MEDIZINISCHE UNIVERSITÄT WIEN

TU WIEN

D I P L O M A R B E I T

FES - Wave

Messsystem zur Untersuchung der Kontraktionseigendynamik FES-aktivierter Oberschenkelmuskulatur bei höheren Stimulationsfrequenzen

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Diplom - Ingenieurs

Betreuer: A.o. Univ.-Prof. DI Dr. Dietmar Rafolt
Zentrum für biomedizinische Technik und Physik
Medizinische Universität Wien

Verfasser: Matthias Krenn
Matrikelnummer: 9925568
3644 Emmersdorf, Westsiedlungsstraße 13

Wien, am 8. April 2007

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt. Ich versichere an Eides statt, dass ich nach bestem Wissen die reine Wahrheit gesagt und nichts verschwiegen habe.

(Ort, Datum) (Unterschrift)

Für meine Familie

An dieser Stelle möchte ich mich besonders bei meinem Betreuer Prof. Dr. Dietmar Rafolt bedanken, für seine Hilfestellungen und für die interessanten Diskussionen zu später Stunde.

Kurzfassung

Bei der funktionellen Elektrostimulation (FES), im Speziellen der transkutanen Nervstimulation (TENS) des Quadrizepsmuskels kann bei höheren Stimulationsfrequenzen oberhalb ca. 30Hz - 50Hz bei einer entsprechenden Stimulationsamplitude eine Eigendynamik der Kontraktion beobachtet werden. Die einzelnen Muskelpartien des 4-köpfigen Muskels kontrahieren nicht glatt, sondern beginnen voneinander mehr oder weniger unabhängig in chaotischer Folge zu kontrahieren. Bei weiterer Erhöhung der Amplitude verschwindet diese Kontraktionswelle („FES - Wave“) wieder, um in einer gleichmäßigen, satten, oft schmerzhaften Kontraktion zu enden.

In der vorliegenden Arbeit wird die Entwicklung einer geeignete Stimulations- und Messausrüstung zur genaueren Untersuchung dieses Phänomens beschrieben. Es wurde ein Stimulator entwickelt, der einfache, manuell einstellbare Stimulationssignale bis hin zu sehr komplexen Stimulationsmustern und Experimentfolgen abgeben kann. Bis zu fünf unabhängige Muster können am PC editiert und in das Gerät geladen werden. Damit ist es beispielsweise möglich, mehrere Ermüdungsprotokolle gefolgt von Messprotokollen automatisch ablaufen zu lassen. Zur Aufzeichnung der „FES - Wave“ wurde ein 30-kanaliges Akzelerometriesystem entwickelt. Erste Testmessungen wurden mit fünf 3-achsigen Beschleunigungssensoren durchgeführt.

Inhaltsverzeichnis

KAPITEL 1 Einleitung

1.1. Beschreibung des Phänomens „FES - Wave“

Die funktionelle Elektrostimulation (FES) hat in den letzten Jahren einen festen Platz zur Unterstützung von Rehabilitation und Training eingenommen. Unter FES versteht man das Auslösen von unwillkürlichen Muskelkontraktionen durch elektrische Stromimpulse. Die Impulse werden mit Stimulationsfrequenzen zwischen 20Hz und 50Hz angewendet und haben sich als ideal Kompromiss zwischen Kraftstärke und Ausdauer erwiesen. Bereits schon ab 30 Hz - 50 Hz kann nun ein Phänomen beobachtet werden, welches als Muskelkontraktionsoszillationen („FES - Wave“) umschrieben werden könnte. Die einzelnen Muskelpartien des 4-köpfigen Quadrizeps Muskels kontrahieren nicht glatt, sondern beginnen ab einem bestimmten Amplitudenbereich voneinander mehr oder weniger unabhängig in chaotischer Folge zu oszillieren. Lokale Kontraktionsschwankungen wandern bzw. oszillieren scheinbar zweidimensional über die gesamte Muskeloberfläche. Bei weiterer Erhöhung der Amplitude verschwindet diese Kontraktionswelle („FES - Wave“) wieder, um in einer gleichmäßigen, satten, oft schmerzhaften Kontraktion zu enden.

Diese Schwingungen sind optisch gut wahrnehmbar, da sie mit einer deutlichen Deformation des Muskels bzw. des Muskelbauches mit einhergehen. Das Typische dabei ist, dass die einzelnen Muskelpartien, ja scheinbar sogar einzelne Muskelabschnitte voneinander unabhängig oszillieren. Die wichtigste Ursache für das Auftreten dieser chaotischen Prozesse dürften nichtlineare Überlagerungen und Rückkoppelungen sein, die durch Volumsdeformationen und damit verbundenen Feldverzerrungen hervorgerufen werden. Die Verschiebung der Motorpoints (= jene Stelle, bei der der Nerv in den Muskel eintritt) unter den Elektroden dürfte in diesem Zusammenhang ebenfalls eine entscheidende Rolle spielen. Das Auftreten dieser Schwingungen ist sehr eng mit dem Problem der richtigen Dosierung der FES-Amplitude und der damit verbundenen Gefahr eines möglichen Muskelkrampfes und in der folge übermäßigen Muskelkaters verbunden, wodurch ein optimales Training gestört wird.

Generell stellt das Thema der "Ermüdung" einen zentralen Punkt in der funktionellen Elektrostimulation dar. Wie hoch und wie lange kann eine Krafterzeugung sein, ohne dass der Muskel zu schnell ermüdet oder gar eine Schädigung erfährt? - Wie sollte ein Kraft- bzw. Ausdauertraining aussehen? - Welche Intervalle sind optimal? usw. Diese Fragen beschäftigen nach wie vor zahlreiche Studien.

Das Auftreten der Kontraktionsoszillationen soll als Indikator des Muskelzustandes helfen, diese Fragen zu beantworten. Grundlegende Untersuchungen müssen zu diesem Zweck durchgeführt werden, wofür ein adäquates Stimulations- und Messinstrument zur Verfügung gestellt werden muss. Die Entwicklung dieser Ausrüstung ist Inhalt dieser Diplomarbeit.

Das Stimulationsgerät muss für diese Anforderungen einen hohen Grad an Flexibilität aufweisen. Außer der Generierung der Frequenzen, bei denen die FES - Wave auftritt, sollen auch Pulsmuster zur Verfügung stehen, die die Ermüdungsdetektion mit Hilfe der M - Wave Analyse erlauben. Dazu sind beispielsweise Einzelimpulsrampen mit steigender Amplitude notwendig. Das periodische Umschalten zwischen 2 Stimulationsfolgen mit dazugehörigen Amplituden sollte ebenfalls automatisch erfolgen, um verschiedene Wechselspiele von Ermüdung und Erholung zu testen. Zu dem Zweck wurde ein sehr komfortabler Editor für den PC oder Notebook entwickelt, der es dem Experimentator erlaubt, seine eigenen Pulsmuster und -abfolgen in nahezu beliebiger Kombination und Dauer zu generieren. 5 unabhängige „Experimente“ können in den Stimulator transferiert werden, die dann vom PC unabhängig verwendet werden können. Der Stimulator beinhaltet 2 Stimulationskanäle, die synchron oder asynchron betrieben werden können.

Zur Aufzeichnung der Muskelbewegungen wurde ein 30-kanaliges Akzelerometrie-system entwickelt. Zehn dreidimensionale Beschleunigungsaufnehmer werden somit zur gleichzeitigen Messung auf der Muskelgruppe platziert. Ein weiterer Analogeingang wird mit einem Sensorsignal zur Messung der Extensionskraft belegt.

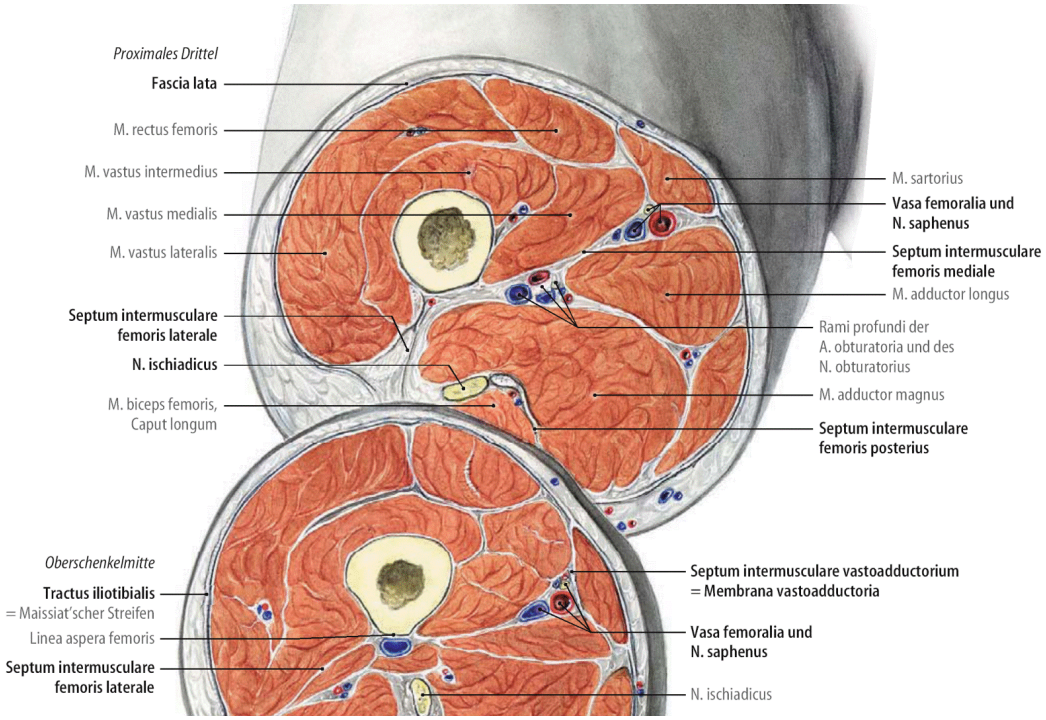


Abbildung 1.1: Querschnitt durch den Oberschenkel
Quelle: (Tillmann, 2005 S. 535)

Oberschenkel als Ziel der Untersuchungen:

Der Quadrizepsmuskel bestehend aus M. rectus femoris, M. vastus medialis, M. vastus intermedialis und M. vastus lateralis (Abb. 1.1) ist eine der wichtigsten Muskelpartien der Skelettmuskulatur und ist deshalb das primäre Ziel unserer Untersuchungen. Selbstverständlich können beliebig andere Muskelgruppen mit dem entwickelten Testsystem untersucht werden.

1.2. Systemüberblick

Die Abbildung 1.2 zeigt das gesamte Messsystem, wobei das Zusammenspiel der einzelnen Elemente dargestellt wird. Die folgende Aufzählung beschreibt die einzelnen Module überblicksmäßig.

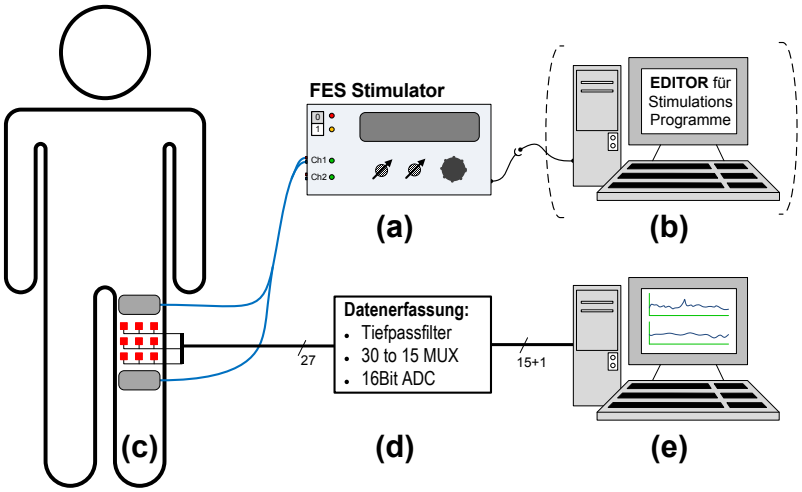


Abbildung 1.2: Schematische Darstellung des gesamten Messsystems
(a) Eingabe der Stimulationsprogramme, (b) 2-Kanal Stimulator, (c) Stimulationselektroden und Sensor-Array bestehend aus Beschleunigungssensoren, (d) Filter Array mit 32 zu 16 Multiplexer, (e) Messwertaufzeichnung am PC/Notebook

(a) Eingabe der Stimulationsprogramme:
Der Stimulator kann fünf verschieden Stimulationsmuster speichern, die von dieser GUI editiert werden könne. Die Eingabe Software verfügt über eine eigene Datenverwaltung mit der effizient Stimulationsprogramm gespeichert und verändert werden könne.

(b) Der 2-Kanal Stimulator liefert biphasische Rechteckimpulse, welche von einer spannungskonstanten Quelle erzeugt werden. Durch die modulare Aufbauweise des Stimulators, könne auch andere Endstufen eingesetzt werden.

(c) Das FES - Wave Phänomen wird mit bis zu zehn Beschleunigungssensoren gemessen. Üblicherweise werden neun Sensoren verwendet, damit an jedem Muskelstrang drei Sensoren angebracht werden können.

(d) Das Filter-Array hat die Aufgabe die Messsignale für die Ni-DAQ Karte vorzubereiten, das inkludiert einen Tiefpassfilter und
30 zu 15 - Multiplexer. Mit dem 16. Kanal wird das Taktsignal aufgezeichnet.

(e) Datenauswertung über DASYLab online Betrachtung der Messergebnisse, für weiter Verarbeitung wird MATLAB verwendet.

KAPITEL 2 Stimulator

Dieses Kapitel unterteilt sich im Wesentlichen in drei Abschnitte. Der erste Teil gibt eine kurze Einführung über die Anwendung verschiedener Elektrostimulatoren. Die weiteren Abschnitte beschäftigen sich konkret mit dem entwickelten Stimulator, wobei zuerst auf die Anforderungen eines Benutzers/einer Benutzerin eingegangen wird. Im Anschluss folgt die Funktionsbeschreibung des Stimulators (2.2) und des Burst-Editors (2.3). Der letzte Teil beinhaltet die technische Realisierung, welche nach den drei Gesichtspunkten – Hardware, µC-Software und Burst-Editor Software aufgeteilt wird.

2.1. Allgemeine Einführung

Unter FES versteht man das Auslösen von unwillkürlichen Muskelkontraktionen der quer gestreiften Muskulatur durch elektrische Stromimpulse.

Im Gegensatz zur Elektrostimulation zur therapeutischen Gewebeanregung, die z.T. auf die Wärmewirkung zurückzuführen ist, hat die funktionelle Elektrostimulation (FES) das Ziel, bestimmte Muskeln zur Kontraktion zu veranlassen. Abgesehen von der gezielten Ausnützung der somit künstlich erzeugten Kraft beispielsweise zur Fortbewegung (Radfahren, Stehen/ Gehen von Querschnitts-Patienten) liegt der therapeutische Benefit im Auftreten von Kräfte (im Muskel selbst, in den Bänder und in und an den Knochen des Skelettsystems), sowohl in der positiven Wirkung der Stoffaustauschprozesse während den bzw. für die Energietransfermechanismen. FES hat daher in den letzten Jahren einen festen Platz zur Unterstützung von Rehabilitation und Training eingenommen. Die Spannungs- bzw. Stromimpulse verändern das elektrische Feld im Gewebe und es kommt zu einer Depolarisation von Neuronen, die eine Muskelkontraktion hervorrufen (Merill, et al., 2005).

Bei Patienten, die sich in der physikalischen Rehabilitation befinden, wird die funktionelle Elektrostimulation immer öfters zur unmittelbaren Therapieunterstützung eingesetzt. Es hat sich der Trend durchgesetzt, dass nach einer Verletzung oder einer Operation, beispielsweise an den Extremitäten, viel früher mit dem Training begonnen wird. FES bietet hier die Möglichkeit, ein schonendes Krafttraining durchzuführen. Im Extremfall kann das bereits im Krankenbett schon vor der klassischen Rehabilitation erfolgen, bevor der Patient selbst dazu in der Lage ist willkürliches Training zu absolvieren.

Mit Hilfe der FES können aber auch beachtliche Kontraktionskräfte erreicht werden, weshalb Stimulatoren vermehrt zur Prophylaxe (z. B. Weltraummedizin) oder rein als

Trainings- und Fitnessgerät eingesetzt wird. Das Zentrum für Biomedizinische Technik und Physik der Medizinischen Universität Wien hat auf diesen Gebieten Pionierarbeit geleistet, beispielsweise beim weltweit ersten Einsatz von FES beim Training der Kosmonauten auf der Raumstation MIR. (Mayr, et al. 2001, Freilinger G. 2002)

Eine weitere angesprochene Personengruppe sind Patienten mit Querschnittslähmung. Für diese kann FES bei moderater Anwendung eine allgemeine Verbesserung der Durchblutung und eine erhöhte Belastbarkeit des Herz-Kreislauf-Systemes bedeuten (Quittan, et al. 2001) und bei intensiver Anwendung sogar zu einer - wenn auch teilweise und eingeschränkte Wiedererlangung der Steh- und Gehfunktion führen (Bijak M, et al. 2002).

Die vorliegende Arbeit beschränkt sich ausschließlich auf die transkutane Nervstimulation (TENS). Das bedeutet zum einen, dass nur Oberflächenelektroden zur Anwendung kommen. Alle Techniken und Methoden mit implantierten Elektroden (Mayr, et al. 2001), wie Beinschrittmacher, Atemschrittmacher oder Blasenschrittmacher werden ausgeklammert, da sie für das zu untersuchende Phänomen der Kontraktionsoszillationen (FES - Wave) den Rahmen der Diplomarbeit weit überschreiten würden.[1)] TENS bedeutet weiters, dass die Muskelfasern nicht direkt, sondern vielmehr die motorischen Nerven, die die Muskeln ansteuern stimuliert werden. Die direkte Muskelstimulation, wie sie bei denervierten Querschnittpatienten mit defekten Motoneuronen angewendet werden muss, wird hier nicht berücksichtigt.[2)]

Um ein möglichst günstiges Verhältnis zwischen Kraftentwicklung und Ermüdung zu erhalten, wird üblicherweise mit Stimulationsfrequenzen zwischen 20Hz und 35Hz gearbeitet. In diesem Bereich ist die Kraft nahezu vollständig fusioniert, das heißt, man erhält eine gleichmäßige Dauerkontraktion mit geringer Restwelligkeit. Bei einer weiteren Erhöhung der Frequenz nimmt die Kraft zunächst noch zu, jedoch steigt die Ermüdung des Muskels überproportional an. Die Tatsache, dass bei der Nervstimulation sehr kurze, meist biphasische Nadelimpulse mit 0,05 bis 1 ms Breite verwendet werden, lässt rein technisch betrachtet weit höhere Stimulationsfrequenzen zu. Theoretisch können mehrere KHz eingestellt werden. Die Grenze ist dann erreicht, wenn sich Impuls an Impuls reiht und eine rechteckförmige Dauerstimulation entsteht. Bei der Niederfrequenzstimulation (bis etwa 1kHz) kann eine Stimulation über der Fusionsfrequenz zum gezielten Ermüdungstraining des Zielmuskels eingesetzt werden.

[1)] Außerdem fehlen derzeit die Hinweise, dass das Phänomen bei implantierten Elektroden, d. h. direkt am Nerv angebrachten Elektroden überhaupt auftritt.

[2)] Direkte Muskelstimulation benötigt viel längere Pulsbreiten (40ms und mehr), so dass entsprechend hohe Stimulationsfrequenz, die für das Phänomen der "FES - Wave" benötigt werden nicht eingestellt werden können.

Mittelfrequenzstimulation im mehreren - kHz - Bereich mit niederfrequenter Amplitudenmodulation wurde früher als Möglichkeit betrachtet, Muskelfasern (2a) gezielt zu trainieren. Heute ist zwar diese Stimulationsart in den Geräten noch implementiert, jedoch erkennen inzwischen viele Anwender keinen Nutzen in dieser Stimulationstechnik.

2.2. Funktionsbeschreibung des Stimulators

Der Stimulator ist für die transkutane funktionelle Elektrostimulation entwickelt und deckt, neben der speziellen Anwendung bei der FES - Wave Messung, einen universellen Verwendungsbereich ab. Neben Einzelpulsen können Stimulationsmuster programmiert werden, welche Frequenzen von bis zu 2 kHz erreichen können. Die Abbildung 2.1 zeigt eine schematische Darstellung des Stimulators und gibt einen Überblick seiner Funktionen, welche in folgender Auflistung beschrieben werden.

Stimulator – Eigenschaften:

1. Zwei getrennt - ansteuerbare Stimulationskanäle
2. Impulse sind rechteckförmig und biphasisch, mit veränderbarer Polarität.
3. Impulsquelle liefert eine konstante Spannung.
4. Die Stimulationsfrequenz reicht von 0,1 Hz bis 2000 Hz.
5. Die Impulsbreiten sind für den positiven und negativen Puls unabhängig einstellbar und liegen im Bereich von 0,16 ms bis 32 ms, wobei beide Impulslängen addiert werden müssen.
6. Über die LCD-Anzeige kann man acht verschiedene Stimulationsprogramme wählen, welche in zwei Gruppen unterteilt werden.
   Die erste Gruppe bilden die Programme „Twitch“, „Int. Trigger“ und „Ext. Trigger“, deren Timing und Amplitude direkt am Gerät einstellbar sind.
   Die fünf weiteren Programme besitzen vordefinierte Protokolle und werden extern über einen Computer und dem Burst - Editor programmiert.
7. Die Amplitude kann während der Stimulation nur bei der ersten Gruppe der Stimulationsprogramme direkt über Drehknöpfe eingestellt werden. In einem zusätzlichen Menüpunkt kann man den Amplitudenbereich vor der Stimulation in einem Bereich von 0 bis 100 % einstellen.
8. Der Burst - Editor ist ein GUI (Graphic User Interface) mit dem die Stimulationsparameter editiert werden, und in weiterer Folge in den Stimulator geladen werden.

Kernstück dieser Aufgaben bildet ein Mikrokontroller, der einen autarken Betrieb ermöglicht, d. h., für den Stimulationsbetrieb ist keine ständige Verbindung zu einem Computer nötig. Wie im letzen Punkt (8) beschrieben, wird der PC nur für eine anwenderfreundliche Programmierung der Burstfolgen verwendet. Die Kommunikation zwischen beiden Geräten erfolgt über die serielle Schnittstelle.

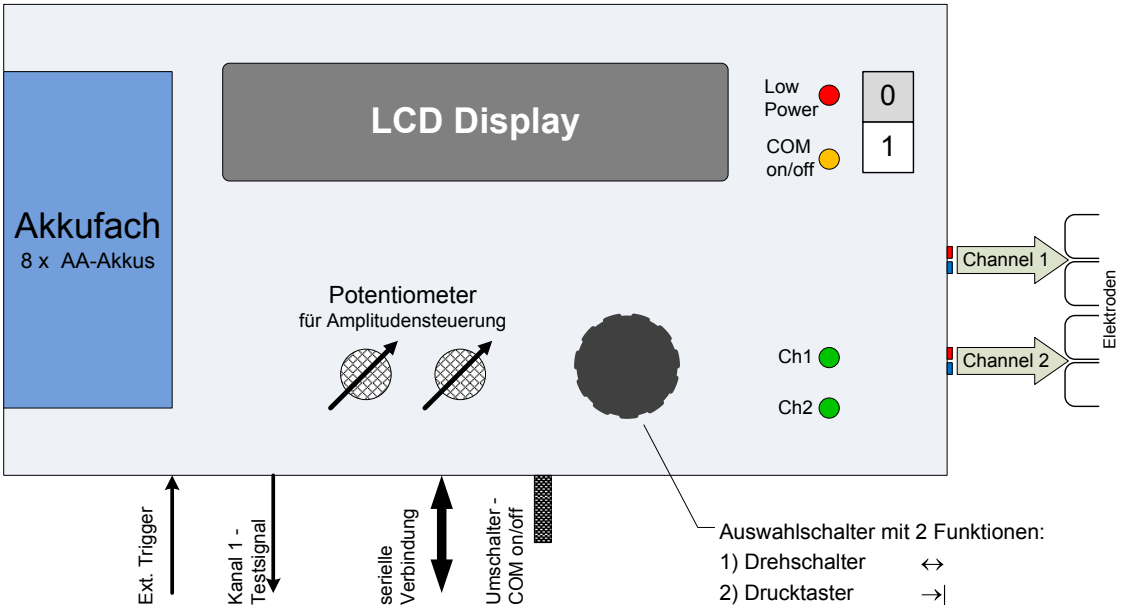


Abbildung 2.1: Schematische Darstellung des Stimulators

2.2.1. Hardware-Profil

In diesem Abschnitt werden die einzelnen Funktionen, die in der Abbildung 2.1 dargestellt werden, kurz beschrieben.

Versorgung

Die Versorgungseinheit ist ein Akkublock bestehend aus 8 x Mignon Akkus, die bei 1,2 V pro Element in Summe eine Versorgungsspannung von 9,6 V liefert. Der Endstufenverstärker ist für eine Betriebsspannung von 10,8 V optimiert, jedoch kann der Stimulator in einem Bereich von 7 bis 12 V betrieben werden. Eine wichtige Anmerkung ist, dass die Versorgungsspannung die Stimulationsamplitude direkt beeinflusst, welche in der folgenden Tabelle gezeigt wird.

Versorgungsspannung	max. Ausgangsspannung
7 V	± 60 V
8 V	± 70 V
9 V	± 75 V
10 V	± 80 V
11 V	± 83 V
12 V	± 85 V

Tabelle 2.1: Ausgangsspannung in Abhängigkeit der Versorgungsspannung

Stromaufnahme (bei 10 V Versorgungsspannung):

- Normalbetrieb: 60 mA
- Serielle Schnittstelle aktiv: 110 mA

Externer Trigger

Der externe Triggereingang ist von der restlichen Stimulatoreinheit über einem Optokoppler galvanisch getrennt - die Isolationsspannung liegt bei 5000 V. Die Durchlassspannung der Eingangsdiode beträgt 1,45 V bei einem Stromfluss von 1,6 mA (Datenblatt HWCN139, 2006). Die Eingangspegel müssen demnach folgende Bedingung erfüllen:

Low-Spannung	< 1 V
High-Spannung	> 4 V

Kanal 1 - Testsignal

Dieser Ausgang besitzt auch einen BNC - Anschluss und liefert ein Testsignal des ersten Stimulationskanals. Die Signalfolge wird vor der Endstufe abgeleitet, wobei die Amplitude den DAC - Wert entspricht, d. h. der Bereich liegt zwischen 0 V und ±5 V. Das Testsignal besitzt die gleiche Signalform wie der Stimulationskanal und kann für weitere externe Endstufen herangezogen werden.

Serielle Verbindung

Ein D-Sub 9 Pin Stecker steht für die Verbindung zu einem Computer zu Verfügung, welche mittels eines handelsüblichen seriellen Kabels direkt an das COM - Port angeschlossen wird. Ein Umschalter aktiviert die serielle Schnittstelle und blockiert gleichzeitig die Versorgung der Stimulator - Endstufe. Damit ist sicher gestellt, dass während der Datenübertragung keine Stimulation möglich ist. Der Zustand wird mit Hilfe des gelben LEDs angezeigt, wobei ein Aufleuchten eine aktive serielle Verbindung signalisiert.

Auswahldrehschalter/-taster

Der Drehschalter/Taster wird bei der Menüsteuerung verwendet und vereint, zwei Funktionen. In der folgenden Erklärung wird das Symbol „↔“ für die Verwendung des Drehschalters und „→|“ für den Taster verwendet.

Amplitudeneinstellung

In den ersten drei Menüpunkten (vgl. Seite 10) kann die Amplitude während der Stimulation verändert werden. Mit den zwei Drehknöpfen kann die Ausgangsspannung für jeden Kanal separat eingestellt werden.

Anzeigenelemente

LCD - Display:

Mit dem Display können zwei Zeilen zu je 24 Zeichen darstellt werden. Dieses Modul wird

bei der Einstellung der Stimulationsprogramme und -parameter, sowie zur Ausgabe wichtiger Fehlermeldungen und des Statusberichtes verwendet.

LED:

Mit den Leuchtdioden werden auf verschiedene Betriebszustände hingewiesen. Die folgende Tabelle solle einen schnellen Überblick der LED-Funktionen geben.

leuchten des LEDs	Bedeutung
Grün	Anzeige der Stimulationsimpulse
Gelb	Serielle Schnittstelle ist aktiv.
Rot	Warnzeichen für zu niedrige Versorgungsspannung. (Anzeichen für entleerten Akku-Bock)

Tabelle 2.2: Beschreibung – LEDs

2.2.2. Beschreibung der LCD-Anzeige und der Menüführung

Menüführung

Die Grundstruktur der Menüführung ist in der Abbildung 2.2 dargestellt. Für die Menüsteuerung wird der Auswahlschalter verwendete, der mit seinen zwei Funktionen ein einfaches Navigieren erlaubt. Mit dem Drehschalter ($\leftrightarrow$) bewegt man sich innerhalb einer horizontalen Ebene und wählt zwischen verschiedenen Programmen aus, oder verändert den Parameterwert. Die Auswahl wird mit dem Tastendruck ($\rightarrow|$) bestätigt und man gelangt eine Ebene tiefer.

Die Elemente der ersten Ebene kann man in drei Gruppen unterteilen.

1. Die ersten Menüpunkte (Twitch, internes Timing, externer Trigger) haben ein einfaches Stimulationsprotokoll, deren Parameter direkt am Gerät einstellbar sind.
2. Die zweite Gruppe sind die vordefinierten Programme (1.Programm, …, 5.Programm), die nahezu beliebig komplexe Stimulationsmuster besitzen können. Die Parameter sind fix vorgegeben und können nur mit dem Burst-Editor (PC) geändert werden.
3. Unter dem letzten Menüpunkt kann man den Amplitudenbereich festlegen. Diese Funktion erhöht die Flexibilität für die Amplitudeneinstellung, welche die ersten drei Menüpunkte verwendet.

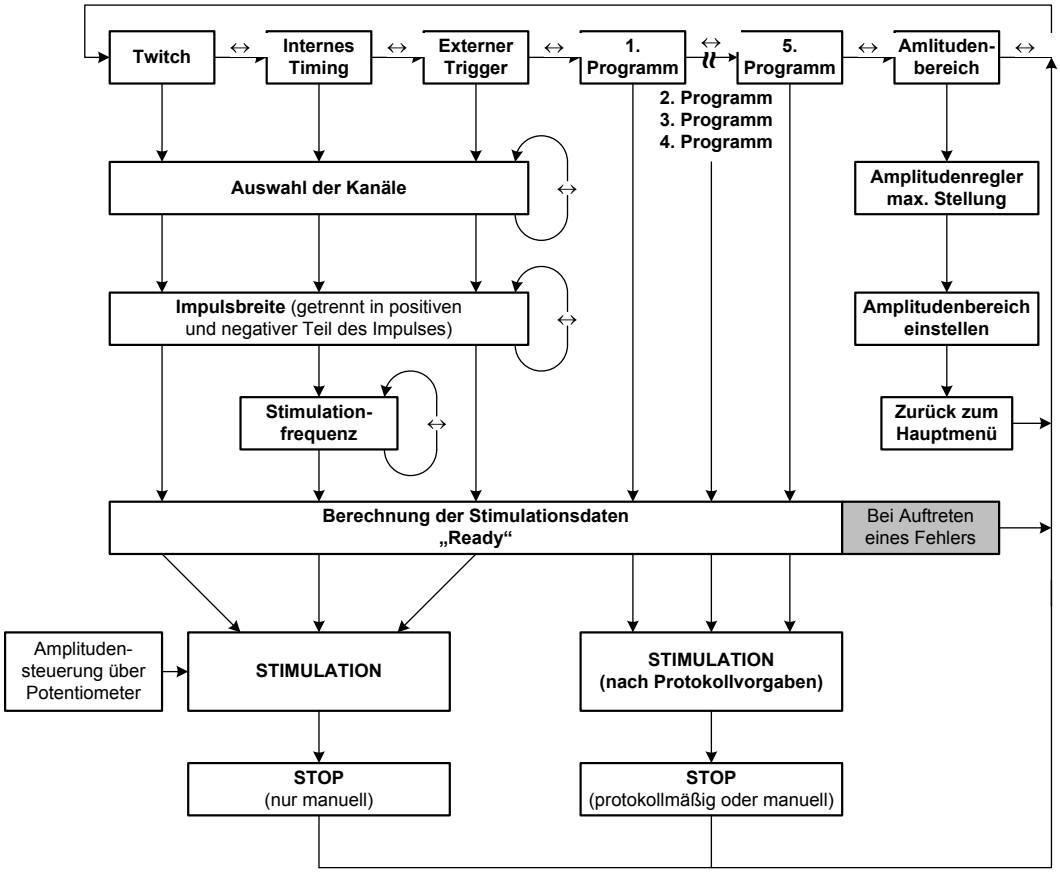


Abbildung 2.2: Prinzipielle Darstellung der Menüführung

Ad 1 – Twitch, internes Timing, externer Trigger

In dieser Gruppe werden die Stimulationsparameter für das jeweilige Programm bestimmt. Der Wertebereich für die einzelnen Einstellungen steht in der Tabelle 2.3 (Seite 20), wobei nicht alle Parameter die in dieser Tabelle genannt werden, an dieser Stelle eine Verwendung finden.

Auswahl der Kanäle [Channel Selection <->]

Im ersten Punkt werden die Stimulationskanäle ausgewählt. Die verschiedenen Einstellungen können über den Auswahldrehschalter (↔) gewählt werden und die anschließende Bestätigung erfolgt mit Taster (→|).

`[Ch1,Ch2 same timing] ↔ [only Ch1] ↔ [only Ch2] ↔ [Ch1,Ch2 independent]`

Hinweis: Wenn die Stimulationsparameter für beide Kanäle den gleichen Wert besitzen, ist es nicht möglich unter dem Menüpunkt „Ch1, Ch2 independent“ ein identisches Timing zu erreichen. Zwischen den beiden Kanälen ist eine fixe Phasenverschiebung zu verzeichnen, die aufgrund der internen Berechnungen zustande kommt. Für diesen Fall ist es nötig „Ch1, Ch2 same timing“ auszuwählen.

Impulsbreite [Set Pulse Width]

Die Kanalauswahl hat direkten Einfluss auf die weiteren Punkte, da man unterschiedliche Kanäle bearbeiten muss. Die Vorgänge sind jedoch sehr ähnlich und damit man Redundanz vermeidet, wird nur der 1.Fall (Ch1, Ch2 same timing) beschrieben.

```
Set Pulse Width
change <->   confirm ->|
```

Das erste Fenster dient nur zur Information – man gelangt durch das Drücken des Auswahltasters (→|) eine Ebene tiefer.

```
Ch1+2 - Pos.Pulse: ◊◊◊ us
Ch1+2 - Neg.Pulse: ◊◊◊ us
```

Die beiden Zeilen werden seriell bearbeitet, d. h., dass man von der zweiten Zeile nicht mehr in die Erste gelangen kann. Die Auswahl der Impulslänge erfolgt mit dem Drehschalter (↔).
Die Schrittweite passt sich automatisch dem Bereich an:
10 µs – Bereich: 80 µs bis 600 µs
100 µs – Bereich: ab 600 µs

Stimulationsfrequenz [Set Stim. Frequency]

Dieser Menüpunkt existiert selbstverständlich nur für das Programm „internes Timing“. Wie in dem Punkt zuvor hat man je nach Auswahl der Kanäle verschiedene Möglichkeiten.

```
Set Stim-Frequency
change <->   confirm ->|
```

Informationsfenster, weiter mit (→|)

```
Ch1+2 - StimFreq:  ◊◊,◊ Hz
```

Auswahlverfahren ist gleich dem vorangegangenen Punkt
Auch hier ändert sich die Schrittweite dynamisch:
0,1 Hz – Bereich: 0,1 Hz bis 2,0 Hz
1,0 Hz – Bereich: 2,0 Hz bis 60,0 Hz
5,0 Hz – Bereich: 60,0 Hz bis 2000,0 Hz

Ready und Stimulation

Diese zwei Punkte sind mit den Menüpunkten (Progam 1, … , Program 5) ident und wird daher gemeinsam in dem folgenden Absatz erklärt.

Ad 2 – Programm 1, …, Programm 5

Ready

Die Stimulationsprogramme werden bevor der Ausgang freigeschalten werden noch einmal überprüft und der Datensatz wird in hardwarenahen Werten umgerechnet. Die Überprüfung erfolgt in zwei Schritten: In dem Ersten wird die Existenz der Daten überprüft,

und im Zweiten, der jeweiligen Wertebereich, beide Fehler führen zu einem Abbruch. Die zweite Fehlermeldung wird in dem Punkt „Fehlermeldungen“ auf der Seite 14 genau beschrieben.

```
Program 1     - error
return to main menu ->|
```

Diese Fehlermeldung kommt, falls die Stimulations-parameter nicht existieren.

```
Program 1     - no data
return to main menu ->|
```

Da es möglich ist Programmplätze leer zu lasse, wird eine irrtümliche Auswahl angezeigt. Nach dieser Fehler-meldung gelangt man sicher im Hauptmenü.

```
Program 1     - data okay
return to main menu ->|
```

Falls diese Meldung auf der LCD - Anzeige erscheint, sind alle Vorgänge fehlerfrei abgearbeitet worden und mit dem Drücken des Auswahltasters beginnt die Stimulation.

Stimulation

```
Stimulation
STOP: Push ->|
```

Die Stimulation läuft und kann jederzeit mit einem Tastendruck abgebrochen werden, auch wenn das Stimulationsprogramm ein definiertes Ende besitzen sollte. Nach jedem Stimulationsabbruch gelangt man in die oberste Menüebene.

Ad 3 – Amplitudensteuerung

In dieser Untergruppe legt man den Amplitudenbereich fest. Als erster Schritt wird der Maximalwert der Potenziometer eruier, um danach die Amplitudenspanne zu definieren. Dieser Vorgang ist nötig um eine feinere Steuerung der Stimulationsamplitude zu ermöglichen.

Datenübertragung: Stimulator - Computer

Die Stimulationsprotokolle für die Programme 1 bis 5 werden über die serielle Schnittstelle in den Stimulator hochgeladen. In diesem kurzen Abschnitt sollen jene Teile beschrieben werden, die der Anwender/ die Anwenderin an der LCD-Anzeige mitverfolgen kann. Das Transferprotokoll wir unter dem Punkt „Serielle Datenübertragung“ auf der Seite 46 detailliert erklärt, indem die Betrachtungsweise bis auf die Byte - Ebene vordringt.

```
ABORT DATA TRANSFER
Push-Button ->|
```

Nachdem der Computer die Handshake - Kennung an den Stimulator gesendet hat, beginnt die Datentransfer - Routine und auf dem Display wird obiger Hinweis ausgeben. Die serielle Kommunikation kann mit dem Drücken des Auswahltasters an jeder Position unterbrochen werden und man kehrt in einen definierten Zustand im Hauptmenü zurück. Dieser Abbruch ist aber nicht sicher, da der Interrupt ausgeführt wird, ohne Rücksicht auf den Zustand der Variablen, d. h., dass eventuell die Konvertierung der Datenblöcke noch nicht abgeschlossen ist. Falls ein Stimulationsprogramm aufgrund eines Abbruchs einen Fehler aufweist, werden diese Programme direkt beim Aufruf gesperrt. Ein fehlerhaftes Programm kann nur durch eine überschreiben der Daten behoben werden.

```
COM: ERROR - ◊◊
Push-Button ->|
```

In der Transfer - Routine werden zwei unterschiedliche Arten von Fehlern erkannt. Die erste Gruppe beinhaltet Fehler im Protokollablauf, die zustande kommen aufgrund nicht interpretierbarer Empfangsdaten. Bei der zweiten Fehlerquelle handelt es sich um Probleme bei der Umrechnung und Speichergenerierung der Stimulationsparameter. Die genaue Fehlerursache wird in der Variable [com_err] gespeichert und wird auf dem LCD-Display ausgegeben, die beiden Symbole ◊◊ sind Platzhalter dieser Variable. Die Tabelle 2.6 auf der Seite 44 zeigt die erkannte Fehlerursache und ermöglicht damit eine einfache Fehlerkorrektur.

```
Transmission: OKAY
Push-Button ->|
```

Nach einer erfolgreichen Datenübertragung gelangt man, wie in den Fällen zuvor, mit dem Auswahltaster (→|) in das Hauptmenü zurück. Die einzelnen Stimulationsparameter werden nicht an dieser Stelle auf ihren gültigen Wertebereich getestet, diese Sicherheitsabfrage wird erst vor der jeweiligen Stimulation durchgeführt.

