩ Pull-Up-Widerstände eingebaut sind.

4.4.2 Fester Aufbau

Um das System praktisch nutzen zu können muss es natürlich fest aufgebaut werden und mit einem Gehäuse versehen werden, das es erlaubt, ihn fest an einem Testobjekt anzubringen und ihn gleichzeitig vor Schäden durch mechanische oder witterungsbedingte Einflüsse schützt.

Um die eigentliche Sensoreinheit möglichst kompakt zu halten, habe ich mich entschieden, den Beschleunigungssensor vom Arduino zu trennen und auf eine kleine Lochrasterplatine zu löten. Diese wird in einen festen Block aus Polyurethanharz² eingegossen, womit sie gleichzeitig gut geschützt und leicht anzubringen ist (Abb. 9).

²OPTICALLY CLEAR POLYURETHANE, RS COMPONENTS, RS 195-984A



Abb. 9: BMA180 in Polyurethanharz eingegossen, die Kunststoffform wurde von den feinmechanischen Werkstätten des Fachbereichs Geowissenschaften gefertigt.

Der Arduino ist in ein ABS Gehäuse (zum Beispiel von BOSS Endloseres) eingebaut (Abb. 12(a) und 12(b)). Die I²C Verbindung mit dem Sensor erfolgt über einen HirschmannsteckerTM mit 7 Polen, von denen 4 beschaltet sind, wie in Abbildung 11 zu sehen ist. Mit dem Logger ist der Arduino über ein USB-Kabel verbunden, über das die Kommunikation per RS232 geführt wird.

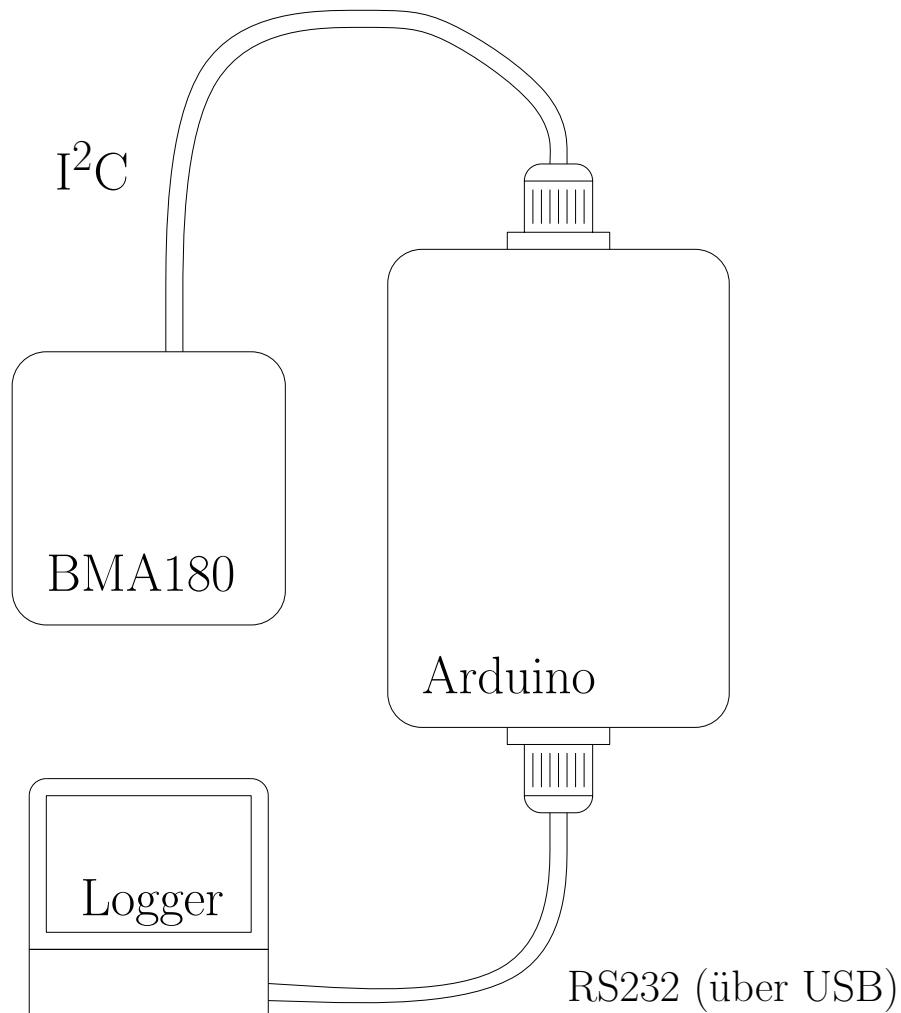


Abb. 10: Verbindungen zwischen BMA180, Arduino und Logger per I²C bzw. RS232

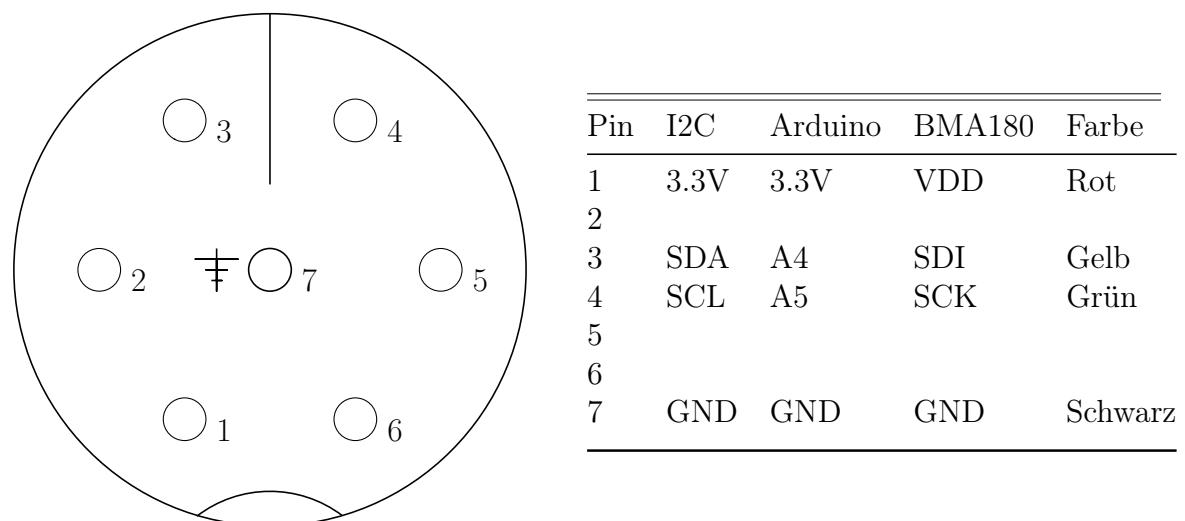
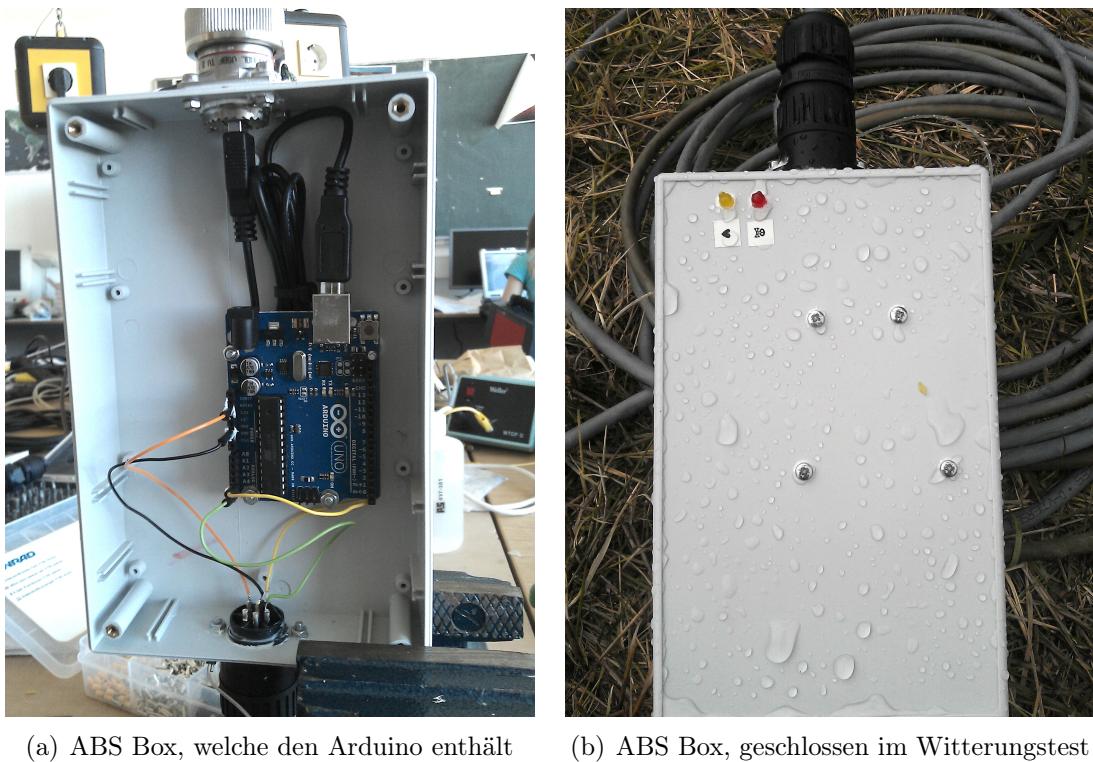


Abb. 11: I²C Beschaltung des HirschmannsteckersTM



(a) ABS Box, welche den Arduino enthält

(b) ABS Box, geschlossen im Witterungstest

Abb. 12: ABS Box mit Arduino

4.5 Software

Die Software besteht aus zwei Teilen: Der Code auf dem Arduino fragt die Daten vom Sensor ab und stellt sie per RS232 zur Verfügung. Der zweite Teil auf dem Logger sammelt die Messungen über den Seriellen Port und schreibt diese gemeinsam mit der jeweils aktuellen Uhrzeit in eine Datei.

Bei der Erläuterung des Codes möchte ich mich auf die wesentlichen Punkte beschränken. Die vollständigen Quelltexte befinden sich im Anhang.

4.5.1 Arduino

Der Arduino ist für die Kommunikation mit dem BMA180 über I²C zuständig. Für diese Aufgabe nutze ich die Arduino Bibliothek bma180³. Sie enthält eine Sammlung von Lese und Schreibroutinen, die ich für meine Zwecke angepasst habe. Hier möchte ich die einzelnen Schritte beschreiben, die notwendig sind, um eine Messung zu erhalten und

³Selbst nach ausführlicher Recherche ist es mir nicht gelungen den Originalauthor der Bibliothek zu ermitteln. Ich habe sie von John Mc Combs, welcher allerdings nicht der Ursprungsaufor ist.
<https://bitbucket.org/johnmccombs/arduino-libraries/src/058c7101c8da/bma180> (August 2012)

diese an den Logger weiterzuleiten.

Die Steuerung des BMA180 erfolgt über direkte Zugriffe auf die Register. Entweder schreibt man einen neuen Wert in ein Register oder man liest ein Register aus. Es gibt fünf Arten von Registern - **test**, **control**, **image**, **status** und **data**. Davon interessieren uns nur die **control** und die **data** Register. Weitere Informationen über die zusätzlichen Register lassen sich im Datenblatt (Sensortec, 2009) nachschlagen.

Die **control** Register dienen dazu, den Beschleunigungssensor zu steuern, vornehmlich also den Messbereich festzulegen und eventuell einen zusätzlichen digitalen Bandfilter einzustellen. Dazu müssen Werte in Register hineingeschrieben werden. Dazu geht man folgendermaßen vor:

In Register schreiben

1. Übertragung initiieren (mit der Adresse des Sensors)
2. Adresse des Registers senden, in welches wir schreiben möchten
3. Daten senden, die wir in das Register schreiben möchten
4. Übertragung beenden

Da ein Register immer ein Byte (8 Bit) groß ist, finden sich oftmals verschiedene Einstellungen in einem Register um Platz zu sparen. Einen korrekten Schreibvorgang auszuführen wird dadurch komplexer. Daher möchte ich den Prozess am Beispiel der Messbereichseinstellung demonstrieren.

offset_y	39h	offset_x<11:0> (msb)		readout_12bit	80h
offset_x	38h	offset_x<11:0> (msb)		00h	80h
offset_t	37h	offset_z<3:0> (lsb)		range<2:0>	00h
offset_lsb2	36h	offset_x<3:0> (lsb)		smp_skip	00h
offset_lsb1	35h	offset_x<3:0> (lsb)		wake-up	80h
gain_z	34h	gain_z<6:0>		shadow_dis	80h
gain_y	33h	gain_y<6:0>		dis_reg	80h
gain_x	32h	gain_x<6:0>			80h
gain_t	31h	gain_t<4:0>		tapsens_dur<2:0>	80h

Abb. 13: Memory Map BMA180, Ausschnitt (Sensortec, 2009)

Die relevanten Bits befinden sich im Register 0x35⁴, Bit eins, zwei und drei⁵ (siehe Abb. 13). Hier die entsprechenden Zeilen aus dem Quellcode:

```
1 void BMA180::setGSensitivity(GSENSITIVITY maxg) //1, 1.5 2 3 4 8 16
2 {
```

⁴Ein vorangestelltes 0x bedeutet, dass es sich um eine Zahl im Hexadezimalsystem handelt

⁵Es wird von rechts nach links gezählt und mit 0 begonnen

```

3     setRegValue (0x35 ,maxg<<1,0xF1) ;
4     gSense = maxg;
5 }
```

Listing 1: Funktion zum Einstellen des Messbereichs aus der bma180 Bibliothek
 maxg ist eine dem Wertebereich zugeordnete Binärzahl zwischen 000 und 110

Die aufgerufene Funktion *setRegValue* (Zeile 3) bekommt als Argumente das zu schreibende Register (0x35), den zu schreibenden Wert (*maxg << 1*) und zusätzlich eine Maske, die angibt, welche Bits des Registers nicht(!) verändert werden sollen.

Da das letzte Bit (Bit Null) eine andere Funktion hat, 'schieben' wir *maxg* mit Hilfe der *bitshift Operation* << um ein Bit nach links.

Bei *gSense* (Zeile 4) handelt es sich um eine Statusvariable, die später benutzt wird, um den genutzten Wertebereich mit zu dokumentieren.

```

1 void BMA180:: setRegValue( int regAdr , int val , int maskPreserve )
2 {
3     int preserve=getValue( regAdr );
4     int orgval=preserve & maskPreserve;
5     Wire.beginTransmission( address );
6     Wire.write( regAdr );
7     Wire.write( orgval | val );
8     int result = Wire.endTransmission();
9     checkResult( result );
10 }
```

Listing 2: Funktion zum Schreiben eines Registers aus der BMA180 Bibliothek

Nun lesen wir zunächst den bisherigen Wert des Registers ein (Zeile 3) und maskieren ihn mit der übergebenen Maske, so dass wir alle Bits übernehmen, außer die, welche wir verändern wollen (Zeile 4). Dann schreiben wir in das Register, wobei wir nach dem oben beschriebenen Schema vorgehen. Beim Senden der Daten verknüpfen wir jedoch *orgval* mittels eines bitweisen *OR* mit dem zu schreibenden Wert (Zeile 7). So haben wir sichergestellt, dass wir nur die relevanten Bits verändern.

In Abbildung 14 ist das Vorgehen noch einmal dargestellt.

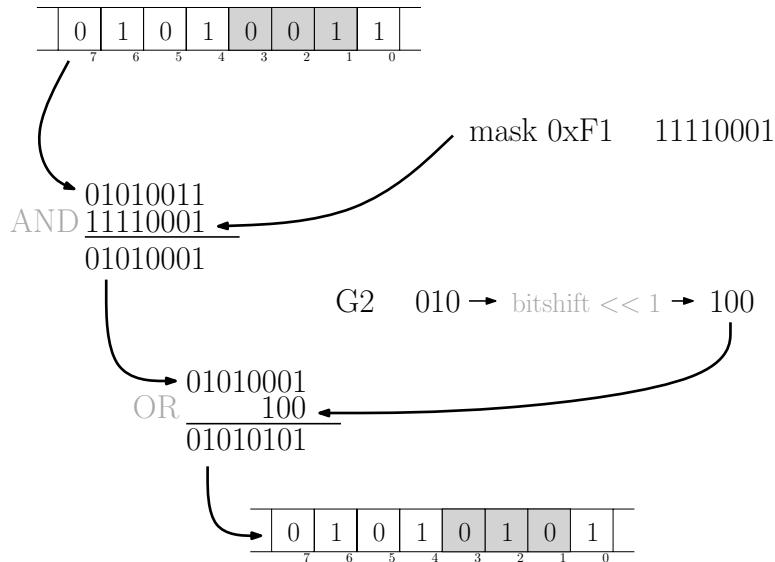


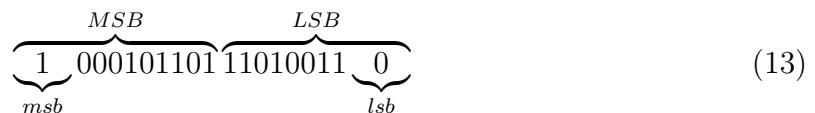
Abb. 14: Beispiel eines Schreibvorgangs in das Register 0x35. Nur die grau markierten Bits werden verändert.

Daten aus Registern lesen

In die data Register werden die gemessenen Beschleunigungsdaten geschrieben. Da der A/D-Wandler des BMA180 eine Auflösung von 14 Bit hat, existieren pro Achse 2 Register mit jeweils 8 Bit. Hier benötigen wir das Konzept der **Bitwertigkeit**:

In einer Zahl haben die Stellen unterschiedlichen Wert. Verändert man zum Beispiel bei der Zahl 4650 die letzte Stelle, so ist der Unterschied maximal 9 (4650 verglichen zu 4659). Wird hingegen die dritte Stelle (von hinten) verändert, so kann der Unterschied bis zu 900 betragen. Analog verhält es sich mit den binären Zahlen. Dabei bezeichnen wir das höchstwertige Bit als **most-significant-bit (msb)** und das geringwertige Bit als **least-significant-bit (lsb)**.

Analog dazu spricht man vom höchstwertigen Byte **most-significant-byte (MSB)** und geringwertigen Byte **least-significant-byte (LSB)**.



Die gemessene Beschleunigung ist also aufgeteilt und muss im Programm wieder zusammengesetzt werden.