Fehlermeldungen

An wesentlichen Stellen im Code werden die Stimulationsprogramme überprüft, um die Sicherheit zu erhöhen. Es können vier verschiedene Fehlermeldungen auftreten, die im folgenden Text genauer beschrieben werden. Für eine genaue Betrachtung wird, wie im Absatz zuvor, auf den Abschnitt „Fehlerprotokoll“ (Seite 44) verwiesen. Die Symbole ◊◊ sind Platzhalter für den Fehlercode, dessen Zahlenwert den Fehler genauer beschreibt.

```
Error: 0 - ◊◊
can't allocate memory
```

Die Fehlermeldung (Error : 0) wird angezeigt, wenn während der Initialisierung der Burstfolgen der nötige Speicherplatz nicht generiert werden kann. Der Wert des Fehlercodes beschreibt die fehlerhafte Variable, der in der Tabelle 2.6 genauer dargestellt wird.

```
Error : 1 - ◊◊
burst calculation
```

Nachdem man ein Stimulationsprogramm gewählt hat, werden die einzelnen Parameter werden auf ihren gültigen Wertebereich überprüft. Falls eine Variable außerhalb seines Bereiches liegt, wird der Vorgang abgebrochen und die Fehlermeldung (Error : 1) ausgeben. Die Tabelle 2.7 geht auf den speziellen Fehler ein.

```
Error : 2
stim pulses interleaving
```

Dieser Fehler tritt nur auf in den zwei Programmen "Twitch" und "Externer Trigger", dabei ist diese Fehlermeldung speziell für das zweite Stimulationsprogramm gedacht. Falls während eines Stimulationsimpulses bereit der nächste Impuls folgen sollte, wird diese Fehlermeldung ausgelöst und das Programm wird abgebrochen.

```
COM: ERROR - ◊◊
Push-Button ->|
```

Diese Fehlermeldung kann bei der seriellen Datenübertragung auftreten. Eine genauere Beschreibung erfolgt im Teil „Datenübertragung: Stimulator - Computer"(Seite 13).

2.3. Funktionsbeschreibung: Burst - Editor

Die grafische Oberfläche zur Eingabe der Burstfolgen (Burst-Editor) ermöglicht eine einfache und flexible Änderung der Stimulationsprogramme. Der Stimulator stellt fünf Speicherplätze für editierbare Prokolle zu Verfügung, die auf dem Stimulator unter den Menüpunkten [Program 1] bis [Program 5] aufgerufen werden können. Wie schon mehrmals erwähnt, erfolgt die Übertragung der Daten über die serielle Schnittstelle, diese Funktion ist in dem Burst-Editor integriert. Demnach kann man die Aufgabe des Burst-Editors in zwei Bereiche teilen, erstens in eine Bearbeitung der Parameter für die jeweiligen Programme und zweitens in die serielle Datenübertragung. Der Burst-Editor wird mit dem Aufruf der Datei **DataToStim.exe** gestartet, wobei sich folgendes Fenster öffnet, diesen Screenshot zeigt die folgende Abbildung 2.3.

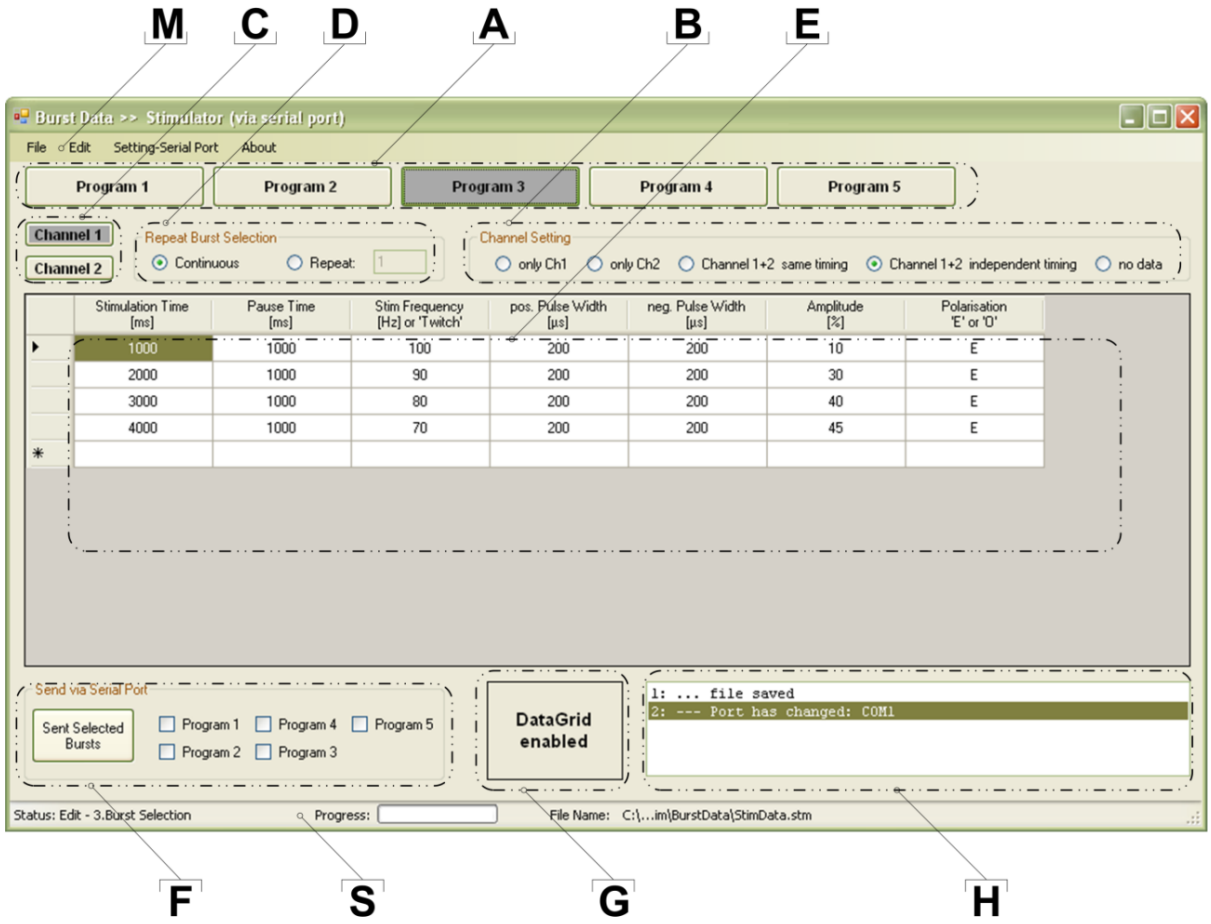


Abbildung 2.3: Screenshot – Burst-Editor

Die gekennzeichneten Elemente des Screenshots werden in alphabetischer Reihenfolge näher beschrieben.

A – Auswahl des Stimulationsprotokolls [Program x]:
Mit diesen fünf Buttons kann man das Stimulationsprotokoll auswählen. Durch einen

Mausklick werden die Stimulationsparameter des ausgewählten Programms angezeigt und editierbar.

B – Auswahl der Stimulationskanäle:

Das Stimulationsprotokoll spricht immer beide Kanäle an, daher besteht in diesem Punkt die Auswahl der Kanäle. Um nur einen Kanal anzusprechen, gibt es die beiden Feldern „only Ch1“ bzw. „only Ch2“. In den Folgenden werden beide Kanäle verwendet, wobei „Channel 1+2 same timing“ nur für eine völlig idente Burstfolge angewendet werden. Den höchsten Grad an Flexibilität erhält man, wenn man den vierten Modus wählt „Channel 1+2 idepentent timing“. *Hinweis:* Wenn man diesen Modus wählt, werden für jedes Stimulationsprotokoll zwei verschiedene Interrupt-Routinen angesprochen, welche seriell ausgeführt werden, d. h., sollten beide Kanäle ein identisches Timing haben, so entsteht zwischen den Stimulationsimpulsen eine Zeitverschiebung.

C – Kanalauswahl:

Man kann nur in dem vierten Modus (vgl. B) eine Auswahl zwischen den zwei beiden Kanälen treffen. Der aktivierte Kanal wird wieder mit einer dunkelgrauen Hintergrundfarbe gekennzeichnet.

D – Wiederholungsrate der Burstfolge:

Dieser Bereich legt die Wiederholungsrate der Burstfolge fest. Als Auswahlmöglichkeiten bestehen „Continuous“ und „Repeat“ mit der Anzahl der Wiederholungen. Wird „Continuous“ ausgewählt, läuft die Stimulation so lange bis der Auswahltaster diese beendet.

E – Datentabelle:

In diesem Feld werden die Parameter der Burstfolge angezeigt und können an dieser Stelle auch editiert. Im Normalfall werden die Daten sofort nach der bestätigten Eingabe aktualisiert, sollte diese Funktion einmal deaktiviert sein, zeigt dies der Button im Bereich G an. Auf die genaue Bedeutung der Parameter wird im folgenden Text noch genauer Bezug genommen. Zur Veranschaulichung der Stimulationsdaten zeigt Abbildung 2.5 eine exemplarische Burstfolge. Der genaue Wertebereich der Parameter wird in der Tabelle 2.3 aufgelistet.

F – Senden der Daten:

In diesem Bereich kann man die einzelnen Programme auswählen, die an den Stimulator gesendet werden sollen. Die Übertragung der Daten erfolgt unmittelbar nach Bestätigung mit der Taste „Send Selected Bursts“.

G – Button zum Aktivieren der Datentabelle:

Dieser Button hat zwei Zustände: „DataGrid enabled“ oder „Press Button to enable DataGrid“. Normalerweise wird die Datenmatrix mit jeder neuen Eingabe automatisch aktualisiert. Es kann jedoch vorkommen, dass für verschiedene Aktionen das Eingabefenster (DataGird) deaktiviert wird, um die Daten wieder abspeichern zu könne muss dieser Button gedrückt werde.

H – Infoliste:

Dieses Service zeigt dem Benutzer/ der Benutzerin Fehlermeldungen oder auch Hinweise an, deren Aktivitäten nicht sichtbar sein würden. Speziell während der seriellen Datenübertragung wird der sequenzielle Fortschritt des Protokolls in diesem Fenster angezeigt.

M – Menüleiste**:**

[File]:

Die Auswahlmöglichkeiten dieses Menüpunkts werden in der Abbildung 2.4 als Screenshot-Teil gezeigt. Die ersten drei Funktionen „Open“, „Save“ und „Save As“ beziehen sich auf die gesamte Datenmatrix aller fünf Burstfolgen, während im Mittelteil nur die Daten des jeweiligen x. Stimulationsprotokoll (x ... beliebige Nummer zwischen 1 und 5) geöffnet bzw. gespeichert werden. Die Dateienstruktur ist speziell für diese Anwendung programmiert und daher mit keiner handelsüblichen Software kompatibel. Damit man die Dateienverwaltung übersichtlicher gestalten kann, wurden die Dateitypen ***.stm** (Protokolle aller Programme) und ***.1st** (nur ein Protokoll) eingeführt.

[Edit]:

Dieser Menüpunkt enthält Element, die das Editieren des Textes erleichtern, wie zum Beispiel „Copy“, „Cut“ und „Select All“. Ein weiteres Feature sind die möglichen Tastenkombinationen, welche die Eingaben bzw. die Änderungen von Protokollen beschleunigen sollen.

[Setting - Serial Port]:

Die Konfiguration der seriellen Schnittstelle ist durch den Mikrokontroller fix vorgeben. Nur der COM - Port kann frei gewählt werde, welcher in der Menüleiste festgelegt wird. In dem Untermenü stehen COM1 und COM2 für eine Schnellauswahl bereit, falls ein anderes Port verwendet wird, kann der Port-Name direkt in einem Textfeld eingegeben werden.

[About]:

Enthält Informationen über die Version und Kontaktadresse des Autors und es Instituts.

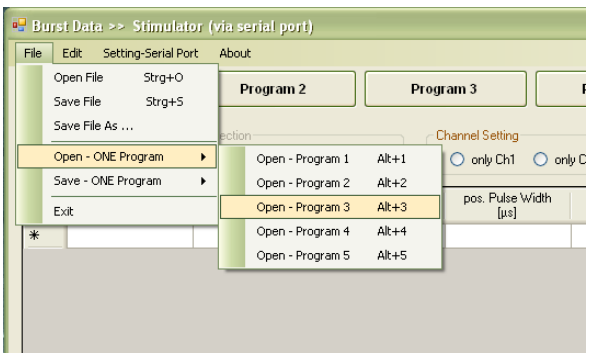


Abbildung 2.4: Screenshot – Menüleiste: [File]

S – Statusleiste:

Die **Statusleiste** dient als zusätzliche Informationszeile und gliedert sich in drei Teilen. Im ersten Bereich [Status: Edit - x.Program] wird die Nummer des Programms angezeigt, das momentan editiert wird. In der Mitte stehen die Felder [Progress:] und ein Balkendiagramm, welche nur im bei dem Datentransfer zum Stimulator aktiv sind. Im Balkendiagramm wird der Fortschritt der Übertragung angezeigt, damit Fehler während der Kommunikation über die serielle Schnittstelle besser erkannt werden könne. Im linken Teil der Statusleiste steht der Name der aktuellen Datei.

Stimulationsprotokoll

In der Abbildung 2.5 werden die Stimulationsparameter in einer Skizze dargestellt. Ein Protokoll beschreibt eine Burstfolge, die entweder kontinuierlich abläuft, oder nach einer bestimmten Anzahl an Wiederholungen beendet wird. Eine weitere wichtige Einstellung ist die Kanalauswahl, bei der man vier Auswahlmöglichkeiten hat: nur Kanal1, nur Kanal2, Kanal 1 und 2 mit gleichem oder unabhängigem Timing.

Die Stimulationsparameter, welche einen Brust beschreiben, kann man in zwei prinzipielle Gruppen unterteilen. Die erste Parametergruppe beschreibt den biphasischen Rechteckimpuls, die von der Amplitude, positiver und negativer Impulslänge, Stimulationsfrequenz und der Polarisation bestimmt werden. Die Burstlänge und eine mögliche Pausenzeit bilden die zweite Gruppe, welche in der folgenden Abbildung im zweiten Graph dargestellt werden.

Ein Stimulationsprotokoll besteht nun aus einer Sammlung diesen Parameter, die jeweils einen Burst (und Pause) beschreiben. Der Stimulator kann in Summe über alle fünf Programm bis zu 350 Bursts speichern, da die Speicherzuweisung dynamisch erfolgt.

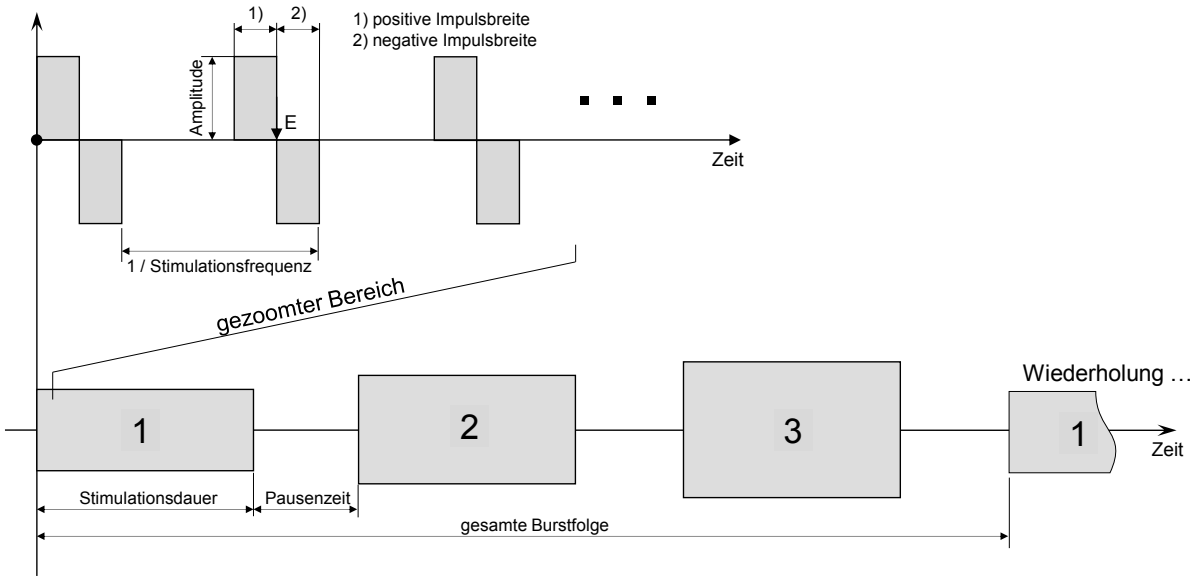


Abbildung 2.5: Darstellung eines Stimulationsprotokoll

Wertebereich

Die Parameter sind nur innerhalb eines bestimmten Wertebereichs gültig, da es anderenfalls zu unkontrollierten Variablen - Überläufen kommen kann. Für die Eingabe der Werte in den Brust-Editor sind die jeweiligen Einheiten fix vorgeben.

	Einheit	Werteberich	Schrittgröße
Stimulationsdauer (Stimulation time)	[ms]	1 ms ..... 1,19 h	1 ms
Pausenzeit (Pause Time)	[ms]	1 ms ..... 1,19 h	1 ms
Stimulationsfrequenz (Stim. Frequency)	[Hz]	0,1 Hz ..... 2 kHz	0,1 Hz
		[oder] Twitch	
positive Impulsbreite (pos. Pulse Width)	[µs]	80 µs ..... 32,7 ms	1 µs
negative Impulsbreite (neg. Pulse Width)	[µs]	80 µs ..... 32,7 ms	1 µs
Amplitude[1)] (Amplitude)	[%]	0 % ..... 100 %	0,1 %
Polarisation (Polarization)		 „E“ oder „O“ 	

Tabelle 2.3: Stimulator: Wertebereich der Burstkennzahlen

[1)] Die maximale Ausgangsamplitude ist abhängig von der Versorgungsspannung (siehe Tabelle 2.1, Seite 8).

2.4. Detaillierte Beschreibung des Stimulators

2.4.1 Hardwarefunktionen

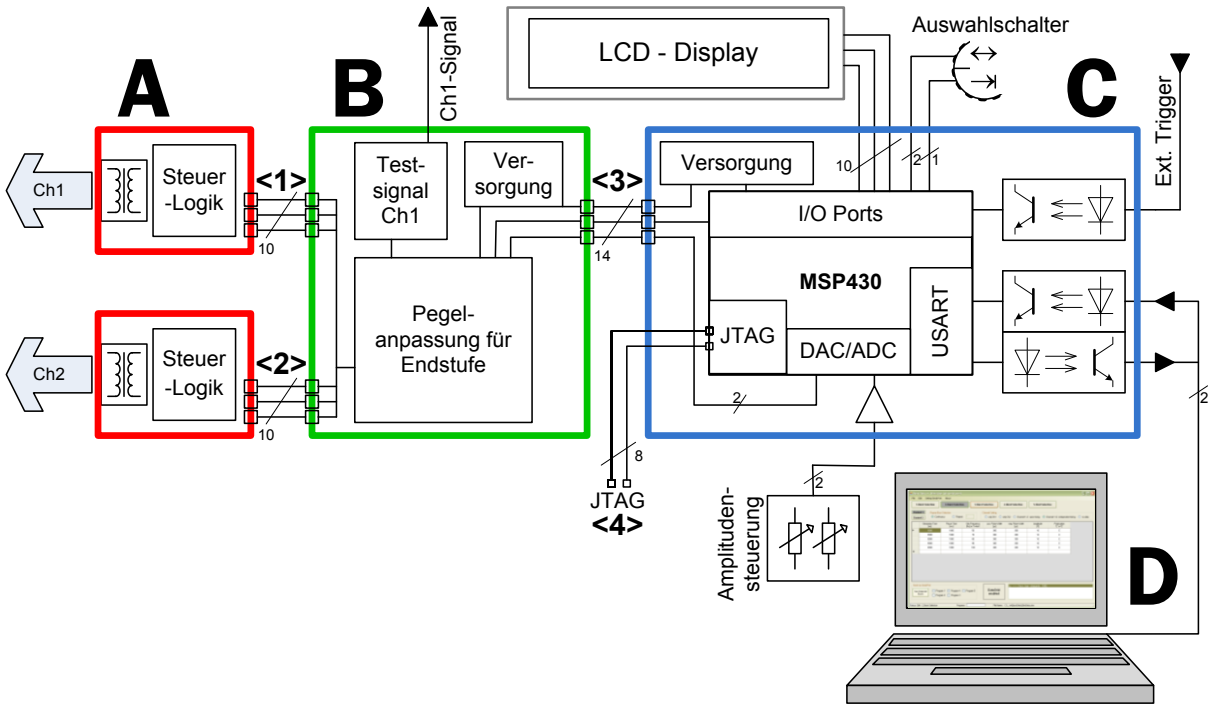


Abbildung 2.6: Detaillierte Skizze des Stimulators
A – Endstufenverstärker, B – Pegelanpassung, C – Mikrokontroller Hauptplatine, D – Burst-Editor. Die Schnittstellen <1>, <2>, <3> und <4> werden im Anhang auf der Seite 104 genau beschrieben.

A - Endstufe:

Die Endstufe verwendet zur Gleichstromentkopplung einen Impulstransformator und wurde am Institut für Biomedizinische Technik von Prof. Lanmüller Hermann entwickelt. Der Stimulationsimpuls wird aus drei Digitaleingängen und einem Analogeingang gebildet. In der folgenden Abbildung werden die Eingangssignale dargestellt, die nötig sind, um einen biphasischen Rechteckimpuls zu generieren.

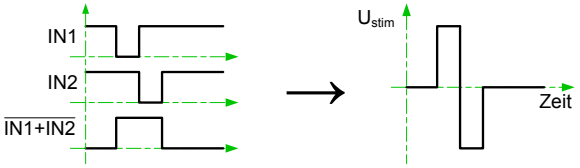


Abbildung 2.7: Pegeldiagramm für die Stimulatorendstufen

Der Spannungsbereich für den Analogeingang liegt bei 0 bis 5 V und regelt die Ausgangsspannung. Die maximale Stimulationsamplitude ist abhängig von der Versorgungsspannung der Endstufen, der Zusammenhang wir in der in der Tabelle 2.1 (Seite 8) gezeigt.

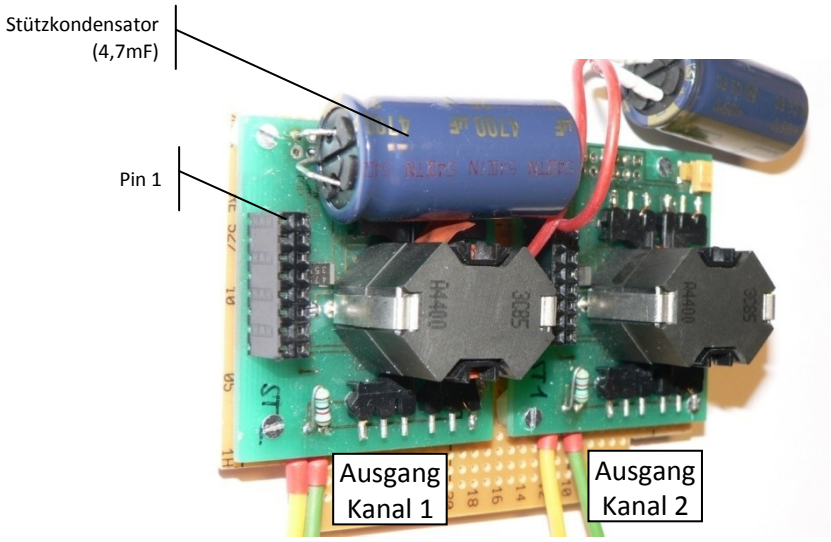


Abbildung 2.8: Foto der Stimulatorendstufe mit Änderungen
Zentrum für Biomedizinische Technik und Physik, Medizinische Universität Wien

B - Pegelanpassung:

Diese Platine hat drei Aufgaben:

1. Generierung des Testsignals von Kanal 1
2. Pegelanpassung zwischen Hauptplatine und Endstufe
3. Versorgung für Endstufenregelung und Optokoppler für die serielle Übertragung

Die Schaltungsteile der ersten beiden Aufgaben sind mit einer roten Umrandung gekennzeichnet. Die Ziffer 1 zeigt den Bereich zur Generierung des Testsignals. Dieses Signal besitzt sie selbe Impulsform wie das Stimulationssignal, nur der Amplitudenbereich ist direkt von dem DAC - Spannungswert abgeleitet. Ein 4 zu 1 Multiplexer wird von den invertierten µC - Ausgängen angesteuert und schaltet den positiven- bzw. negativen DAC - Wert oder das Massepotenzial durch.

Das 2. Schaltungselement zeigt die Pegelanpassung für einen Kanal. Die Digitalsignale für den Stimulationsimpuls müssen, von der Hauptplatine kommend, noch auf die richtige Form gebracht werden. Der Mikrokontroller verwendet pro Stimulationskanal zwei I/O Port Pins, welche mit dieser Schaltung auf die notwendigen drei Digitalsignale erweitert wird.

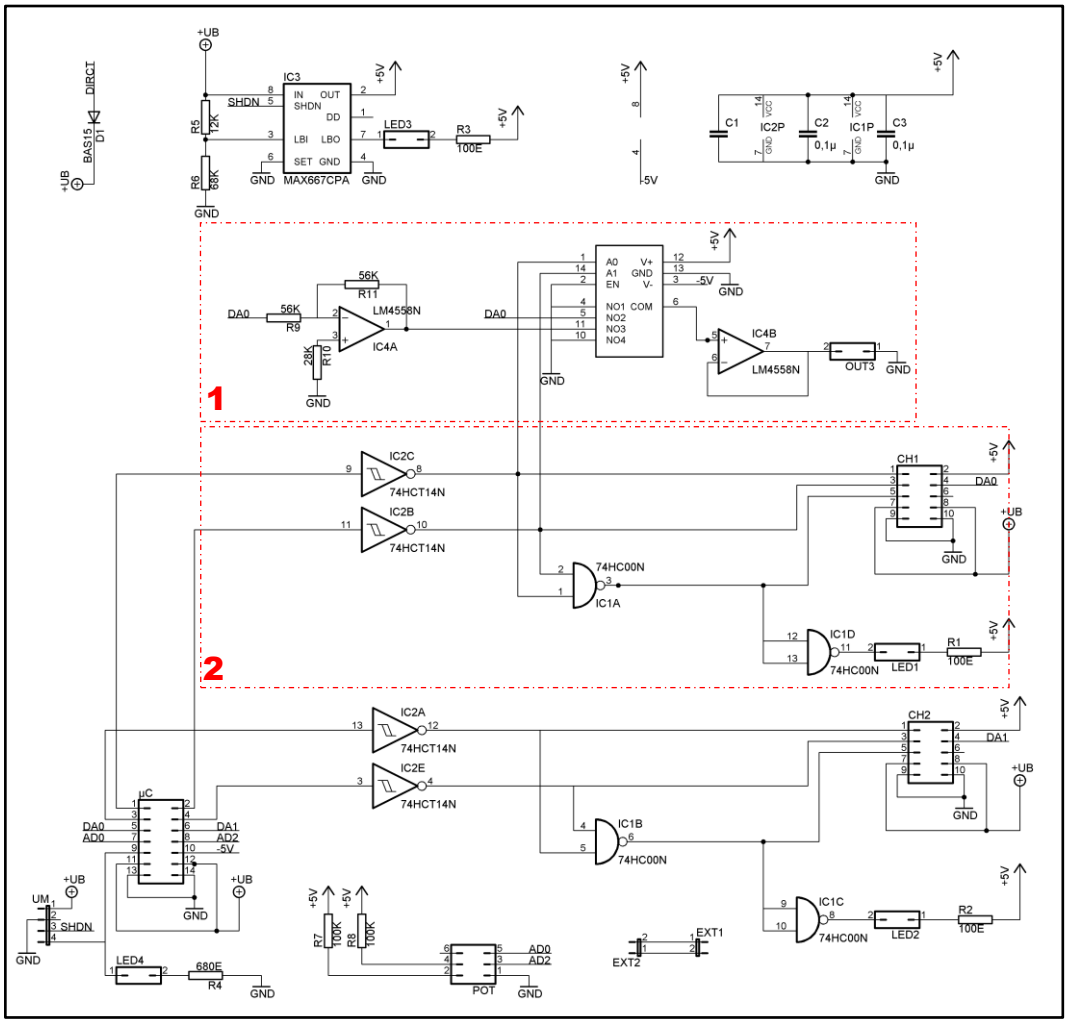


Abbildung 2.9: Schaltplan der Pegelanpassung

C – Hauptplatine:

Die Hauptplatine ist von DI Martin Fend von der Medizinischen Universität in Graz übernommen und beinhaltet die Mikrokontrollerschaltung und deren Peripherie. Im Zentrum diese Schaltung steht der Mikrokontroller - MSP430F1611, der über die JTAG - Schnittstelle programmiert werden kann. Die Pinbelegung der verwendeten JTAG - Leitungen <4> ist im Anhang genauer beschrieben (siehe Tabelle 5.3 - Seite 104). Von den handelsüblichen JTAG - Stecker, die 14-poligen sind, werden in dieser Schaltung nur 8 Eingänge verwendet.

D – GUI, serielle Kommunikation:

Die Eingabemöglichkeit der Stimulationsprotokolle durch einen Computer soll an dieser Stelle nur zur Vollständigkeit erwähnt werden. Wichtig in diesem Zusammenhang ist, dass die serielle Schnittstelle nur RxD - und TxD - Leitung verwendet und von der Stimulationseinheit galvanisch getrennt ist.

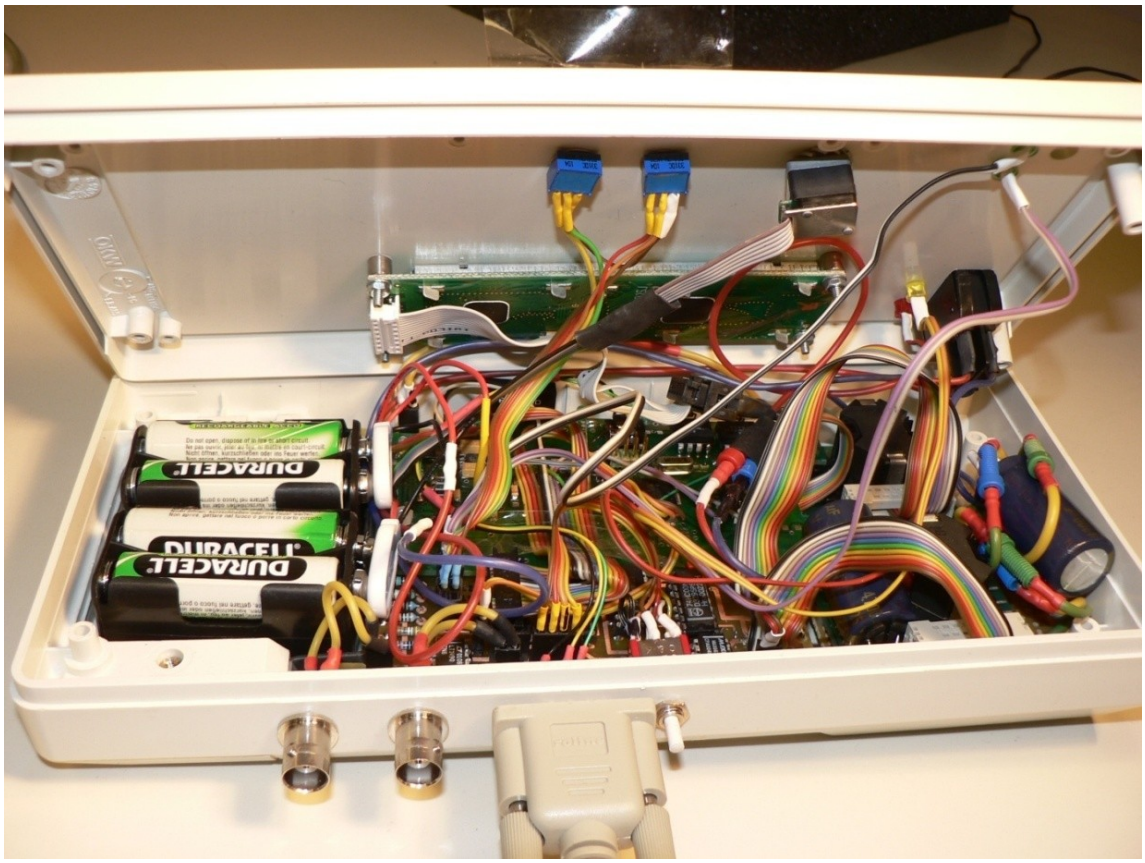


Abbildung 2.10: Stimulator Innenansicht

2.4.2. Beschreibung der µC - Softwareteile

Die Mikrokontrollersoftware ist in der Programmiersprache C geschrieben. Der Code wird mit einem GNU-Compiler übersetzt und über einem Programmieradapter (JTAG) in den Befehlsspeicher des Kontrollers geladen. Der MSPGCC ist portiert von dem Open Source Compiler GNU und ist an die Mikrokontrollerfamilie MSP430 von Texas Instrument angepasst (Underwood, 2003). Die nötige Software und das Benützungshandbuch kann man von der Sourceforge - Internetseite unter der Adresse http://mspgcc.sourceforge.net downloaden.

Der Mikrokontroller wird über den parallelen JTAG - Adapter angesteuert, welcher die Funktionen zum Löschen, Schreiben und Lesen des MSP430 - Speichers (Flash ROM und RAM) enthält, aber auch die Möglichkeit eines Debuggens liefert. Da der Compiler kommandozeilenorientiert ist, verwendet man zur Vereinfachung die Datei „makefile“, welche die Eigenschaften des Mikrokontrollers für den MSP - GNU C Compilier festlegt und beim Kompilieren und Downloaden benötigt wird.

Beispielsdatei - makefile

```
# makfile configuration
NAME            = 2chStim
OBJECTS         = 2chStim_main.o lcd.o
CPU             = msp430x1611

CFLAGS          = -mmcu=${CPU} -O2 -Wall -g

#switch the compiler (for the internal make rules)
CC              = msp430-gcc

.PHONY: all FORCE clean download download-jtag download-bsl dist

#all should be the first target. it's built when make is run without args
all: ${NAME}.elf ${NAME}.a43 ${NAME}.lst

#additional rules for files
${NAME}.elf: ${OBJECTS}
     ${CC} -mmcu=${CPU} -o $@ ${OBJECTS}
${NAME}.a43: ${NAME}.elf
     msp430-objcopy -O ihex $^ $@
${NAME}.lst: ${NAME}.elf
     msp430-objdump -dSt $^ >$@
download-jtag: all
     msp430-jtag -e ${NAME}.elf

clean:    rm -f ${NAME}.elf ${NAME}.a43 ${NAME}.lst ${OBJECTS}
#project dependencies
2chStim_main.o:  2chStim_main.c  hardware.h lcd.h
lcd.o :          lcd.c           hardware.h lcd.h
```

Das gesamte Projekt beinhaltet folgende Dateien:

makefile . . . . . Wie oben erklärt, benötigt man diese Datei für die Compiler - Anwendungen (make.exe) und (download.exe).

2chStim_main.c . . . Das Hauptprogramm wird im folgenden Text ausführlich behandelt.

hardware.h . . . . . Damit die Initialisierung übersichtlicher gestaltet werden kann, sind die wichtigsten Konstanten in dieser Headerdatei zusammengefasst.

led.c / led.h . . . . . Diese Dateien beinhalten Funktionen, welche die Ansteuerung des LCD - Displays regelt. Sie sind von Martin Fend übernommen und werden in diesem Rahmen nicht genauer beschrieben.

*.a43, *.elf, *.o . . . kompilierte Dateien

In weiterer Folge wird das Hauptprogramm genauer beschrieben. Die Grundstruktur des Programms ist ereignisgesteuert, d. h., dass Funktionen nur nach Auftreten eines externen oder internen Interrupts durchgeführt werden. Die Interrupts werden von folgenden Elementengeneriert: I/O Port, ADC, USART oder Timer.

Die folgenden Abschnitte geben den angekündigten, detaillierten Einblick in das µC - Programm, welche durch die wesentlichen Codepassagen ergänzt werden. Die Zeilennummern der Codeteile entsprechen der fortlaufenden Nummerierung innerhalb der Datei 2chStim_main.c.

Hauptroutine - void main()

Das Hauptprogramm beinhaltet die Initialisierung der internen und externen Peripherie - Modulen, sowie der globalen Variablen und eine Endlosschleife in der das Programm im Normalbetrieb abläuft. Die Abbildung 2.11 zeigt das Flussdiagramm des Hauptprogramms.

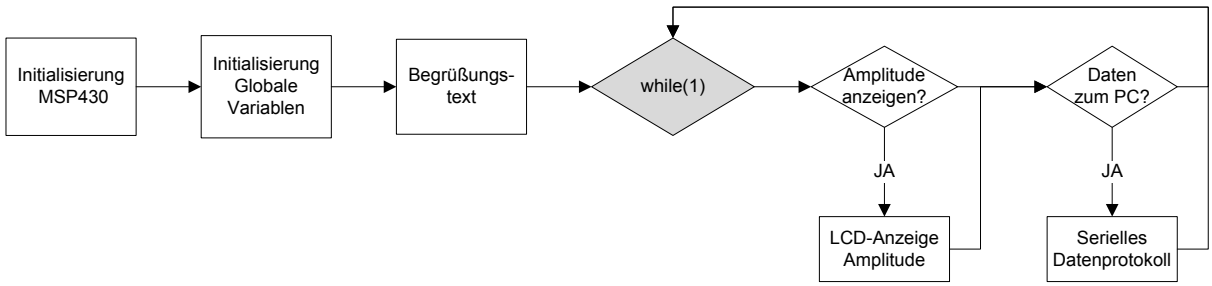


Abbildung 2.11: Flussdiagramm - void main()

Bei der folgenden Beschreibung der einzelnen Kontrollregister werden nicht alle möglichen Varianten lehrbuchsmäßig erklärt, sondern nur Bezug auf die verwendetet Variante genommen.

Initialisierung der Systemtaktung und des Watchdog Timers

Das Kontrollregister des Watchdog Timers enthält eine Passwortkennung, die mit dem Byte WDTPW gesetzt wird. Alle anderen Funktionen dieses Timers werden nicht verwendet (WDTHOLD). Die Systemtaktung wird in zwei Register gesetzt, namentlich in BCSCTL1 und 2. An dem Eingang X_{in} und X_{out} befindet sich der externe Oszillator mit 8MHz. Diese Taktrate wird in weiterer folge nicht mehr geteilt. Der Mikrokontroller würde auch die Möglichkeit einer Taktverringerung oder eine interne niederfrequente Taktquelle bieten, damit der Energiebedarf verringert wird (Nagy, 2003). Die Variante wird aber in diesem Projekt nicht verwendet wird.

```
1653 :    // ------------- <1> initialization : clock and watchdog
1654 :    WDTCTL = WDTPW|WDTHOLD;             //Init watchdog timer
1655 :
1656 :    BCSCTL1 &= ~XT2OFF;                 // ACLK = LFXT1 = HF XTAL
1657 :    do {   IFG1 &= ~OFIFG;              // Clear OSCFault flag
1658 :             for (i=0xFF; i>0; i--);    // Time for flag to set
1659 :     }while((IFG1 & OFIFG));            // OSCFault flag still set?
1660 :
1661 :     BCSCTL2 |= SELM_XT2CLK + SELS + DIVS_DIV1 + DIVM_DIV1; // SELS select source for SMCLK
```

Initialisierung der I/O Ports

Die I/O Port Pins müssen für ihre Verwendung in mehreren Stufen initialisiert werden. Der Mikrokontroller biete für die meisten Anschlüsse, neben der Verwendung als Ein-/ Ausgang, zusätzliche Funktionen an, die mit dem PxSEL - Register ausgewählt werden. Diese Sonderfunktionen sind der Interrupt - Eingang, ADC/DAC und eine serielle

Schnittstelle. Bei der Initialisierung einer Interruptleitung wird jeder Eingang einzeln aktiviert und die Sprungrichtung der Signalform bestimmt. Wird der Port - Pin als normaler Ein-/Ausgang verwendet, muss die Signalrichtung zusätzlich mit PxDIR bestimmt werden.

Initialisierung der Timer A und B

Die Taktfrequenz der beiden Timer wird auf 1MHz gesetzt, indem man die Systemfrequenz (TASSEL_SMCLK) durch acht teilt (ID_DIV8). Mit den Kontrollregistern TxCCTL0 und TxCCTL1 wird die Interruptsteuerung initialisiert. Beide Timer werden in Modus 1 (MC_1) betrieben, bei dem das Timerrun - Register (TAR) bis zu dem Wert TxCCR0 hinaufzählt und nach Erreichen dieser Schwelle wieder bei 0 anfängt. Bei diesem Vorgang wird eine „overflow“ - Interrupt ausgelöst. Das Register TxCCR1 legt eine weitere Schwelle fest, die ebenso bei Überschreitung einen Timer-Interrupt auslöst.

```
1684 :      // ------------- <3> initialization : Timer A + Timer B
1685 :      TACTL = TASSEL_SMCLK + ID_DIV8 + TACLR + TAIFG + TAIE;  // 8Mhz SMCLK, 8Mhz/8 = 1MHz
1686 :      TACCTL0 = CCIE;                                         // CCR0 interrupt enabled
1687 :      TACCTL1 = CCIE;                                         // CCR1 interrupt enabled
1688 :      TACCR0  = 800;
1689 :      TACCR1  = 500;
1690 :
1691 :      TBCTL = TBSSEL_SMCLK + ID_DIV8 + TBCLR + TBIFG + TBIE;
1692 :      TBCCTL0 = CCIE;
1693 :      TBCCTL1 = CCIE;
1694 :      TBCCR0 = 800;
1695 :      TBCCR1 = 500;
```

Initialisierung des ADC und DAC

Die ADC/DAC - Funktion ist am Port 6 untergebracht, wobei die Port - Pins P6.1 und P6.3 für den ADC verwendet werden und P6.6 und P6.7 DAC - Ausgänge sind. Als Referenzspannung wird die interne 2,5 V Quelle gewählt (REF2_5V). Mit dem Register ADC12IE wird der Interrupt aktiviert, der nach Abschluss der Konvertierung ausgelöst wird.

```
1697 :      // ------------- <4> initialization : ADC, DAC
1698 :      P6SEL = 0x0A;                                     // P6.1 and P6.3 ADC option selected
1699 :
1700 :      ADC12CTL0 = REF2_5V + REFON + ADC12ON + MSC + SHT0_13; // ADC12 on, set sampling time
1701 :      delay(1000);                                          // Time VRef to settle
1702 :      ADC12CTL1  = SHP + ADC12DIV_0 + CONSEQ_3;             // Use sampling timer,single seq.
1703 :      ADC12MCTL0 = INCH_1;                                  // ref+=AVcc, channel = A1
1704 :      ADC12MCTL1 = INCH_3+EOS;                              // ref+=AVcc, channel = A3,end seq
1705 :      ADC12IE    = 0x0A;                                    // Enable ADC12IFG.3
1706 :
1707 :      DAC12_0CTL = DAC12IR + DAC12AMP_5 + DAC12ENC + DAC12GRP;
1708 :      DAC12_1CTL = DAC12IR + DAC12AMP_5 + DAC12ENC;
```

Initialisierung der seriellen Schnittstelle

Der verwendete Mikrokontroller besitzt zwei identische USART - Module (Universal Synchronous/Asynchronous Receive/Transmit), die voneinander unabhängig betrieben werden können. Für die serielle Kommunikation zu dem Computer wird das zweite Modul als UART verwendet, die Betriebsart für das RS232 - Protokoll (Bierl, 2004 S. 183ff). Die

Initialisierung erfordert das Setzen von sechs Kontrollbytes und die Aktvierung des Interrupts. Die serielle Schnittstelle enthält folgende Konfiguration: 8 Datenbits (CHAR), 1 Stopbit, DTR und CTS deaktiviert, keine Parität und eine Baudrate von 115200 Baud/s. Bei einer Baudrate von 115200 Baud/s und einer Systemtaktung von 8 MHz ist der theoretische Teilerwert 69,44… ($8{,}0MHz/115200\frac{Baud}{s} = 69{,}\dot{4}$). Das Teilerverhältnis wird in drei Registern gespeichert. So wird der ganzzahlige Teil in den Baud - Registern UBR01 und UBR11 geschrieben und die Kommastellen werden durch einen 8 - Bit Modulator (UMCTL1) angenähert. Dieser Modulator ist ein spezielles Feature des MSP430 und bietet höhere Flexibilität. Mit der folgenden Formel berechnet man die Baudrate, welche auch die Modulatorfunktion inkludiert.

$$\boldsymbol{Baudrate} = \frac{\boldsymbol{BRCLK}}{\boldsymbol{UBR01} + \boldsymbol{UBR11} \cdot \boldsymbol{16^2} + \frac{\boldsymbol{1}}{\boldsymbol{8}} \cdot \sum_{i=0}^{7} \boldsymbol{m_i}}$$

UMCTL1	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0	… Modulator-Register

Die Verwendung des Modulators zeigt das folgende Beispiel:

UMCTL1	0	0	1	0	1	1	0	0	
Teilerzahl	69	69	70	69	70	70	69	69	→ 69,375

```
1710 :      // ------------- <5> initialization : UART
1711 :      UCTL1  = CHAR;            // [0001 0000], 8-bit character
1712 :      UTCTL1 = SSEL1;           // [0010 0000], UCLK = SMCLK
1713 :      URCTL1 = 0x00;            // [0000 0000]
1714 :      UBR01  = 0x45;            // [0100 0101], 8 MHz / 115200 = 69,444d (≈ 0x45)
1715 :      UBR11  = 0x00;            // [0000 0000]
1716 :      UMCTL1 = 0x2C;            // [0010 1100] = modulation = 0010 1100b
1717 :      ME2   |= UTXE1 + URXE1;   // Enable USART1 TXD/RXD
1718 :      IE2   |= URXIE1;          // Enable USART1 RX interrupt
```

Initialisierung der globalen Variablen, Begrüßungstext

Globale Variablen werden am Anfang des Programms deklariert, und gelten in allen folgenden Funktionen (Müller, et al., 2005 S. 189). Nach der Initialisierung der internen Module werden diese Variablen in einen definierten Zustand gebracht. Auf eine genaue Auflistung wird an dieser Stelle verzichtet.

Der Begrüßungstext zeigt an, dass der Initialisierungsteil abgeschlossen ist und der Stimulator für Anweisungen bereit ist.

Begrüßungstext:

```
2-Channel Stimulator
-FesWave-     March 2007
```

Endlosschleife - while(1)

In der Endlosschleife wird die Wiederholungsbedingung immer erfüllt und lässt das Hauptprogramm ad infinitum laufen. Das Programm reagiert noch auf Interrupt-Ereignisse, bei denen das Hauptprogramm unterbrochen und die dazugehörige Interrupt-Routine

ausgeführt wird. Innerhalb der Endlosschleife werden zwei Prozeduren abgearbeitet, die zeitlich nicht kritisch sind. Diese Funktionen sind „Anzeigen der Amplitudenwerte“ und „das Senden der Daten für die serielle Kommunikation“. Beide Teile werden in den Abschnitten auf den folgenden Seiten genauer beschrieben.

Datenstruktur

Alle Parameter, die ein Stimulationsprogramm beschreiben, sind in dem selbstdefinierten Typ (Prog) zusammengefasst. Innerhalb dieses Typs gibt es vier dynamische Variablen : TimA, TimB, *iAmp_a* und *iAmp_b*. Mit dieser Flexibilität können die einzelnen Programme unterschiedlich lange Burstfolge besitzen, das den Speicherbedarf minimiert. Nachteil dieses Verfahrens ist, dass bei jeder Änderung der Stimulationsparameter der Speicherplatz gelöscht und wieder neu zugewiesen werden muss. Diese zusätzliche Fehlerquelle erfordert eine zusätzliche Überprüfung der Daten. Etwaige Fehler werden auf dem LCD-Display angezeigt und das Stimulationsprogramm wird gesperrt. Auf der Seite 44 wird das Fehlerprotokoll genauer beschrieben.

```
 76 : typedef struct {
 77 :     float fStimTime;       // stimulation time [1ms]
 78 :                            // within the parameters of 1ms -to- 4294s
 79 :     float fPauseTime;      // pause time [1ms]
 80 :                            // within the parameters of 1ms -to- 4294s
 81 :     int iStimFreq;         // stimulation freq.  [1dHz] OR -1 for Twitch
 82 :                            // within the parameters of 0.1 Hz -to- 2000Hz
 83 :     unsigned int uiPWidth; // pulse width of the positive pulse [1us]
 84 :                            // within the parameters of 80us -to- 32767us
 85 :     unsigned int uiNWidth; // pulse width of the negative pulse [1us]
 86 :                            // within the parameters of 80us -to- 32767us
 87 :     char cPol;             // Polaristation - (E) eve   (O) odd
 88 : }TData;
 89 :
 90 :
 91 : typedef struct  {
 92 :     unsigned int ProgInfo;
 93 :     int RepTA;
 94 :     int RepTB;
 95 :     int* iAmp_a;        // Amplitude of Timer A - [one-tenth of a percent, (1 ‰)]
 96 :                         // 1000 is equivalent to 4,88V DAC output value
 97 :     int* iAmp_b;        // Amplitude of Timer B - [one-tenth of a percent, (1 ‰)]
 98 :                         // 1000 is equivalent to 4,88V DAC output value
 99 :     TData* TimA;
100 :     TData* TimB;
101 : }Prog;
102 :
103 :
104 : typedef struct {
105 :     unsigned int uiTCCR0_1;
106 :     unsigned int uiTCCR0_2;
107 :     unsigned int uiTCCR0_s;
108 :     unsigned int uiTCCR1_s;
109 :     unsigned long ulPauseTime;
110 :     unsigned long ulStimTime;
111 :     unsigned long ulStimPer;
112 :     int iSAmp_a;
113 :     int iSAmp_b;
114 :     unsigned char OUT1;
115 :     unsigned char OUT2;
116 : } StimData;
117 :
118 : Prog Set;
119 : Prog B1;
120 : Prog B2;
121 : Prog B3;
122 : Prog B4;
123 : Prog B5;
124 : StimData *StimTA = NULL;
125 : StimData *StimTB = NULL;
```

Der Codeteil zeigt die Initialisierung der Stimulationsprogramme, wobei nur das Programm 1 exemplarisch dargestellt ist. Die Protokollvariablen (Set, B1, …, B5) müssen vor jeder Veränderung ihre Speicherzuordnung mit free(…) frei geben, um danach mit der Funktion malloc(…) wieder einen Bereich im Speicher zu belegen. Diese scheinbar komplizierte Art der dynamischen Speicherzuweisung ist notwendig, da der MSPGCC die Funktion realloc(…) nicht unterstützt.

Mit dem Unterprogramm TData BurstTiming(…) werden die Stimulationsparameter übersichtlich gespeichert.

```
1897 : TData BurstTiming(float a, float b, float c, unsigned int d, unsigned int e, char f )
1898 : {   TData tmp;
1899 :     tmp.fStimTime = a;     tmp.fPauseTime = b;    tmp.iStimFreq = c;
1900 :     tmp.uiPWidth = d;      tmp.uiNWidth = e;    tmp.cPol = f;
1901 :     return tmp;
1902 : }
1903 :
1904 : void InitProgram(void)
1905 : {
1906 :     dat_err = 0;
1907 :
1908 :     free(Set.TimA);free(Set.TimB);free(Set.iAmp_a);free(Set.iAmp_b);
1909 :     free(B1.TimA); free(B1.TimB); free(B1.iAmp_a); free(B1.iAmp_b);
1910 :     free(B2.TimA); free(B2.TimB); free(B2.iAmp_a); free(B2.iAmp_b);
1911 :     free(B3.TimA); free(B3.TimB); free(B3.iAmp_a); free(B3.iAmp_b);
1912 :     free(B4.TimA); free(B4.TimB); free(B4.iAmp_a); free(B4.iAmp_b);
1913 :     free(B5.TimA); free(B5.TimB); free(B5.iAmp_a); free(B5.iAmp_b);
 ... :
 ... :                ......
 ... :
1939 : //--------------- Program 1
1940 :     B1.ProgInfo = BothSame;
1941 :     B1.RepTA    = ContStim;
1942 :     B1.RepTB    = ContStim;
1943 :     BurstLength[P1][T_A] = 3;
1944 :
1945 :     B1.TimA = (TData*)malloc(BurstLength[P1][T_A]*sizeof(TData));
1946 :     B1.iAmp_a = (unsigned int*)malloc(BurstLength[P1][T_A]*sizeof(unsigned int));
1947 :     B1.iAmp_b = (unsigned int*)malloc(BurstLength[P1][T_A]*sizeof(unsigned int));
1948 :
1949 :     if(B1.TimA == NULL || B1.iAmp_a == NULL || B1.iAmp_b == NULL) {
1950 :         B1.ProgInfo = NoData; dat_err |= 0x004;
1951 :     } else {
1952 :         dat_err &= ~0x004;
1953 :         B1.TimA[0] = BurstTiming(3000,1000,2000,100,100,'E');
1954 :         B1.TimA[1] = BurstTiming(3000,1000,1000,200,200,'O');
1955 :         B1.TimA[2] = BurstTiming(3000,1000, 500,300,300,'E');
1956 :
1957 :         B1.iAmp_a[0] = 100;        B1.iAmp_b[0] = 100;
1958 :         B1.iAmp_a[1] = 200;        B1.iAmp_b[1] = 200;
1959 :         B1.iAmp_a[2] = 300;        B1.iAmp_b[2] = 300;
1960 :     }
 ... :
 ... :                ......
 ... :
2046 :
2047 :     if(dat_err != 0)   { ErrorMessage(0); }
2048 : }
```

Menüführung

Die Struktur der Menüführung mit Programmauswahl und Parametereinstellung wurde bereits auf Seite 11 erklärt. Die Abbildung 2.2 gibt einen schematischen Überblick, auf den sich auch die folgende Beschreibung lehnt. Die Grundelemente des µC-Programms sind zwei Interrupt - Routinen, die von dem Auswahldrehschalter/-taster aktivierte werden. In

dieser Beschreibung werden die beiden Funktionen separat behandelt, wobei zunächst die Steuerung mithilfe des Auswahltasters betrachtet wird. Der Taster ist hardwaremäßig mit dem Pin 1.2 des Mikrokontrollers verbunden und löst einen Interrupt aus, wenn er gedrückt wird.

interrupt (PORT1_VECTOR) INT_P1(void)

Der Mikrokontroller fasst alle Interruptquellen eines Ports zu einem Interrupt-Vektor zusammen, wobei eine Differenzierung der einzelnen Quellen erst innerhalb des Unterprogramms stattfinden kann. Der Port1 wird neben dem Auswahltaster auch für den Eingang des externen Triggers verwendet, dessen Codeteil (von Zeile 212 bis 227) später beschrieben wird.

Die Menüführung ist in vier Ebenen gegliedert, welche mit der Variable *status* festgelegt werden. In der folgenden Auflistung werden die möglichen Zustände kurz erklärt.

1. SET — Dieser Teil beinhaltet alle Einstellmöglichkeiten des Stimulators, so werden an dieser Stelle die Programme ausgewählt und in weiterer Folge die Stimulationsparameter gesetzt. Die dazugehörigen Unterprogramme werden im Anschluss an diesem Abschnitt beschrieben.
2. ReadyToRun — In dieser Ebene werde die Stimulationsprogramme in StimTA bzw. StimTB Variablen umgerechnet und die Daten werden noch einmal kontrolliert
3. RUN — Beginn der Muskelstimulation, wobei je nach ausgewähltem Programm die nötigen Features eingestellt werden müssen, wie zum Beispiel die Aktivierung der ADC - Interrupt Routine.
4. STOP — Alle Stimulationsprogramme können zu jeder Zeit mit dem Taster abgebrochen werden. Die Methode StimSTOP(…) wird aufgerufen und die Ausgänge werden in einen sicheren Zustand gebracht. Danach kommt man automatisch in die erste Menüebene und kann wieder ein Stimulationsprogramm wählen. Wenn ein Stimulationsprogramm eine endliche Stimulationszeit besitzt, kommt man nach ablaufen dieser Zeit auch automatisch in das Hauptmenü.

```
208 : interrupt (PORT1_VECTOR) INT_P1(void)
209 : {
210 :     int i;
211 :
212 :     if(P1IFG & BIT7)     {
... :
... :                   // PROGRAMMTEIL - SIEHE  EXTERNER TRIGGER //
... :
227 :     }
228 :
229 :
```

```
230 :     if(P1IFG & BIT2) {
231 :
232 :         if(ReceiveFlag == 1)      {
233 :             for(i=0;i<10;i++) buff[i]=0;
234 :             if(indata != NULL) free(indata);
235 :
236 :             PCToStim = ReceiveHello;
237 :             ReceiveFlag = 0;
238 :             once = 1;
239 :
240 :             menu_push = SetProg;
241 :             status = SET;
242 :         }
243 :
244 :
245 :         switch(status)  {
246 :
247 :              case SET:
248 :                     switch(menu_push)  {
249 :                         case SetProg:   SetProgram();       break;
250 :                         case SetCh:     SetChannel();       break;
251 :                         case SetPulse:  SetPulseWidth();    break;
252 :                         case SetFreq:   SetFrequency();     break;
253 :                         case SetAmp:    SetAmplitude();     break;
254 :                     }
255 :                     break;
256 :
257 :
258 :              case ReadyToRUN:
259 :                    ReadyOut();
260 :                    status = RUN;
261 :                    break;
262 :
263 :              case RUN:
264 :                     if(NoProg <= 2)  {
265 :                         ADC12CTL1 &= ~CONSEQ_3;
266 :                         ADC12CTL1 |=  CONSEQ_1;
267 :                     }
268 :                     if(NoProg == 0)   {
269 :                         if(menu_rot != StopTwitch)
270 :                            lcdScreen("Twitch ->|","main menu <->");
271 :                         menu_rot = StopTwitch;
272 :                     } else {
273 :                         status = STOP;
274 :                         menu_rot = BlockRotary;
275 :                         lcdScreen("Stimulation ","STOP: Push ->|");
276 :                     }
277 :                     if(NoProg == 2)     {
278 :                         P1IE |= 0x80;
279 :                         extTrig = 1;
280 :                     } else {
281 :                         if(STM_ProgInfo == BothIndep)   {
282 :                             TB_status = stm_PulseStart;
283 :                             TBCTL |= MC_1;
284 :                             (*StimTA).OUT1 |= (*StimTB).OUT1;
285 :                         }
286 :                         TA_status = stm_PulseStart;
287 :                         TACTL |= MC_1;
288 :                         P3OUT |= (*StimTA).OUT1;
289 :                     }
290 :                     break;
291 :
292 :              case STOP:
293 :                     StimSTOP(0);
294 :                     break;
295 :         }
296 :         P1IFG &= ~BIT2;
297 :     }
298 :     P1IFG = 0;
299 : }
```

Die Elemente der Ebene „SET“ werden noch genauer beschrieben, da sie einen umfangreichen Teil der µC - Software ausmachen

void SetProgram(void)

Dieses Unterprogramm setzt die Variable NoProg (Number of Program) und gibt das aktuelle Stimulationsprogramm auf der LCD-Anzeige aus. Der Werteberich der Variable NoProg geht von 0 bis 8, mit folgender Zuweisung:

0 – Twitch	3 – 1. Programm	6 – 4. Programm
1 – externer Trigger	4 – 2. Programm	7 – 5. Programm
2 – internes Timing	5 – 3. Programm	8 – Amplituden

Tabelle 2.4: Variable NoProg

Mit den folgenden Methoden werden die Stimulationsparameter eingestellt und werden nur während der Programme - „Twitch, externer Trigger und internes Timing“ verwendet.

void SetChannel(void)

Die Auswahl des Kanals wird in der Variable *ProgInfo* gespeichert, die im weitern Programmablauf sehr wichtig ist. Die Zahlenwerte dieser Variable sind folgendermaßen zugeordnet:

0 (no data)	–	Stimulationsprogramm enthält keine Daten
1 (OnlyCh1)	–	es wird ausschließlich der Kanal 1 verwendet
2 (OnlyCh2)	–	es wird ausschließlich der Kanal 2 verwendet
3 (BothSame)	–	beide Kanäle besitzen ein identische Timing
4 (BothIndep)	–	beide Kanäle könne völlig unabhängig von einander gesteuert werden

Tabelle 2.5: Variable ProgInfo

void SetPulseWidth(void), void SetFrequency(void)

Beide Unterprogramme sind von der Struktur ident und zeigen den Ausgangszustande der jeweiligen Parameter, d. h., der aktuelle Parameterwert wird an dem LCD - Display angezeigt. Innerhalb dieser Methode gibt es für die Navigation zwei wichtige Variablen *jumpto* und *rotat*. Die zweite Variable legt den Parameter fest, der zur Editierung aktiviert wurde. Der Parameterwert wird mit dem Drehschalter eingestellt, in der dazugehörigen Interrupt - Routine. Die Variable *jumpto* zeigt auf den Parameter, der bei dem nächsten Drücken des Tasters angezeigt werden soll. Notwendig ist diese Variable, da die Parameteranzeige je nach Kanalauswahl variiert.

Der folgende Codeabschnitt zeigt nur die Methode SetFrequency, welche alle wesentlichen Strukturelementen zeigt.

```
727 : void SetFrequency(void)
728 : {
729 :     switch(jumpto)    {
730 :         case 0: lcdBlinkOff();
731 :                 lcdScreen("Set Stim-Frequency","change <->   confirm ->|");
732 :                 menu_rot = BlockRotary;
733 :                 jumpto = 1;
734 :                 Set.TimA[0].iStimFreq = 100;
735 :                 Set.TimB[0].iStimFreq = 100;
736 :
737 :                 break;
738 :         case 1:lcdClear();
739 :                 lcdPosL1(0);
740 :                 lcdOutString("Ch    StimFreq:      Hz ");
741 :                 ScreenCh(1);
742 :                 if(Set.ProgInfo == BothIndep) {
743 :                     lcdPosL2(0); lcdOutString("Ch2 - StimFreq:       Hz ");
744 :                     jumpto = 2;
745 :                 }
746 :                 else    {
747 :                     status = ReadyToRUN;
```

```
748 :                       jumpto = 0;
749 :                   }
750 :                   lcdPosL1(15);
751 :                   printf("%4d.%1d",(int)(Set.TimA[0].iStimFreq/10),Set.TimA[0].iStimFreq%10);
752 :                   lcdPosL1(23);
753 :                   lcdBlinkOn();
754 :                   menu_rot = SetFreq;
755 :                   rotat = 1;
756 :                   break;
757 :
758 :           case 2: lcdBlinkOff();
759 :                   lcdPosL2(0);
760 :                   lcdOutString("Ch2 - StimFreq:       Hz ");
761 :                   lcdPosL2(15);
762 :                   printf("%4d.%1d",(int)(Set.TimB[0].iStimFreq/10),Set.TimB[0].iStimFreq%10);
763 :                   lcdPosL2(23);
764 :                   lcdBlinkOn();
765 :                   menu_rot = SetFreq;
766 :                   rotat = 2;
767 :                   jumpto = 0;
768 :                   status = ReadyToRUN;
769 :                   break;
770 :       }
771 : }
```

void ScreenCh(int line)

Diese Methode wird von den beiden Unterprogrammen SetPulseWidth und SetFrequency verwendet, da je nach Kanalauswahl die richtige Bezeichnung auf dem LCD-Display ausgeben werden soll. Es gibt demnach drei verschiedene Kanalbezeichnungen : Ch1, Ch2 und Ch1+2.

interrupt(PORT2_VECTOR) INT_P2(void)

Diese Interrupt - Routine wird durch den Drehschalter angesteuert. Der Schalter belegt die Portpins 2.1, 2.2, 2.3, 2.4 und setzt durch ein Ereignis den Interrupt - Vektor des 2.Ports. Zu Beginn wird die Drehrichtung bestimmt und die Variable *rot* wird auf 1 gesetzt für eine Rechtsdrehung und -1 für die Gegenrichtung.

In weitere Folge werden die Parameter verändert, oder im Hauptmenü ein Stimulationsprogramm ausgewählt. Die Vorgangsweise ist dabei immer gleich, so wird mit den Variablen *menu_rot* und *rotat* der Parameter bestimmt und sein Wert wird anschließend mit einem Vielfachen von *rot* in- bzw. dekrementiert.

```
302 : interrupt (PORT2_VECTOR) INT_P2(void)
303 : {
304 :     static int rot = 0;
305 :
306 :     switch (P2IFG) {
307 :        case 1: if((P2IN & DREH_A) == DREH_A) rot =-1; else rot = 1;  break;
308 :        case 2: if((P2IN & DREH_A) == DREH_A) rot = 1; else rot =-1;  break;
309 :        case 4: if((P2IN & DREH_B) == DREH_B) rot = 1; else rot =-1;  break;
310 :        case 8: if((P2IN & DREH_B) == DREH_B) rot =-1; else rot = 1;  break;
311 :     }
312 :
313 :
314 :     if(menu_rot == SetProg)   {
315 :         NoProg += rot;
316 :         if(NoProg > 8) NoProg = 0;
317 :         if(NoProg < 0) NoProg = 8;
318 :         SetProgram();
319 :     }
320 :
```

```
321 :      if(menu_rot == SetCh)    {
322 :          useChTmp += rot;
323 :          if(useChTmp > 3) useChTmp = 0;
324 :          if(useChTmp < 0) useChTmp = 3;
325 :          SetChannel();
326 :      }
327 :
328 :      if(menu_rot == SetPulse) {
329 :          if(rotat == 1) {
330 :              if(Set.TimA[0].uiPWidth > 595)  Set.TimA[0].uiPWidth += rot*100;
331 :              else                            Set.TimA[0].uiPWidth += rot*10;
332 :              if(Set.TimA[0].uiPWidth <= 80 || Set.TimA[0].uiPWidth >= 32760) {
333 :                 Set.TimA[0].uiPWidth = 80;
334 :                 lcdPosL2(0); lcdOutString("!! value out of range   ");
335 :              }
336 :              else { lcdPosL2(0); lcdOutString("Ch    Neg.Pulse:     us ");
337 :                     ScreenCh(2);
338 :              }
339 :              lcdPosL1(16);
340 :              printf("%5d",Set.TimA[0].uiPWidth);
341 :              lcdPosL1(23);
342 :          }
343 :          if(rotat == 2)    {
344 :              if(Set.TimA[0].uiNWidth > 595) Set.TimA[0].uiNWidth += rot*100;
345 :              else                           Set.TimA[0].uiNWidth += rot*10;
346 :              if(Set.TimA[0].uiNWidth <= 80 || Set.TimA[0].uiNWidth >= 32760) {
347 :                  Set.TimA[0].uiNWidth = 80;
348 :                  lcdPosL1(0); lcdOutString("!! value out of range   ");
349 :              }
350 :              else { lcdPosL1(0);  lcdOutString("Ch    Pos.Pulse:     us ");
351 :                     ScreenCh(1);
352 :                     lcdPosL1(16); printf("%5d",Set.TimA[0].uiPWidth);
353 :              }
354 :              lcdPosL2(16);
355 :              printf("%5d",Set.TimA[0].uiNWidth);
356 :              lcdPosL2(23);
357 :          }
... :
... :                               // ... ähnlicher Teil
... :
400 :      }
401 :      if(menu_rot == SetFreq) {
402 :
403 :         if(rotat == 1) {
404 :             if(Set.TimA[0].iStimFreq >= 1)     {
405 :                  if( Set.TimA[0].iStimFreq >= 20)     {
406 :                      if( Set.TimA[0].iStimFreq >= 600)    {
407 :                           Set.TimA[0].iStimFreq += 50*rot;
408 :                      } else {
409 :                           Set.TimA[0].iStimFreq += 10*rot;
410 :                      }
411 :                  } else {
412 :                      Set.TimA[0].iStimFreq += rot;
413 :                  }
414 :              }
415 :              if(Set.TimA[0].iStimFreq < 1) {
416 :                  Set.TimA[0].iStimFreq = 1;
417 :                  lcdPosL2(0); lcdOutString("!! value out of range   ");
418 :              } else {
419 :                  if(Set.ProgInfo == BothIndep)     {
420 :                      lcdPosL2(0); lcdOutString("Ch2 - StimFreq:       Hz ");
421 :                  }else {
422 :                      lcdPosL2(0); lcdOutString("                        ");
423 :                  }
424 :              }
425 :               lcdPosL1(15);
426 :               printf("%4d.%1d",(int)(Set.TimA[0].iStimFreq/10),Set.TimA[0].iStimFreq%10);
427 :               lcdPosL1(23);
428 :          }
429 :          if(rotat == 2)    {
... :
... :                               // ... ähnlicher Teil
... :
456 :
457 :      }
458 :
459 :      if(menu_rot == StopTwitch)    {
460 :          StimSTOP(0);
461 :      }
462 :
463 :      P2IFG = 0;
464 : }
```

Ausgang - Stimulationsimpulse

Bevor die Stimulation gestartet werden kann, müssen die Parameter noch in den hardwarenahen Type StimData umgerechnet werden. Beide Timer (A und B) besitzen eine eigene Datenvariable dieses Typs - *StimTA* bzw. *StimTB*. Die Vorteile dieser Vorgansweise liegen darin, dass die Stimulationsimpulse schneller berechnet werden können und eine strikte Trennung zwischen den Daten und der Stimulationsmethode besteht.

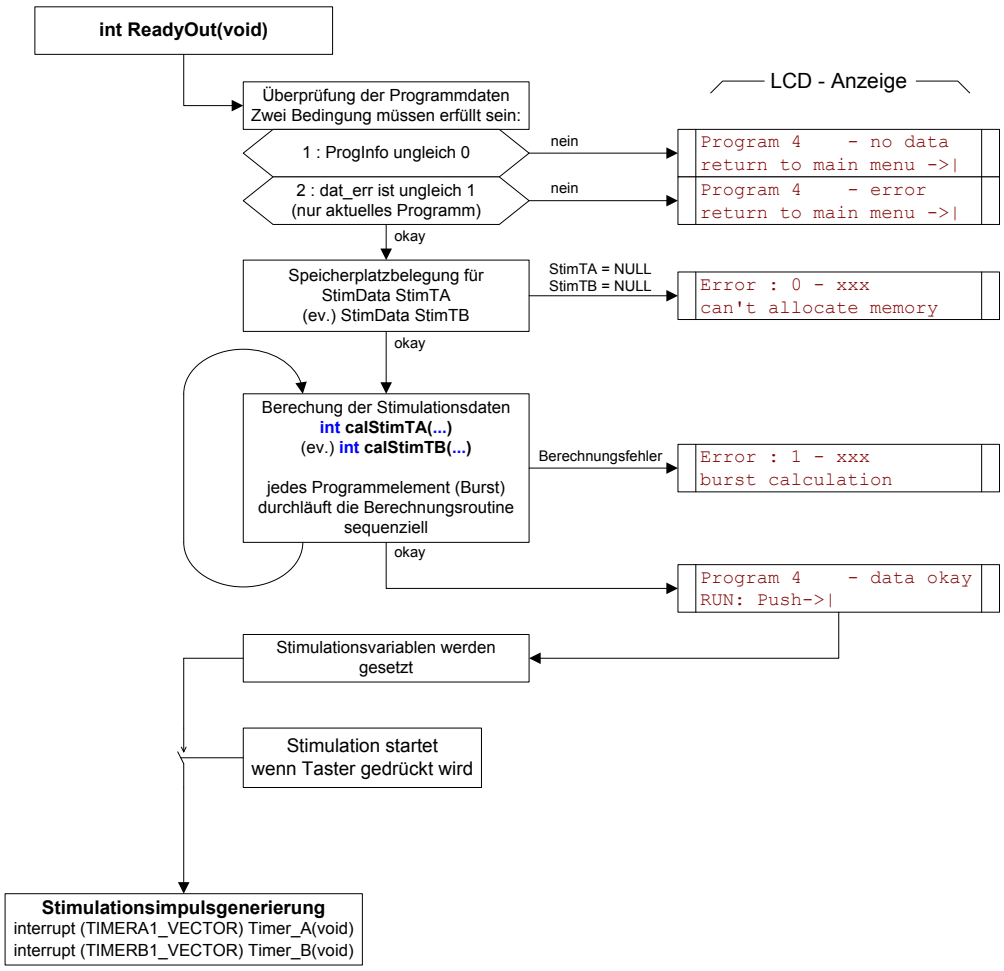


Abbildung 2.12: Flussdiagramm der StimTA, StimTB Berechnung

Das Unterprogramm **ReadyOut(…)** wird im folgenden Flussdiagramm dargestellt und beinhaltet im Wesentlichen die eben erwähnte Parameterkonvertierung. Im ersten Schritt wird der Programmstatus abgefragt, der in den Variablen *ProgInfo* und *dat_err* zu finden ist. Der Datenvektor *dat_err* wird ganz speziell auf das vorliegende Programm untersucht, d. h., falls ein Programm fehlerhaft ist, können trotzdem alle anderen verwendet werden.

Die Variablen *StimTA* und *StimTB* müssen immer neu generiert werden, aus diesem Grund werden die Speicherregionen zunächst mit dem Befehl free(…)freigegeben und danach ein neuer Speicherblock mit der aktuellen Programmgröße belegt. Wenn kein

Speicher allokiert werden kann, wir die Stimulation abgebrochen und man kehrt in die erste Menüebene zurück.

Im Anschluss dieser Vorbereitungen werden die Daten umgerechnet in den Type StimData. Das Unterprogramm calStimTA(…) und calStimTB(…) konvertieren die Stimulationsparameter, indem jeder Burst des Programms sequenziell bearbeitet wird.

```
1369 :    case 6:
1370 :      for(n=0;n<STM_SizeTA; n++)   {
1371 :           cal_err |= calStimTA(n, &B4.TimA[n], &B4.iAmp_a[n], &B4.iAmp_b[n]);
1372 :      }
1373 :      if(STM_ProgInfo == 4)     {
1374 :           for(n=0;n<STM_SizeTB; n++)   {
1375 :                cal_err |= calStimTB(n, &B4.TimB[n], &B4.iAmp_b[n]);
1376 :           }
1377 :      }            break;
```

Im Anschluss an die Umrechnung werden die Startzustände für die Stimulation gesetzt. Für die Programme, „Twitch“ - „int. Timing“ und „ext. Trigger“ werden an dieser Stelle die ADC - Interrupt Routine aktiviert.

int calStimTA(…), int calStimTB(…)

Der folgende Abschnitt nimmt nur auf die Methode calStimTA(…) einen direkten Bezug. Das Unterprogramm calStimTB(…) besitzt die gleiche Struktur wie die Methode für den Timer A, jedoch kommt sie nur für Programme mit unabhängigen Timing zum Einsatz. Diese Funktion muss daher nur die Amplitude für den Kanal 2 umrechnen und braucht in dem Programmteil <5> (siehe Zeile 1491) keine weiteren Unterscheidungen zu treffen.

```
1434 : int calStimTA(int i, TData* tmp, int* Amp1, int* Amp2)
1435 : {
1436 :      int Out1 = 0;
1437 :      int Out2 = 0;
1438 :      unsigned long modulo;
1439 :      unsigned long ultmp;
1440 :
1441 : // ----------- <1> control input values
1442 :     if(tmp == NULL) return 0x02;
1443 :     if((*tmp).iStimFreq > 20000 || (*tmp).iStimFreq < 1) { if((*tmp).iStimFreq != Twitch) ►
                return 0x10; }
1444 :     if((*tmp).uiNWidth < 80 || (*tmp).uiPWidth < 80)     return 0x20;
1445 :     if(((*tmp).uiNWidth + (*tmp).uiPWidth) > 32700)      return 0x40;
1446 : // ----------- <end 1>
1447 :
1448 : // ------------- <2> calculate pulse length
1449 :     (*(StimTA+i)).uiTCCR0_s = (*tmp).uiPWidth + (*tmp).uiNWidth;
1450 :
1451 :     if((*tmp).cPol ==  'E')   (*(StimTA+i)).uiTCCR1_s = (*tmp).uiPWidth;
1452 :     else                        (*(StimTA+i)).uiTCCR1_s = (*tmp).uiNWidth;
1453 : // ------------- <end 2>
1454 :
1455 : // ------------- <3> calculate register value of the time between two impulses
1456 :      if(NoProg == 0 || NoProg == 2) {
1457 :         (*(StimTA+i)).uiTCCR0_1 = 0;
1458 :         (*(StimTA+i)).uiTCCR0_2 = 0;
1459 :      } else {
1460 :           if((*tmp).iStimFreq == Twitch){    ►
                        (*(StimTA+i)).ulStimPer =  (unsigned long)(1000 * (*tmp).fStimTime  );}
1461 :           else {  (*(StimTA+i)).ulStimPer =  (unsigned long)(10000000/(*tmp).iStimFreq);}
1462 :
1463 :           ultmp = (unsigned long)( (*(StimTA+i)).ulStimPer - (*(StimTA+i)).uiTCCR0_s );
1464 :           modulo = ultmp % 0xFFFF;
1465 :           if(ultmp >= 0xFFFF) {
1466 :                          modulo += 0xFFFF; modulo /= 2;
1467 :                          (*(StimTA+i)).uiTCCR0_1 = (unsigned int)(modulo);
1468 :           } else {  (*(StimTA+i)).uiTCCR0_1 = (unsigned int)(modulo);
1469 :           }
```

```
1470 :            (*(StimTA+i)).ulStimTime  = (unsigned long)((*tmp).fStimTime*1000);
1471 :       }
1472 : // ------------- <end 3>
1473 :
1474 : // ------------- <4> calculate register value of the time between two impulses
1475 :     if((*tmp).fPauseTime > 0.08)    {
1476 :            (*(StimTA+i)).ulPauseTime = (unsigned long)((*tmp).fPauseTime*1000);
1477 :            ultmp = (*(StimTA+i)).ulPauseTime;
1478 :            modulo = ultmp % 0xFFFF;
1479 :            if(ultmp >= 0xFFFF) {
1480 :                 modulo += 0xFFFF; modulo /= 2;
1481 :                 (*(StimTA+i)).uiTCCR0_2 = (unsigned int)(modulo);
1482 :            } else {
1483 :                 (*(StimTA+i)).uiTCCR0_2 = (unsigned int)(modulo);
1484 :            }
1485 :     }
1486 :      else {
1487 :            (*(StimTA+i)).uiTCCR0_2 = 0;
1488 :       }
1489 : // ------------- <end 4>
1490 :
1491 : // ------------ <5> define output values
1492 :      switch(STM_ProgInfo){
1493 :            case 1:
1494 :                 if( (*tmp).cPol == 'E') Out1 = 0x02;
1495 :                 else                    Out1 = 0x04;
1496 :                 Out2 = 0x06; break;
1497 :            case 2:
1498 :                 if( (*tmp).cPol == 'E') Out1 = 0x08;
1499 :                 else                    Out1 = 0x10;
1500 :                 Out2 = 0x18; break;
1501 :            case 3:
1502 :                 if( (*tmp).cPol == 'E') Out1 = 0x0A;
1503 :                 else                    Out1 = 0x14;
1504 :                 Out2 = 0x1E; break;
1505 :            case 4:
1506 :                 if( (*tmp).cPol == 'E') Out1 = 0x02;
1507 :                 else                    Out1 = 0x04;
1508 :                 Out2 = 0x06; break;
1509 :       }
1510 :       (*(StimTA+i)).OUT1 = Out1;
1511 :       (*(StimTA+i)).OUT2 = Out2;
1512 : // ------------- <end 5>
1513 :
1514 : // ------------ <6> check and set the amplitude
1515 :       if(*Amp1 > 1000) *Amp1 = 1000;
1516 :       if(*Amp2 > 1000) *Amp2 = 1000;
1517 :       if(*Amp1 < 0)    *Amp1 = 0;
1518 :       if(*Amp2 < 0)    *Amp2 = 0;
1519 :
1520 :       (*(StimTA+i)).iSAmp_a = *Amp1;
1521 :       if(STM_ProgInfo == 3)  (*(StimTA+i)).iSAmp_b = *Amp2;
1522 :       else                   (*(StimTA+i)).iSAmp_b = 0;
1523 : // ------------ <end 6>
1524 :
1525 :
1526 :     return 0;
1527 : }
```

In dem ersten Programmabschnitt <1> werden die Wertebereiche der einzelnen Variablen kontorlliert. Diese Überprüfung soll vor nicht erkennbaren Variablenüberläufe und vor Programmabstürzen schützen, wenn mathematische Operationen nicht ausgeführt werden können.