Um die Beschleunigungsdaten vom Sensor zu laden, ist so vorzugehen:

1. Übertragung initiieren (mit der Adresse des Sensors)

2. Adresse des Registers senden, bei dem wir **anfangen** wollen zu lesen
3. Übertragung beenden
4. So viele Bytes abrufen, wie wir haben möchten
5. Übertragung beenden

```

1 void BMA180::readAccel()
2 {
3     unsigned int result;
4
5     Wire.beginTransmission(address);
6     Wire.write(0x02);
7     Wire.endTransmission();
8     Wire.requestFrom((int)address, 7);
9     if(Wire.available()==7)
10    {
11        int lsb = Wire.read()>>2;
12        int msb = Wire.read();
13        x=(msb<<6)+lsb ;
14        if (x&0x2000) x|=0xc000; // set full 2 complement for neg values
15        lsb = Wire.read()>>2;
16        msb = Wire.read();
17        y=(msb<<6)+lsb ;
18        if (y&0x2000) y|=0xc000 ;
19        lsb = Wire.read()>>2;
20        msb = Wire.read();
21        z=(msb<<6)+lsb ;
22        if (z&0x2000) z|=0xc000 ;
23        temp = Wire.read();
24        if (temp&0x80) temp|=0xff00 ;
25    }
26    result = Wire.endTransmission();
27 }
```

Listing 3: Funktion zum Auslesen der Beschleunigungsdaten, x,y,z sind 2 Byte Integerzahlen

Das Register, bei dem wir beginnen wollen zu lesen, ist 0x02, (Zeile 6) und wir benötigen die nächsten 7 Bytes (Zeile 8). Jeweils zwei Bytes MSB und LSB für die drei Achsen und als letztes Byte lesen wir zusätzlich die aktuelle Temperatur aus. Diese wird jedoch im Weiteren nicht verwendet.

Mit `Wire.requestFrom(address, n)` fordern wir einen Stapel von n Bytes vom Sensor (Zeile 8). `Wire.read()` entfernt immer das oberste Byte von diesem Stapel und gibt seinen Wert aus. Da das LSB an Stelle null und eins für den Messwert nicht relevante Bits enthält, entfernen wir diese per Bitshift (Zeile 11). Dann verschieben wir das MSB im ganzen um sechs Stellen nach links, um es dann mit dem LSB zu addieren und damit den kompletten Messwert zu erhalten (Zeile 13).

Da die Daten im sogenannten Zweierkomplement (näheres in (Tietze and Schenk, 2002)) ausgegeben werden und es sich bei x (sowie bei y und z) um Integer Variablen handelt, müssen wir das Vorzeichenbit an vorderster Stelle setzen. Dies geschieht in Zeile 14 (bzw. Zeile 18 für y und Zeile 24 für z).

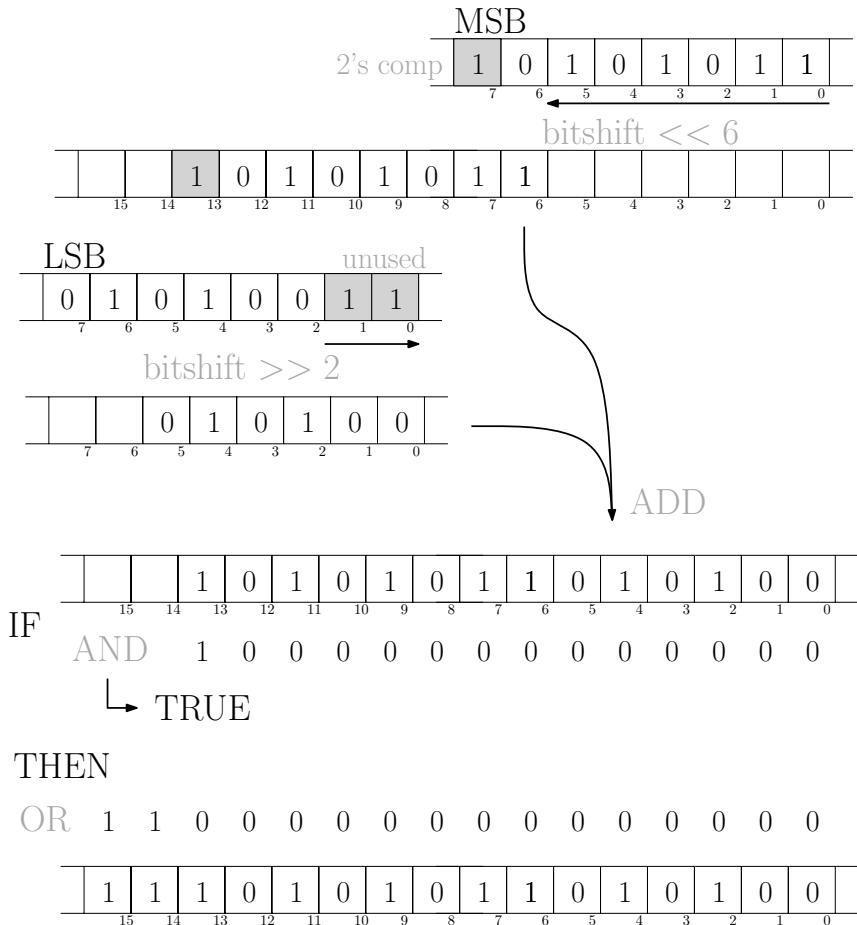


Abb. 15: Auslesen und zusammenfügen der Daten. MSB und LSB werden addiert. Mittels IF Abfrage und Masking wird überprüft, ob es sich um eine negative Zahl handelt; Falls ja, so wird das korrekte Zweierkomplement gebildet und als Integer gespeichert.

4.5.2 Restitution

Jetzt haben wir eine Beschleunigungsmessung, jedoch ist diese noch in gerätespezifischen 'count' Einheiten. Diese müssen nun die physikalisch bestimmte Einheit der Beschleunigung umgerechnet werden. Der Vorgang wird als Restitution bezeichnet und der entsprechende Code ist in Listing 4 zu sehen.

```

1 float BMA180::getXValFloat()
2 {
3     // normalize (if x is maximum (8191) and GSENSE=1.0 then 1.0
4     return (float)x/8191.0*getGSense();
5 }
```

Listing 4: Restitution der aufgenommenen Beschleunigungsdaten. Ausgabe als Vielfaches von der Erdbeschleunigung $g = 9.81m/s^2$

Aus dem Datenblatt (Sensortec, 2009) lässt sich entnehmen, dass 10 0000 0000 0000 den geringstmöglichen Wert in jedem Messbereich darstellt und 01 1111 1111 1111 die größten. Das erste Bit gibt dabei das Vorzeichen an (Zweierkomplement). Teilt man den Wert durch 8191, was in Binärdarstellung 1 1111 1111 entspricht, erhält man nach einer Multiplikation mit dem Messbereich (GSENSE) die Beschleunigung als Vielfaches der Erdbeschleunigung $g = 9.81m/s^2$.

4.5.3 PC Logger

Das Programm, welches auf dem PC läuft und die Daten aufzeichnet, ist ein Python Script, welches ich von Matthias Hort übernommen und angepasst habe. Sobald es gestartet ist und eine kurze Beschreibung für das aktuelle Experiment eingegeben wurde, nimmt es sämtliche Nachrichten auf dem eingestellten Comport entgegen und schreibt diese mit der jeweils aktuellen Systemzeit in eine Datei.

Die exakte Zeit ist sehr wichtig, um die Ergebnisse später mit anderen Messungen verglichen zu können. Zu diesem Zweck ist der PC mit einem NTP (Network Time Protocol) Server verbunden, welcher stets die korrekte Uhrzeit von GPS-Satelliten empfängt und im Netzwerk zur Synchronisation bereitstellt. Ein Programm auf dem PC (Domain Time II) korrigiert mit dieser GPS Zeit die computerinterne Uhr.

5 Berechnung der Wegwerte aus den Beschleunigungsdaten

5.1 Numerische Integration

Um aus den gemessenen Beschleunigungen die gesuchten Wegwerte (Ausschläge) zu ermitteln, müssen die Messwerte zweifach über die Zeit integriert werden. Da die zu integrierenden Funktionen als begrenzte Anzahl von Abtastpunkten in diskreten Zeitabständen vorliegen, bietet es sich an, numerische Integrationsverfahren anzuwenden. Die dazu möglichen Verfahren, wie z.B. Rechteckverfahren, Trapezformel, Rombergverfahren, unterscheiden sich durch die Art der Interpolation zwischen den Abtastpunkten (Bronstein et al., 1995, S. 760ff). Die Wahl des Integrationsverfahrens beeinflusst dabei die Genauigkeit des Ergebnisses.

In dieser Arbeit verwende ich die Trapezformel und komme damit für den Weg auf folgende Rekursionsformel:

$$y_{i+1} = \Delta t^2 \cdot a_i - y_{i-1} + 2 \cdot y_i \quad (14)$$

Mit dieser Formel kann aus den vorherigen Wegwerten y_{i-1} und y_i sowie den gemessenen Beschleunigungen a_i und dem Zeitintervall Δt der nächste Wegwert bestimmt werden. Die Startwerte für $i = 1$ sind vorerst nicht bekannt und können z.B. $y_0 = y_1 = 0$ gewählt werden. Dies führt allerdings zu systematischen Fehlern, die später behoben werden müssen.