Die Berechnung Timervariablen für die Stimulationsimplusform besitzt die Schwierigkeit zwei deutlich unterschiedlichen Zeitspannen zu kombinieren. Die nötige Auslösung der Impulsbreite bestimmt die Timerfrequenz. In diesem konkreten Fall ist die Frequenz auf 1 MHz gesetzt, das bedeute die kleinste Zeiteinheit entspricht 1 µs. Für die positve und negative Impulsbreite werden die beide Variablen *uiTCCR0_s* und *uiTCCR1_s* gestetzt, wobei innerhalb eines Timerüberlaufes der gesamte Impuls abeschlossen sein

muss, das bedeute der längstmögliche Impuls (positve und negative Impulsbreite addiert) dauert 32,7 m.

Die folgenden Abschnitte zur Berechnung der Stimulationsperiode <3> und der Pausendauer <4> besitzt die selbe Struktur und wird gemeinsam berschrieben. Zur Verdeutlichung zeigt die Abbildung 2.13 den Zusammenhang zwischen dem Timer Runregister und der Ausgangsimpulse.

Die gesamte Zeitspanne wird zunächst auf die Timereinheit umgerechnet. Die Variablen *uiTCCR0_1* und *uiTCCR0_2* bilden dabei das Ergebnis der der Modulo-Funkiton (mit einem maximalen Timerüberlauf 0xFFFF). Falls das Ergenbis der mathematischen Operation zu klein ist (im konkreten Fall kleiner als 80µs) würde das Programm in der Interrupt Routine hängen bleiben. Um diesem Fehler vorzubeugen werden die Variablen mit einem Wert 0xFFFF gemittelt, dieser Vorgang wird nur durchgeführt, wenn die Zeitpanne größer als 0xFFFF Timereinheiten ist.

Das Beispiel zeigt alle drei möglichen Fälle, die in der Abbildung von links nach rechts dargestellt sind:

1. TCCR0_1 Die Variablenlänge wurde gemittelt und somit sind zählt das Timerregisterzweimal bis zu dem Wert von TCCR0_1 hoch.
2. TCCR0_2 Die Variable wird zunächst auf den gemittelten Modulowert gesetzt, nachdem der Rest durchlaufen wurde, läuft das Timerregister bis zur maximalen Länge.
3. TCCR0_1 Die Zeitspanne ist unter den 32,7 ms, daher braucht man nur einen Timerdurchlauf.

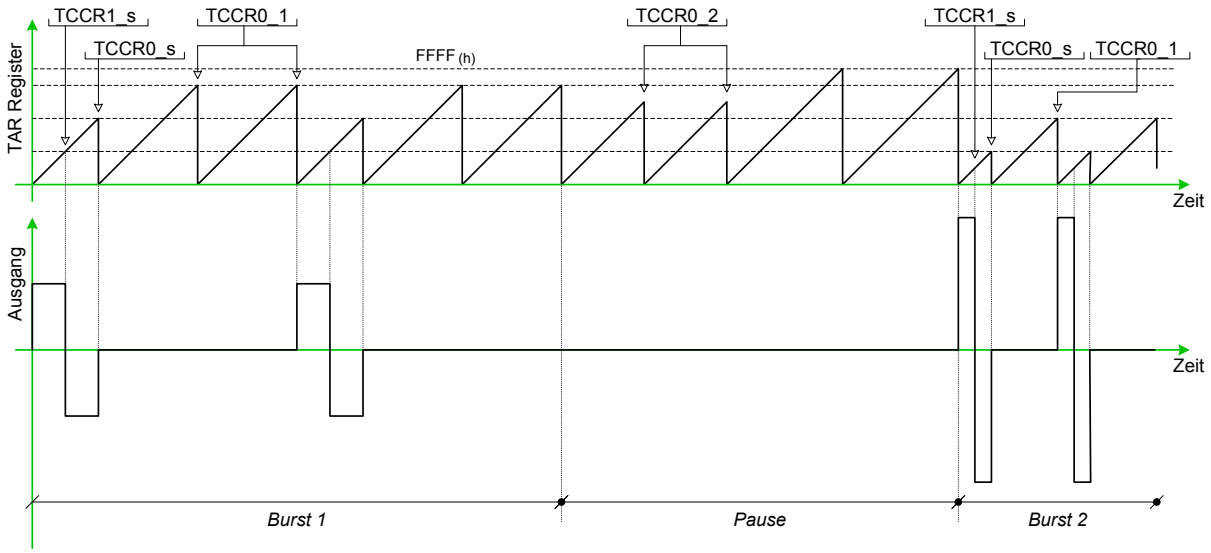


Abbildung 2.13: TAR Registerverlauf zur Impulsgenerierung

interrupt (TIMERA1_VECTOR) Timer_A(), interrupt (TIMERB1_VECTOR) Timer_B()

Der folgende Source Code zeigt die Interrupt Routine des Timers A. Da der Programmteil des Timers B sehr ähnlich ist, wird er nicht explizit dargestellt.

```
1025 : interrupt (TIMERA1_VECTOR) Timer_A(void)
1026 : {
1027 :    char DACValue_out = 0;
1028 :    int newNoB;
1029 :
1030 :     switch(TAIV)    {
1031 :        case 0xA:
1032 :             if( NoProg == 0 || NoProg == 2)   {
1033 :                 P3OUT &= ~(*StimTA).OUT2;
1034 :                 TACTL &= ~MC_1;
1035 :                 ADC12CTL1 |= CONSEQ_3;
1036 :                 ADC12CTL0 |= ENC+ADC12SC;
1037 :             }
1038 :             else    {
1039 :              switch(TA_status) {
1040 :                 case stm_PulseEnd:
1041 :                    P3OUT &= ~(*(StimTA+NoB_TA)).OUT2;
1042 :                    TACCR0 = (*(StimTA+NoB_TA)).uiTCCR0_1;
1043 :                    ulPerTA = 0; itTA = 0;
1044 :
1045 :                    if(NoProg == 1) {
1046 :                       ADC12CTL0 |= ENC+ADC12SC;
1047 :                       ulTmpTA = 1;
1048 :                    } else {
1049 :                       ulTmpTA += (*(StimTA+NoB_TA)).ulStimPer;
1050 :                       if(ulTmpTA > (*(StimTA+NoB_TA)).ulStimTime) {
1051 :                          ////////  set trigger for m-wave measurement  ////////
1052 :                          DACValue_out = 1;
1053 :                          ulTmpTA = 0;
1054 :                       }
1055 :                    }
1056 :                    TA_status = stm_InterPulse;
1057 :                    break;
1058 :                 case stm_InterPulse:
1059 :                    itTA++;
1060 :                    if(itTA <= 2)    {
1061 :                       ulPerTA += (*(StimTA+NoB_TA)).uiTCCR0_1;
1062 :                    }else{
1063 :                       TACCR0 = 0xFFFF;
1064 :                       ulPerTA += 0xFFFF;
1065 :                    }
1066 :                    if(ulPerTA >=(*(StimTA+NoB_TA)).ulStimPer-(*(StimTA+NoB_TA)).uiTCCR0_s) {
1067 :                       if(ulTmpTA == 0)   {
1068 :                          if((*(StimTA+NoB_TA)).uiTCCR0_2 < 2) {
1069 :                           TA_status = stm_NewBurst;
1070 :                          } else {
1071 :                           TACCR0 = (*(StimTA+NoB_TA)).uiTCCR0_2;
1072 :                           TA_status = stm_StimBreak;
1073 :                          }
1074 :                       } else {
1075 :                          TA_status = stm_PulseStart;
1076 :                          TACCR0 = (*(StimTA+NoB_TA)).uiTCCR0_s;
1077 :                       }
1078 :                       itTA = 0;
1079 :                    }
1080 :                    break;
1081 :                 case stm_StimBreak:
1082 :                    itTA++;
1083 :                    if(itTA <= 2)    {
1084 :                       ulTmpTA += (*(StimTA+NoB_TA)).uiTCCR0_2;
1085 :                    } else {
1086 :                       TACCR0 = 0xFFFF;
1087 :                       ulTmpTA += 0xFFFF;
1088 :                    }
1089 :                    if(ulTmpTA >= (*(StimTA+NoB_TA)).ulPauseTime)  {
1090 :                       TA_status = stm_NewBurst;
1091 :                    }
1092 :                    break;
1093 :              }
1094 :           }
1095 :
1096 :           if(TA_status == stm_NewBurst)  {
1097 :              NoB_TA++;
1098 :              if(NoB_TA >= STM_SizeTA) { NoB_TA = 0;     repTA++; }
1099 :              TACCR0 = (*(StimTA+NoB_TA)).uiTCCR0_s;
1100 :              TACCR1 = (*(StimTA+NoB_TA)).uiTCCR1_s;
1101 :              TA_status = stm_PulseStart;
1102 :           }
1103 :
```

```
1104 :            if(DACValue_out == 1)     {
1105 :               newNoB = NoB_TA + 1;
1106 :               if(newNoB >= STM_SizeTA) newNoB = 0;
1107 :                  switch(STM_ProgInfo) {
1108 :                      case 1: DAC12_0DAT = (*(StimTA+newNoB)).iSAmp_a * 4; break;
1109 :                      case 2: DAC12_1DAT = (*(StimTA+newNoB)).iSAmp_a * 4; break;
1110 :                      case 3: DAC12_0DAT = (*(StimTA+newNoB)).iSAmp_a * 4;
1111 :                              DAC12_1DAT = (*(StimTA+newNoB)).iSAmp_b * 4; break;
1112 :                      case 4: DAC12_0DAT = (*(StimTA+newNoB)).iSAmp_a * 4; break;
1113 :                  }
1114 :               DACValue_out = 0;
1115 :            }
1116 :
1117 :              if(repTA >= STM_RepTA && STM_RepTA != ContStim)  {
1118 :                  TACTL &= ~MC_1;
1119 :               P3OUT &= ~(*(StimTA+NoB_TA)).OUT2;
1120 :                  StimEndFlag++;
1121 :                  if(StimEndFlag >= 2) StimSTOP(0);
1122 :              } else {
1123 :               if(TA_status == stm_PulseStart ) {
1124 :                   P3OUT |= (*(StimTA+NoB_TA)).OUT1;
1125 :                   ulTmpTA = 0;
1126 :               }
1127 :            }
1128 :            break;// <end of TAIV == 0xA>
1129 :
1130 :         case 0x2:
1131 :              if(TA_status == stm_PulseStart)   {
1132 :                  P3OUT ^= (*(StimTA+NoB_TA)).OUT2;
1133 :               TA_status = stm_PulseEnd;
1134 :              }
1135 :              break;    // <end of TAIV == 0x2>
1136 :      }
1137 : }
```

Externer Trigger

Der Trigger ermöglicht eine externe Steuerung der Stimulationsfrequenz und ist über eine galvanische Trennung mit dem Pin 1.7 des Mikrokontrollers verbunden. Bei der Beschreibung der Menüführung wurde bereits hingewiesen, dass der Interruptvektor nur für ein gesamtes Port zu Verfügung steht. In diesem Teil wird die Funktion der externen Triggerung beschrieben.

Der externe Trigger wird nur dann Aktiv, falls das gleichnamige Stimulationsprogramm ausgewählt ist. Wenn eine Stimulation gestartet wird, d. h., die Variable *status* ist gleich RUN, kommt es zur Freigabe dieser Routine, indem extTrig = 1 gesetzt wird.

Nach einer Sicherheitskontrolle wird der Timer A und eventuell der Timer B gestartet und der Stimulationsimpuls wird generiert. Bei der vorangegangenen Kontrolle wird überprüft, ob der letzte Impuls bereits beendet ist. Im Fehlerfall wird die Stimulation abgebrochen und auf dem LCD - Display wird die Fehlermeldung (Error : 2) ausgegeben.

```
208 : interrupt (PORT1_VECTOR) INT_P1(void)
209 : {
210 :     int i;
211 :
212 :     if(P1IFG & BIT7)      {
213 :         if(extTrig == 1) {
214 :            if((P3OUT & 0x1E) != 0) { extTrig = 0; P1IE &= ~0x80; ErrorMessage(2); }
215 :            else{
216 :              if(STM_ProgInfo == BothIndep)   {
217 :                 TB_status = stm_PulseStart;
218 :                 (*StimTA).OUT1 |= (*StimTB).OUT1;
219 :                  TBCTL |= MC_1;
220 :               }
221 :                 TACTL |= MC_1;
222 :                 P3OUT |= (*StimTA).OUT1;
```

```
223 :                 TA_status = stm_PulseStart;
224 :               }
225 :           }
226 :           P1IFG &= ~BIT7;
227 :       }
228 :
229 :
230 :       if(P1IFG & BIT2) {
231 :
... :
... :                   // PROGRAMMTEIL  SIEHE  MENÜFÜHRUNG //
... :
297 :       }
298 :       P1IFG = 0;
299 : }
```

ADC – Einstellungen

In dem Menüpunkt „Amplitudeneinstellung“ kann der Amplitudenbereich in zwei Schritten bestimmt werden. Der dazugehörige Abschnitt beginnt ab der Zeile 473 und endet bei 490.

Zu der Einstellung des Maximalwertes werden beide Potenziometer in den rechten Anschlag gebracht und danach wird diese Position mit dem Taster bestätigt
Der Amplitudenbereich kann in dem Intervall von 0 bis 100% beliebig eingestellt werden

Der restliche Teil der Interrupt - Routine (Zeile 493 - 510) beinhaltet die Amplitudensteuerung während der Stimulation. Dabei ist vor allem die Kanalzuweisung wichtig und zur Steigerung der Sicherheit werden die Amplitudenwerte noch einmal überprüft.

```
466 : interrupt (ADC_VECTOR) ADC12ISR(void)
467 : {
468 :     float ftmp1=0,ftmp2=0;
469 :
470 :     ftmp1 = (float)(ADC12MEM0);     // Move results, IFG is cleared
471 :     ftmp2 = (float)(ADC12MEM1);     // Move results, IFG is cleared
472 :
473 :     if(NoProg == 8) {
474 :
475 :         if(jumpto == 1) {
476 :             maxAmp1 = ftmp1;
477 :             maxAmp2 = ftmp2;
478 :
479 :             *Set.iAmp_a = (int)(maxAmp1/4.096);
480 :             *Set.iAmp_b = (int)(maxAmp2/4.096);
481 :         }
482 :
483 :         if(jumpto == 2) {
484 :             k_Amp1 = 1000*ftmp1/(maxAmp1*maxAmp1);
485 :             k_Amp2 = 1000*ftmp2/(maxAmp2*maxAmp2);
486 :
487 :             *Set.iAmp_a = (int)(1000*ftmp1/maxAmp1+0.1);
488 :             *Set.iAmp_b = (int)(1000*ftmp2/maxAmp2+0.1);
489 :         }
490 :     }
491 :     else    {
492 :
493 :           if(STM_ProgInfo == 2)     {
494 :                *Set.iAmp_a = (int)(ftmp2*k_Amp2+0.1);
495 :                if(*Set.iAmp_a > 1000) *Set.iAmp_a = 1000;
496 :                if(*Set.iAmp_a < 0)    *Set.iAmp_a = 0;
497 :                DAC12_1DAT = (int)(*Set.iAmp_a*4);
498 :           } else {
499 :                *Set.iAmp_a = (int)(ftmp1*k_Amp1+0.1);
500 :                if(*Set.iAmp_a > 1000) *Set.iAmp_a = 1000;
501 :                if(*Set.iAmp_a < 0)    *Set.iAmp_a = 0;
502 :                DAC12_0DAT = (int)(*Set.iAmp_a*4);
503 :           }
504 :
505 :           if(STM_ProgInfo >= 3)     {
```

```
506 :                 *Set.iAmp_b = (int)(ftmp2*k_Amp2+0.1);
507 :                 if(*Set.iAmp_b > 1000) *Set.iAmp_b = 1000;
508 :                 if(*Set.iAmp_b < 0)  *Set.iAmp_b = 0;
509 :                 DAC12_1DAT = (int)(*Set.iAmp_b*4);
510 :             }
511 :     }
512 : }
```

Das folgende Unterprogramm liefert für die Auswahl des Amplitudenbereichs die nötigen Display Ausgabe und setzt die ADC - Eigenschaften.

```
515 : void SetAmplitude(void)
516 : {
517 :     menu_rot = BlockRotary;
518 :
519 :     switch(jumpto)  {
520 :         case 0:
521 :             ADC12CTL1 |=  CONSEQ_3; ADC12CTL0 |= ENC+ADC12SC;
522 :             lcdScreen("max position","to confirm ->|");
523 :             Set.ProgInfo = BothIndep;
524 :             LCDShowAmpl = 1;
525 :             jumpto=1;
526 :             break;
527 :
528 :         case 1:
529 :             lcdScreen("amp. range","to confirm ->|");
530 :             jumpto=2;
531 :             break;
532 :
533 :         case 2:
534 :             lcdScreen("back to menu","press button ->|");
535 :             ADC12CTL0 &= ~(ENC+ADC12SC);
536 :             jumpto=0;
537 :             menu_push = SetProg;
538 :             NoProg = 0;
539 :             LCDShowAmpl = 0;
540 :             break;
541 :     }
542 : }
```

Während der Stimulation werden die aktuellen Amplitudenwerte auf dem Display angezeigt. Dieses Feature steht nur für die Programme - Twitch, ext. Trigger und int. Timing zu Verfügung. Der folgende Codeteil ist ein Element der Hauptroutine, da diese Aufgabe nicht notwendigerweise einem fixen Zeitschema folgen muss. Die Variable *LCDShowAmpl* wird in dem Unterprogramm ReadyOut(…) gesetzt. In der Zeile 1768 wird gezeigt, wie man mit dem printf - Befehl, welcher nur ganze Zahlen verarbeiten kann, auch eine Zahl mit einer Kommastelle ausgeben kann.

```
1752 :         // ----------- <L2> show ADC-value live on LCD display
1753 :         if(LCDShowAmpl == 1)  {
1754 :             for (i=10; i>0; i--)    delay(10000);
1755 :         }
1756 :         if(LCDShowAmpl == 1)  {
1757 :             if(status == RUN && menu_rot != StopTwitch)    {
1758 :                 if(*Set.iAmp_a > 200 || *Set.iAmp_b > 200)  {
1759 :                     lcdPosL2(23); lcdBlinkOn();
1760 :                 }
1761 :                 else lcdBlinkOff();
1762 :             }
1763 :             else lcdBlinkOff();
1764 :
1765 :             if(NoProg == 8 || status == RUN || status == STOP)    {
1766 :                 if(Set.ProgInfo == BothSame || Set.ProgInfo == BothIndep)    {
```

```
1767 :                      lcdPosL1(13);
1768 :                      printf(" Ch1:%3i.%1i ",(int)(*Set.iAmp_a/10),*Set.iAmp_a%10);
1769 :                      lcdPosL2(13);
1770 :                      printf(" Ch2:%3i.%1i",(int)(*Set.iAmp_b/10),*Set.iAmp_b%10);
1771 :                  }
1772 :                  else    {
1773 :                      lcdPosL1(13);
1774 :                      if(Set.ProgInfo == OnlyCh1)
1775 :                           printf(" Ch1:%3d.%1i ",(int)(*Set.iAmp_a/10),*Set.iAmp_a%10);
1776 :                      if(Set.ProgInfo == OnlyCh2)
1777 :                           printf(" Ch2:%3d.%1i" ,(int)(*Set.iAmp_a/10),*Set.iAmp_a%10);
1778 :                  }
1779 :              }
1780 :          }
```

Fehlerprotokoll

Erkannte Fehler werden mithilfe des Unterprogramms ErrorMessage auf dem LCD-Display ausgegeben. Es werden drei verschiedene Fehlerquellen angezeigt. Die ersten beiden Fehlermeldungen werden im Text nach dem Code genauer beschrieben. Während der Stimulation mit den Programmen „Ext. Trigger“ und „Twitch“ kann es zu der Fehlermeldung [Error 2] kommen, wenn die Stimulationsfrequenz zu groß wird und die Impulse sich überlappen.

```
1631 :  void ErrorMessage(int j)
1632 :  {
1633 :      StimSTOP(1);
1634 :
1635 :      switch(j)  {
1636 :         case 0:     lcdScreen("Error : 0 -","can't allocate memory");
1637 :                     lcdPosL1(11);printf("%3d",dat_err);break;
1638 :         case 1:     lcdScreen("Error : 1 -","burst calculation");
1639 :                     lcdPosL1(11);printf("%3d",cal_err);break;
1640 :         case 2:     lcdScreen("Error : 2","stim pulses interleaving"); break;
1641 :      }
1642 :
1643 :      NoProg = 0;
1644 :      status = SET;
1645 :      menu_push = SetProg;
1646 :  }
```

Variablen: dat_err und com_err

Werden Stimulationsprogramme initialisiert oder neu geladen, so muss dafür der nötige Speicherplatz generiert werden. Die Fehlervariablen *dat_err* / *com_err* werden an der entsprechenden Bit-Stelle gleich 1 gesetzt, falls dem Stimulationsprogramm keine gültige Speicheradresse zugewiesen werden kann. Die Variable *com_err* ist nur während der Datenübertragung aktiv und wird danach auf die Variable *dat_err* überschrieben. Dies vermeidet nicht nur Redundanz, sondern erleichtert die Überwachung der einzelnen Programme auf diesen gravierenden Fehler.

Nr.	Hex-Wert	Variable	Nr.	Hex-Wert	Variable
0	0001 h	Set.Ch1	0	0001 h	indata[]
1	0002 h	Set.Ch2	1	0002 h	
2	0004 h	B1.Ch1	2	0004 h	B1.Ch1
3	0008 h	B1.Ch2	3	0008 h	B1.Ch2
4	0010 h	B2.Ch1	4	0010 h	B2.Ch1
5	0020 h	B2.Ch2	5	0020 h	B2.Ch2
6	0040 h	B3.Ch1	6	0040 h	B3.Ch1
7	0080 h	B3.Ch2	7	0080 h	B3.Ch2
8	0100 h	B4.Ch1	8	0100 h	B4.Ch1
9	0200 h	B4.Ch2	9	0200 h	B4.Ch2
10	0400 h	B5.Ch1	10	0400 h	B5.Ch1
11	0800 h	B5.Ch2	11	0800 h	B5.Ch2
12	1000 h		12	1000 h	
13	2000 h		13	2000 h	
14	4000 h		14	4000 h	
15	8000 h		15	8000 h	

Tabelle 2.6: Beschreibung: dat_err, com_err

Variable: cal_err

Die Variable *cal_err* zeigt erkannte Fehler an, die während einer Umrechnung von den Stimulationsparametern zu den hardwarenahen Werten gemacht werden. Die folgende Tabelle zeigt die Zuweisung der möglichen Fehler, die kurz in einem Beispiel erklärt wird.

Beispiel: Wenn die Stimulationsfrequenz und die positive Impulsbreite im ungültigen Wertebereich liegen, dann besitzt die Variable folgenden Wert:

$$cal\_err = \ 0000\ 0000\ 1100\ 0000_b = 00C0_h$$

Nr.	Hex-Wert	Variable	Fehlerbeschreibung
0	0001 h	StimTA	nötiger Speicherplatz konnte nicht generiert werden
1	0002 h		
2	0004 h	calTimerTA(*p,…)	*p ist Null-Pointer
3	0008 h		
4	0010 h	.fStimFreq	außerhalb des gültigen Wertebereichs
5	0020 h	.uiPWidth, .uiNWidth	außerhalb des gültigen Wertebereichs
6	0040 h	.uiPWidth + .uiNWidth	außerhalb des gültigen Wertebereichs
7	0080 h		
8	0100 h	StimTB	nötiger Speicherplatz konnte nicht generiert werden
9	0200 h		
10	0400 h	calTimerTB(*p,…)	*p ist Null-Pointer
11	0800 h		
12	1000 h	.fStimFreq	außerhalb des gültigen Wertebereichs
13	2000 h	.uiPWidth, .uiNWidth	außerhalb des gültigen Wertebereichs
14	4000 h	.uiPWidth + .uiNWidth	außerhalb des gültigen Wertebereichs
15	8000 h		

Tabelle 2.7: Beschreibung: cal_err

Serielle Datenübertragung

Das Flussdiagramm beschreibt den Ablauf der seriellen Übertragung. Die Vorgänge werden dabei in zwei Gruppen geteilt, in Senden und Empfangen von Daten.

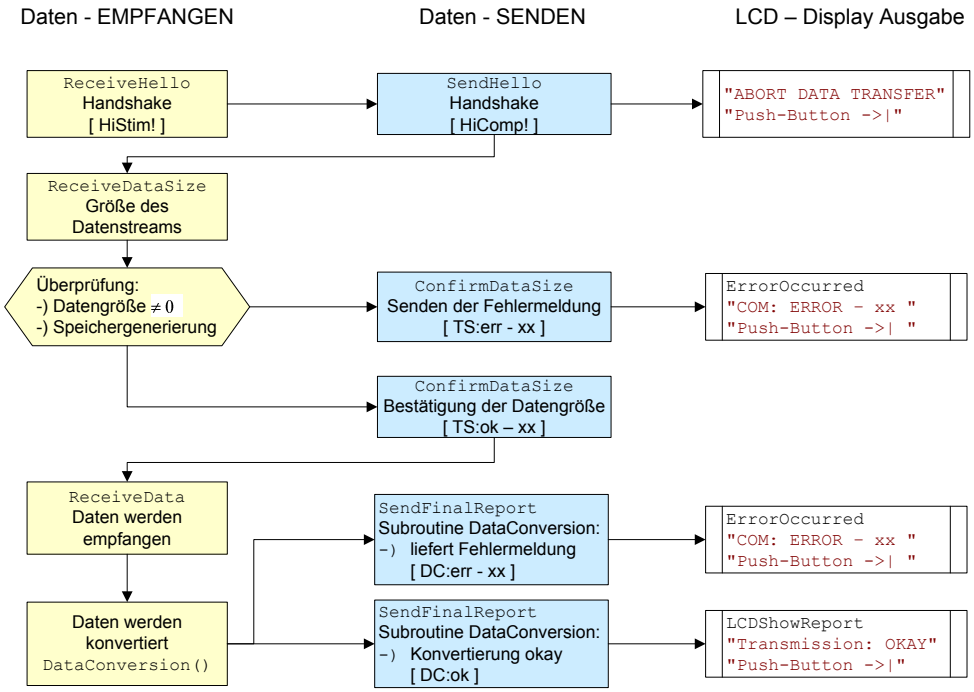


Abbildung 2.14: Flussdiagramm der seriellen Kommunikation

Empfangen von Daten

Die serielle Schnittstelle arbeitet Interrupt - gesteuert, d. h., jedes empfangene Byte löste einen Interrupt aus. In dieser Routine gibt es zwei wichtige Variablen - buff[comIT] und indata[comIT].

- buff[comIT] – Diese Variable dient als Speicher der empfangenen Bytes.
- indata[comIT] – In diesem Datenvektor werden die Stimulationsparameter gespeichert

```
777 : interrupt (UART1RX_VECTOR) Usart1_RX (void)
778 : {
779 :     ReceiveFlag = 1;
780 :
781 :     switch(PCToStim)    {
782 :         case ReceiveHello:
783 :             if(comIT >= 10) comIT = 0;
784 :             buff[comIT] = RXBUF1;
785 :             if( buff[comIT-6] == 'H' && buff[comIT-5] == 'i' &&
786 :                 buff[comIT-4] == 'S' && buff[comIT-3] == 't' &&
787 :                 buff[comIT-2] == 'i' && buff[comIT-1] == 'm'   && buff[comIT] =='!') {
788 :                 comIT = 0;
789 :                 PCToStim = SendHello;
790 :             }
791 :             break;
792 :
793 :         case ReceiveDataSize:
794 :             TransSize = RXBUF1;
795 :             if(TransSize == 0)  PCToStim = ErrorOccurred;
796 :             else                PCToStim = ConfirmDataSize;
797 :             comIT = 0;
```

```
798 :             break;
799 :
800 :         case ReceiveData:
801 :             if(comIT >= TransSize) comIT = 0;
802 :             indata[comIT] = RXBUF1;
803 :             if(indata[comIT-4]=='-' && indata[comIT-3]=='E' && indata[comIT-2]=='o' &&
804 :                indata[comIT-1]=='D' && indata[comIT]=='-')    {
805 :                 PCToStim = SendFinalReport;
806 :                 comIT = 0;
807 :             }
808 :             break;
809 :
810 :         default:    RXBUF1;
811 :
812 :     }
813 :
814 :     comIT++;
815 : }
```

Senden

Die Funktionen zum Senden der Daten an den Computer sind in dem Hauptprogramm (void main()) eingebettet. Die Übertragungsstruktur ist einfach zu erkenne und wird daher nicht näher beschrieben.

```
1782 :         // ----------- <L3> serial communication : sending data to PC
1783 :
1784 :         // show messege on LCD display during communication
1785 :         if(ReceiveFlag == 1 && once == 1)  {
1786 :             lcdScreen("ABORT DATA TRANSFER:","Push-Button ->|");
1787 :             once = 0;
1788 :             menu_rot = BlockRotary;
1789 :         }
1790 :
1791 :         // communication protocoll (see also interrupt routine)
1792 :         switch(PCToStim)    {
1793 :             case SendHello:
1794 :                 SendText("HiComp!");
1795 :                 PCToStim = ReceiveDataSize;
1796 :                 comIT = 0;
1797 :                 break;
1798 :
1799 :             case ConfirmDataSize:
1800 :                 TransSize *= 30;
1801 :                 indata = (char*)malloc(TransSize*sizeof(char)); // allocat memory block
1802 :                 comIT = 0;
1803 :                 if(indata == NULL)  {
1804 :                    sprintf(str,"TS:err - %c",TransSize);
1805 :                     SendText(str);
1806 :                     PCToStim = ErrorOccurred;
1807 :                     com_err = 0x01;
1808 :                 }
1809 :                 else    {
1810 :                     sprintf(str,"TS:ok - %c",TransSize);
1811 :                     SendText(str);
1812 :                     PCToStim = ReceiveData;
1813 :                 }
1814 :                 break;
1815 :
1816 :             case SendFinalReport:
1817 :                 com_err = DataConversion();             // subroutine DataConversion()
1818 :                 comIT = 0;
1819 :                 if(com_err == 0)  {
1820 :                     SendText("DC:ok");
1821 :                     PCToStim = LCDShowReport;
1822 :                 }
1823 :                 else    {
1824 :                     sprintf(str,"DC:err - %c",(char)(com_err));
1825 :                     SendText(str);
1826 :                     dat_err |= com_err;
1827 :                     ErrorMessage(0);
1828 :                     PCToStim = ErrorOccurred;
1829 :                 }
1830 :                 break;
1831 :
1832 :             case LCDShowReport:
1833 :                 lcdScreen("Transmission: OKAY","Push-Button ->|");
1834 :                 PCToStim = BlockCommunication;
1835 :                 ReceiveFlag = 1;
```

```
1836 :                  break;
1837 :          }
1838 :          // LCD - message when an error is occured
1839 :          if(PCToStim == ErrorOccurred)    {
1840 :              lcdScreen("COM: ERROR - ","Push-Button ->|");
1841 :              lcdPosL2(14);
1842 :              printf("%d",com_err);
1843 :              PCToStim = BlockCommunication;
1844 :              ReceiveFlag = 1;
1845 :          }
1846 :      }
1847 : }
```

Die folgende Funktion übernimmt das Senden der Daten an den Computer. Der Text wird byteweise gesendet, wobei das letzte Byte immer den Wert 0x0A hat (siehe Zeile 1858)

```
1850 : void SendText(char *Text)
1851 : {  int i;
1852 :    i=0;
1853 :    while (Text[i]!='\0')          // transmit string via UART
1854 :    {
1855 :       TXBUF1=Text[i++];
1856 :       while ((UTCTL1&0x01)==0);
1857 :    }
1858 :    TXBUF1=0x0A;                   // defined end character
1859 :    while ((UTCTL1&0x01)==0);
1860 : }
```

Protokoll zur Datenübertragung

Die Variable **indata[comIT]** speichert den gesamten Stimulationsparameter eines Programms. Das eindimensionale Feld von Byte-Werten folgt einem strikten Protokoll, das in der folgenden Tabelle gezeigt wird. Die Daten werden zurückgewonnen und der jeweiligen Programmvariable (Prog B1, …, Prog B5 - siehe Seite 29) zugeordnet. Der Codeabschnitt zur Datenkonvertierung wird an dieser Stelle nicht gezeigt, da im Wesentlichen die Elemente das Übertragungsprotokoll nur byteweise durchlaufen. Dieser Abschnitt ist der Funktion ReadDataStream(…) des Burst - Editors sehr ähnlich (vgl. Seite 54).

Übertragungsprotokoll - beide Kanäle mit *unabhängigem* Timing

#	**P**	**R**	**O**	**G**	◊	**I**	◊	**C**	◊	**B**	◊	**R**	◊	*	◊	◊	◊	◊	:	◊	◊	◊	◊	:	◊	◊	◊	:
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6	7	8	9	10	11	12	13
					n		i		ch		b		r		stimulation time [ms]					pause time [ms]					stim. frequency [mHz]			

◊	◊	:	◊	◊	:	◊	◊	:	◊	*	◊	◊	◊	◊	:	…	:	◊	*	**C**	◊	**B**	◊	**R**	◊	*
29	30	31	32	33	34	35	36	37	38	39	40	41	42	45	46	…	65	66	67	68	68	69	70	71	72	73
14	15	16	17	18	19	20	21	22	23	24	0	1	2	3	4	…	22	23	24	0	1	2	3	4	5	6
pos.pulse width [µs]			neg.pulse width [µs]			amplitude [‰]			p		stimulation time [ms]					…		p			ch		b		r	

◊	◊	◊	◊	:	◊	◊	◊	◊	:	◊	◊	◊	:	◊	◊	:	◊	◊	:	◊	…	**–**	**E**	**o**	**D**	**–**
74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	…	+0	+1	+2	+3	+4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	23	…	0	1	2	3	4
stimulation time [ms]					pause time [ms]					stim. frequency [mHz]				pos.pulse width [µs]			neg.pulse width [µs]				…	end of data				

Übertragungsprotokoll - beider Kanäle mit *identischem* Timing

#	P	R	O	G	◊	I	◊	C	◊	B	◊	R	◊	*	◊	◊	◊	◊	:	◊	◊	◊	◊	:	◊	◊	◊	:
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6	7	8	9	10	11	12	13
					n		i		ch		b		r		stimulation time [ms]					pause time [ms]					stim. frequency [mHz]			

◊	◊	:	◊	◊	:	◊	◊	:	◊	◊	:	◊	*	...	–	E	o	D	–
29	30	31	32	33	34	35	36	37	38	39	40	41	42	...	+0	+1	+2	+3	+4
14	15	16	17	18	19	20	21	22	23	24	25	26	27	...	0	1	2	3	4
pos.pulse width [µs]			neg.pulse width [µs]			Amp-Ch1 [‰]			Amp-Ch2 [‰]			p		...	end of data				

Tabelle 2.8: Protokoll für serielle Datenübertagung

n … Programm-Nummer, siehe Tabelle 2.4 (Seite 33)
i … Kanal - Infobyte, siehe Tabelle 2.5 (Seite 33)
ch … Kanal(1 - Kanal 1 / 2 - Kanal 2)
b … Datenmenge (beliebige Nummer)
r … Wiederholungsrate der Burstfolge (c - kontinuierliche Stimulation / beliebige Nummer)
p … Polarisation (E / O)

Beispiel: Der empfangene Datenstrom 00h, 00h, 3Ah, 98h wird umgerechnet und man erhält für die Stimulationszeit 15 Sekunden.

*	00 h		00 h		3A h		98 h		:
	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0	
14	15		16		17		18		19
	stimulation time [ms]								

$$0 \cdot 16^7 + \cdots + 0 \cdot 16^4 + 3 \cdot 16^3 + A \cdot 16^2 + 9 \cdot 16^1 + 8 \cdot 16^0 = 15000ms$$

```
1010 : long calHexToDec(int pos, char len)
1011 : {
1012 :      int i;
1013 :      long erg =  0;
1014 :      unsigned long mul = 1;
1015 :
1016 :      for(i=len-1;i>=0;i--)     {
1017 :         erg +=indata[pos+i]*mul;
1018 :         mul = mul << 8;
1019 :      }
1020 :      return erg;
1021 : }
```

2.5. Burst-Editor – Visual C++ 2005

Der Burst - Editor wurde in der Programmiersprache Visual C++ 2005 geschrieben, welche zu der Familie der Visual Studio - Entwicklungstools von Microsoft gehört. Diese IDE (Integrated Development Environment) inkludiert die Programmiersprachen Visual Basic 2005, Microsoft Visual C#® 2005, Visual C++ 2005 und Microsoft Visual J#® 2005.

Visual Studio 2005 verwendet das dotNET Framework 2.0 als Klassenbibliothek und die Laufzeitumgebung (CLR - Common Language Runtime) für Anwendungen. Das dotNET Framework arbeitet als Interface zwischen dem Betriebssystem und der Anwendung, dabei werden ausschließlich Windows-Betriebssysteme unterstützt. Visual C++ 2005 besitzt, neben der dotNET - Basis, die Möglichkeit native Programme zu kompilieren, die auch Unmanaged genannt werden. Diese Anwendungen benötigen keine Laufzeitumgebung und auf Hardwareelementen kann man über Pointer direkt zugreifen. In dem Visual Studio Paket verfügt nur Visual C++ die Möglichkeit Programm managed oder unmanaged zu behandlen, und kann damit ältern nativen Code mit Hilfe von Wrapperklassen in die dotNET Umgebung einbinden.

Das folgende Programm (Burst - Editor) basiert auf Managed Code. Für Visual C++ gibt es eine eigene Laufzeitumgebung (CLI - Common Language Infrastructure), die verschiedene Elemente beinhaltet, wie z. B. Debugging, Speicher managment, Compiling, Security und Exception Handling. (Skibo, et al., 2006). Seit Dezember 2005 liegt ein offiziell von der ECMA ratifizierter Standard für C++/CLI vor (vgl. ECMA - http://www.ecma-international.org).

Visual Studio eignet sich gut für die Erstellung von grafischen Oberflächen (GUI). Mit der Drag & Drop Technik lassen sich Element von einer Toolbox direkt auf einem Ausgabefenster platzieren und einrichten. (Miller, et al., 2006). Die Eigenschaften dieser Elemente werden können einfach initialisiert werden, da alle möglichen Parameter in einem Fenster aufgelistet sind. Diese grafische Programmierung, die von der IDE zu Verfügung gestellt wird, erspart das Schreiben von aufwendigem Code und dezimiert so den Programmieraufwand.

2.5.1. Aufbau

Das Visual C++ 2005 Projekt wird in der Abbildung 2.15 schematisch dargestellt, und listet die Klassen auf die hinter der grafischen Oberfläche stehen. Der Programmablauf wird in drei Abschnitten unterteilt und in dieser Einteilung beschrieben.

Die Klasse „Editor“ erzeugt die grafische Oberfläche und bildet das Herzstück des Burst- Editors, da es die Funktionen der beiden anderen Programmteile (Datenverwaltung und Datentransfer) einbindet.