5.2 Problematik des Integrierens Noisebehafteter Daten

Wie alle nicht synthetischen Daten enthalten die aufgenommenen Beschleunigungsmessungen ein gewisses Rauschen. Schwankungen in der Spannungsversorgung, elektromagnetische Einstreuungen oder thermisches Rauschen sind einige Beispiele für mögliche Ursachen. Solange das Signal deutlich größer ist als das Rauschen, ist das kaum ein Problem. Man spricht von einem guten Signal- zu Rauschverhältnis.

Bei der Integration der Beschleunigungswerten zu Wegwerten handelt es sich jedoch gewissermaßen um eine Addition aller Messwerte und damit einer Addition allen Rauschens. Je länger die Zeit ist, über die integriert wird, desto größer wird der Fehler in den Wegdaten.

Das Resultat ist in Abbildung 16 zu erkennen. Ich habe in Matlab eine Sinusfunktion mit 1000 Stützstellen numerisch integriert. Dabei habe ich zunächst keinen Noise hinzugefügt. Bei der unteren Abbildung jedoch liegt auf dem Sinus ein Rauschen mit einer Varianz von 0.3. Es ist eine deutliche Drift der resultierenden Daten zu sehen. Bei der

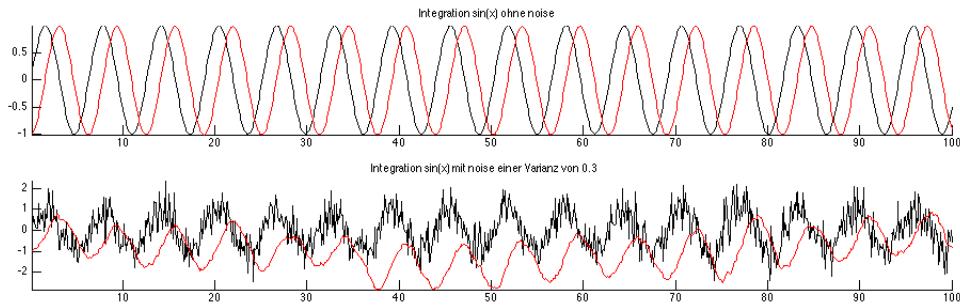


Abb. 16: Numerische Integration von $\sin(x)$. Oben: ohne Noise Unten: Mit Noise einer Varianz von 0.3

zweifachen Integration, die man benötigt um aus den Beschleunigungsdaten Wegdaten zu berechnen, tritt dieser Effekt sogar noch verstärkt auf. Die Fehler der Messungen überlagern die eigentlichen Messergebnisse und führen damit zu völlig falschen Wegstrecken.

Um diesen Effekt zu verhindern, müssen die Daten gefiltert werden. Die sich dafür bietenden Möglichkeiten möchte ich in der jeweiligen Auswertung besprechen.

6 Experimente in Waakirchen

Um das System in der Anwendung zu testen, habe ich im Mai 2012 an Eruptionsdynamikexperimenten in Waakirchen, in der Nähe von München, teilgenommen. Diese Experimente wurden gemeinsam mit der Ludwig-Maximilians-Universität München, der Universität Catania und dem Istituto Nazionale di Geofisica e Vulcanologia (Sezione di Catania and Dept. of Seismology and Tectonophysics, Rome) durchgeführt.

Bei diesen Experimenten wurden künstliche Explosionen in Autoklaven (Spieler et al., 2004) verschiedener Geometrie (von 16 cm bis 40 cm Länge) ausgelöst. Es kamen dabei zwei Aufbauten zum Einsatz, von denen einer das Erhitzen der Proben auf bis zu 850°C und damit der Wirklichkeit nähere Umgebungsbedingungen ermöglichte. Bei den verwendeten Samples handelte es sich um Proben von verschiedenen Vulkanen (sowohl Asche als auch festes Lavagestein) und Analogmaterialien. Die Experimente wurden wiederholt durchgeführt und dabei potentielle Einflussgrößen wie Korngröße, Mischverhältnis, Druck und Temperatur variiert.

Untersucht wurde dabei eine Vielzahl von Parametern durch den Einsatz von Hochgeschwindigkeits-Thermo- und optischen Kameras, Piezo Sensoren, einem akustischen Array und unserem Doppler Radar.

Mein Ziel war dabei vor Allem, den Prototypen im Feld zu testen und Informationen über die Größenordnung der an der Radarantenne auftretenden Geschwindigkeiten zu gewinnen. Zusätzlich wollte ich die Bewegung des Autoclavenaufbaus untersuchen, da bei vorherigen Experimenten in den Radardaten in einigen Fällen negative Geschwindigkeiten auftraten, für die es bisher keine vollständige Erklärung gibt.

7 Ergebnisse der Messungen in Waakirchen

7.1 Messungen an der Radarantenne

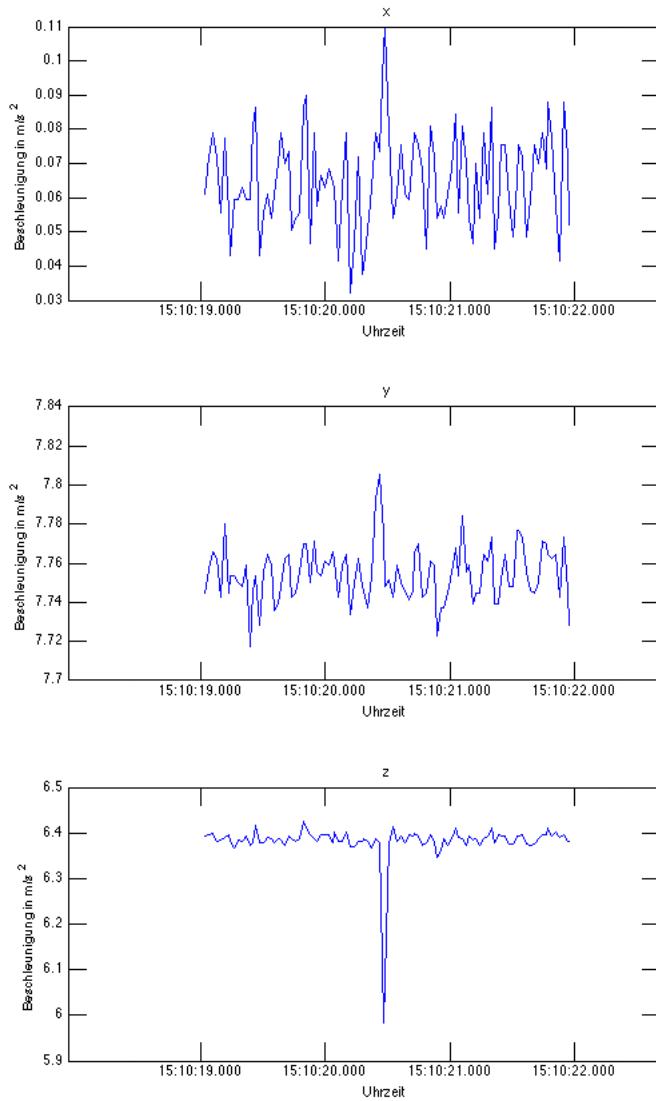


Abb. 17: Restituierte Beschleunigungsdaten in 3 Komponenten während Experiment 7.
Die X-Achse entspricht der Querachse (Nickachse) des Radars, die Y-Achse seiner Hochachse und die Z-Achse der Längsachse, welche in Radarstrahlrichtung ausgerichtet ist.

In Abbildung 17 sind exemplarisch die Beschleunigungsdaten aus Experiment 7 zu sehen. Es wurde mit 32 Hz aufgenommen und der Messbereich auf $\pm 1.5g$ eingestellt. (Auf

andere verweisen? alle anhängen??) Auf der Z-Achse ist ein deutlicher Ausschlag um ca. $0.4m/s^2$ zu sehen, welcher durch die Druckwelle der Explosion zustande kommt.

Die Bewegung der Schüssel wird allerdings lediglich durch ein einziges Sample erfasst. Es ist nicht sichergestellt, ob hier wirklich das Maximum an Beschleunigung abgebildet ist und wie die Bewegung genau verläuft. Eine Bestimmung der Geschwindigkeit oder des Weges durch Integration ist damit praktisch nicht möglich.

An dieser Stelle sei auch auf das Nyquist Theorem verwiesen. Es besagt, dass ein Signal immer mindestens mit der doppelten Frequenz des hochfrequentesten Anteils im Signal abgetastet werden muss, damit es komplett rekonstruiert werden kann. (vgl. Shannon, 1949)

Betrachtet man lediglich das Sample, bei dem die Beschleunigung auftritt und interpoliert linear zu den nächsten beiden Samples, welche noch keine Beschleunigung anzeigen, so lässt sich dennoch eine Geschwindigkeit abschätzen. Diese ergibt sich demnach als:

$$v = \frac{1}{2} a \cdot t \quad (15)$$

Im betrachteten Experiment 7 ist $v = 0.012m/s$. Diese Geschwindigkeit ist sehr gering und kann vom Radar nicht aufgelöst werden. Aus den eben genannten Gründen ist dieses Ergebnis vorsichtig zu betrachten. Eine höhere Abtastrate ist notwendig um sicherere Aussagen treffen zu können.

Vergleicht man den Zeitpunkt des Maximalausschlags (15:10:20.465 Uhr) mit dem vom Radar ermittelten Explosionszeitpunkt (15:10:20.274 Uhr), so ergibt sich ein Unterschied von 191 Millisekunden. Diese Differenz lässt sich nicht durch die langsamere Ausbreitungsgeschwindigkeit der Druckwelle erklären. Es gab offensichtlich ein Problem mit der Zeitsynchronisation, so dass das Radar nicht zu jedem Zeitpunkt eine korrekte GPS Zeit besaß.

Korrelation zwischen Geschwindigkeit (Radargemessen) und gemessener Beschleunigung? bringt mit den 3 Experimenten, wo alles gemeinsam zu sehen ist wenig...

7.2 Messungen am Autoklavenaufbau



Abb. 18: Messaufbau: Bewegungsmessung des Autoklavenaufbaus. Der Beschleunigungssensor ist am Bein des Aufbaus befestigt (roter Kreis).

8 Entwicklung des Schwingungsmesssystems: ASC 5511LN-002

Mit den bisher gewonnenen Erkenntnissen konnte ich nun das zweite Schwingungsmesssystem mit dem ASC Chip (ASC 5511LN-002) realisieren. Durch die höhere Auflösung und Samplerate und geringeres Rauschen, ergibt sich ein besser nutzbares Signal. Einen Vergleich der relevanten technischen Daten enthält Tabelle 2.

Beschleunigungsaufnehmer	Auflösung	Noise	Temperatur Drift
BMA180	0.244 mg	$150\mu g/\sqrt{Hz}$	0.5 mg/k
ASC 5511LN-002	0.077 mg	$5\mu g/\sqrt{Hz}$	0.2 mg/k

Tab. 2: Vergleich der relevanten technischen Daten der beiden Sensoren bezogen auf einem Messbereich von $\pm 2g$.

Das Auflösungsvermögen des ASC 5511LN-002 bezieht sich hier auf die Verwendung mit dem *Diamond-MM-16-AT PC/104 Analog I/O Module*

8.1 ASC 5511LN-002



Abb. 19: ASC 5511LN-010 von Advanced Sensors Calibration

Bei dem ASC 5511LN-002 handelt es sich ebenfalls um einen kapazitiven MEMS Sensor. Er ist allerdings von hochwertigerer Qualität als der BMA180 und gibt das Signal analog als Spannung aus. Aus diesem Grund wird ein externer AD-Wandler benötigt. Durch das Aluminiumgehäuse ist die Elektronik bereits gut geschützt und die symmetrische Ausführung der Signalkabel sorgt dafür, dass äußere elektromagnetische Einflüsse

auf dem Weg zum AD-Wandler vermieden werden. Das Signal wird als auf zwei Leitungen ausgegeben, wobei das Signal auf der einen Leitung invertiert ist (Abb. 21). Störungen wirken auf beide Adern und somit kann man sie an der Empfangsstation per Differenzbildung leicht entfernen.

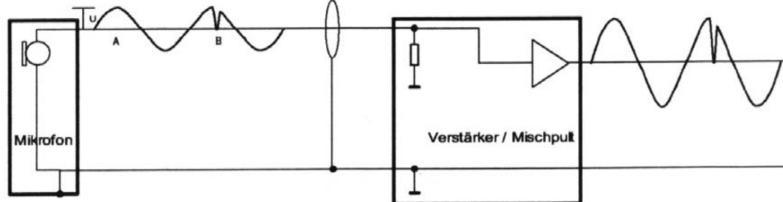


Abb. 20: Unsymmetrische Signalübertragung mit Störung 'Zacke'. (Sengpiel, 2001)

Bei unsymmetrischer Signalführung wird das Störsignal ebenso stark verstärkt, wie das Nutzsignal (Abb. 20). Mathematisch lässt sich dies so ausdrücken (Sengpiel, 2001):

$$U_a = v \cdot (U_e + U_{stoer} = v \cdot U_e + v \cdot U_{stoer}) \quad (16)$$

wobei U_a dem Ausgangssignal, U_e dem Eingangssignal, U_{stoer} dem Störsignal und v dem Verstärkungsfaktor entspricht.

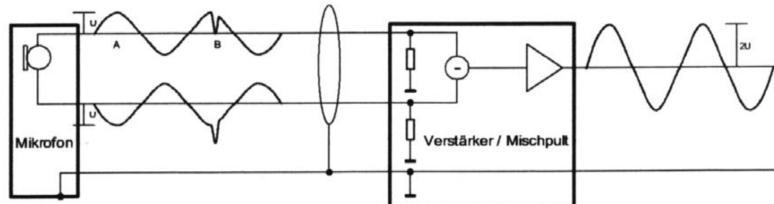


Abb. 21: Symmetrische Signalübertragung mit sich aufhebender Störung. (Sengpiel, 2001)

Symmetrische Signalführung hingegen sorgt dafür, dass sich der Störeinfluss auslöscht:

$$U_a = U_e + U_{stoer} - (-U_e + U_{stoer}) = 2 \cdot U_e \quad (17)$$

Das Resultat ist ein besseres Signal zu Rausch Verhältnis.

8.2 Diamond-MM-16-AT PC/104 Analog I/O Module

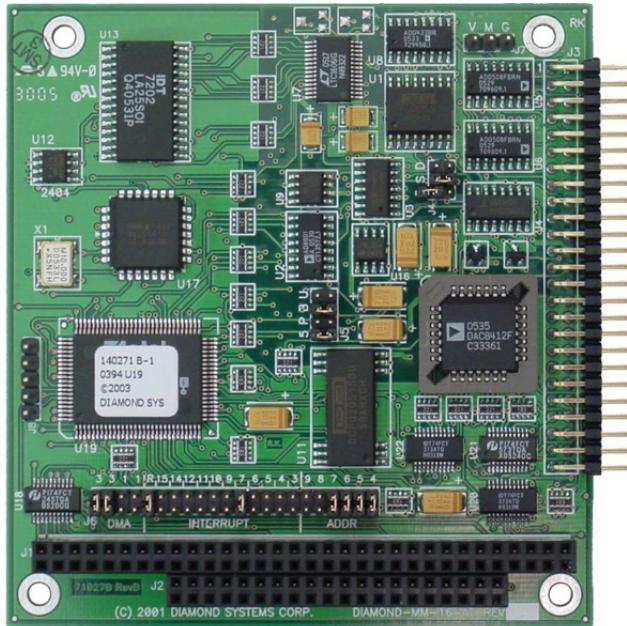


Abb. 22: Diamond-MM-16-AT PC/104 Analog I/O Module

Als AD-Wandler verwende ich ein *Diamond-MM-16-AT PC/104 Analog I/O Module*. Das Board verfügt über 8 Eingänge (16 wenn man sie nicht als Differentialeingänge verwendet) und eine maximale Samplingrate von 100kHz. Außerdem besitzt es einen 512-Sample *FIFO Speicher* und verwendet *Interrupts*, worauf ich in den beiden nächsten Punkten eingehen möchte. Mit dem Computer kommuniziert das Board über den *PC/104 Bus*.

8.2.1 Interrupts

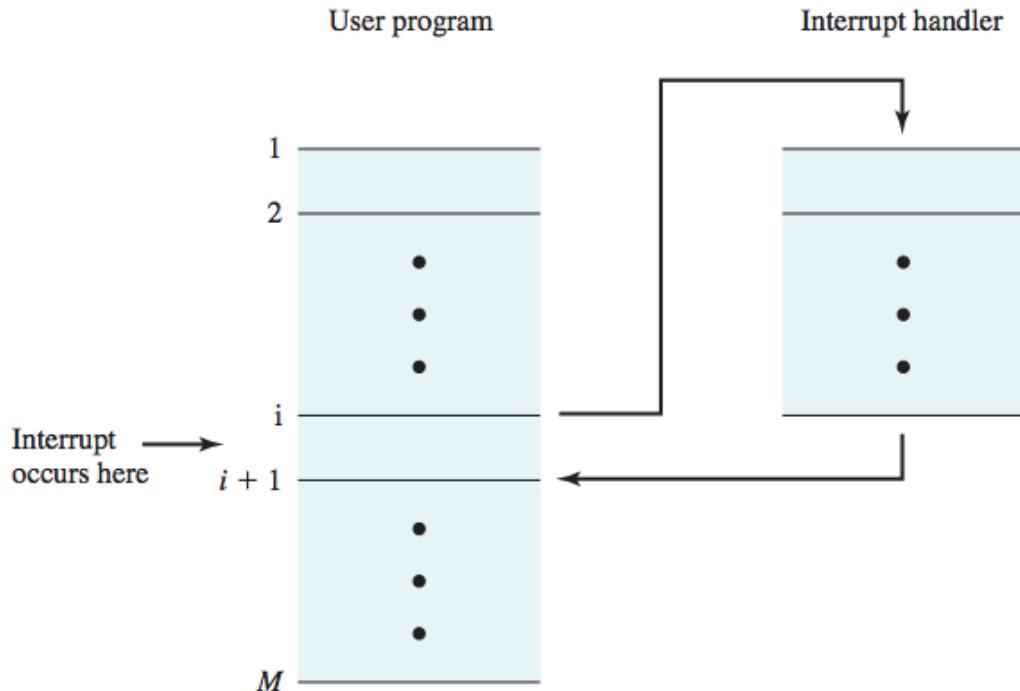


Abb. 23: Ablauf eines Interrupts (Stallings, 2000)

Ein Interrupt beschreibt in der Informatik eine kurzzeitige Unterbrechung des normalen Programmflusses, um eine andere Operation auszuführen. Diese andere Operation ist Zeitkritisch und ihr exaktes Auftreten nicht vorher bestimmbar. Sobald sie beendet ist, setzt das Programm seine Ausführung an der vorher unterbrochenen Stelle fort.