Projektdateien:

DataToStim.sln [1)]	–	Projekt-Datei
DataToStim.ncb [1)]	–	VC++ Intellisense Datenbank
…\DataToStim.cpp [1)]	–	Quelldatei für das Hauptprogramm
…\Editor.resx [1)]	–	enthält die graphischen Elemente
…\Editor.h [1)]	–	beinhaltet die Komponenten für das Eingabefenster
…\DataHandling.h	–	Headerdatei - Verwaltung der Stimulationsprogramme
…\DataHandling.cpp	–	Quellcode ---- || ----
…\SerialCom.h	–	Headerdatei - serieller Datentransfer
…\SerialCom.cpp	–	Quellcode ---- || ----
…\stdafx.h, …\stdafx.cpp[1)]	–	werden von dem PreCompiler verwendet
…\AssemblyInfo.cpp[1)]	–	beinhaltet Anwenderattribute und Assembly-Metadaten
…\DataToStim.vcproj[1)]	–	Projekt-Datei, die Informationen über das generierende Datei, Konfiguration und Plattform

[1)] Diese Dateien werden von dem Form-Assistenten automatisch generiert.

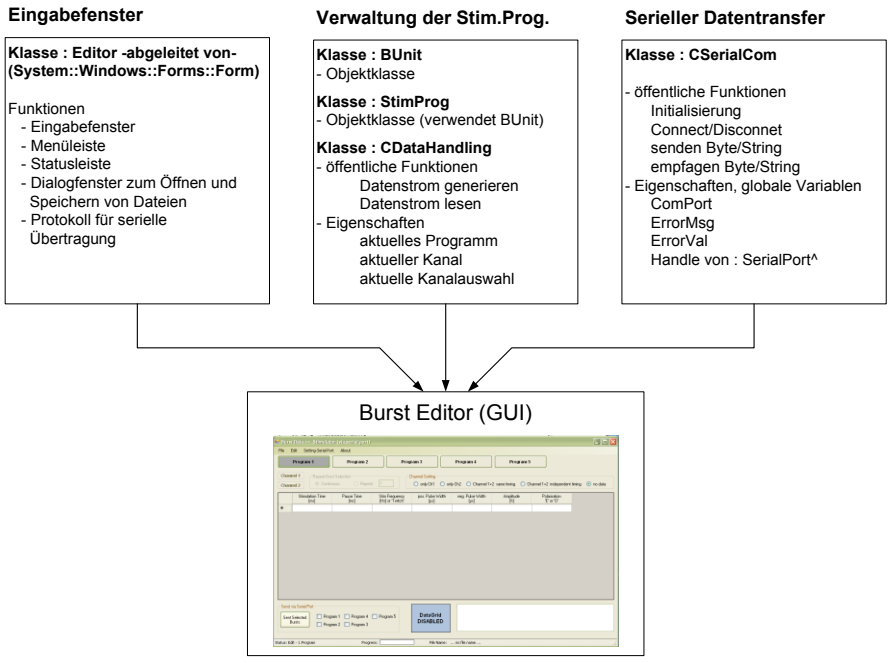


Abbildung 2.15: Klassenstruktur des Burst-Editors

2.5.2. Verwaltung der Stimulationsprogramme

Dateien: DataHandling.h, DataHandling.cpp

In der Headerdatei werden zwei verschiedene Arten von Klassen verwendet, deren Unterschied kurz beschrieben wird. Bei der „ref class“ und „value class“ handelt es sich um C++/CLI- Elemente. Verweisklassen (ref class) sind Datenstrukturen, die Variablen, Methoden und eingebettete Parameter enthalten. Der Name Verweis kommt daher, dass es sich um einen Datentyp handelt, der durch einen Verweis auf den tatsächlichen Wert des Typs dargestellt wird. Soll ein Verweistyp einer Variablen zugewiesen werden, so wird keine Kopie erzeugt sondern die Variable verweist auf den ursprünglichen Wert.(Schumann, et al., 2006).

Werttypen (value class) können Variablen, Methoden und eingebettete Typen enthalten. Der Unterschied zu anderen Klassentypen ist, dass sie weder benutzerdefinierte Destruktoren, Finalizer, Standard-Konstruktoren, Kopier-Konstruktoren noch Zuweisungsoperatoren besitzen können. Werttypen wurden entworfen, um der virtuellen Maschine ein effizientes Kopieren von Daten zu ermöglichen. Wenn einer Variablen ein Werttyp zugewiesen werden soll, wird dieser Variablen eine neue Kopie des Werts übergeben, dies steh im Gegensatz zu einem Verweistyp.(Schumann, et al., 2006).

Bei der Verwaltung der Stimulationsprogramme werden Werttypen angelegt, um die Parameter übersichtlicher zu bearbeiten. Die Aufgabe dieses Abschnittes ist, dass die Programmdaten der einzelne Burstfolgen für den Datentransfer zu dem Stimulator bereitgestellt werden und dass die Parameter-Sets in Dateien (*.stm, *.1st) abgespeichert werden können.

Klasse: value class BUnit

Die Wertklasse „value class BUnit“ beinhaltet die Variablen, welche die Parameter eines Burst beschreiben. In der Abbildung 2.5 (Seite 19) werden die Parameter genauer beschrieben.

```
 7 : value class BUnit
 8 : {  public:
 9 :      Double dStimTime;     // stimulation time [ms]
10 :      Double dPauseTime;    // pause time [ms]
11 :      Double dStimFreq;     // stimulation frequency [Hz]
12 :      Double dPwidth;       // positive pulse width [µs]
13 :      Double dNwidth;       // negative pulse width [µs]
14 :      Double dAmpl;         // amplitude - equal for neg. and pos. pulse  [%]
15 :      Double dAmplCh2;      // amplitude - Channel 2
16 :      Char cPol;            // polarisation - 'E'(eve) or 'O'(odd)
17 : };
```

Klasse: value class StimProg

Die Klasse „StimProg“ enthält alle Parameter, die ein Stimulationsprogramm beschreiben. Die Variablen ProgInfo, RepeatCh1 und RepeatCh2 beschreiben die

Kanalauswahl bzw. die Anzahl der Burstfolgen - Wiederholungen. Die gesamten Stimulationsparameter sind in einer Liste (List<BUnit>) gespeichert, welche die Methoden für eine dynamische Speicherzuweisung bereits inkludieren. Die Datenverwaltung hat die gleiche Struktur, wie sie auch in der Mikrokontrollersoftware verwendet wird, nur besitzt VC++ einfacher Funktionen zur flexiblen Datenspeicherung.

```
19 : value class StimProg
20 : {  public:
21 :     Byte ProgInfo;
22 :     // 0 - no data
23 :     // 1 - onlyCh1
24 :     // 2 - onlyCh2
25 :     // 3 - both channels with same timeing
26 :     // 4 - both channels with independent timeing
27 :
28 :     Int16 RepeatCh1;
29 :     Int16 RepeatCh2;
30 :     // continous = 0;
31 :     // any other integer number (except 0)
32 :
33 :     List<BUnit>^ BDataCh1;
34 :     List<BUnit>^ BDataCh2;
35 : };
```

Klasse: ref class CDataHandling

Diese Klasse bindet die beiden Werttypen (StimProg, BUnit) ein und enthält alle Funktionen für die Datenverwaltung.

Die öffentlichen Variable StimData ist ein Feld mit fünf Elementen, die jeweils ein Stimulationsprogramm beinhalten, d. h., alle Daten werden in dieser Variable gespeichert. Die zweite öffentliche Variable ist die Liste DataStream, welche ebenso die Stimulationsprogramme enthält, jedoch in konvertierter Form, nach dem seriellen Transferprotokoll.

Die Umrechnung der beiden Variablen erfolgt mit den Methoden „GenDataStream“ und „ReadDataStream“ bzw. „ReadDataStream_OneProg“. Die erste Funktion erzeugt den Datenvektor (DataStream), wobei alle Stimulationsprogramme oder nur Einzelne konvertiert werden können. Diese Methode wird für das Speichern der Burstfolgen verwendet (*.stm, *.1st), aber auch für den seriellen Datentransfer, bei dem aber immer nur einzelne Stimulationsparamter gesendet werden. Die Rückkonvertierung ist nur für das Einlesen von *.stm/*.1st - Dateien möglich, wofür es die zweitgenannten Methoden gibt.

Die aktuellen Werten des Stimulationsprogramms (ActNoProg), Kanalauswahl (ActNoCh) und Kanalverwendung (ActProgInfo) werden in den Klassen-Eigenschaften (properties) gesetzt. Ein Hinweis bezüglich der Nomenklatur von Variablennamen: Die Silbe „No“ bedeutet, dass es sich um eine Anzahl handlet (No - Number of).

```
37 : ref class CDataHandling
38 : {
39 : public:
40 :     CDataHandling();
41 :     ~CDataHandling();
42 :
43 :     property int ActNoProg     {
44 :        int get()               { return _ActNoProg;   }
45 :        void set(int iBurst )   { _ActNoProg = iBurst; }
46 :     }
47 :     property int ActNoCh  {
48 :        int get()          { return _ActNoCh; }
49 :        void set(int iCh ) { _ActNoCh = iCh;  }
50 :     }
51 :     property int ActProgInfo   {
52 :        int get()          { return _ActProgInfo; }
53 :        void set(int PIn ) { _ActProgInfo = PIn;  }
54 :     }
55 :          // generate data stream (write *.stm/*.1st - files)
56 :     void GenDataStream(int Mode);        // Mode: 0..all programs 1-5..number of program
57 :
58 :          // read data stream (read *.stm/*.1st - files)
59 :     void ReadDataStream(void);               // all programs
60 :     void ReadDataStream_OneProg(int Mode);   // Mode: 1-5..number of program
61 :
62 : private:
63 :     void Reset_AllBursts(void);
64 :     void Reset_OneBurst(int NoB);
65 :          // subroutines for stream generation
66 :     void Ch1DataStream(int NoB);             // generate ch1 stream
67 :     void Ch2DataStream(int NoB);             // generate ch2 stream
68 :     void calDecToHex(double dErg, int len);  // data conversion
69 :          // subroutines for stream reading
70 :     int FindString(String^ tmp, int start);  // find starting string
71 :     double calHexToDec(int pos, int len);    // data conversion
72 :
73 : private:
74 :     int _ActNoProg;
75 :     int _ActNoCh;
76 :     int _ActProgInfo;
77 :
78 : public:
79 :     array<StimProg>^ StimData;          // data of all actual programs
80 :     List<Byte>^ DataStream;             // byte-list of the data stream
81 : };
```

Source Code

Konstruktor, Dekonstruktor und Initialisierung

In dem Konstruktor werden die alle Variablen initialisiert und in den Fällen „StimData“ und „DataStream“ zuvor noch generiert.

```
 1 : #include "stdafx.h"
 2 : #include "DataHandling.h"
 3 :
 4 : CDataHandling::CDataHandling() {
 5 :
 6 :    // generate variables
 7 :    StimData = gcnew array<StimProg>(5);
 8 :    for(int i=0; i<5; i++) {
 9 :       StimData[i].BDataCh1 = gcnew List<BUnit>();
10 :       StimData[i].BDataCh2 = gcnew List<BUnit>();
11 :    }
12 :    DataStream = gcnew List<Byte>();
13 :
14 :    // initialize variables
15 :    for(int i=0; i<5; i++) {
16 :       StimData[i].ProgInfo = 0;
17 :       StimData[i].RepeatCh1 = 0;
18 :       StimData[i].RepeatCh2 = 0;
19 :
20 :       StimData[i].BDataCh1->Clear();
21 :       StimData[i].BDataCh2->Clear();
22 :    }
```

```
23 : }
24 :
25 : CDataHandling::~CDataHandling()     {
26 : }
27 :
28 : void CDataHandling::Reset_AllBursts()    {
29 :    for(int i=0; i<5; i++) {
30 :       StimData[i].ProgInfo = 0;
31 :       StimData[i].RepeatCh1 = 0;
32 :       StimData[i].RepeatCh2 = 0;
33 :       StimData[i].BDataCh1->Clear();
34 :       StimData[i].BDataCh2->Clear();
35 :    }
36 : }
37 :
38 : void CDataHandling::Reset_OneBurst(int NoB)   {
39 :       StimData[NoB].ProgInfo = 0;
40 :       StimData[NoB].RepeatCh1 = 0;
41 :       StimData[NoB].RepeatCh2 = 0;
42 :       StimData[NoB].BDataCh1->Clear();
43 :       StimData[NoB].BDataCh2->Clear();
44 : }
```

Generierung des Datenvektors

Die öffentliche Methode **GenDataStream(…)** konvertiert die Stimulationsparameter, welche in dem Array - StimData gespeichert sind, in einen Datenvektor. Die Zeichenkette wird nach dem Protokoll von Tabelle 2.8 (Seite 49) gebildet und nimmt dadurch zwei Anwendungsmöglichkeiten an. Die erste Verwendung ist das speichern der Daten in eine Datei, wobei man alle Programme (*.stm) sichern kann, oder nur Einzelne (*.1st). Da das zugrunde liegende Protokoll für die serielle Datenübertragung konzipiert ist, stellt diese Methode auch den Datenvektor für den Datentransfer zu dem Stimulator bereit.

Mit der Variable - **int Mode** - wird die Betriebsart der Methode bestimmt. Wenn Mode einen Wert von 0 bis 4 aufweißt, werden nur einzelne Programme konvertier und zwar entsprechend ihrer Nummer, z. B. *Mode = 0 …. Konvertierung des 1.Programms*. Hat die Variable den Wert 99, so werden alle fünf Stimulationsprogramme in den Datenvektor umgewandelt.

In den Zeilen 67 bis 71 werden die Parameter der Burstfolgen bearbeitet, wobei der jeweilige Kanal berücksichtigt werden muss. Die beiden Funktionen Ch1DataStream(…) und Ch2DataStream(…) sind von der Struktur gleich und bilden den Datenkörper des Übertragungsprotokolls.

```
46 : void CDataHandling::GenDataStream(int Mode)
47 : {
48 :    int i, start, end;
49 :    BUnit B;
50 :
51 :    DataStream->Clear();
52 :
53 :    if(Mode == 99)  {
54 :       DataStream->Add(0);     // Mode - Flag
55 :       start=0; end=4;
56 :    } else   {
57 :       start=Mode; end=Mode;
58 :    }
59 :
60 :    for(i=start; i<=end; i++) {
61 :       DataStream->Add('#');DataStream->Add('P');DataStream->Add('R');
62 :       DataStream->Add('O');DataStream->Add('G');
63 :       DataStream->Add(i+1);
64 :       DataStream->Add('I');DataStream->Add(StimData[i].ProgInfo);
65 :
66 :       switch(StimData[i].ProgInfo)   {
```

```
 67 :          case 1: Ch1DataStream(i); break;
 68 :          case 2: Ch2DataStream(i); break;
 69 :          case 3: Ch1DataStream(i); break;
 70 :          case 4: Ch1DataStream(i);
 71 :                  Ch2DataStream(i); break;
 72 :       }
 73 :       DataStream->Add('-');
 74 :       DataStream->Add('E');DataStream->Add('o');DataStream->Add('D');
 75 :       DataStream->Add('-');
 76 :    }
 77 : }
 78 :
 79 : void CDataHandling::Ch1DataStream(int NoB)    {
 80 :    BUnit B;
 81 :    int i;
 82 :
 83 :    DataStream->Add('C');  DataStream->Add(1);
 84 :    DataStream->Add('B');
 85 :    i =  StimData[NoB].BDataCh1->Count::get();
 86 :    calDecToHex(StimData[NoB].BDataCh1->Count::get(),1);
 87 :    DataStream->Add('R');
 88 :    calDecToHex(StimData[NoB].RepeatCh1,1);
 89 :    DataStream->Add('*');
 90 :
 91 :    for each( B in StimData[NoB].BDataCh1)  {
 92 :       calDecToHex(B.dStimTime,4);          DataStream->Add(':');
 93 :       calDecToHex(B.dPauseTime,4);         DataStream->Add(':');
 94 :       calDecToHex(B.dStimFreq*1000,3);     DataStream->Add(':');
 95 :       calDecToHex(B.dPwidth,2);            DataStream->Add(':');
 96 :       calDecToHex(B.dNwidth,2);            DataStream->Add(':');
 97 :       calDecToHex(B.dAmpl*10,2);           DataStream->Add(':');
 98 :       if(StimData[NoB].ProgInfo == 3)     {
 99 :          calDecToHex(B.dAmplCh2*10,2); DataStream->Add(':');
100 :
101 :       }
102 :       DataStream->Add((Byte)B.cPol);    DataStream->Add('*');
103 :
104 :    }
105 : }
106 :
107 : void CDataHandling::Ch2DataStream(int NoB)    {
... :
... :         // Diese Routine besitzt die gleiche Struktur wie Ch1DataStream(...)
... :
129 : }
```

Umrechnungen von der Dezimalzahl zu dem hexadezimalen Zahlensystem

Diese Funktion wandelt eine Dezimalzahl in eine Hexadezimalzahl um, wobei die Länge des Ergebnisses fix vorgegeben wird.

Beispiel: calDecToHex(1000,4) … Die Zahl 1000 soll konvertiert werden, wofür 4 Bytes zu Verfügung stehen. Das Ergebnis lautet: 00_h,00_h,03_h,$E8_h$ und wird an das Listenende der Variable DataStream angehängt. (das tiefer gestellte h deutet auf eine Hexadezimalzahl hin).

```
131 : ////   decimal number --> hexadecimal number with fixed length
132 : void CDataHandling::calDecToHex(double dErg, int len)
133 : {
134 :     Int64 ulTmp;
135 :
136 :     Byte HexNumber[4];
137 :     Byte a, b;
138 :     int i;
139 :
140 :     ulTmp = (unsigned long)(dErg);
141 :
142 :     for(i=0; i<len; i++)  {
143 :        a = (Byte)(ulTmp % 16);
144 :        ulTmp = (Int64)(ulTmp/16);
145 :        b = (Byte)(ulTmp % 16);
146 :        ulTmp = (Int64)(ulTmp/16);
147 :        HexNumber[i] = (b << 4) + a;
148 :     }
149 :
150 :     for(i=len-1; i>=0; i--)
151 :        DataStream->Add(HexNumber[i]);
152 : }
```

Auslesen des Datenvektors

Die Methoden ReadDataStream und ReadDataStream_OneProg sind für das Einlesen der Dateien notwendig, dabei wird unterschieden, ob nur ein Programm (*.1st) oder alle Programme (*.stm) geöffnet werden. Der Unterschied zwischen ReadDataStream und ReadDataStream_OneProg ist minimal und wird an dieser Stelle nicht explizit beschrieben.

Bei der Rückkonvertierung wird zuerst der Headertext „#PROG“ in dem Datenvektor gesucht und im Anschluss nach dem Protokoll die Stimulationsparameter berechnet. Jedes Stimulationsprogramm liegt zwischen den beiden Symbolketten „#PROG“ und „-EoD-“. Die private Funktion FindString(…) sucht ab einem gewissen Startwert ein Textelement in dem Datenvektor und liefert bei erfolgreicher Suche die Position zurück.

Beispiel: FindString(''#Prog'', 0) …. Der Datenvektor wird von Beginn an durchsucht, und liefert in diesem Beispiel den Index-Wert 79 zurück.

◊	◊	◊	◊	◊	#	P	R	O	G	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊	◊
74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102
					⇑																							

Die zweite private Funktion, die von ReadDataStream(…) verwendet wird, ist DataBodyLoop(…). Diese Methode folgte dem fixen Datenprotokoll und berechnet die Dezimalwerte.

```
154 : void CDataHandling::ReadDataStream(void) {
155 :     int index;
156 :
157 :     int NoB;
158 :     int NoCh;
159 :     int ChSize;
160 :     int ProgFlag;
161 :
162 :     if( DataStream[0]!=0 )  {
163 :        MessageBox::Show("- file signature is incorrect -\n first byte should be '0'" ▶
                          ,"Error", MessageBoxButtons::OK, MessageBoxIcon::Error);
164 :        return;
165 :     }
166 :     Reset_AllBursts();
167 :
168 :     index = 0;
169 :
170 :     for( int i=0; i<=4; i++)   {
171 :
172 :        index = FindString("#PROG",index);
173 :        if(index < 0) return;
174 :        NoB = DataStream[index+5]-1;
175 :        if(NoB < 0 || NoB > 5) return;
176 :        ProgFlag = DataStream[index+7];
177 :        StimData[NoB].ProgInfo = ProgFlag;
178 :        if(StimData[NoB].ProgInfo < 0 || StimData[NoB].ProgInfo > 4 || ▶
              DataStream[index+6] != 'I')   return;
179 :
180 :        index += 8;
181 :
182 :        if(ProgFlag != 0)   {
183 :           if(DataStream[index] != 'C') return;
184 :           NoCh = DataStream[index+1];
185 :           ChSize = DataStream[index+3];
186 :           if(NoCh == 1) StimData[NoB].RepeatCh1 = DataStream[index+5];
187 :           else          StimData[NoB].RepeatCh2 = DataStream[index+5];
188 :
189 :           if(DataStream[index+6] != '*') return;
190 :
191 :           index += 7;
192 :
193 :           for(int j=0; j < ChSize; j++)  {
```

```
194 :                if(NoCh == 1) StimData[NoB].BDataCh1->Add(DataBodyLoop(index,ProgFlag));
195 :                else          StimData[NoB].BDataCh2->Add(DataBodyLoop(index,0));
196 :
197 :                if(ProgFlag == 3)    index += 28;
198 :                else              index += 25;
199 :
200 :                if(DataStream[index-1] != '*') return;
201 :             }
202 :          }
203 :
204 :          if(ProgFlag == 4)    {
205 :             if(DataStream[index] != 'C') return;
206 :             NoCh = DataStream[index+1];
207 :             ChSize = DataStream[index+3];
208 :             if(NoCh == 1) StimData[NoB].RepeatCh1 = DataStream[index+5];
209 :             else          StimData[NoB].RepeatCh2 = DataStream[index+5];
210 :
211 :             if(DataStream[index+6] != '*') return;
212 :
213 :             index += 7;
214 :
215 :             for(int j=0; j < ChSize; j++)  {
216 :                 StimData[NoB].BDataCh2->Add(DataBodyLoop(index,0));
217 :
218 :                 if(ProgFlag == 3)    index += 28;
219 :                 else              index += 25;
220 :
221 :                 if(DataStream[index-1] != '*') return;
222 :             }
223 :          }
224 :          //index += 5;
225 :       }
226 :
227 : }
228 :
229 : int CDataHandling::FindString(String^ tmp, int start)   {
230 :    int index, match=0, len;
231 :
232 :    index = start;
233 :    len = tmp->Length::get();
234 :
235 :    while(match < len)   {
236 :       match = 0;
237 :       for(int i=0; i<len; i++)
238 :          if( DataStream[(int)index+i] == tmp[i])       match++;
239 :       index++;
240 :       if(index == DataStream->Capacity::get()) return -1;
241 :    }
242 :    index += -1;
243 :    return index;
244 : }
246 :
247 : BUnit CDataHandling::DataBodyLoop(int index, int ProgInfo)   {
248 :
249 :    double res;
250 :    Int64 mul;
251 :    BUnit B;
252 :    int jump[] = {4, 4, 3, 2, 2, 2, 2, 1};
253 :
254 :    B.dAmplCh2 = 0;
255 :
256 :    for(int i=0; i<8; i++)  {
257 :
258 :       res = 0;
259 :       mul = 1;
260 :       if(i != 6 || ProgInfo == 3){
261 :
262 :          for(int j=jump[i]-1; j>= 0; j--)  {
263 :             res += DataStream[index+j] * mul;
264 :             mul = mul << 8;
265 :          }
266 :          index += jump[i]+1;
267 :       }
268 :       switch(i) {
269 :          case 0: B.dStimTime = res; break;
270 :          case 1: B.dPauseTime = res; break;
271 :          case 2: B.dStimFreq = res/1000; break;
272 :          case 3: B.dPwidth = res; break;
273 :          case 4: B.dNwidth = res; break;
274 :          case 5: B.dAmpl = res/10; break;
246 :          case 6: B.dAmplCh2 = res; break;
247 :          case 7: B.cPol = (char)(res); break;
248 :       }
249 :    }
250 :
251 :    return B;
252 : }
253 :
```

```
254 : void CDataHandling::ReadDataStream_OneProg(int NoB)     {
... :
... : //  Diese Routine besitzt eine ähnliche Struktur zum Codeteil - ReadDataStream
... :
345 : }
```

Umrechnungen von einer hexadezimalen Zahl in das Dezimalsystem

Die Berechnung der Dezimalwerte aus dem Datenvektor gestaltet sich sehr einfach. Der Index und die Länge der Hexadezimalzahl bestimmen die Konvertierung.

Beispiel: calHexToDec(88 , 4) entspricht in diesem Fall die Zahl 1000_d.

Wert	:	00_h	00_h	03_h	$E8_h$	:
Positionsnummer	87	88	89	90	91	92

```
347 : //// hexadecimal number in a data stream --> decimal number
348 : double CDataHandling::calHexToDec(int pos, int len)
349 : {
350 :       int i;
351 :       double erg = 0;
352 :       unsigned long mul;
353 :
354 :       mul = 0x01;
355 :       for(i=len-1;i>=0;i--)     {
356 :             erg += (double)(DataStream[pos+i]*mul);
357 :             mul = mul << 8;
358 :       }
359 :
360 :       return erg;
361 : }
```

2.5.3. Datentransfer

Dateien: SerialCom.h, SerialCom.cpp

Klasse: ref class CSerialCom

Diese Klasse enthält die Grundfunktionen für die serielle Datenübertragung, diese sind Funktionen für die Initialisierung und das Lesen bzw. Schreiben auf das COM - Port. Die Ref-Klasse besitzt drei Eigenschaften, wobei zwei nur Informationen über eine etwaige Fehlermeldung besitzt und die dritte Eigenschaft das COM - Port bestimmt.

```
 1 : #include "stdafx.h"
 2 :
 3 : using namespace System;
 4 : using namespace System::IO::Ports;
 5 : using namespace System::Windows::Forms;
 6 : using namespace System::Collections::Generic;
 7 :
 8 : ref class CSerialCom
 9 : {
10 :   public:
11 :       CSerialCom();
12 :       ~CSerialCom();
13 :
14 :       void InitCom();
15 :
16 :       void Connect();
17 :       void Disconnect();
```

```
18 :
19 :       void    sendByte(Byte b);
20 :       void    sendString(String^ str);
21 :       Byte    getByte();
22 :       String^ getString();
23 :
24 :       property String^ ComName {
25 :          String^ get()     { return strComName; }
26 :          void set(String^ str ){ strComName = str; }
27 :       }
28 :       property String^ ErrorMsg{
29 :          String^ get()     { return errMSG; }
30 :       }
31 :       property int ErrorValue  {
32 :          int get()    { return errVAL; }
33 :       }
34 :
35 :
36 :    private:
37 :       SerialPort^ ComPort;
38 :       String^ strComName;
39 :       String^ errMSG;
40 :       int     errVAL;
41 : };
```

Source Code

Konstruktor, Dekonstruktor und Initialisierung

Die serielle Schnittstelle enthält folgende Konfiguration: 8 Datenbits (CHAR), 1 Stopbit, DTR und CTS deaktiviert, keine Parität und eine Baudrate von 115200 Baud/s. Diese Einstellung kann nicht mehr geändert werden, da sie auf den Stimulator abgestimmt ist.

```
 4 : CSerialCom::CSerialCom() {
 5 :       errMSG = "";
 6 :       strComName = "COM1";
 7 :       ComPort = gcnew SerialPort();
 8 : }
 9 :
10 : CSerialCom::~CSerialCom(){
11 : }
12 :
13 : void CSerialCom::InitCom()     {
14 :       ComPort->PortName::set(strComName);
15 :       ComPort->DataBits::set(8);
16 :       ComPort->BaudRate::set(115200);
17 :       ComPort->Parity::set(System::IO::Ports::Parity::None);
18 :       ComPort->StopBits::set(System::IO::Ports::StopBits::One);
19 :       ComPort->DtrEnable::set(false);
20 :       ComPort->RtsEnable::set(false);
21 :       ComPort->ReadTimeout::set(500);
22 :       ComPort->WriteTimeout::set(500);
23 : }
24 :
25 : void CSerialCom::Connect()     {
26 :       errMSG = "";
27 :       try  {    ComPort->Open();
28 :       } catch(Exception^ ex)    {
29 :            errMSG = ex->ToString();
30 :       }
31 : }
32 :
33 : void CSerialCom::Disconnect()  {
34 :       errMSG = "";
35 :       try  {    ComPort->Close();
36 :       } catch(Exception^ ex)    {
37 :            errMSG = ex->ToString();
38 :       }
39 : }
```

Senden und Empfangen von Daten

Die folgenden Grundfunktionen sind selbsterklärend, da es sich nur um das Senden bzw. Empfangen von einzelnen Bytes oder Textstrings handlet. Das Protokoll für die serielle Datenübertragung ist direkt im Form-Blatt (Editor.h) eingebettet und wird an dieser Stelle erklärt.

```
41 : void CSerialCom::sendByte(Byte b)   {
42 :       errMSG = "";
43 :       errVAL = 0;
44 :       array<unsigned char,1>^ buffer = gcnew array<unsigned char,1>(1);
45 :       buffer[0] = (unsigned char)(b);
46 :
47 :       try  {     ComPort->Write(buffer,0,1);
48 :       } catch(Exception^ ex)    {
49 :            errMSG = ex->ToString();
50 :            errVAL = 1;
51 :       }
52 : }
53 :
54 : void CSerialCom::sendString(String^ str) {
55 :       errMSG = "";
56 :       errVAL = 0;
57 :       try  { ComPort->WriteLine(str);
58 :       } catch(Exception^ ex)    {
59 :            errMSG = ex->ToString();
60 :            errVAL = 1;
61 :       }
62 : }
63 :
64 : Byte CSerialCom::getByte()     {
65 :       Byte b=0;
66 :       errMSG = "";
67 :       errVAL = 0;
68 :       try  {     b = ComPort->ReadByte();
69 :       } catch(Exception^ ex)    {
70 :            errMSG = ex->ToString();
71 :            errVAL = 2;
72 :            b = 0;
73 :       }
74 :
75 :       return b;
76 : }
77 :
78 : String^ CSerialCom::getString()     {
79 :       String^ str="";
80 :       errMSG = "";
81 :       errVAL = 0;
82 :       try  {     str = ComPort->ReadLine();
83 :       } catch(Exception^ ex)    {
84 :            errMSG = ex->ToString();
85 :            errVAL = 2;
86 :            str = "";
87 :       }
88 :
89 :       return str;
90 : }
```

2.5.4. Eingabefenster

Datei: Editor.h

In dieser Klasse wird die grafische Oberfläche des Burst - Editors erzeugt, wobei alle Element initialisiert und alle Events erzeugt werden. Neben dieser Headerdatei ist auch noch die Datei Editor.resx wichtig, da sie die gesamten grafischen Elemente enthält. Diese Datei wird von dem Visual C++ Editor automatisch generiert.

Die folgenden Codeteile sind auf die wesentlichen Abschnitte beschränkt, da vor allem die Funktion beschrieben werden soll und nicht der von Visual C++ selbst generierten Code.

Konstruktor/Initialisierung

In dem Konstruktor stehen alle Initialisierungen der grafischen Elemente und der Variablen. Der folgende Codeteil soll einen Überblick geben, da die Initialisierung beinahe 1000 Zeilen benötigt, wird sie an dieser Stelle nicht aufgelistet.

```
   1 : #pragma once
   2 : #include "SerialCom.h"
   3 : #include "DataHandling.h"
   4 :
   5 : namespace DataToStim {
   6 :
   7 :      # define ALL_PROG   99
   8 :
   9 :      using namespace System;
  10 :      using namespace System::IO;
  11 :      using namespace System::Data;
  12 :      using namespace System::Drawing;
  13 :      using namespace System::Collections;
  14 :      using namespace System::ComponentModel;
  15 :      using namespace System::Windows::Forms;
  16 :
  17 :      public ref class Editor : public System::Windows::Forms::Form
  18 :      {
  19 :      public:
  20 :           Editor(void)
  21 :           {
  22 :                InitializeComponent();
  23 :                InitializeIndividual();
  24 :           }
  25 :
  26 :      protected:
  27 :           ~Editor()
  28 :           {
  29 :                if (components)
  30 :                {
  31 :                     delete components;
  32 :                }
  33 :           }
 ... :
 ... :               //  Initialisierung des Form-Fensters
 ... :
 134 : #pragma region Windows Form Designer generated code
 135 :           void InitializeComponent(void)
 139 :           {
 ... :
 ... :               //  Initialisierung der einzelnen Komponenteneigenschaften
 ... :
1039 :           }
1040 :
1041 :           void InitializeIndividual(void)
1042 :           {
1043 :                OpenFlag = 0;
1044 :                SaveFlag = 0;
1045 :                ActFileName = "";
1046 :                BoxCount =  0;
1047 :                bEnableDataGrid = false;
1048 :
1049 :                BData.ActNoProg::set(0);
1050 :                BData.ActProgInfo::set(0);
1051 :                BData.ActNoCh::set(1);
1052 :
1053 :                Refresh_BurstSelection();
1054 :                SetRDB_ProgInfo();
1055 :                Refresh_ProgInfo();
1056 :                Refresh_Channel();
1057 :                Refresh_RepBSel();
1058 :                Refresh_DataGrid();
1059 :
1060 :           }
```

Menüleiste und Statuszeile

Die Funktionen der Menüleiste und der Statuszeile werden auf der Seite 18 beschrieben. Die einzelnen Menüelemente können mit Drag&Drop Technik erstellt werden, welche die Programmierarbeit erleichtert. Auch die beiden Dialogfenster zum Öffnen und Speichern von Dateien stellt Visual C++ zu Verfügung. Der Programmcode wiederholt sich für die einzelnen Elemente und wird nicht angezeigt, da es sich um triviale Probleme handelt.

Einlesen der Parameter aus dem DataGrid

Das DataGrid enthält alle Stimulationsparameter eines Programms. Falls ein Tabellenelement verändert wird die Methode ReadIn_DataGrid(…) aufgerufen und die Array - Variable StimData wird aktualisiert. Jedes Element wird auf seine Gültigkeit überprüft, wobei der ASCII - Code in eine Zahl umgerechnet werden muss. Falls es bei diesem Vorgang ein Fehler auftritt, wird dies in einer Info - Fenster angezeigt.

```
1475 : private: int ReadIn_DataGrid(void){
1476 :         BUnit B;
1477 :         String^ tmpFreq;
1478 :         String^ tmpPol;
1479 :
1480 :         int i=0;
1481 :         Windows::Forms::RowStyle row;
1482 :
1483 :         int NoB = BData.ActNoProg::get();
1484 :         int NoCh = BData.ActNoCh::get();
1485 :
1486 :         switch(NoCh)  {
1487 :            case 1: BData.StimData[NoB].BDataCh1->Clear(); break;
1488 :            case 2: BData.StimData[NoB].BDataCh2->Clear(); break;
1489 :         }
1490 :
1491 :         for(int i = 0; i < DataGrid->RowCount::get()-1; i++)   {
1492 :           try{ B.dStimTime = Convert::ToDouble(DataGrid[0,i]->Value::get());
1493 :           }catch( Exception^ ex){
1494 :            MessageBox::Show("incorrect input - at [Stimulation Time] in Row " +   ▶
                                  Convert::ToString(i+1) + "\n\n" + ex->ToString(),"Error   ▶
                                  Message",MessageBoxButtons::OK,MessageBoxIcon::Exclamation);
1495 :             return 1;
1496 :           }
1497 :
1498 :           try{ B.dPauseTime = Convert::ToDouble(DataGrid[1,i]->Value::get());
1499 :           }catch( Exception^ ex){
1500 :            MessageBox::Show("incorrect input - at [Pause Time] in Row " +   ▶
                                  Convert::ToString(i+1) + "\n\n" + ex->ToString(),"Error   ▶
                                  Message",MessageBoxButtons::OK,MessageBoxIcon::Exclamation);
1501 :             return 1;
1502 :           }
1503 :
1504 :           try{
1505 :             tmpFreq = Convert::ToString(DataGrid[2,i]->Value::get());
1506 :             if(tmpFreq->Contains("T") || tmpFreq->Contains("t"))    {
1507 :               B.dStimFreq = 0;
1508 :             }
1509 :             else
1510 :              B.dStimFreq = Convert::ToDouble(DataGrid[2,i]->Value::get());
1511 :
1512 :           }catch( Exception^ ex){
1513 :            MessageBox::Show("incorrect input - at [Stim Frequency] in Row " +   ▶
                                  Convert::ToString(i+1) + "\n\n" + ex->ToString(),"Error   ▶
                                  Message",MessageBoxButtons::OK,MessageBoxIcon::Exclamation);
1514 :             return 1;
1515 :           }
1516 :
1517 :           try{ B.dPwidth = Convert::ToDouble(DataGrid[3,i]->Value::get());
1518 :           }catch( Exception^ ex){
1519 :            MessageBox::Show("incorrect input - at [pos. Pulse Width] in Row " +   ▶
                                  Convert::ToString(i+1) + "\n\n" + ex->ToString(),"Error   ▶
                                  Message",MessageBoxButtons::OK,MessageBoxIcon::Exclamation);
1520 :             return 1;
1521 :           }
1522 :
```

```
1523 :          try{ B.dNwidth = Convert::ToDouble(DataGrid[4,i]->Value::get());
1524 :          }catch( Exception^ ex){
1525 :           MessageBox::Show("incorrect input - at [neg. Pulse Width] in Row " +   ▶
                                 Convert::ToString(i+1) + "\n\n" + ex->ToString(),"Error   ▶
                                 Message",MessageBoxButtons::OK,MessageBoxIcon::Exclamation);
1526 :            return 1;
1527 :          }
1528 :
1529 :          try{ B.dAmpl = Convert::ToDouble(DataGrid[5,i]->Value::get());
1530 :          }catch( Exception^ ex){
1531 :           MessageBox::Show("incorrect input - at [Amplitude] in Row " +   ▶
                                 Convert::ToString(i+1) + "\n\n" + ex->ToString(),"Error   ▶
                                 Message",MessageBoxButtons::OK,MessageBoxIcon::Exclamation);
1532 :            return 1;
1533 :          }
1534 :
1535 :          try{
1536 :            tmpPol = Convert::ToString(DataGrid[7,i]->Value::get());
1537 :            if(tmpPol->Contains("O") || tmpPol->Contains("o")){
1538 :              B.cPol = 'O';
1539 :            }else   {
1540 :              B.cPol = 'E';
1541 :            }
1542 :          }catch( Exception^ ex){
1543 :           MessageBox::Show("incorrect input - at [Polarisation] in Row " +   ▶
                                 Convert::ToString(i+1) + "\n\n" + ex->ToString(),"Error   ▶
                                 Message",MessageBoxButtons::OK,MessageBoxIcon::Exclamation);
1544 :            return 1;
1545 :          }
1546 :
1547 :          if(BData.ActProgInfo::get() == 3)     {
1548 :            try  {B.dAmplCh2 = Convert::ToDouble(DataGrid[6,i]->Value::get());
1549 :            }catch( Exception^ ex)    {
1550 :             MessageBox::Show("incorrect input - at [Amplitude Ch1] in Row " +   ▶
                                   Convert::ToString(i+1) + "\n\n" + ex->ToString(),"Error   ▶
                                   Message",MessageBoxButtons::OK,MessageBoxIcon::Exclamation);
1551 :              return 1;
1552 :            }
1553 :          }
1554 :
1555 :          switch(NoCh)  {
1556 :           case 1: BData.StimData[NoB].BDataCh1->Add(B); break;
1557 :           case 2: BData.StimData[NoB].BDataCh2->Add(B); break;
1558 :          }
1559 :
1560 :          // enhance datagrid output
1561 :          EnableDataGrid(false);
1562 :          if(B.dStimFreq == 0)  DataGrid[2,i]->Value::set("Twitch");
1563 :          if(B.cPol == 'E')     DataGrid[7,i]->Value::set("E");
1564 :          else          DataGrid[7,i]->Value::set("O");
1565 :          EnableDataGrid(true);
1566 :
1567 :        }
1568 :        return 0;
1569 :      }
1570 :
```

Serielle Kommunikation

Bevor Daten zum Stimulator gesendet werden können, muss der richtige Portname eingegeben werden. In der Menüleiste kann der COM - Port bestimmt werden, dabei hat man die Auswahl zwischen COM1 und COM2, oder man gibt den Portnamen direkt ein.