Genutzt werden Interrupts zum Beispiel von Ein- und Ausgabegeräten, wie Maus und Tastatur. Ohne Interrupts müssten alle Programme zyklisch nachfragen, ob es eine neue Eingabe gibt und diese dann entsprechend bearbeiten (sogenanntes Polling). Da ein Programm in dieser Zeit nichts anderes machen kann, ist diese Art der Abfrage höchst ineffizient.

Die am weitesten verbreitete Analogie zur Verdeutlichung des Prinzips ist eine Wohnungstür mit Klingel. Man kann den ganzen Tag in der Wohnung seinen Aufgaben nachgehen und zwischendurch klingeln Gäste, mit denen man sich dann kurz beschäftigt. Das Klingeln ist hier also der Interrupt. Dann kann man wieder weiter mit der Arbeit machen bis der nächste Klingelt. Hat man nun aber keine Klingel, so muss man ständig zur Tür rennen und nachsehen, ob eventuell jemand dort steht und gerne hereinkommen möchte (Polling).

An dieser Analogie kann man auch erkennen, dass Interrupts zusätzliche Hardware (eine Klingel) erfordern.

8.2.2 FIFO Speicher

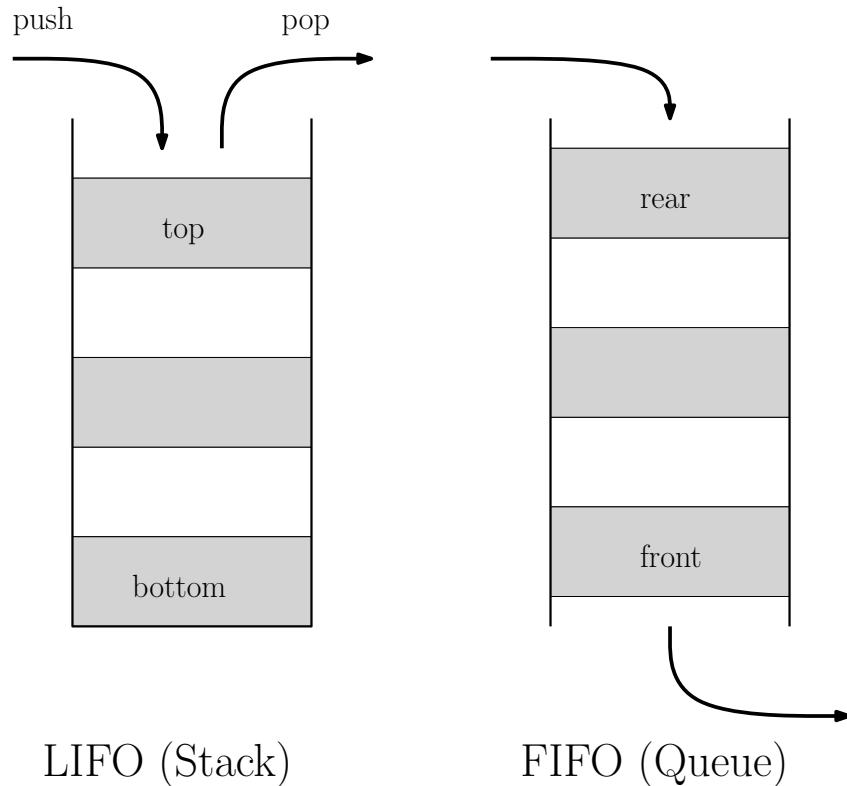


Abb. 24: Vergleich von FIFO und LIFO

Ein FIFO (First In First Out) ist ein Speicherverfahren, bei dem die Elemente, die als erstes gespeichert werden, auch als erstes wieder entnommen werden. Es wird auch als Warteschlangenprinzip (Queue) bezeichnet und in vielen Bereichen der Datenverarbeitung verwendet. Besonders bei Schnittstellen und Kommunikationsprotokollen ist es wichtig, dass die Daten in der Reihenfolge verarbeitet werden, in der sie angekommen sind, beziehungsweise abgesendet wurden. Das gegensätzliche Prinzip ist das LIFO (Last In First Out), auch Stapelprinzip (engl. Stack) genannt.

Im AD-Wandler sorgt der FIFO dafür, dass wirklich alle digitalisierten Samples vom Computer gespeichert werden, auch wenn dieser aufgrund von starker Prozesserbelastung nicht in der Lage ist diese sofort zu verarbeiten. Die digitalisierten Daten werden unabhängig vom PC immer sofort in den FIFO gespeichert und können dort angefordert

werden. Ein Problem ergibt sich erst wenn der PC die Sample so langsam ausliest, dass der FIFO Speicher voll wird. In diesem Fall spricht man von einem Speicherüberlauf und bereits aufgenommene Daten werden überschrieben.

Zusammen mit Interrupts erlaubt der FIFO Speicher eine konstante Messwertdigitalisierung mit hoher Frequenz, ohne den Prozessor des PCs zu stark zu beanspruchen. Dazu werden die Daten kontinuierlich vom Board in den FIFO geschrieben und sobald ein festgelegter Grenzwert (threshold) erreicht ist, löst es einen Interrupt aus. Darauf reagiert nun der PC, indem er das komplette Datenpaket auf einmal (also so viele Samples, wie der threshold Wert eingestellt ist) aus dem FIFO liest und auf seine Festplatte speichert. Währenddessen werden weiterhin Daten digitalisiert und es geht kein Sample verloren. Würde bei jedem neuen Sample ein Interrupt ausgelöst werden und die Daten jedes Mal einzeln abgerufen und gespeichert, so würde dies bei hohen Abtastfrequenzen zu extremer Prozessorbelastung führen. Der FIFO dient also gleichzeitig auch zur Verringerung der Interrupt Rate:

$$\text{Interrupt Rate} = \frac{\text{A/D Frequenz} \cdot \text{Anzahl der aufgenommenen Kanäle}}{\text{FIFO threshold}} \quad (18)$$

Bei einem Kanal und einer Abtastfrequenz von 100000Hz resultiert, bei einem threshold von 256, ein Unterschied von 100000 zu ca. 391 Interrupts pro Sekunde.

8.3 Software

Für das Diamond Board existiert die *Universal Driver Software*⁶, eine C-basierte Programmzbibliothek, die eine große Zahl von Funktionen zur Datenaquisition zur Verfügung stellt. Die Low-Level-Programmierung, bei der auf die einzelnen Register zugegriffen wird, entfällt damit.

Bei der Entwicklung habe ich mich an der *DSCADScanInt Demo*⁷ aus den Beispielen der Treibersoftware orientiert.

Ich möchte wiederum nur die wichtigsten Teile des Quellcodes erläutern. Der komplette Code findet sich im Anhang.

Das Programm wird mit zwei Parametern aufgerufen: Dem Namen des Logfiles, in das

⁶<http://www.diamondsystems.com/products/dscud>

⁷http://files.diamondsystems.com/cdrom/Software/Universal%20Driver/Demos/DMM16-AT_5.91_Demo.zip (September 2012)

die Daten geschrieben werden und der Anzahl an Samples, die aufgenommen werden sollen.

Wenn im folgenden von einem Buffer gesprochen wird, so ist damit nicht der FIFO gemeint, sondern ein Speicher, in dem die Samples gespeichert werden, sobald sie aus dem FIFO herauskommen. Dieser Buffer wird nur von Interrupts beschrieben, daher wird zusätzliche Logik benötigt, um die Daten in der richtigen Reihenfolge aus diesem Buffer auszulesen.

Der Programmablauf ist folgendermaßen gegliedert:

1. Initialisierung des Treibers
2. Initialisierung des Boards (mit Bus-Adresse und Interrupt Level)
3. Konfigurationsvariablen für den AD-Wandler setzen
4. Konfigurieren des Interrupts und des FIFOs
5. Loop, der alle x Sekunden überprüft, ob neue Daten vorliegen
 - Daten in der richtigen Reihenfolge aus dem Buffer lesen und samt Zeitstempel in eine Datei schreiben

Initialisierung des Treibers

In Listing 5 wird zunächst der Treiber für das Board initialisiert, dabei wird überprüft, ob die korrekte Treiberversion installiert ist.

```

1 if( dscInit( DSC_VERSION ) != DE_NONE )
2 {
3   dscGetLastError(&errparams);
4   fprintf( stderr , "dscInit error: %s %s\n" , dscGetErrorString( errparams
      .ErrCode) , errparams.errstring );
5   return 0;
6 }
```

Listing 5: Initialisierung der Treibersoftware

Initialisierung des Boards

Als nächstes wird das Board selbst initialisiert (Lst. 6). Dazu notwendig sind die Bus-Adresse (Zeile 3) und das Interrupt-Level (Zeile 4), die auf dem Board per Jumper eingestellt werden können. Es ist dadurch möglich mehrere Boards gleichzeitig in einem Computer zu betreiben.

```

1 printf( "\nDMMI6AT BOARD INITIALIZATION:\n" );
2
3 dsccb.base_address = 0x300;
4 dsccb.int_level = (BYTE) 7;
5
6 if(dscInitBoard(DSC_DMMI6AT, &dsccb, &dscb)!= DE_NONE)
7 {
8     dscGetLastError(&errparams);
9     fprintf( stderr, "dscInitBoard error: %s %s\n", dscGetErrorString(
10         errparams.ErrCode), errparams.errstring );
11 }

```

Listing 6: Initialisierung des Boards

Der Ablauf der Konfiguration ist in allen Punkten einheitlich. Zunächst werden die entsprechenden Werte in eine Struktur geschrieben, dann wird diese an das Board übergeben. Sollten dabei Fehler auftreten, so werden diese ausgegeben und das Programm bricht ab.