Das COM - Port kann getestet werden, indem man Daten sendet und das Echo empfängt. Bei dem Echo-Modem werden die Pins 2 und 3 verbunden. Die Funktion MStrip_C_PortTest_Click sendet den String „Hallo Echo!“ und falls der gleiche Text wieder empfangen wird, ist der Test erfolgreich und wird mit einem <okay> bestätigt.

```
1744 : private: System::Void MStrip_C_PortTest_Click(System::Object^  sender, ▶
                                                      System::EventArgs^  e)  {
1745 :        InfoList("--- Port Test  ... sending data");
1746 :        String^ str = "";
1747 :        Com.InitCom();
1748 :        Com.Connect();
1749 :        Com.sendString("Hallo Echo!");
1750 :        str = Com.getString();
1751 :        if(str == "Hallo Echo!") InfoList("    <okay>     ... echo received");
1752 :        else             InfoList("    <error>    ... no/wrong data received");
```

```
1753 :         Com.Disconnect();
1754 :
1755 :       }
1756 : private: System::Void MStrip_C_ComXTxt_TextChanged(System::Object^  sender, ►
                                                          System::EventArgs^  e) {
1757 :         MStrip_C_Com1->Checked::set(false);
1758 :         MStrip_C_Com2->Checked::set(false);
1759 :         MStrip_C_ComX->Checked::set(true);
1760 :         Com.ComName::set(MStrip_C_ComXTxt->Text::get());
1761 :       }
1762 : private: System::Void MStrip_C_Com1_Click(System::Object^  sender, System::EventArgs^  e) {
1763 :         MStrip_C_Com1->Checked::set(true);
1764 :         MStrip_C_Com2->Checked::set(false);
1765 :         MStrip_C_ComX->Checked::set(false);
1766 :         Com.ComName::set("COM1");
1767 :         InfoList("--- Port has changed: COM1");
1768 :
1769 :       }
1770 : private: System::Void MStrip_C_Com2_Click(System::Object^  sender, System::EventArgs^  e) {
1771 :         MStrip_C_Com1->Checked::set(false);
1772 :         MStrip_C_Com2->Checked::set(true);
1773 :         MStrip_C_ComX->Checked::set(false);
1774 :         Com.ComName::set("COM2");
1775 :         InfoList("--- Port has changed: COM2");
1776 :
1777 :
1778 :       }
1779 : private: System::Void MStrip_C_ComX_Click(System::Object^  sender, System::EventArgs^  e) {
1780 :         InfoList("--- Port has changed: "+ MStrip_C_ComXTxt->Text::get());
1781 :       }
```

Daten zu dem Stimulator senden

Das Protokoll zur seriellen Datenübertragung wurde bereits bei der Stimulator - Software genau beschrieben. Das Flussdiagramm in der Abbildung 2.14 (Seite 46) zeigt die wesentlichen Elemente des Protokolls und kann auch für das Unterprogramm btnSendData_Click(…) zur bildlichen Darstellung verwendet werden. Die Bezeichnungen Senden/Empfangen müssen jedoch vertauscht werden, da man sich auf der anderen Seite des Übertragungskanals befindet.

Die Übertragung der Stimulationsprogramme erfolgt sequenziell, d. h., jedes Programm durchläuft das gesamte Protokoll unabhängig von den anderen Programmen. Das bringt den Vorteil, dass die Simulationsprogramme flexibel übertragen werden können.

Die Grundstruktur des Unterprogramms bildet eine for - Schleife, welche über alle Programme iteriert und nur die Ausgewählten zum Senden freigibt. Im Anschluss lässt sich das Übertragungsprotokoll in vier Ebenen unterteilen:

1. (handshake) – überprüfen der Verbindung
2. (check data size) – senden der Länge des Datenvektors
3. (data transmission) – die Stimulationsparameter werden gesendet
4. (final protocol) – Statusbericht von Stimulator

```
1783 : private: System::Void btnSendData_Click(System::Object^  sender, System::EventArgs^  e) {
1784 :
1785 :         String^ info = "";
1786 :         String^ str = "";
1787 :         bool execute = false;
1788 :
1789 :         Byte bTmp=0;
1790 :
1791 :         unsigned int DataSize;
1792 :         int iter;
```

```
1793 :         int error;
1794 :
1795 :         StatusStrip_Progress->Style::set(System::Windows::Forms::ProgressBarStyle::Marquee);
1796 :         Com.InitCom();
1797 :         Com.Connect();
1798 :
1799 :         for(int i=0; i<5; i++)
1800 :         {
1801 :           switch(i)  {
1802 :             case 0: execute = ckdBSel1->Checked::get();break;
1803 :             case 1: execute = ckdBSel2->Checked::get();break;
1804 :             case 2: execute = ckdBSel3->Checked::get();break;
1805 :             case 3: execute = ckdBSel4->Checked::get();break;
1806 :             case 4: execute = ckdBSel5->Checked::get();break;
1807 :           }
1808 :
1809 :           if(execute == true)    {
1810 :
1811 :            info = "--- SENDING DATA OF [ Program " + Convert::ToString(i+1) + " ] ---";
1812 :            InfoList(info);
1813 :
1814 :            BData.GenDataStream(i);
1815 :            DataSize = BData.DataStream->Count::get();
1816 :            DataSize /= 30;
1817 :            DataSize++;
1818 :
1819 :            error = 0;
1820 :            iter = 0;
1821 :
1822 :            while(iter<=8) {
1823 :
1824 :              switch(iter) {
1825 :                case 0:                 //   1 - handshake
1826 :                  Com.sendString("HiStim!");
1827 :                  if(Com.ErrorValue::get() != 0) {
1828 :                    InfoList("<error> handshake failed - cannot send data");
1829 :                    iter = 100;
1830 :                  }break;
1831 :                case 1:
1832 :                  str = Com.getString();
1833 :                  if(Com.ErrorValue::get() != 0) {
1834 :                    InfoList("<error> handshake failed - cannot read data");
1835 :                    iter = 100;
1836 :                  } else {
1837 :                    if(str == "HiComp!")    InfoList("<okay> handeshake succesful");
1838 :                    else{
1839 :                      InfoList("<error> handshake failed - received wrong handshake-word");
1840 :                      iter = 100;
1841 :                    }
1842 :                  }
1843 :                  break;
1844 :                case 2:                 // 2 -  check data size
1845 :                  Com.sendByte(Convert::ToByte(DataSize));
1846 :                  if(Com.ErrorValue::get() != 0) {
1847 :                    InfoList("<error> transmit data size: failed - cannot send byte");
1848 :                    iter = 100;
1849 :                  }
1850 :                  break;
1851 :                case 3:
1852 :                  str = Com.getString();
1853 :                  if(Com.ErrorValue::get() != 0) {
1854 :                    InfoList("<error> transmit data size: failed - cannot read data");
1855 :                    iter = 100;
1856 :                  } else {
1857 :                    if(str->Contains("TS:ok"))   {
1858 :                      InfoList("<okay> transmission of data size (" + str + ")");
1859 :                    }else  {
1860 :                      if(str->Contains("TS:err")) InfoList("<error> send data size");
1861 :                      else InfoList("<error> send data size: fatal error");
1862 :                      iter = 100;
1863 :                    }
1864 :                  }
1865 :                  break;
1866 :                case 4:                 // 3 - data transmission
1867 :                  for(int i=0; i<BData.DataStream->Count::get(); i++){
1868 :                    Com.sendByte(BData.DataStream[i]);
1869 :                    if(Com.ErrorValue::get() != 0) {
1870 :                      InfoList("<error> data transission: failed (cannot send byte)");
1871 :                      iter = 100;
1872 :                      break;
1873 :                    }
1874 :                  }
1875 :                  break;
1876 :                case 5:                 // 4 - final protocol
1877 :                  str = Com.getString();
1878 :                  InfoList(str);
1879 :                  break;
1880 :              }
```

```
1881 :             iter++;
1882 :           }
1883 :         }
1884 :       }
1885 :
1886 :       Com.Disconnect();
1887 :
1888 :
1889 :       StatusStrip_Progress->        ►
             Style::set(System::Windows::Forms::ProgressBarStyle::Continuous);
1890 :
1891 :     }
1892 :
1893 : };
1894 : }
```

KAPITEL 3 Messsystem

Das Messsystem besteht aus drei wesentlichen Gruppen: der Sensorik, der hardwareseitigen Signalverarbeitung und der Datenerfassung am Computer. Der erste Teil dieses Kapitels ist der Sensorik gewidmet, das die verwendeten Beschleunigungssensoren beschreibt, aber auch einen größeren Bogen über theoretische Überlegungen spannt. Die Signale der Sensoren werden hardwaremäßig aufbereit, das im zweiten Abschnitt gezeigt wird. Die Datenerfassung und deren Visualisierung werden nur kurzbündig im dritten Teil beschrieben.

3.1. Beschleunigungssensor

3.1.1. Grundlagen über Beschleunigungssensoren

Unter der Beschleunigung ($\vec{a}$) versteht man die zeitliche Änderung der Geschwindigkeit ($\vec{v}$), stellt also die zweite Ableitung einer Ortsposition nach der Zeit dar. Nach dem Grundsatz Kraft = Masse x Beschleunigung ($\vec{F} = m \cdot \vec{a}$) wird für die Messung der Beschleunigung die Auslenkung einer seismischen Masse registriert, die von einem Aufnehmer nach verschiedenen Prinzipien in ein elektrisches Signal umgewandelt wird.

Das Anwendungsgebiet von Beschleunigungssensoren deckt einen großen Bereich ab. So wird in der Automobilindustrie der Sensor in der Auslöseelektronik zur Airbagaktivierung verwendet. Aber auch im Maschinenbau spielen sie bei der Maschinendiagnose und überwachung (z. B. zur Vibrationsmessung) eine wichtige Rolle (Glück, 2005). Mithilfe der MEMS - Technologie (mikroelektromechanische Sensoren) können Low-Cost - Bauelemente gefertigt werden, welche eine Verwendung in vielen Applikationen ermöglicht. Zum Beispiel kann mit der Ausnutzung der Erdbeschleunigung der Neigungswinkel oder das Fallen eines Gerätes detektieren werden. Letztere Anwendung wird auch in der Medizintechnik zum Patienten - Monitoring verwendet. Da die untere Grenze des Frequenzbereiches bei den meisten Beschleunigungssensoren 0 Hz ist, kann auf die Positionsänderung des Sensors rückgerechnet werden. In der wissenschaftlichen Arbeit „Accelerometer for Mobile Robot Positioning“ (Hugh, et al., 2001) werden speziell auf die Probleme der Positionsberechnung eingegangen.

Akzelerometrische Messsysteme in der Medizin:

Sowohl in verschiedenen Gebieten der medizinischen Forschung als auch im klinischen Bereich werden Akzelerometer seit einigen Jahren vermehrt eingesetzt. (Gallasch, et al., 1993).

Die Anwendungsgebiete sind: Kinematik von Körpersegmenten - wie zum Beispiel die Ganganalyse, posturaler Tremor, Schlafuntersuchungen … (vgl. (Mathie, et al., 2004)(Chen, et al., 2005)(Trost, et al., 2005)(Culhane, et al., 2005)(Bernmark, et al., 2002)). Im Vergleich zu bildgebenden Messmethoden auf Basis von Markern haben Akzelerometer den Vorteil, dass das Inertialsystem im bewegten System selbst gebildet wird. Hierbei ist der Graviationsvektor zur genauen Bestimmung der Wegtrajektorie ein Störfaktor.

Mikrobewegung und Körperschall: Verschiedene Arten von Tremor - vgl. folgende Literaturhinweise (Gallasch, et al., 1994)(Hallett, 1998), Phonokardiographie, Ballistokardiogrphie - zeichnet den Rückstoß auf, welchen der Körper infolge des Blutausstoßes des Herzens (Systole) erfährt, Seismokardiographie - registriert das direkte seismische Signal infolge der Herztätigkeit am Brustbein, Atemgeräusche, turbulente Strömungen in Blutgefäße, Mechanomyographie MMG - (vgl.(Rafolt, et al., 2002) (Cescon, et al., 2004)(Watakabe, et al., 2003)).

Strukturanalyse: Zur Untersuchung der mechanischen Struktur von Skelett und Gewebe (Frakturanalyse, Untersuchung schädigender Wirkung von Werkzeugen etc.)

Grundstruktur

Beschleunigungssensoren bestehen in den meisten Fällen aus einer seismischen Masse (Probemasse), die mittels mechanischer Feder so aufgehängt wird, dass sie in einer Richtung frei beweglich ist. Unter Feder versteht man in diesem Zusammenhang eine Struktur, die bei Auslenkung eine Rückstellkraft entwickelt, die meistens proportional zur Wegdifferenz ist. Diese Struktur wird aus einer rundum eingespannten Membrane sein, kann aber auch aus DMS - bestückten Biegebalken bestehen (ENTRAN). Wird der Sensor in dieser beweglichen Richtung beschleunigt, kommt es aufgrund der Trägheit zu einer Auslenkung der seismischen Masse um den Betrag x. Diese Positionsveränderung kann mittels induktiver, piezoresistiver, piezoelektrischer oder kapazitiver Methoden gemessen werden. (Glück, 2005).

Im einfachsten Fall lassen sich die mechanischen Elemente des Beschleunigungssensors als ein System bestehend aus der Probemasse, einer Feder und dem Dämpfungsglied modellieren. Die Abbildung 3.1,a zeigt das mechanisch Modell eines Beschleunigungssensors, bei der die Probemasse nur in horizontaler Richtung bewegt werden kann. Das lineare System wird charakterisiert durch die seismische Masse m $([m] = 1kg)$, durch die Dämpfungskonstante b $([b] = [F]/[v] = N \cdot s/m)$ und durch die Federkonstante k $([k] = [F]/[x] = 1\ N/m)$. Das Koordinatensystem in dieser Abbildung

bezieht sich auf den Massenschwerpunkt in der Ruhelage ($a_m = 0\ m/s^2$). Wird die Aufhängung der Probemasse, einer Beschleunigung a in Messrichtung ausgesetzt, so ergibt sich nach der Formel 3.1 eine Trägheitskraft (F_T), wobei a_m die Beschleunigung der seismischen Masse ist. (Wüstling, 1997). Die zeitlichen Ableitungen werden in der newtonschen Schreibweise dargestellt: $\dot{x}(t) \triangleq \frac{d^2x(t)}{d\,t^2}$

$$F_T = m \cdot a_m(t) = m \cdot (\ddot{x}(t) - a(t)) \qquad \textbf{3.1}$$

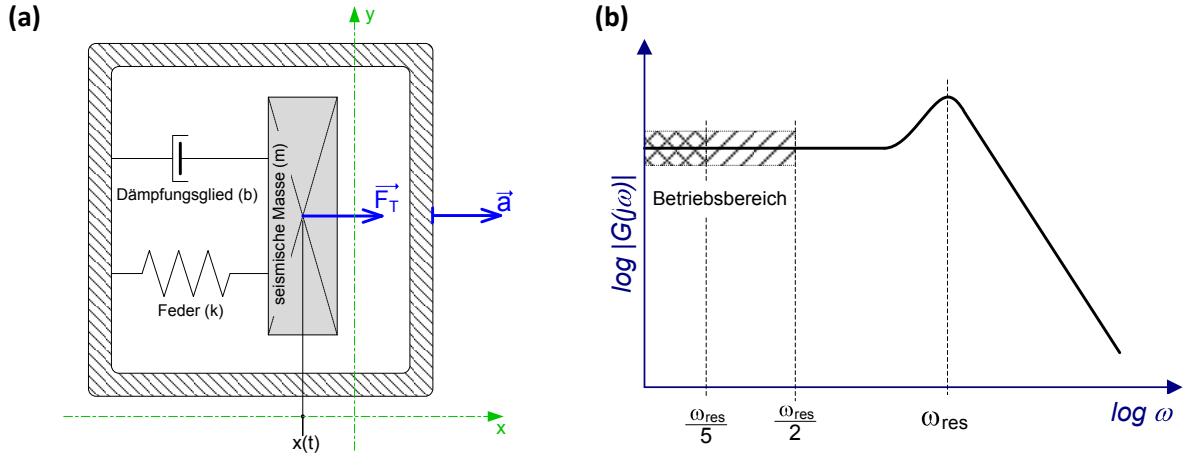


Abbildung 3.1: Modellierung des Beschleunigungssensors
(a) Mechanische Model des Beschleunigungssensors
(b) Übertragungsfunktion G(jω) des mechanischen Models,

Das mechanische Modell des Beschleunigungssensors gibt Aufschlüsse über das dynamische Verhalten. So begründet sich die Differenzialgleichung 3.2 auf das 2. Newtonschen Axiom (Aktionsprinzips), welches aussagt, dass auf die Summe aller Kräfte gleich ist mit der Trägheit des Körpers.(Fraden, 1993 S. 171)

$$F_T = -b \cdot \dot{x}(t) - k \cdot x(t) \qquad \textbf{3.2}$$

$$m \cdot a(t) = m \cdot \ddot{x}(t) + b \cdot \dot{x}(t) + k \cdot x(t) \qquad \textbf{3.3}$$

In der Gleichung 3.3 wurde F_T substituiert und die Terme der beiden Variablen $a(t)$ und $x(t)$ getrennt. In dem nächsten Schritt wird die Laplacetransformation $\mathcal{L}\{\cdot\}$ angewendet um die Systemeigenschaften im Bildbereich zu zeigen. Aus der folgenden Gleichung erkennt man, dass das Model eine Tiefpasscharakteristik hat. Der Frequenzgang der Übertragungsfunktion wird in Abbildung 3.1, (b) dargestellt.

$$G(s) = \frac{\mathcal{L}\{x(t)\}}{\mathcal{L}\{a(t)\}} = \frac{m}{m \cdot s^2 + b \cdot s + k} \qquad \textbf{3.4}$$

Dieser Graph zeigt die Resonanzfrequenz ω_{res}, welche eine wichtige Systemeigenschaft ist. Dieser Wert bestimmt den Frequenzbereich, dessen obere Grenzfrequenz in dem Intervall von 1/5 bis 1/2 der Resonanzfrequenz liegt (Shieh, et al.,

2001). Aus der Gleichung 3.5 kann man nach Umformung feststellen, dass diese Systemeigenschaft direkt von der Dämpfungs- und Federkonstante abhängt.

Messprinzipien der Beschleunigung

In diesem Abschnitt werden verschiedene Beschleunigungssensorarten oberflächlich beschrieben. Das oben genannte Sensormodel kann für alle folgenden Beschleunigungssensoren angewendet werden. Ein struktureller Unterschied besteht nur zu piezoelektrischen Akzelerometer, bei denen die seismische Masse nicht über einen Biegebalken aufgehängt ist.

Das folgende Diagramm zeigt den Verwendungsbereich der einzelnen Beschleunigungssensoren, in Bezug auf die obere Grenzfrequenz und den Bereich der Beschleunigungswerte. Alle Akzelerometer, bis auf piezoelektrische Sensoren, besitzen einen untere Grenzfrequenz von 0 Hz. Mit diesen Beschleunigungssensoren kann durch zeitliche Integration die Sensorgeschwindigkeit und in weitere Folge die Positionsänderung berechnet werden.

Die Graphen zeigen den Zusammenhang, dass Sensoren mit einem größeren Beschleunigungsbereich auch eine höhere Grenzfrequenz besitzen. Betrachtet man dafür die Modellierung, so erkennt man, dass durch eine steifere Feder die seismische Masse stärker fixiert wird und somit die Resonanzfrequenz nach oben verschoben wird.

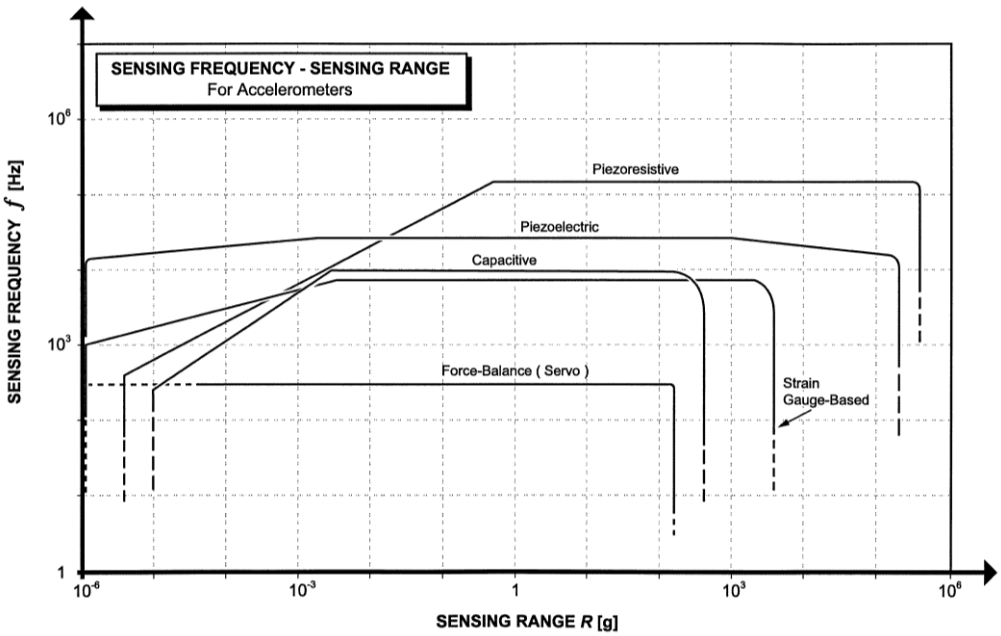


Abbildung 3.2: Anwendungsbereich von Beschleunigungssensoren
Quelle: (Shieh, et al., 2001 S. 472)

Kapazitiver Sensoren

Die kapazitiven Akzelerometer detektieren die Positionsänderung der seismischen Masse über den Abstand zwischen zwei Kondenstorelektroden. Eine Elektrode ist stationär und ist mit dem Gehäuse verbunden, im Gegensatz dazu steht die Zweite, welche mit der beweglichen Masse in Verbindung steht. Die meisten kapazitive Beschleunigungssensoren

sind in MEMS - Technologie gefertigt, die in dem folgenden Bild exemplarisch dargestellt wird.

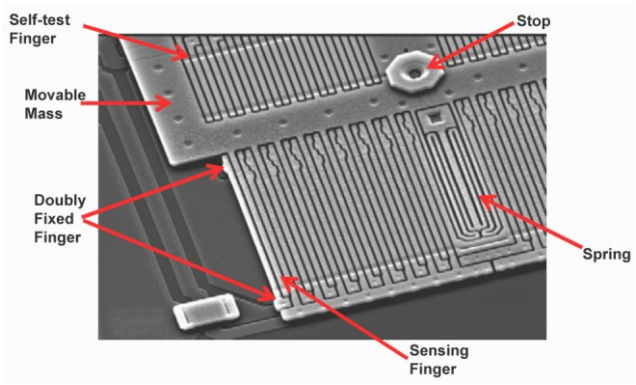


Abbildung 3.3: Kapazitiver Beschleunigungssensor
Sensorzelle, deren seismischen Masse an beiden Enden befestigt ist.
Quelle: Freescale; MMA7260Q - Course

In dem Buch „Sensorschaltungstechnik“ sind einige Messschaltungen aufgelistet, die zum weitern Studium empfohlen werden (Schmidt, 2002). Die Sensoren dieser Technologie zeichnen sich durch ihre Größe und niedrigen Preis aus. Durch die miniaturisierte Bauweise können die Probleme von Rauschen und nicht - Linearität besser behandlet werden, wobei die Genauigkeit und Auflösung, in Bezug auf die anderen Beschleunigungssensoren, schlechter ausfällt.

Piezoresistive Sensoren und DMS - basierende Sensoren

Beide Akzelerometertypen messen die Dehnung des Biegebalkens, der die seismische Masse in der Ruhelage hält. Die Messung der Biegespannung unterscheidet beide Typen und wirkt sich augenscheinlich auf die Baugröße des Beschleunigungssensors aus.

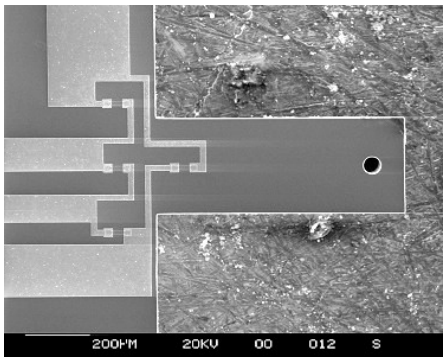


Abbildung 3.4: Piezoresistive Beschleunigungssensor
Biegebalken mit Widerstandsbrücke – Quelle: National Physical Laboratory, UK
(http://www.npl.co.uk/materials/functional/images/piezoresistive.jpg, 10.04.2007)

Piezoresistive Sensoren werden heutzutage meist aus einem Stück Silizium gefertigt. Bei diesen Verfahren ist das Sensorelement ca. 1 mm^2 groß, wodurch man einen Frequenzbereich bis zu 1 MHz erreichen kann. Die Abbildung 3.4 zeigt ein Beispiel für so ein Sensorelement, bei dem die Widerstände zu einer Brücke zusammengeschaltet sind.

Bei DMS - basierte Beschleunigungssensoren wird die Biegung des Balkens mit Folien-Dehnmessstreifen gemessen. Verglichen mit den piezoresistiven Sensoren ist dieses Messelement wesentlich größer und somit wird der Messbereich und die obere Grenzfrequenz niedriger.

Piezoelektrische Akzelerometer

Bei piezoelektrischen Beschleunigungssensoren gibt es keinen Biegebalken, stattdessen wird die seismische Masse direkt auf den piezoelektrischen Quarz montiert. Diese Sensoren sind nicht für statische Anwendungen geeignet, da der Frequenzbereich von ca. 0,1 Hz bis 30 kHz liegt. Hervorzuheben sind die guten Eigenschaften der Rauschunterdrückung und der Linearität.

Servo - Beschleunigungsaufnehmer

Diese Sensoren werden auch kraftkompensiert Beschleunigungsaufnehmer genannt. Das Messsignal wird in diesem Fall nicht durch die Positionsänderung der seismischen Masse erzeugt, sondern es wird versucht über eine Rückkopplung die Masse in ihrer Initialposition zu halten. Traditionell wir die Rückstellkraft durch Schwingspulen erzeugt, dessen Spulenstrom proportional der Beschleunigung ist. Bei modernen kapazitiven MEMS Sensoren, wird dieses Rückkopplungssignal elektrostatisch erzeugt. Durch diese Kraftkompensierung kann eine deutlich höhere Auflösung und Genauigkeit erreicht werden, jedoch können keine hohe Frequenzen gemessen werden (< 300 Hz).

3.1.2. Beschleunigungssensoren - MMA7260Q und LIS3L02AQ

Auswahlkriterien

Der kapazitive Beschleunigungssensor LIS3L02AQ der Firma STMicroeletronics wurde als Sensorelement gewählt. Für vorangegangene Tests wurde auch der Sensor MMA7260Q der Firma Freescale eingesetzt. Im folgenden Absatz werden die wesentlichen Eigenschaften kurz dargestellt, wobei für die weitere Messung nur der Akzelerometer LIS3L02AQ von Interesse ist.

Die Auswahl der Beschleunigungssensoren ist nach den folgenden vier wesentlichen Eigenschaften getroffen worden:

1. kleine Sensormasse und Abmessungen
2. Messbereich bis 2 g (= 19,62 m/s^2)
3. geringe Rauschspannung
4. kleiner Sensorpreis (Sensor-Array besteh aus 9 Sensoren)

Eigenschaften der Beschleunigungssensoren:

	LIS3L02AQ	MMA7260Q
Versorgungsspannung	von 2,4 V bis 3,6 V	von 2,2 V bis 3,6 V
Strombedarf	850 µA	500 µA
Messbereiche	2g / 6g	1.5g/2g/4g/6g
Abmessungen	7 mm x 7 mm x 1,8 mm	6 mm x 6 mm x 1,45 mm
Gehäusebauform	QFN - 44 Pins	QFN - 16 Pins
Rauschdichte	50 $mg/\sqrt{Hz}$	350 $mg/\sqrt{Hz}$
Ausgangswiderstand	110 kΩ	
Preis (vgl. Spoerle.com)	ca. 13,- EUR	ca. 13,- EUR

Tabelle 3.1: Eigenschaften der Beschleunigungssensoren

Histogramm- und Varianzmessung

Bei der Messung des Histogramms und der Varianz befinden sich beide Sensoren in der Ruhelage. Bei der Messkonfiguration wurde das Filter Array mit der Verstärkung 1 ($R_G = \infty$) verwendet, wobei die Multiplexer-Funktion in diesem Fall nicht verwendet wurde. Die Daten werden im Anschluss von einer Ni-DAQ Karte (PCI-6221) erfasst und für eine computergestützte Weiterverarbeitung bereitgestellt. Das analoge Signal wird dabei mit einer Samplingfrequenz von 10 kHz abgetastet. Für die Auswertung wurde das Programm DASYLab verwendet, dessen Schaltplan in der Abbildung 3.5 gezeigt wird.

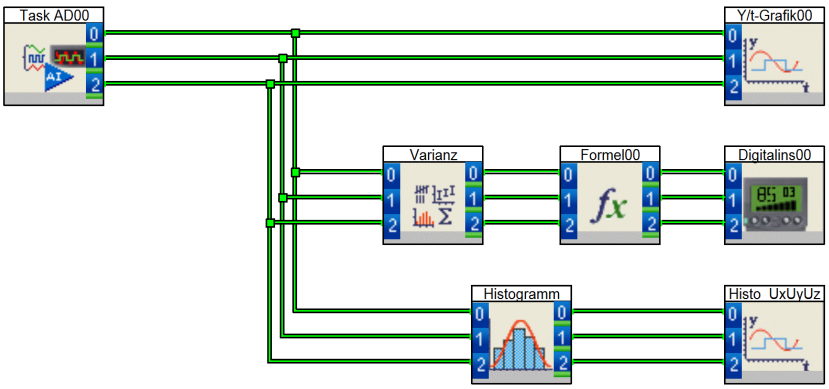


Abbildung 3.5: DASYLab - Schaltbild zur Histogramm- und Varianzmessung

Histogramme:

Die Messzeit betrug ca. 8 min, wodurch für die Ermittlung des Histogramms ca. $5*10^6$ Messwerte zu Verfügung stehen. Damit man beide Sensoren besser miteinander vergleichen kann, wurde die gleiche Intervallbreite gewählt. Die Sensorspannung wird in dem Bereich von 60 mV in 1000 Intervalle unterteilt, das für jedes Intervall einen Spannungswert von 60 µV ergibt. Auf der Abszisse der folgenden Histogramme wird der

Spannungswert $U_Z(t)$ aufgetragen, während die Ordinate die Häufigkeit darstellt. Die Häufigkeit wird auf den jeweiligen Maximalwert skaliert, d. h., der größte Wert eines Intervalls entspricht 100%.

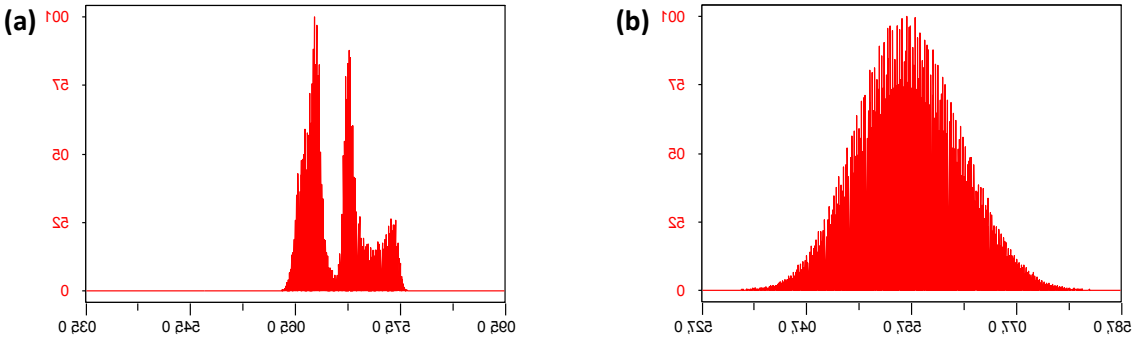


Abbildung 3.6: Histogramme der Rauschspannung beider Beschleunigungssensor
(a) LIS3L02AQ3, (b) MMA7260Q

Varianzmessung:

$$\sigma_{U_X}{}^2 = \frac{1}{N} \cdot \sum_{n=1}^{N} (\overline{U_X} - U_X[n])^2 \qquad \textbf{3.6}$$

	LIS3L02AQ3	MMA7260Q
x - Achse U_x	20,11 μV^2	55,45 μV^2
y - Achse U_y	20,00 μV^2	55,46 μV^2
z - Achse U_z	19,52 μV^2	77,97 μV^2

Tabelle 3.2: Varianzmessung der Beschleunigungssensoren

Foto beider Akzelerometer

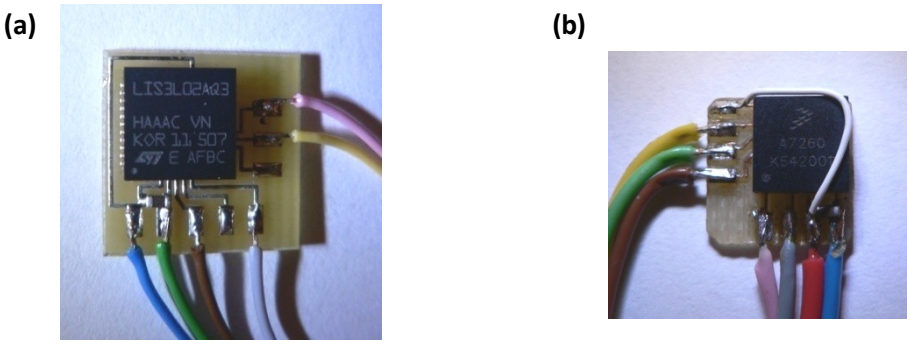


Abbildung 3.7: Foto – Beschleunigungssensor
(a) LIS3L02AQ3, (b) MMA7260Q

Achsenorientierung der Ausgangsspannungen

Die Ausgangsspannungen sind einem kartesischen Koordinatensystem zugeordnet (U_X, U_Y, U_Z). Die folgende Skizze zeigt die Achsenbezeichnung im Bezug auf das Chipgehäuse. Der Punkt in der linken unteren Ecke beschreibt den Pin 1.

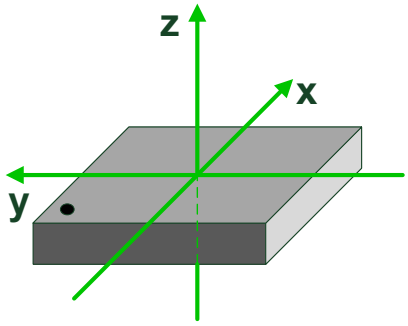


Abbildung 3.8: Achsenorientierung des Beschleunigungssensor
Die Achsen-Richtungen sind für beide Sensoren identisch

Sensorfeld

Das Sensorfeld soll in dieser Stelle nur kurz erwähnt werden, da eine genauere Betrachtung im Kapitel 4 erfolgt. An der Oberschenkelmuskulatur werden normalerweise neun Akzelerometer (vgl. Abbildung 4.16 - Seite 97) angebracht, wobei zunächst eine Datenserie mit fünf Sensoren aufgenommen wurde (vgl. Abbildung 4.2 - Seite 85). Die Sensoren werden mit einem doppelseitigen Klebeband an der Haut befestigt, wobei geachtet werden muss, dass keine Relativbewegungen zur Haut stattfinden können. Das Achsensystem bezieht sich die Oberschenkelgeometrie, so liegt die x-Achse radial (quer) zum Oberschenkelknochen, die y-Achse parallel zum Femur in distaler Richtung und die z-Achse tritt aus der Muskeloberfläche heraus.

3.2. Filter-Array

Das Sensorfeld besteht aus maximal 10 Sensoren mit je drei Ausgangssignalen (U_X, U_Y, U_Z). Es müssen aus diesem Grund 30 Signale aufgezeichnet werden könne. Die Daten werden mit dem Filter-Array für die Ni-DAQ Messkarte aufbereitet, wofür es zwei Funktionen integriert - eine Tiefpassfilterung und das Zeitmultiplexen der Kanäle. In der folgenden Abbildung 3.9 werden die einzelnen Elemente dargestellt:

A – 68-poliger Anschluss an die Ni-DAQ Karte (PC)
B – Versorgungsplatine
C – Redel-Anschluss für Beschleunigungssensor
D – Versorgung der Mux-Filter Platinen (+10V , -10 V, GND)
E – Mux-Filter Platine

Für die Anschlüsse der Beschleunigungssensoren gibt es an der Frontseite zehn 7-polige Redel - Buchse. Die genaue Pinbelegung wird im Anhang B - Abbildung 5.7 gezeigt. An den Buchsen in dem Filter-Array ist der Pin 6 (Auswahl des Beschleunigungsbereich) direkt mit der Masse verbunden, d. h., die Sensoren arbeiten im Bereich +/- 2g.

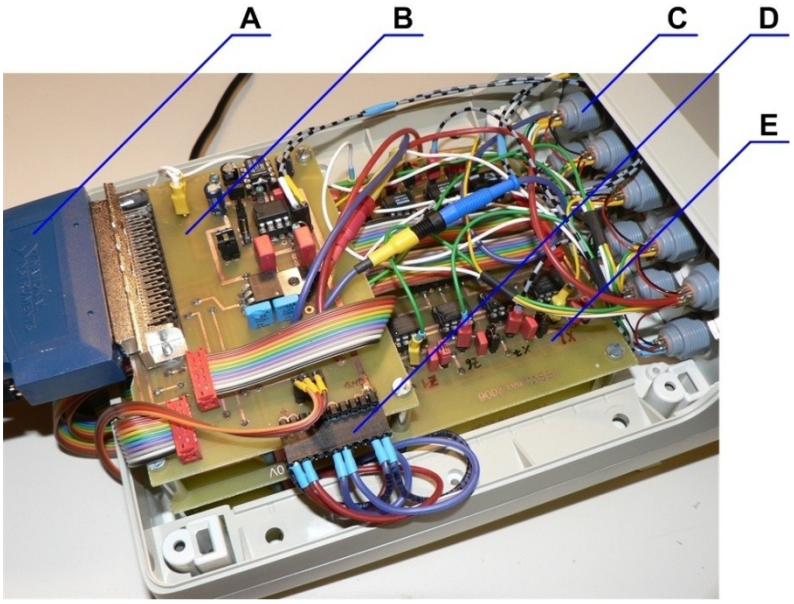


Abbildung 3.9: Foto – Filter-Array

3.2.1. Versorgungsplatine

Die Platine stellt die verschiedenen Versorgungsspannungen bereit, so benötigen die Beschleunigungssensoren 3,3 V und das Filter Array +/- 10 V. Die symmetrische Versorgung des Filter Arrays ist an die Ni - DAQ Karte angepasst. Ein wichtiger Teil dieser Platine ist auch die Taktgenerierung, welche als Grundbaustein den NE555 verwendet. Mit Jumpern wird die Auswahl getroffen, ob das Taktsignal für die Steuerung der Multiplexer verwendet wird, oder ob die Multiplexer in einer konstanten Stellung die Messsignal and die Ni-DAQ Karte weiterleitet. Diese Jumperstellungen werden im Anhang deutlicher beschrieben (vgl. Abbildung 5.9 - Seite 107).