Setzen der Konfigurationsvariablen für den AD-Wandler

Der AD-Wandler kann in verschiedenen Messbereichen arbeiten und außerdem eine gewisse Signalverstärkung (gain) vornehmen. Die dazu notwendigen Einstellungen sind in Listing 7 aufgeführt. Der *ASC 5511LN-002* gibt eine maximale Spannung von 5V aus, daher ist RANGE_5 der richtige Eintrag. In unserem Fall muss die Spannung auch nicht verstärkt werden.

Über eine längere Zeit und vor allem mit der Änderung der Temperatur neigen AD-Wandler dazu, eine gewisse Drift aufzuweisen. Dies liegt daran, dass sich der Ohmsche Widerstand, welcher als Referenz zur gemessenen Spannung dient, mit der Temperatur ändert. Das *Diamond-MM-16-AT* verfügt über einen Autokalibrationsmechanismus. Indem es vorher exakt bekannte Ohmsche Widerstände misst und die Messungen mit den bekannten Werten vergleicht, kann die genaue Drift bestimmt und herausgerechnet werden (siehe auch Miller, 2006).

Ob diese Autokalibration genutzt werden soll, legt `load_cal` in Zeile 4 fest.

Zeile 5 setzt den aktuellen Kanal, bei dem die Digitalisierung beginnen soll fest.

```

1 dscadsettings.range = RANGE_5;
2 dscadsettings.polarity = UNIPOLAR;
3 dscadsettings.gain = GAIN_1;

```

```

4 dscadsettings.load_cal = (BYTE)TRUE;
5 dscadsettings.current_channel = 0;

```

Listing 7: Konfigurieren des AD-Wandlers

Konfigurieren des Interrupts und des FIFOs

An dieser Stelle (Lst. 8) wird festgelegt, dass der FIFO benutzt werden soll (Zeile 9) und wie groß der threshold ist, bei dem ein Interrupt ausgelöst wird (Zeile 11). Die Größe des FIFOs (**fifo_depth**) ist durch die Hardware auf 256 Byte festgelegt.

Low_channel (Zeile 5) und **high_channel** (Zeile 6) geben an, welche der 8 Eingänge des Boards genutzt werden sollen. Obwohl der Beschleunigungssensor nur 3 Komponenten hat, verwende ich hier 4 Kanäle, da dies die Ausleselogik vereinfacht.

In Zeile 2 wird die Samplingrate festgelegt. Sie ist immer auf alle Kanäle bezogen, so dass sich bei einer Samplingfrequenz von 1000 Hz und 4 Kanälen eine letztendliche Abtastrate von 250 Mal pro Sekunde ergibt.

Theoretisch lassen sich mit dem Board Samplingraten von bis zu 100 000 Mal pro Sekunde erreichen. Dazu ist jedoch eine genaue Kalibrierung aller Parameter und weitere Codeoptimierung notwendig, was nicht mehr Teil dieser Arbeit sein soll.

```

1 dscaioint.num_conversions = num_conversions;
2 dscaioint.conversion_rate = 1000;
3 dscaioint.cycle = (BYTE)TRUE;
4 dscaioint.internal_clock = (BYTE)TRUE;
5 dscaioint.low_channel = 0;
6 dscaioint.high_channel = 3;
7 dscaioint.external_gate_enable = (BYTE)FALSE;
8 dscaioint.internal_clock_gate = (BYTE)FALSE;
9 dscaioint fifo_enab = (BYTE)TRUE;
10 dscaioint fifo_depth = 256;
11 dscaioint.dump_threshold = 256;

```

Listing 8: Setzen der Variablen für den Interrupt

In Zeile 3 lässt sich zwischen *One-Shot-Mode* und *Recycle-Mode* wechseln. Im One-Shot-Mode wird ein Buffer ein Mal mit Daten gefüllt und danach beendet sich die Routine. **Num_conversions** gibt dabei die Größe des Buffers an, und damit wie viele Samples aufgenommen werden sollen.

Im Recycle-Mode wird bei Erreichen des Bufferendes zum Anfang des Buffers gesprungen und dort die alten Daten überschrieben. Der Parameter **num_conversions** gibt in diesem

Fall die Größe des Buffers an. Somit enthält dieser immer die letzten n Samples, wobei n die Buffergröße ist.

Um herauszufinden, an welcher Stelle im Buffer grade geschrieben wurde, dienen die beiden Variablen `DSCS.transfers` und `DSCS.total_transfers`. `DSCS.transfers` gibt an, wie viele Samples im aktuellen Zyklus bereits im Buffer gespeichert wurden und wird immer auf 0 gesetzt wenn ein neuer Zyklus beginnt. `DSCS.total_transfers` gibt die Gesamtzahl an bisher digitalisierter Werte in der gesamten Operation an.

Damit ist es möglich, die Daten in der korrekten Reihenfolge in eine Datei zu speichern.

Die weiteren in Listing 8 aufgeführten Variablen sind für unseren Anwendungsfall nicht relevant.

Auslesen und Speichern der Daten

Wie bereits beschrieben verläuft das Speichern der neuen Samples in den zirkulären Buffer automatisch durch die Interrupts. Sobald der `dump_threshold` erreicht ist, werden die Daten vom FIFO in den Buffer übertragen. Um sie von dort aus in eine Datei zu schreiben, bedienen wir uns einer Programmschleife, die mehrmals pro Sekunde überprüft, ob neue Samples vorliegen (Lst. 9).

```

1 DWORD sleep_ms = 300;
2
3 do {
4     dscSleep(sleep_ms);
5     dscGetStatus(dscb, &dscs);
6
7     if (dscs.overflow) {
8         printf("Operation failed: FIFO overflowed\n");
9         break;
10    }
11
12    if (dscs.total_transfers == last_total_transfers) {
13        printf("Operation failed: no new samples taken in %d ms\n",
14            sleep_ms);
15        break;
16    }
17    new_sample_count = dscs.total_transfers - last_total_transfers;
18
19    /* Number of new samples should never exceed the size of the
       circular buffer. If it does it means that either "sleep_ms"

```

```

should be smaller so you check status more often, or "
num_conversions" should be bigger so the circular buffer is
bigger */

20    if ( new_sample_count > num_conversions ) {
21        printf("Operation failed: not processing data fast enough. %d
22              samples lost\n",
23              new_sample_count - num_conversions);
24        break;
25    }

26 /* AUSLESEN UND SPEICHERN DER DATEN */
27 /* FOLGT IM NAECHSTEN LISTING */

28

29     last_transfers = dscs.transfers;
30     last_total_transfers = dscs.total_transfers;

31

32     if ( dscs.total_transfers >= stop_after_transfers )
33         break;

34

35 } while ( dscs.op_type != OP_TYPE_NONE );
36
37 dscCancelOp( dscb );

```

Listing 9: Programmschleife, die den Buffer auf neue Daten überprüft und diese in eine Datei schreibt. Der Code zum Auslesen und Schreiben der Samples ist in diesem Listing entfernt und separat in Listing 10 und 11 abgebildet.

In Zeile 1 wird mit `sleep_ms = 300` festgelegt, dass die Schleife ca. 3 Mal pro Sekunde ausgeführt wird. Zunächst wird überprüft, ob der FIFO übergelaufen ist (Zeile 7), es überhaupt neue Samples seit des letzten Schleifendurchlaufs gab (Zeile 12) und ob die Daten rechtzeitig aus dem Buffer in eine Datei geschrieben wurden (Zeile 17 und 20). Schlägt einer dieser Checks fehl, so wird das Programm mit einer entsprechenden Fehlermeldung beendet.

Treten keine Fehler auf, werden die Daten aus dem zirkulären Buffer herausgelesen und in eine Datei gespeichert. Dies geschieht so lange, bis die Zahl der aufzunehmenden Samples erreicht ist (Zeile 32 und 33).

Da der Buffer zirkulär ist, also beim Erreichen des Endes wieder vom Anfang beschrieben wird, gibt es zwei Möglichkeiten wo die neuen Daten darin gespeichert wurden. Im einfachen Fall befinden sie sich zwischen der zuletzt ausgelesenen Position und dem

Bufferende. Dieser Fall wird der Code in Listing 10 ausgeführt.

```

1 if ( dscs.transfers > last_transfers ) {
2   for ( i = last_transfers; i < dscs.transfers; i++ ) {
3     if ( i%4 == 0 ) {
4       uhr = mvTime::Now(); // get actual time
5       for ( k = 0; k < 3; k++ ) { // conversion into volts
6         dscADCodeToVoltage(dscb, dscadsettings, dscaoint.sample_values[
7           i+k], &voltage[k]);
8         fprintf(logFile, "%s counts: %5d %5d %5d Volts: %5.3lf %5.3lf
9           %5.3lf\n", uhr.GetTimeStamp().c_str(), dscaoint.sample_values[
10             i], dscaoint.sample_values[i+1], dscaoint.sample_values[i+
11               2], voltage[0], voltage[1], voltage[2]);
12     }
13   }
14 }
```

Listing 10: Auslesen und Speichern der Daten. Fall 1: Im zirkulären Buffer wurden neue Samples nur zwischen der zuletzt ausgelesenen Position und dem Ende des Speichers geschrieben.