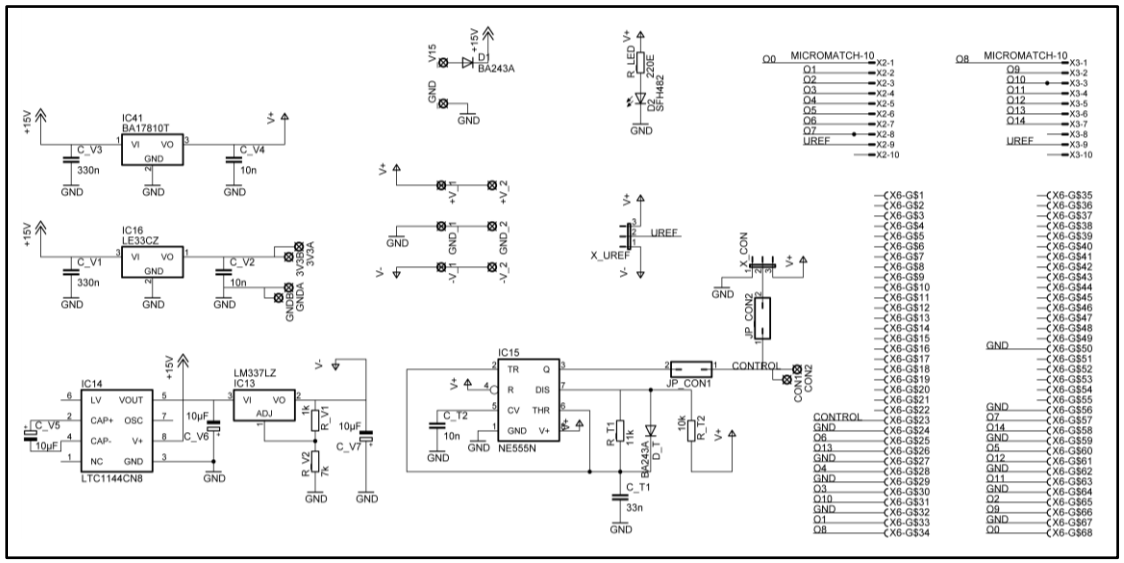


Abbildung 3.10: Filter-Array – Versorgungsplatine

3.2.2. Filter/Multiplexer - Platine

In der Abbildung 3.11 wird der Signalweg zweier korrespondierenden Sensoren beschrieben. Der Tiefpassfilter (Formel 3.7) am Eingang wird aus dem Sensorinnenwiderstand und eine Kondensator gebildet, der mit dem positiven Eingang des Instrumentenverstärkers verbunden ist. Der Multiplexer am Ende des Tiefpasses wird von dem Taktsignal auf der Versorgungsplatine angesteuert. Ein wichtiger Hinweis ist, dass die Abtastfrequenz der Ni-DAQ Karte und die Taktfrequenz des Multiplexers nicht synchronisierst sind.

$$f_G = \frac{1}{2\pi \cdot R_i \cdot C_{11}} \approx 140\ Hz \qquad \textbf{3.7}$$

In dem folgenden Schaltplan (Abbildung 3.12) beschreibt die Platine für eine Ebene, d. h., es werden 16 Eingangskanäle zu 8 Ausgängen zusammengefasst. Bei jedem Instrumentenverstärker kann der R_G - Widerstand unabhängig von den anderen gewählt werden. Die R_G - Leitungen werden dafür an einen eigene Steckerleiste (M1,M2) geführt. (siehe Anhang C)

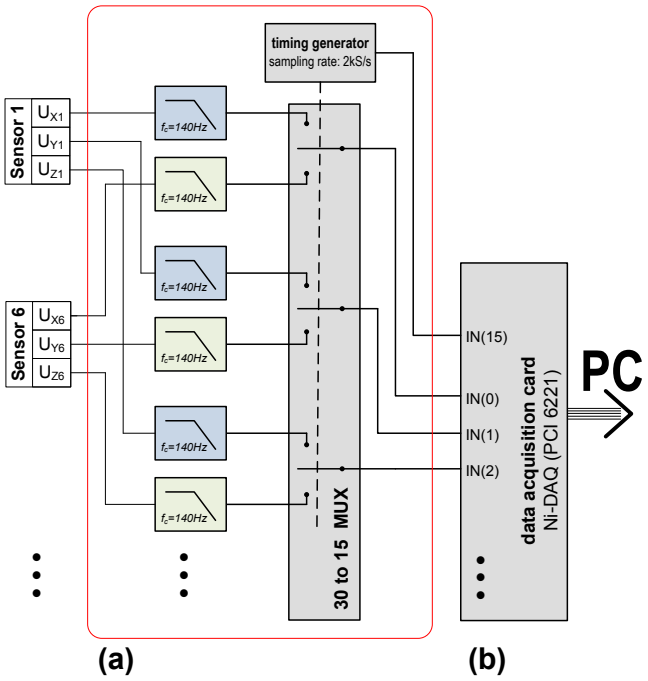


Abbildung 3.11: Prinzipschaltplan – Filter-Array
(a) Signalvorbereitung (b) NI - Messkarte

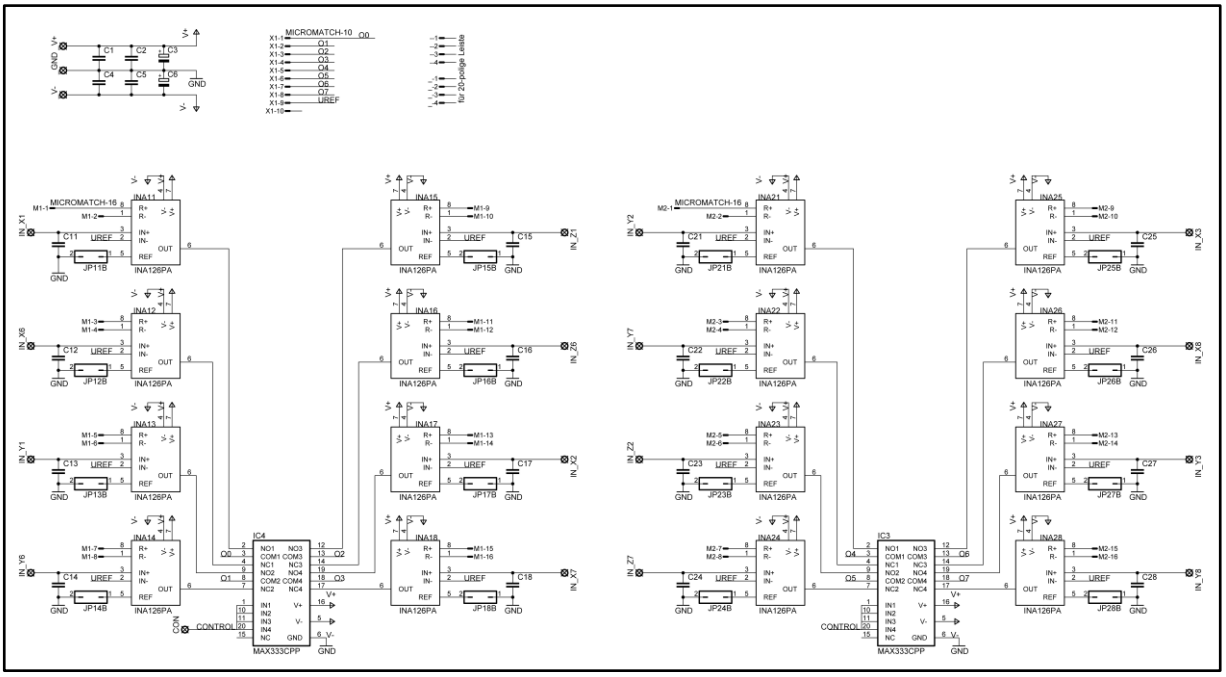


Abbildung 3.12: Filter/Multiplexer - Platine

3.3. Datenerfassung

Von dem Filter-Array werden die analogen Signale an eine PC - Messkarte weitergeleitet, welche die Analog-Digitalkonvertierung durchführt. Die weiteren Berechnungen und die Visualisierung der Daten werden mit den beiden Softwarepaketen DASYLab und MATLAB vollzogen.

3.3.1. Ni-DAQ Datenerfassungskarte

Zur Datenerfassung wird die Messkarte PCI-6221 der Firma National Instruments verwendete. Die folgende Auflistung zeigt die wichtigsten Eigenschaften dieser Messkarte:

- PCI - Messkarte
- 16 pseudodifferenzielle Analogeingänge; 16Bit Auflösung; 250 kHz Abtastrate
- 2 analoge Ausgänge; 16 Bit Auflösung
- 24 digitale I/O-Kanäle
- 2-Bit-Counter

3.3.2. Visualisierungssoftware DASYLab 9.0

Mit DASYLab werden die Daten in eine Datei gespeichert und gleichzeitig verschiedenen Graphen online auf dem Bildschirm angezeigt.

Programmfeatures:

- das Werkzeug zur grafischen Entwicklung von Messapplikationen
- großer Funktionsvorrat für On- und Offline- Berechnungen wie Mittelungen und Filter, Arithmetik, Trigger, FFT, logische Verknüpfungen uva.
- Online-Darstellung der Messdaten als Y/t-und X/Y-Grafik, Schreiber, Zeiger- oder numerischer Anzeige
- Abspeicherung der Messdaten in verschiedenen Datenformaten
- integrierte Layoutfunktion zur Erstellung von Bedienoberflächen und Berichten

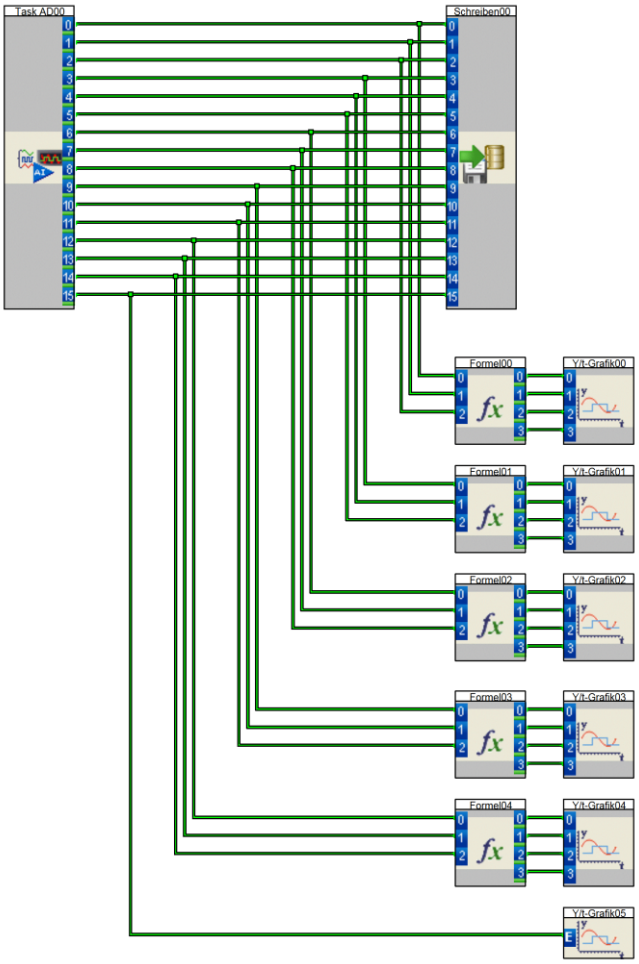


Abbildung 3.13: DASYLab - Schaltplan zum Einlesen der Daten

Die folgende Abbildung zeigt die Elemente des Programms, die bei der Messung im Kapitel 4 verwendet wurde. Die verwendeten Schaltsymbole beinhalten vier unterschiedliche Funktionen: Zu Beginn werden die Daten aus der Ni-DAQ Karte eingelesen und ohne Bearbeitung in eine Datei gespeichert. Bei der Auswahl des Dateiformats bietet DASYLab verschiedene Typen an. In diesem konkreten Fall wurden die Daten in ASCII-Format gespeichert. Vor der Visualisierung kann der Datenstrom noch Berechnungen durchlaufen, so wie in dem folgenden Beispiel, das den Betrag der Eingangssignale berechnet. Jeder Sensor wird separat in einem Fenster angezeigt.

3.3.3. Demultiplexer mit MATLAB

Das Filter-Array liefert als Ausgang das Multiplex - Signal, das aus zwei Sensorsignalen zusammengesetzt wird. Das abgetastet Signal wird in der folgenden Abbildung dargestellt, wobei deutlich die Signale unterschieden werden können. Neben dem Messsignal wird auch das Taktsignal dargestellt, der für die Rekonstruktion nötig ist.

Für das Demultiplexing gibt es zwei wesentliche Probleme:

1. Taktfrequenz läuft nicht synchron mit der Abtastfrequenz
2. Zeitverzögerungen der Multiplexer beim Umschalten der Sensoren

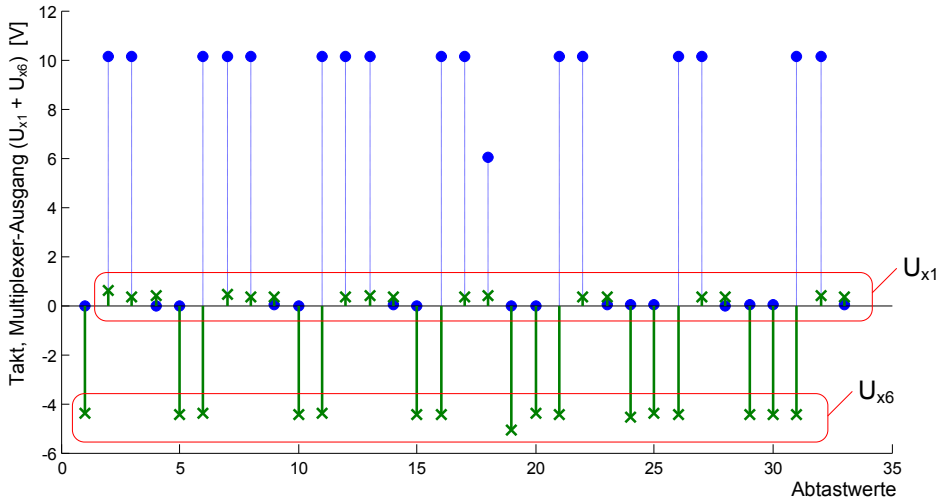


Abbildung 3.14: Taktsignal (•) , U_X-Wert von Sensor 1 und 6 (x)

Der folgende Codeabschnitt zeigt ein Bespiel, das eine Abtastsignal erzeugt, dass zu jedem Takt genau einen Abtastwert zuordnet. Nachteil dieser Art der Zuweisung ist, dass der zeitliche Jitter bis zu einem Abtastsample erhöht wird.

Programm-Listing

```
clear all
% ----------- <1> Öffnen der Daten
load raw_data.mat
%Matrix hat 3 Spalten
%   1...Zeit, 2...Daten, 3...Taktvektor

IndexHigh = find(raw(:,3)> 5);
IndexLow  = find(raw(:,3)<=5);

% --- <2> Berechnung der High-Abtastpunte
High1 = zeros(length(raw(:,3))+1,1);
High2 = zeros(length(raw(:,3))+1,1);

High1(IndexHigh)   = 1;
High2(IndexHigh+1) = 1;
High = High1 & High2;
for i = 1:length(High)
    if(High(i) == 1)
        High(i+1) = 0;
    end
end

% --- <3> Berechnung der Low-Abtastpunte
Low1 = zeros(length(raw(:,3))+1,1);
Low2 = zeros(length(raw(:,3))+1,1);

Low1(IndexLow)   = 1;
Low2(IndexLow+1) = 1;
Low = Low1 & Low2;
for i = 1:length(Low)
    if(Low(i) == 1)
        Low(i+1) = 0;
    end
end
% --- <4> DEMULTIPLEXEN des Datenvetors raw(:,2)
U_x1 = zeros(length(raw),1);
U_x6 = zeros(length(raw),1);

U_x1 = raw(find(Hight>5),2);
U_x6 = raw(find(Low > 5),2);

% --- <5> Ausgabe der Abtastpunkte
n = 50000:50039;
figure(1)
subplot(2,1,1)
    stem([-Low1(n),High1(n)]);
subplot(2,1,2)
    stem([-Low(n),High(n)]);
```

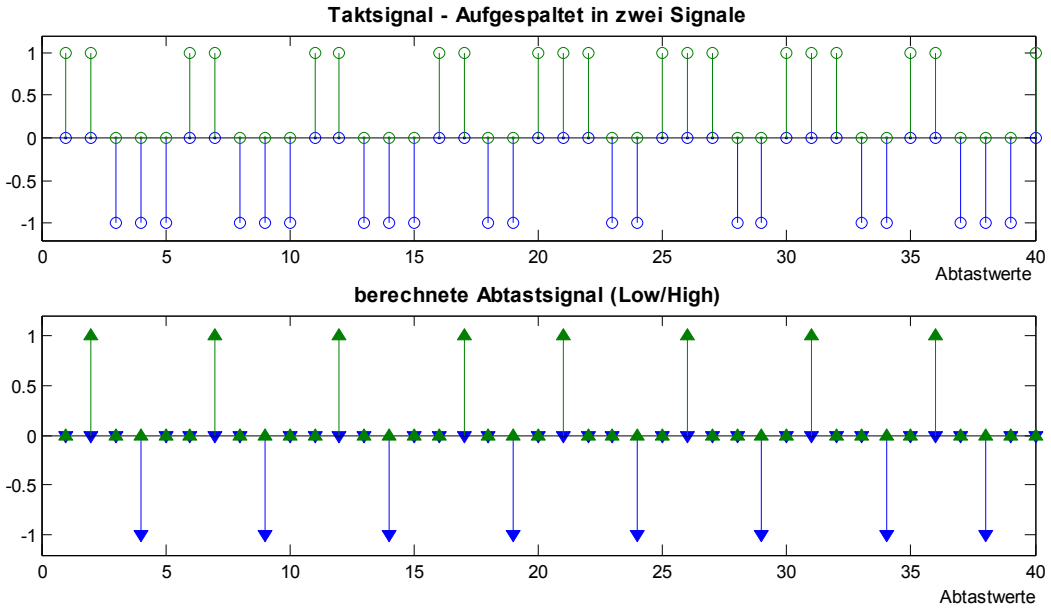


Abbildung 3.15: Generiertes Abtastsignal

Mit dem neuem Abtastsignal können die beiden Signale differenziert werden, das in der folgenden Abbildung gezeigt wird.

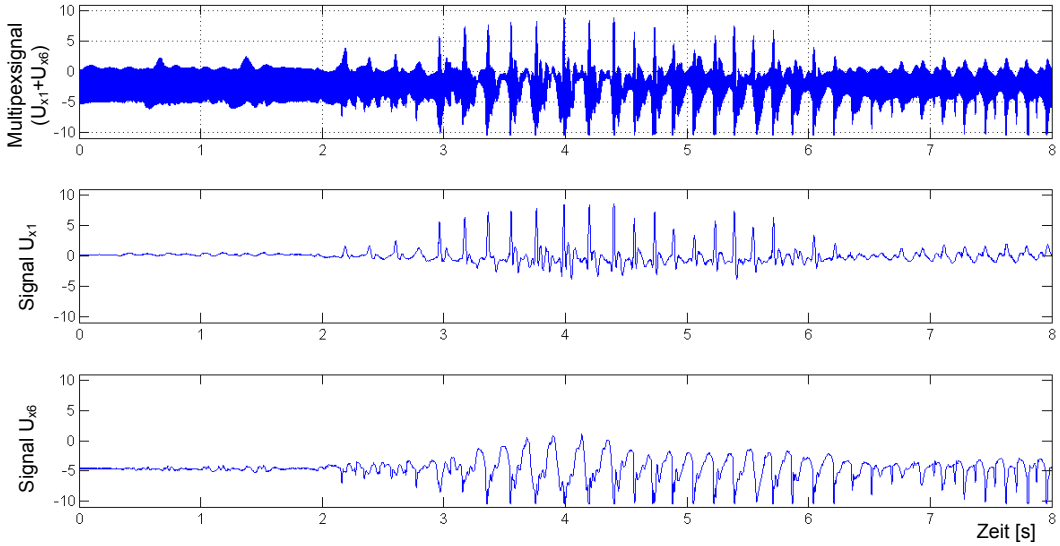


Abbildung 3.16: Ergebnis des Demultipexer-Programms

KAPITEL 4 Messergebnis

Dieses Kapitel zeigt einen exemplarischen Datensatz der Beschleunigungssignale. Es werden verschiedenen Experimente absolviert, die eine Messung bei beabsichtigter Kontraktionen, einer Twitchmessung und Messungen bei verschiedenen Stimulationsfrequenzen inkludieren.

In dem ersten Unterpunkt wird zunächst die Messkonfiguration beschrieben, bei der vor allem die Positionierung der Beschleunigungssensor gezeigt werden. Im Anschluss daran werden die Messergebnisse dargestellt. Das Phänomen der Kontraktionseigendynamik FES-aktivierter Oberschenkelmuskulatur wird in diesem Kapitel nach unterschiedlichen Gesichtspunkten der Signalverarbeitung betrachtet. Den Abschluss bildet ein kurzer Ausblick auf weitere Messungen, die eine quantitative Differenzierung des Phänomens FES - Wave erreichen sollten.

4.1. Messkonfiguration

Auf dem Foto der Messanordnung sind alle wesentlichen Elemente des Messsystems:

A – 2-Kanal Stimulator
B – Filter-Array
C – Stimulationselektrode
D – Beschleunigungssensor (LIS3L02AQ3)
E – Fixierung des Beins

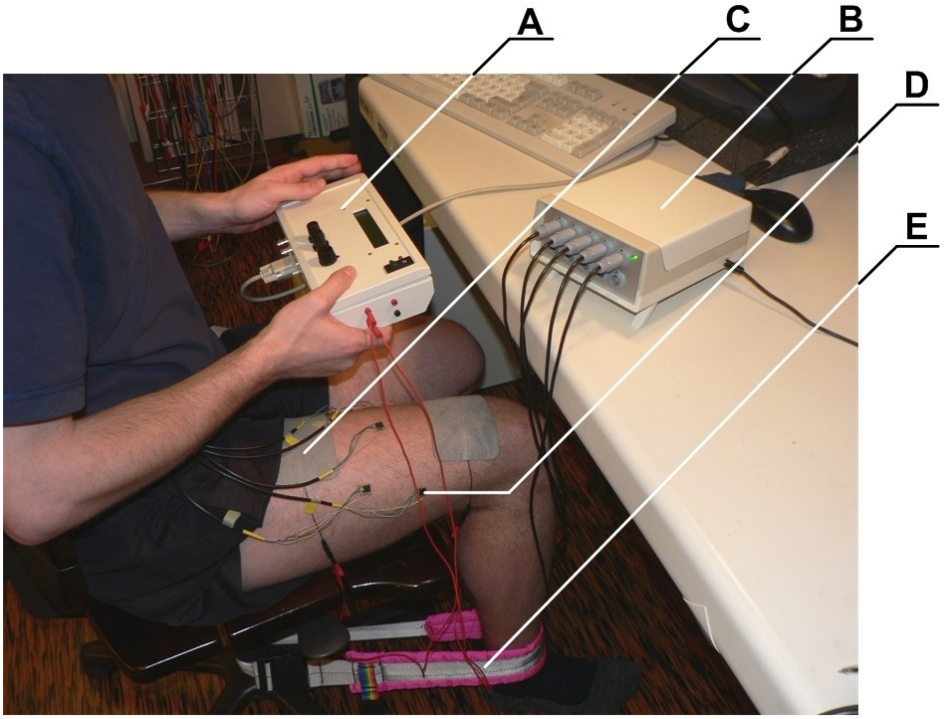


Abbildung 4.1: Foto der Messanordnung

Sensorik, Datenverarbeitung:

Bei dieser Referenzmessung besteht das Sensorfeld nur aus fünf Beschleunigungssensoren. Jeder Sensor hat drei Signalausgänge, welche der Beschleunigung im kartesischen Koordinatensystem entsprechen. Durch diese dezimierte Anzahl an Sensoren müssen die Signale nicht gemultiplext werden, d. h., die 15 Kanäle werden direkt an die Ni-DAQ Karte weitergeleitet, deren Abtastfrequenz bei 10 kHz liegt.

Die folgende Abbildung 4.2 zeigt die Platzierung der Beschleunigungssensoren. Die Sensoren sind so angebracht, dass sie jeweils Zwei auf den Muskelbäuchen des M. vastus lateralis bzw. des M. vastus medialis liegen und ein Sensor die Bewegung des M. rectus femoris aufnimmt.

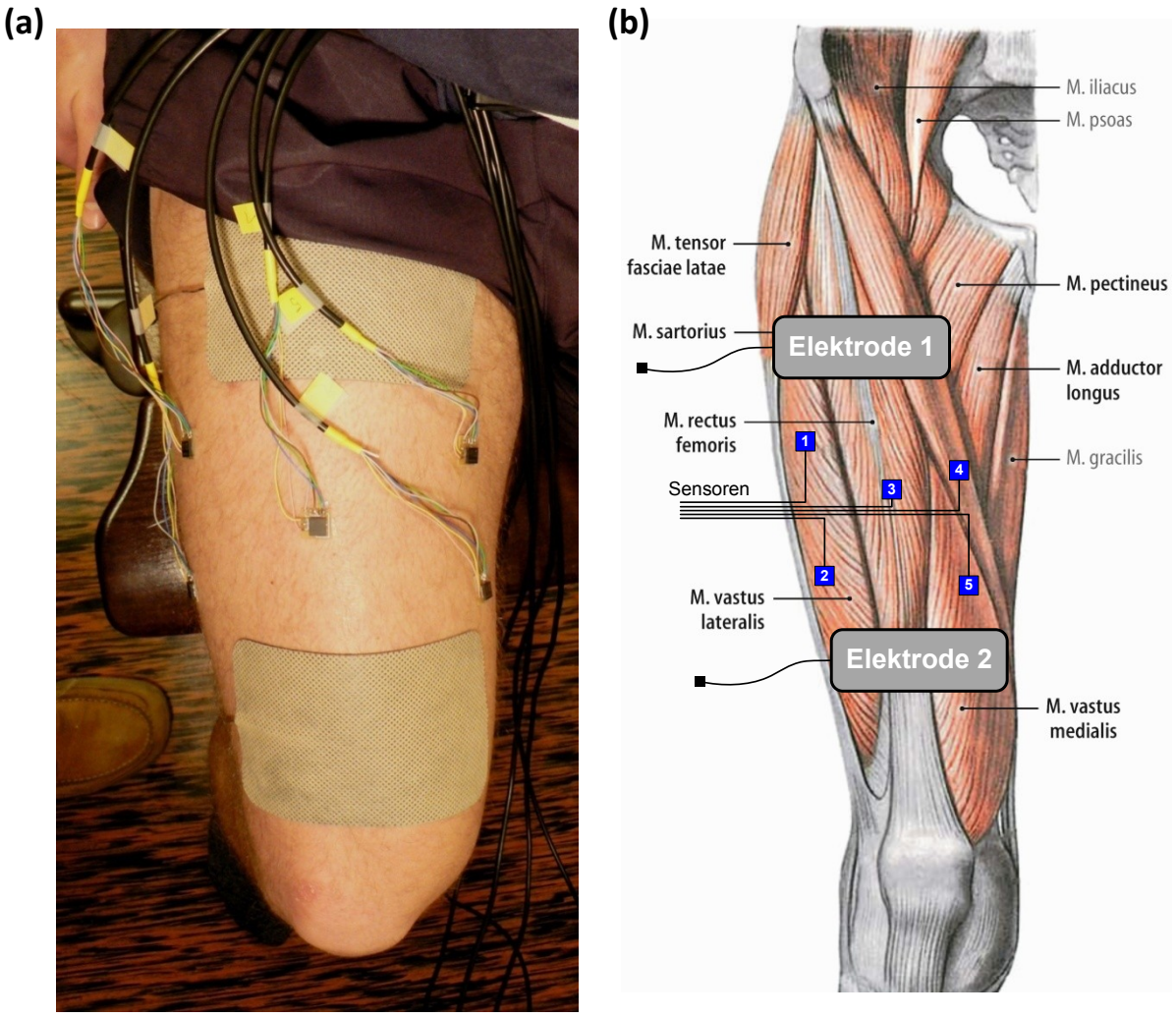


Abbildung 4.2: Platzierung der Beschleunigungssensoren
(a) Foto der fünf Beschleunigungssensoren während der Messung
(b) Positionierung der Sensoren in Bezug auf die Muskelbäuche des M. quadriceps femoris. Quelle der anatomischen Abbildung: (Tillmann, 2005 S. 472)

Um die Visualisierung der Daten übersichtlicher zu gestalten, werden in weiterer Folge nicht alle Signale einzeln angezeigt, sondern nur der Betrag des Beschleunigungssignals:

$$\|U_i\| = \sqrt{{U_{x,i}}^2 + {U_{y,i}}^2 + {U_{z,i}}^2} \quad \textbf{4.1}$$

i … Beschleunigungssensor (1,…,5)
$U_{x,i}$ … Beschleunigungssignal in x-Achse [V]
$U_{y,i}$ … Beschleunigungssignal in y-Achse [V]
$U_{z,i}$ … Beschleunigungssignal in z-Achse [V]

Eine wichtige Bemerkung zu dem Messsystem ist, dass ausschließlich isometrische Muskelkontraktion ausgewertet werden. Das Bein wird mit einem Gurt in einer rechtwinkeligen Position gehalten. In der Abbildung 4.1 ist der Gurt zu erkennen und mit den Buchstaben „E“ bezeichnet.

4.2. Messungen ohne elektrischer Stimulation

Bei der ersten Versuchsanordnung werden die Beschleunigungssignale bei willentlicher Kontraktion untersucht, d. h. ohne elektrische Stimulation. In der folgenden Abbildung werden die Beträge der orthogonalen Signale des Sensors dargestellt (Gleichung 4.1).

1. Messung - Impulskraft:

Bei dieser Messung wurde von dem Probend eine Impulskraft gegen den Fixierungsgurt ausgeübt. Bei dieser versuchten Kniestreckung (isometrische Kontraktion) erkennt man sehr gut die Oberflächenschwingung.

2. Messung - willentliche Maximalkraft (MVC):

Auch bei diesem Versuch wurde eine isometrische Kraft gegen den Gurt ausgeübt, wobei die Aktivierung der maximalen Kraft entspricht (MVC) und über einen längeren Zeitpunkt angehalten werden muss. Das Muskelzittern währende der Kontraktion kann man sehr deutliche in den Messergebnissen sehen.

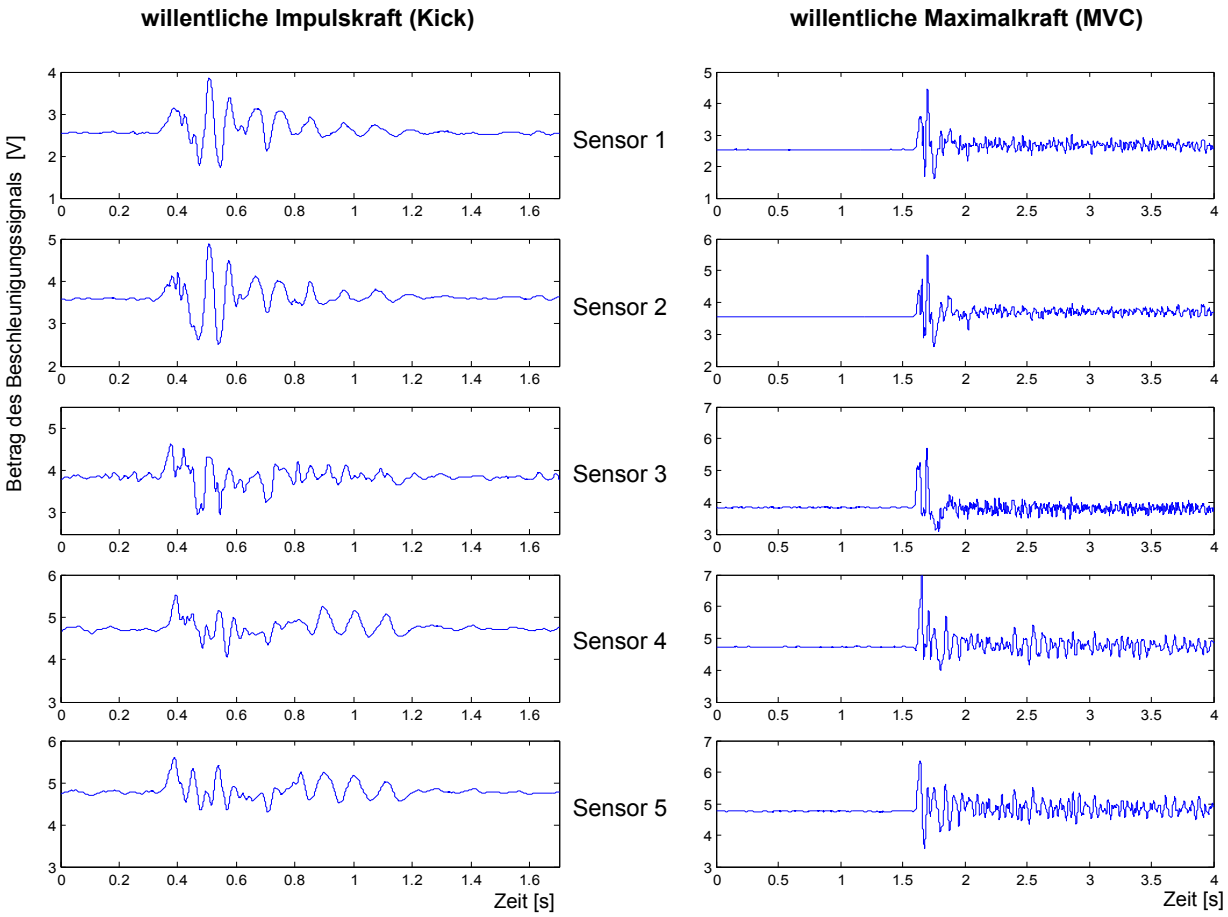


Abbildung 4.3: Impulskraft, Maximalkraft

4.3. Twitchmessung

Der erste Graph zeigt die Stimulationsspannung, die aufgrund ihrer relativ kurzen Breite nur als Nadelimpulse erkennbar ist. Die mechanische Antwort auf diese Stimulationsimpulse wird im unteren Graphen dargestellt. Da alle Signal sehr ähnlich aussehen wir auf diese Redundanz verzichtet und nur das Beschleunigungssignals des 4. Sensors in z-Richtung dargestellt.

Die wissenschaftliche Arbeit „Validation of an accelerometer for determination of muscle belly radial displacement“ von (Zagar, et al., 2005) beschäftigt sich speziell mit diesem Thema und geht auf weitern Facetten ein.

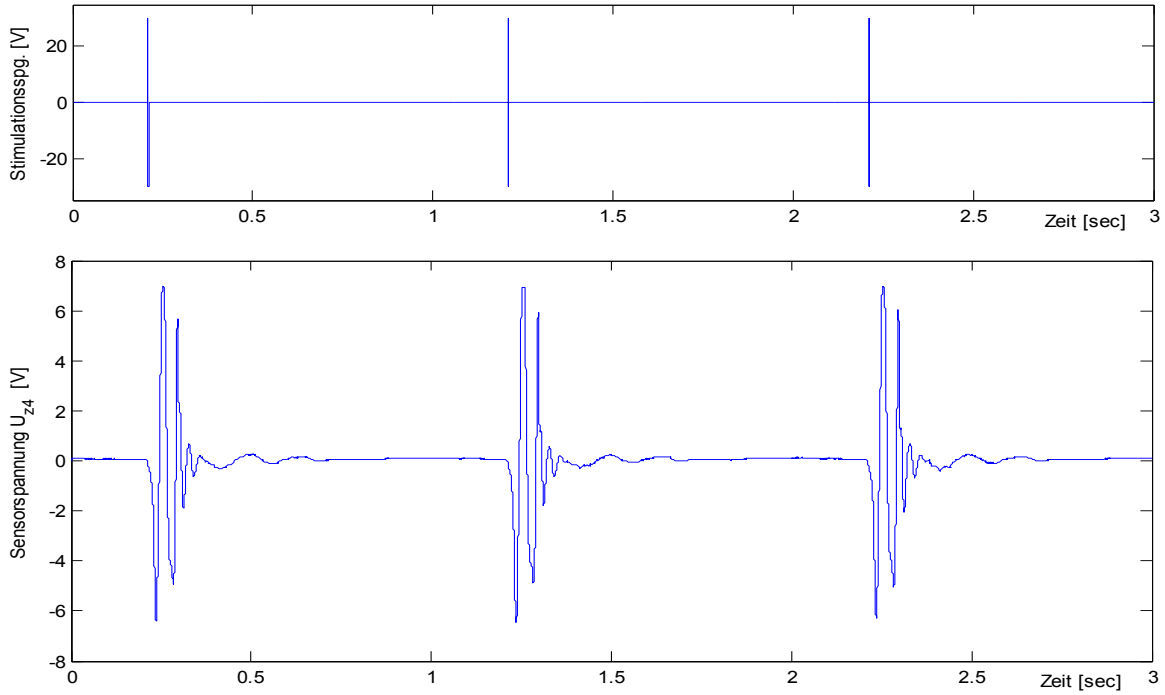


Abbildung 4.4: Twitchmessung

4.4. Messung bei verschieden Stimulationsfrequenzen

4.4.1. Vergleich der Rohdaten

Die kommenden Abbildungen zeigen die Rohdaten der Beschleunigung in Abhängigkeit der Stimulationsfrequenz, wobei wieder der Betrag der Sensorsignale (Gleichung 4.1) gebildet wird. In dem dargestellten Zeitbereich ist die Stimulationsspannung konstant. Die Messung wurde bei fünf verschiedenen Stimulationsfrequenzen durchgeführt - 10 Hz, 30 Hz, 60 Hz, 100 Hz und 150 Hz.

Das Phänomen der „FES - Wave“ wird ab der Stimulationsfrequenz von 100 Hz in den Daten sichtbar und nimmt bei dem Versuch von 150 Hz amplitudenmäßig zu. In den Beschleunigungssignalen bei einer Stimulationsfrequenz von 10 Hz und 30 Hz erkennt man deutlich die Muskelzuckungen aufgrund der Stimulationsimpulse.

Bei der vorliegenden Messung befindet sich der Quadriceps bei 60 Hz in einer stabilen Kontraktion - fusioniert und ohne FES - Wave. Eine wichtige Anmerkung ist, dass FES - Wave Phänomen wurde auch bei Stimulationsfrequenzen unter 60 Hz beobachtet, jedoch muss die Amplitude der Stimulationsspannung in einen größeren Wertebereich liegen.