In Zeile 1 findet die Überprüfung statt, ob die aktuelle Position im Buffer (`dscs.transfers` größer ist als die zuletzt ausgelesene Position `last_transfers` (Also Fall 1 eingetreten ist). Um die nächsten Zeilen zu verstehen, ist es wichtig zu wissen, wie die einzelnen Samples im Buffer strukturiert sind. Abbildung 25 zeigt diese Struktur.

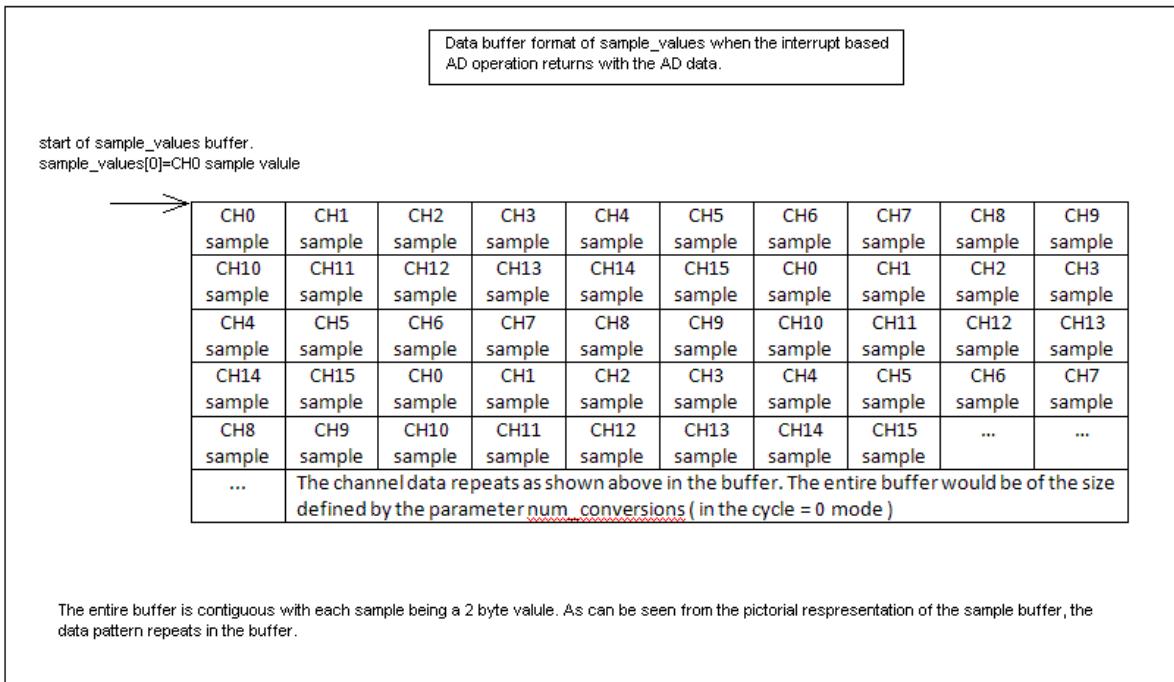


Abb. 25: Struktur des Buffers, in dem die Samples stehen. (Systems, 2008)

Alle Kanäle werden hintereinander gespeichert. Um diese nun nach Kanal sortiert auszugeben, nutze ich in Zeile 3 den Modulo Operator damit i immer Kanal 0 enthält und die anderen Kanäle mit $i + 1$ und $i + 2$ auszulesen sind.

Die Werte müssen noch von AD-Counts mittels der Funktion `dscADCodeToVoltage` in Volt umgewandelt werden (Zeilen 5 bis 7).

Das Schreiben in eine Datei findet in Zeile 8 statt. Dabei werden sowohl die umgewandelten Voltzahlen als auch die ursprünglichen Counts gespeichert und mit einem Zeitstempel versehen.

Im zweiten Fall wurden neue Daten sowohl nach der zuletzt ausgelesenen Position geschrieben, als auch wieder an den Anfang des Buffers. Fall 2 ist in Listing 11 zu sehen und beinahe identisch mit Fall 1.

Es gibt allerdings zwei for Schleifen. Die erste liest Samples, die nach der zuletzt gelesenen Position gespeichert wurden (Zeile 2) und die zweite liest die Samples, welche sich zwischen dem Anfang und der aktuellen Position befinden (Zeile 10).

```

1 } else if ( dscs.transfers <= last_transfers ) {
2   for ( i = last_transfers; i < num_conversions; i++ ) // after the
      last_transfers
3     if ( i%4 == 0 ) {

```

```

4      uhr = mvTime::Now();      // get actual time
5      for ( k = 0; k < 3; k++ ) { // conversion into volts
6          dscADCodeToVoltage(dscb, dscadsettings, dscaioint.
7              sample_values[i+k], &voltage[k]);
8      }
9      fprintf(logFile, "%s counts: %5d %5d %5d Volts: %5.3lf %5.3lf
10         %5.3lf \n", uhr.GetTimeStamp().c_str(), dscaioint.sample_values
11         [i], dscaioint.sample_values[i+1], dscaioint.sample_values[i
12         +2], voltage[0], voltage[1], voltage[2]);
13      if ( i%4 == 0 ) {
14          uhr = mvTime::Now();      // get actual time
15          for ( k = 0; k < 3; k++ ) { // conversion into volts
16              dscADCodeToVoltage(dscb, dscadsettings, dscaioint.
17                  sample_values[i+k], &voltage[k]);
18      }

```

Listing 11: Auslesen und Speichern der Daten. Fall 2: Im zirkulären Buffer wurden neue Samples zwischen der zuletzt ausgelesenen Position und dem Ende und zusätzlich am Anfang des Speichers geschrieben.

Die Zeitstempel erhalte ich durch eine Methode aus der `mvTime` Klasse. Diese stammt aus der Radarserver Software von Malte Vöge.

Der Nachteil bei dieser Art der Datenaufnahme ist, dass ein neuer Zeitstempel nur bei jedem Schleifendurchlauf (drei Mal pro Sekunde) entsteht. Es haben daher immer mehrere Samples den gleichen Zeitstempel. In der Auswertung muss also zunächst ein neuer Zeitstempel generiert werden.

9 Experimente am Geomatikum

10 Verbesserungen

bessere Zeitstempel, nicht nur 3 Mal pro Sekunde

Literatur

Datasheet ADXL05. *1g to 5g Single Chip Accelerometer with Signal Conditioning*, 1996.

Arduino. Arduino wire library, 2012. URL
<http://www.arduino.cc/en/Reference/Wire>.

I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt am Main, 2. überarbeitete und erweiterte Auflage edition, 1995.

Yozo Kanda. Piezoresistance effect of silicon. *Sensors and Actuators A: Physical*, 28(2): 83–91, July 1991.

Lasse Klingbeil. *Entwicklung eines modularen und skalierbaren Sensorsystems zur Erfassung von Position und Orientierung bewegter Objekte*. PhD thesis, Universität Bonn, 2006.

D. Meschede. *Gerthsen Physik*. Springer Berlin Heidelberg, 12. Auflage edition, September 2001.

Jonathan Miller. *The benefits of autocalibration*. Diamond Systems, 2006.

NXP. *UM10204 I2C-bus specification and user manual*. NXP, rev4 edition, February 2012.

Schmidt. *Sensorschaltungstechnik*. Vogel, 2002.

Eberhard Sengpiel. Symmetrische und unsymmetrische signalübertragung. Tutorium UdK Berlin, March 2001. URL
<http://www.sengpielaudio.com/SymmetrischeUndUnsymmetrischeSignaluebertragung.pdf>.

Bosch Sensortec. *BMA180 Digital, triaxial acceleration sensor Data sheet*, 2.1 edition, December 2009.

Claude E. Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, January 1949.

S.J. Sherman, W.K. Tsang, T.A. Core, R.S. Payne, D.E. Quinn, K.H.-L. Chau, J.A. Farash, and S.K. Baum. A low cost monolithic accelerometer; product/technology update. In *Electron Devices Meeting, 1992. IEDM '92. Technical Digest., International*, pages 501–504, 1992.

Oliver Spieler, Ben Kennedy, Ulrich Kueppers, Donald B. Dingwell, Bettina Scheul, and Jacopo Taddeucci. The fragmentation threshold of pyroclastic rocks. *Earth and Planetary Science Letters*, 226(1-2):139–148, September 2004.

William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall International, 6 edition, 2000.

Diamond Systems. *Universal Driver Manual v5.92*, 2008.

Ulrich Tietze and Christoph Schenk. *Halbleiter-Schaltungstechnik*. Number 3-540-42849-6. Springer, Berlin, 12 edition, 2002.