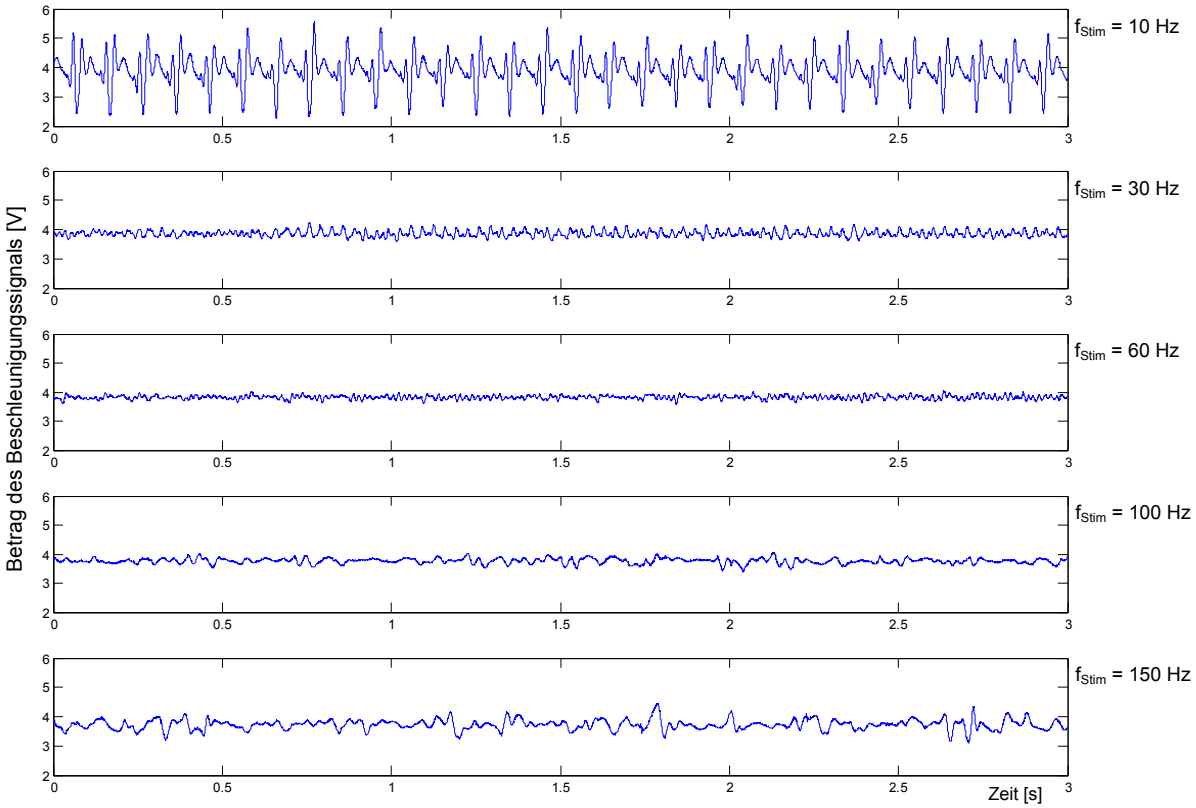


Abbildung 4.5: Vergleich Sensor 3 (M. rectus femoris)

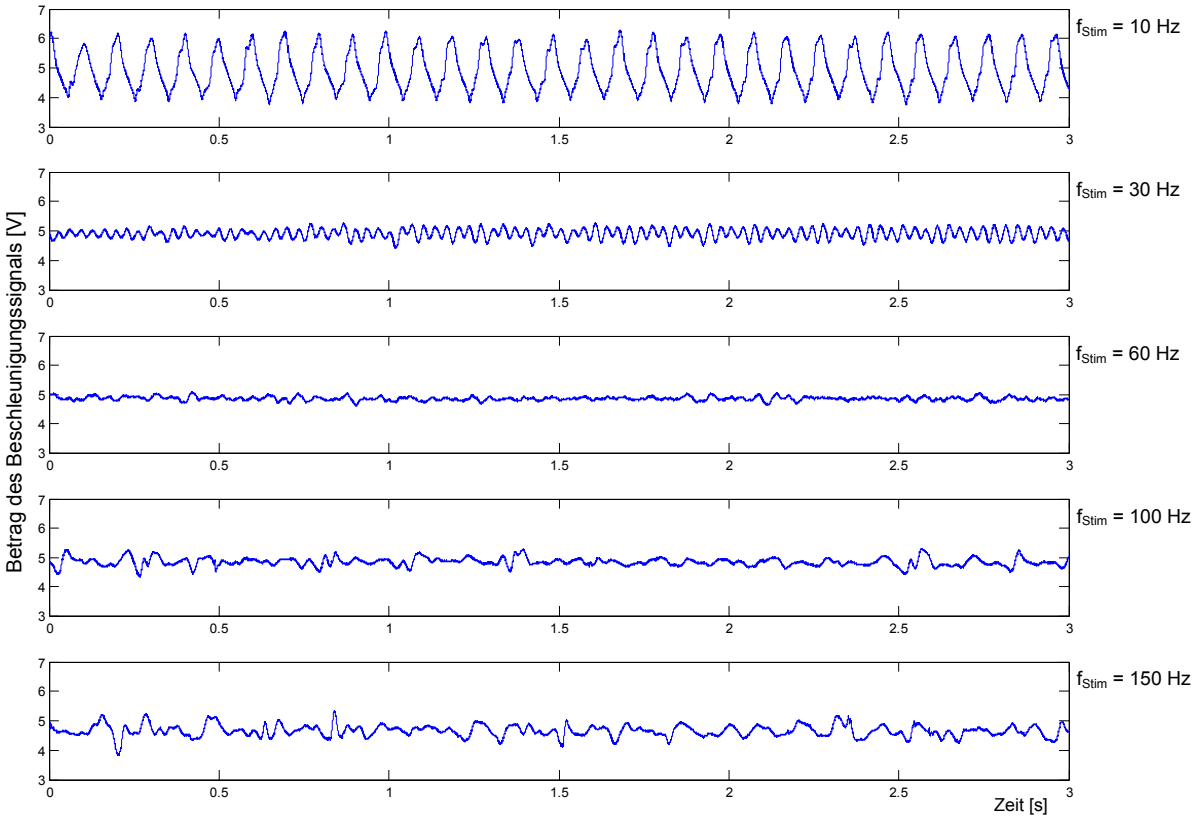


Abbildung 4.6: Vergleich Sensor 4 (M. vastus medialis)

4.4.2. Berechnung der Positionsänderungen

Aus der Beschleunigung wird durch zweimaliges Integrieren die Wegdifferenz berechnet. Das Blockschaltbild beschreibt den Rechengang der in MATLAB implementiert worden ist.

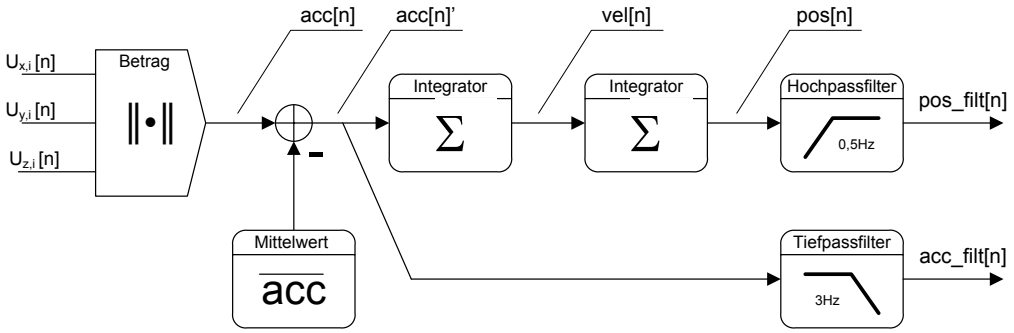


Abbildung 4.7: Blockschaltbild - Berechnung der Positionsänderung

In den folgenden Abbildungen werden verschiedene Eigenschaften der Signale *add[n]*, *pos_filt[n]* und *acc_filt[n*] dargestellt. Der Betrag der Beschleunigungssignale (Gleichung 4.1) wird im ersten Schritt von dem Mittelwert befreit, damit Integrationsrampen vermieden werden.

Hochpassfilterung des Positionssignals (pos_filt[n]): Neben dem Mittelwert, der im ersten Schritt kompensiert worden ist, stört auch das Kippen des Beschleunigungssensor das Messergebnis. Durch diesen Vorgang enthält das zweifach Integrierte Signal eine langsame Schwingung dessen Frequenzbereich bei 0,1 Hz bis 0,3 Hz liegt und auch betragsmäßig weit über dem Nutzsignal liegt. Der Hochpass filtert diese langsamen Frequenzen aus dem Signal.

Tiefpassfilterung von Beschleunigungssignals (acc_filt[n]): Die Oberflächenschwingung die von dem Phänomen der „FES - Wave“ hervorgerufen werden, befindet sich in einem sehr niedrigen Frequenzbereich. Aus diesem Grund betrachtet man auch die Beschleunigungssignale in diesem Bereich um bessere Aussage treffen zu können.

Mittelwert:

$$\overline{acc} = \frac{1}{N} \cdot \sum_{j=1}^{N} acc[j] \qquad \textbf{4.2}$$

Integrator:

$$vel[n] = k_1 \cdot \sum_{j=1}^{n} acc[j]$$

$$pos[n] = k_2 \cdot \sum_{j=1}^{n} vel[j] \qquad \textbf{4.3}$$

MATLAB - Programm

```
  1 : close all;
  2 : clear acc vel pos acc_filt vel_filt pos_filt mag
  3 : % load raw10.mat
  4 : % load raw30.mat
  5 : % load raw60.mat
  6 : % load raw100.mat
  7 : load raw150.mat  % öffnen der Rohdaten-Matrix
  8 :
  9 : data = raw150;
 10 : filename = 'figures/aap_150Hz_';
 11 :
 12 : n = 1:30000;       % EINSTELLEN DER ZEITSPANNE
 13 : time = n * 10^-4;
 14 : index = n + 155000;
 15 :
 16 : %---------------<1> Generierung der Berchnungsvariablen
 17 : acc = zeros(length(n),5);
 18 : vel = zeros(length(n),5);
 19 : pos = zeros(length(n),5);
 20 : acc_filt = zeros(length(n),5);
 21 : vel_filt = zeros(length(n),5);
 22 : pos_filt = zeros(length(n),5);
 23 :
 24 : for i = 1:5
 25 :
 26 :     %---------------<2> Berechnung des Betrags
 27 :     mag(:,i) = sqrt( data(:,i*3-1).*data(:,i*3-1) + ...
 28 :                      data(:,i*3  ).*data(:,i*3  ) + ...
 29 :                      data(:,i*3+1).*data(:,i*3+1)      )
 30 :
 31 :     %---------------<3> acc - mittelwertfrei
 32 :     acc(:,i) = mag(index,i) - mean(mag(index,i));
 33 :
 34 :     %---------------<4> Berchnung von vel durch aufsummieren von acc
 35 :     vel(1,i) = acc(1,i);
 36 :     for j = 2:length(acc)
 37 :         vel(j,i) = vel(j-1,i) + acc(j,i);
 38 :     end
 39 :
 40 :     %---------------<5> Berchnung von pos ... aufsummieren von vel
 41 :     pos(1,i) = vel(1,i);
 42 :     for j = 2:length(n)
 43 :         pos(j,i) = pos(j-1,i) + vel(j,i);
 44 :     end
 45 :
 46 :     %---------------<6> Tiefpassfilterung des Beschleunigungssiganl
 47 :     [b,a] = butter(4,3*2/10000);
 48 :     acc_filt(:,i)=filter(b,a,acc(:,i));
 49 :     acc_filt(:,i)=acc_filt(:,i) - mean(acc_filt(:,i));
 50 :
 51 :     %---------------<7> Hochpassfilterung des Signals pos und vel
 52 :     [b,a] = butter(2,0.5*2/10000,'high');
 53 :     vel_filt(:,i)=filtfilt(b,a,vel(:,i));
 54 :     pos_filt(:,i)=filtfilt(b,a,pos(:,i))/(0.6083*10^6);
 55 :
 56 : end
 57 :
 58 : %---------------<8> FFT Berechnung und Ausgabe
 59 : wertebereich = [0 5 0 0.12;0 5 0 0.20;0 5 0 0.12;0 5 0 0.20;0 5 0 0.12];
 60 : for i = 1:5
 61 :     samplerate=10000;
 62 :     t_anf=1/samplerate;
 63 :     t_end=length(n)/samplerate;
 64 :
 65 :     %---------------<8.1> FFT des Beschleunigungssignals
 66 :     Zeit_axis=[t_anf,t_end,-1,1];
 67 :     Frq_axis=[0,60,0,0.0085];
 68 :     fk_FFT_plot(acc(:,i), t_anf,t_end, samplerate,Zeit_axis,Frq_axis,0.2);
 69 :
 70 :     %---------------<8.2> FFT der Position
 71 :     Zeit_axis=[t_anf,t_end,-1,1];
 72 :     Frq_axis = wertebereich(i,:);
 73 :     fk_FFT_plot(pos_filt(:,i), t_anf,t_end, samplerate,Zeit_axis,Frq_axis,0.8);
 74 : end
 75 : %---------------<9>  Speichern der FFT-Graphen
 84 : %---------------<10> Ausgabe der Daten acc, acc_filt, pos_filt
101 : %---------------<11> Speichern der Graphen
```

Betrachtung im Zeitbereich

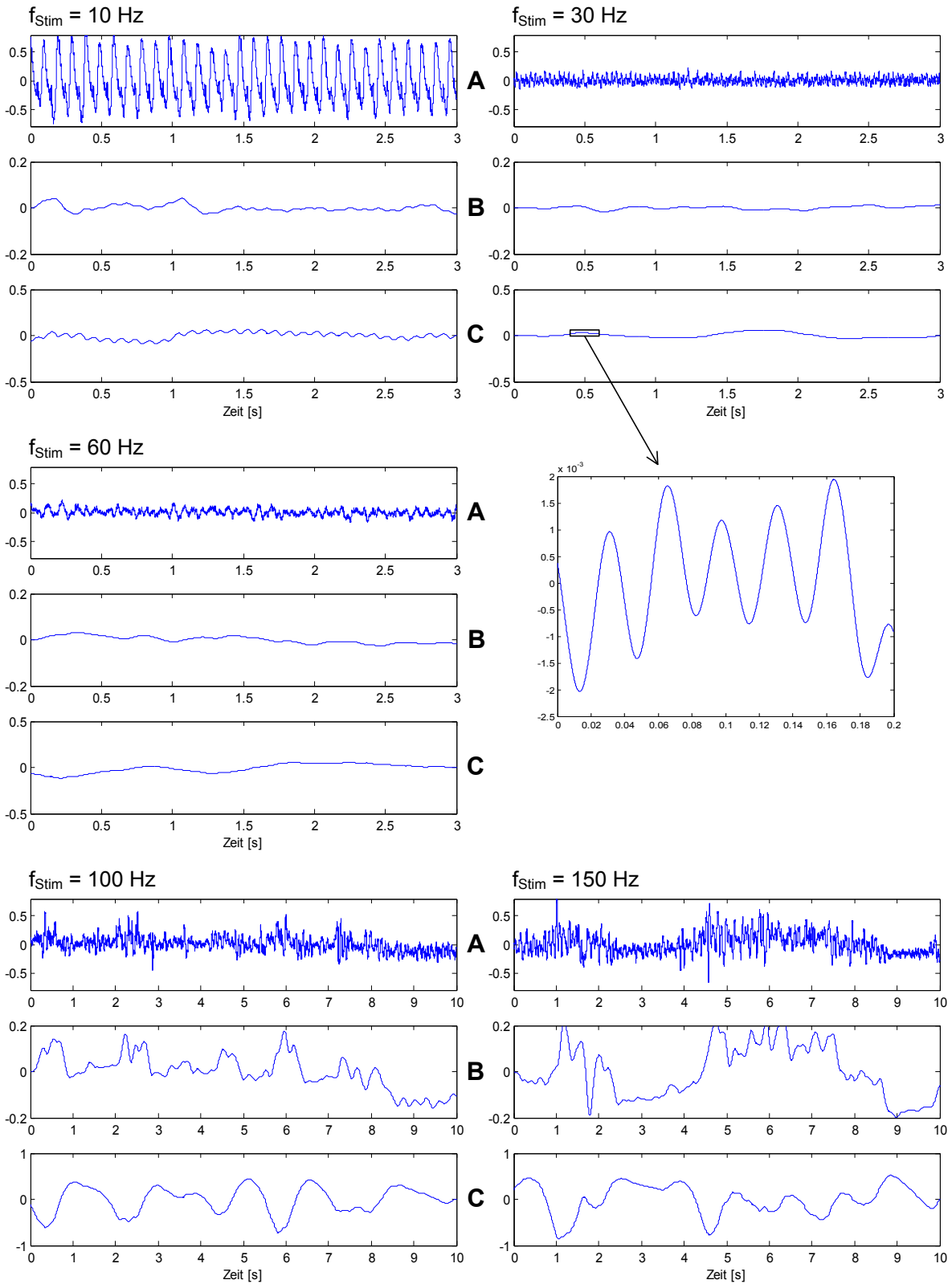


Abbildung 4.8: Beschleunigungssensor 1: Beschleunigungs- und Positionssignal
A - Betrag der Beschleunigung ohne Filterung
B - Betrag der Beschleunigung nach der Tiefpassfilterung (f_c = 3 Hz)
C - Berechnete Positionsänderung (vgl. Gleichung 4.3)

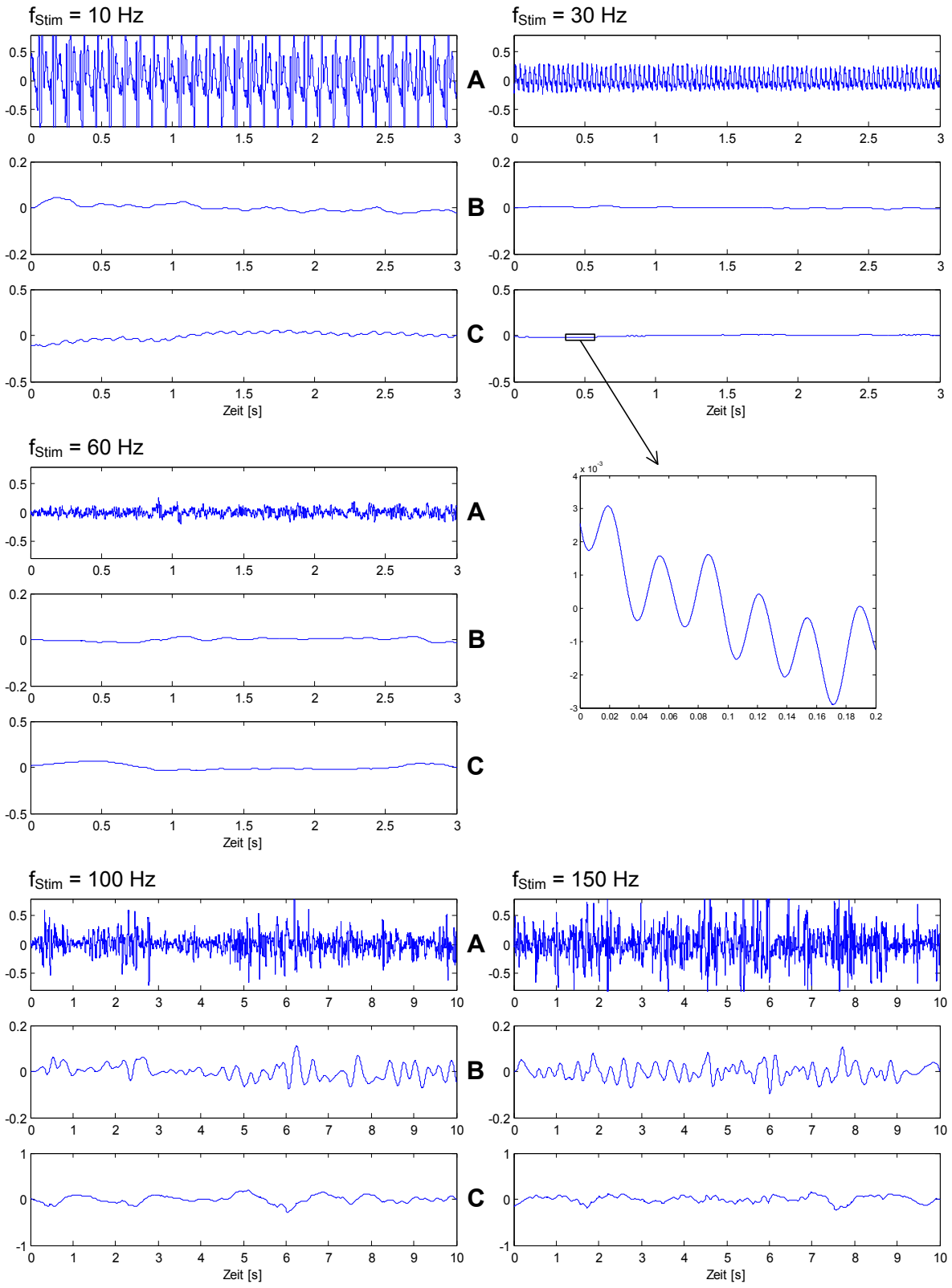


Abbildung 4.9: Beschleunigungssensor 3: Beschleunigungs- und Positionssignal
A - Betrag der Beschleunigung ohne Filterung
B - Betrag der Beschleunigung nach der Tiefpassfilterung (f_c = 3 Hz)
C - Berechnete Positionsänderung (vgl. Gleichung 4.3)

Betrachtung im Frequenzbereich

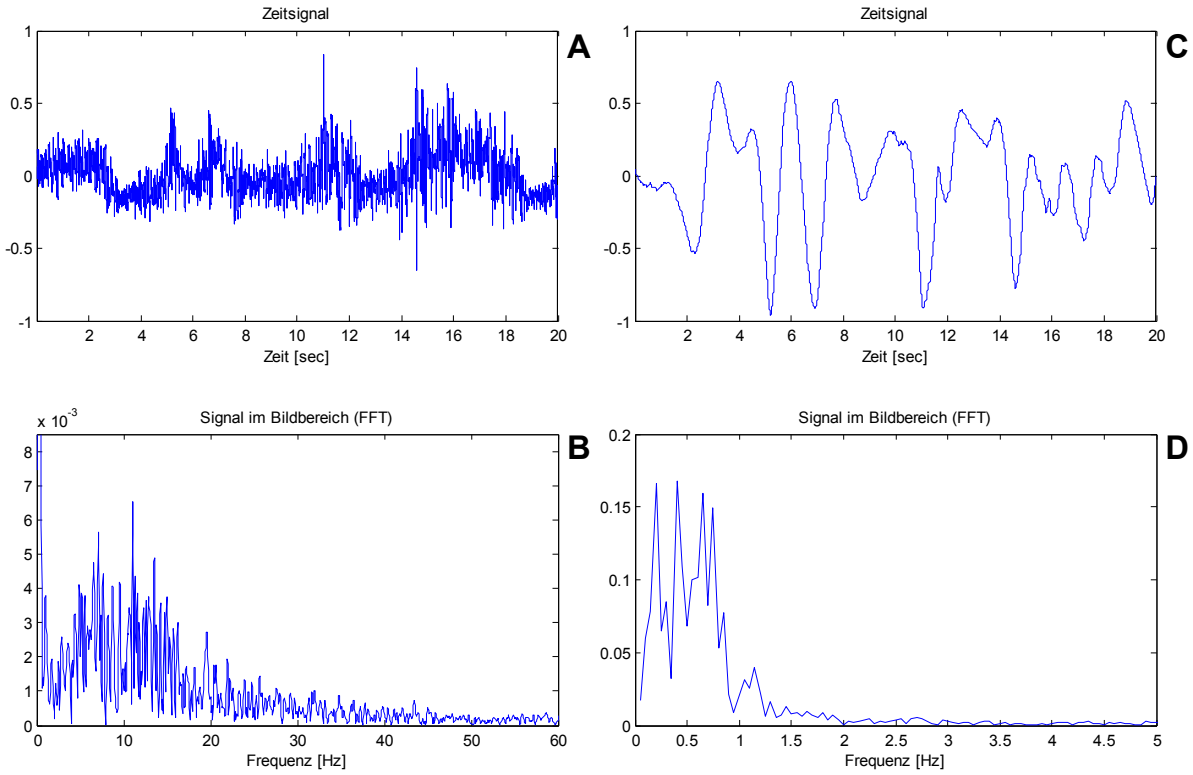


Abbildung 4.10: FFT - Berechnung des Sensor 1 (M. vastus lateralis) bei f_{stim} = 150 Hz
A,B … Betrag der Beschleunigung, C,D … Positionsänderung

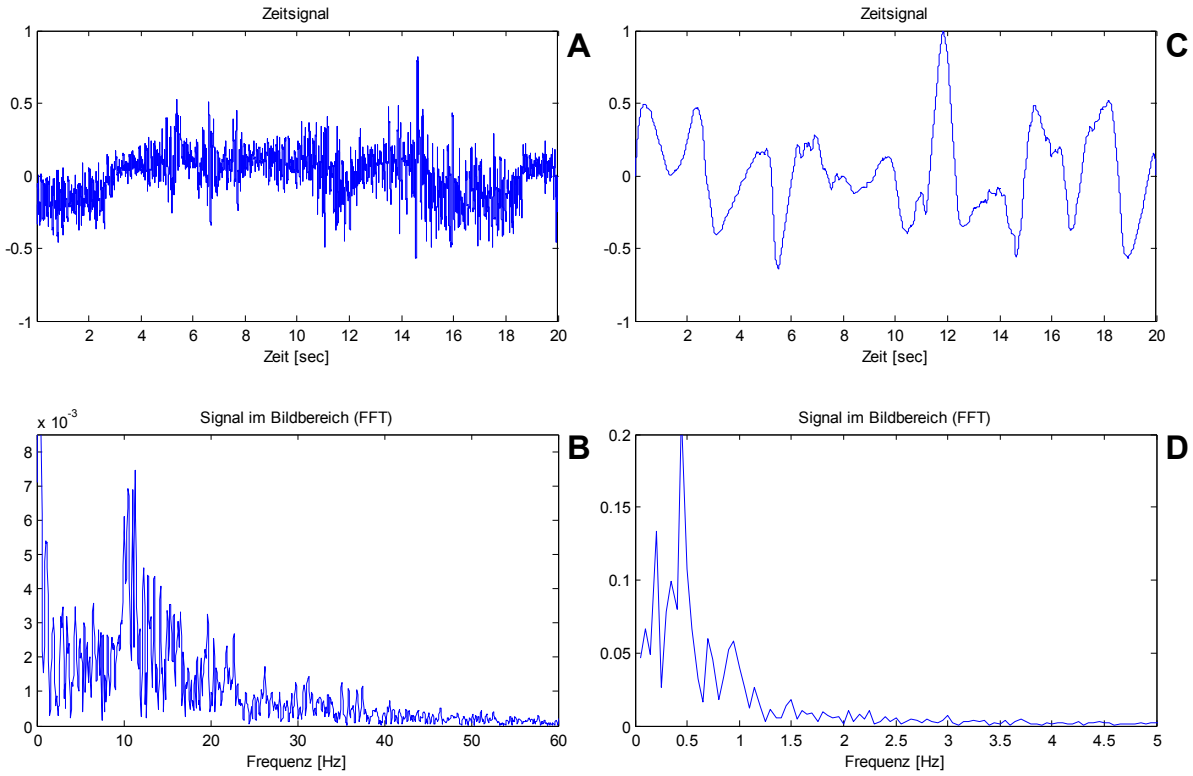


Abbildung 4.11: FFT - Berechnung des Sensor 2 (M. vastus lateralis) bei f_{stim} = 150 Hz
A,B … Betrag der Beschleunigung, C,D … Positionsänderung

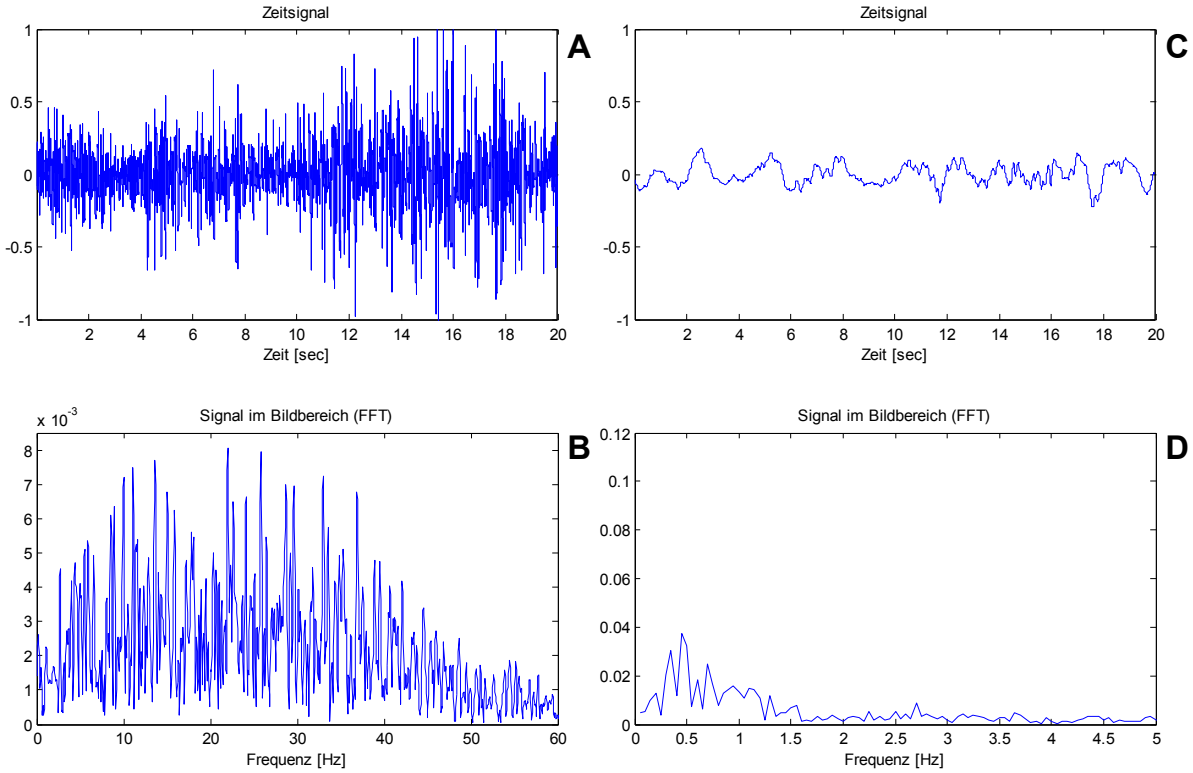


Abbildung 4.12: FFT - Berechnung des Sensor 3 (M. rectus femoris) bei f_{stim} = 150 Hz
A,B … Betrag der Beschleunigung, C,D … Positionsänderung

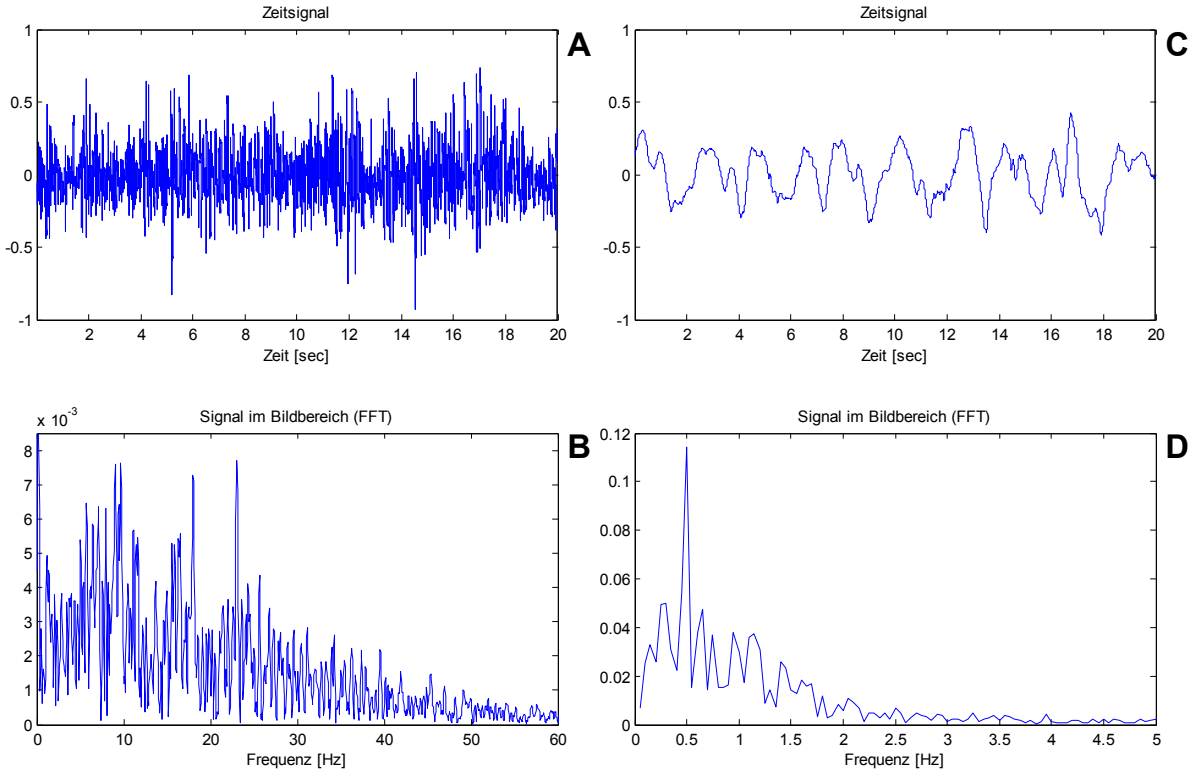


Abbildung 4.13: FFT - Berechnung des Sensor 4 (M. vastus medialis) bei f_{stim} = 150 Hz
A,B … Betrag der Beschleunigung, C,D … Positionsänderung

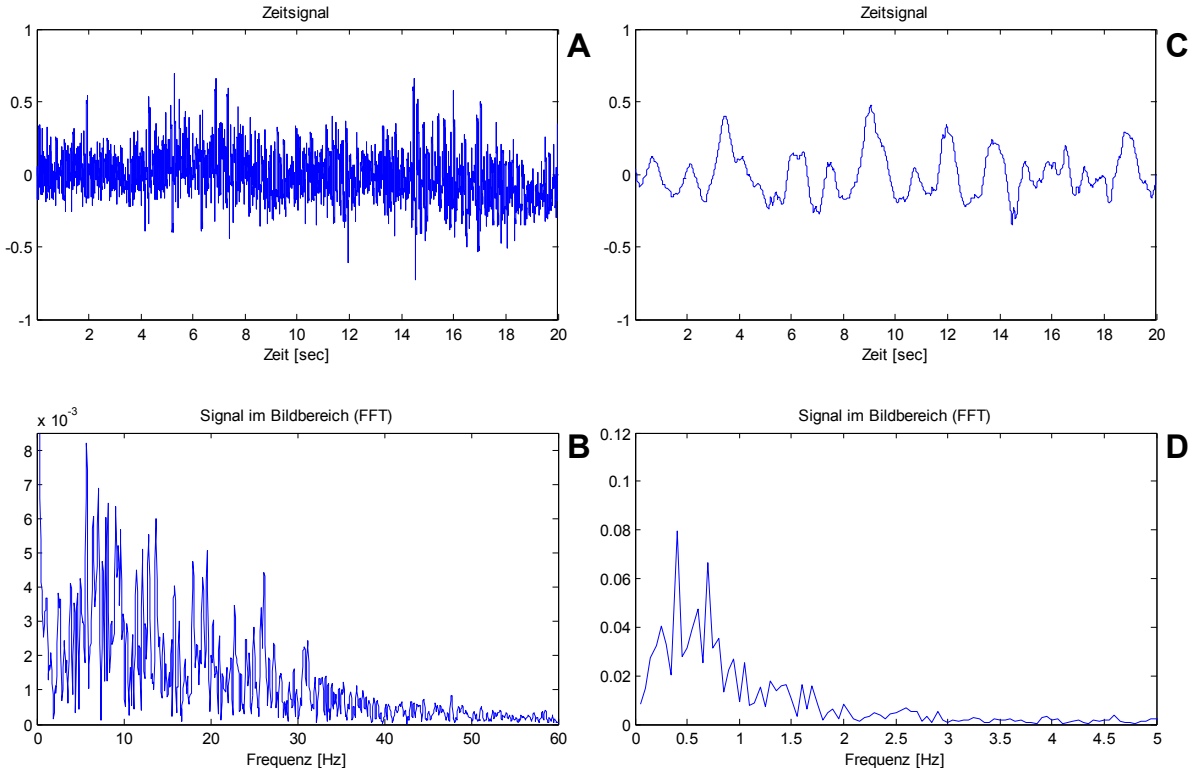


Abbildung 4.14: FFT - Berechnung des Sensor 5 (M. vastus medialis) bei f_{stim} = 150 Hz
A,B … Betrag der Beschleunigung, C,D … Positionsänderung

4.4.4. Beschleunigungssignal - Sensor 1 / Sensor 3

Die folgenden beiden Graphen zeigen den Vergleich der Beschleunigungssignale von zwei Sensoren. Der Sensor 3 ist auf dem Muskelbauch des M. rectus femoris angebracht und der Sensor 1 steht im Zusammenhang mit dem Muskelbauch des M. vastus lateralis.

Die Stimulationseinstellungen (Amplitude, Impulsdauer, Frequenz) wurden so gewählt, dass augenscheinliche das Phänomen „FES - Wave“ am stärksten auftritt. Die beiden Graphen zeigen, dass bei dieser Schwelle der M. rectus femoris stark kontrahiert und kaum eine Relativbewegung ausübt. Im Gegensatz dazu steht der M. vastus lateralis, der eine starke Eigendynamik aufweist. Dieses Ergebnis könnte verwendet werden um das Phänomen der „FES - Wave“ zu Klassifizierung.

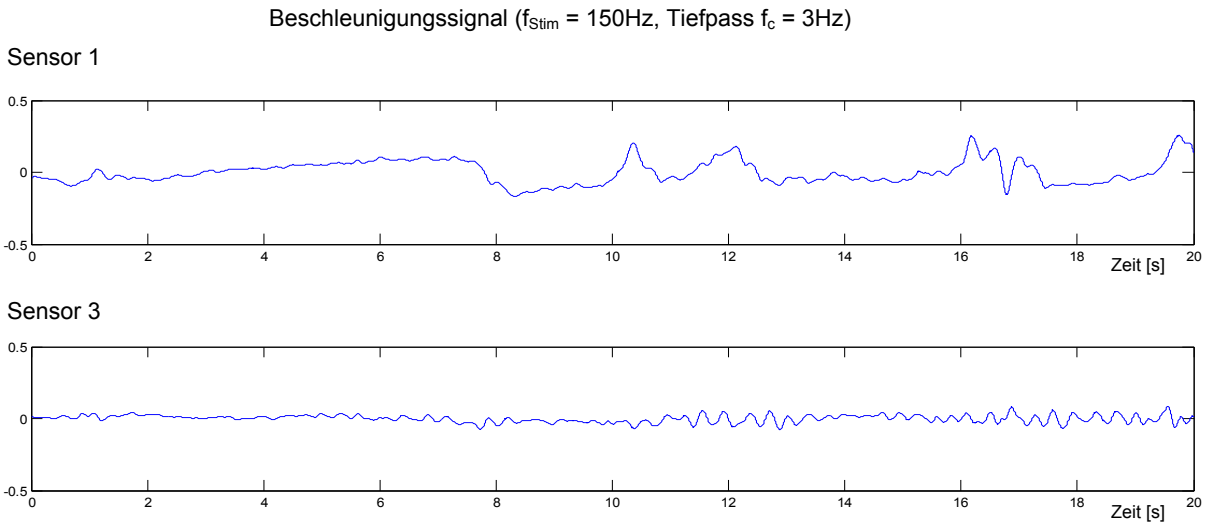


Abbildung 4.15: Tiefpassgefiltertes Sensorsignal

4.5 Ausblick

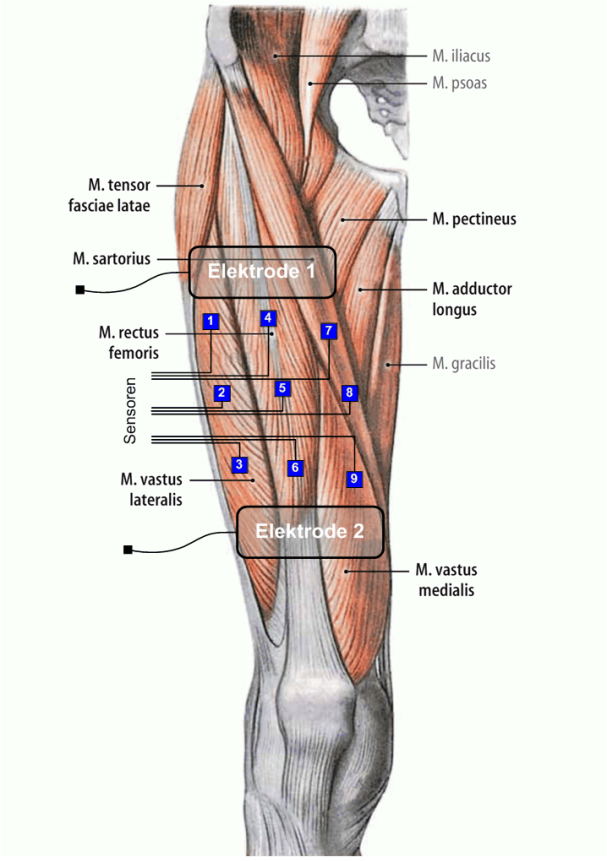


Abbildung 4.16: **Messanordnung mit neun Sensoren**
Quelle der anatomischen Abbildung: (Tillmann, 2005 S. 472)

Überlegungen zur Messung der FES - Wave:

- Erweiterung auf neun Sensoren (siehe Abbildung)
- Einbeziehung eines Kraftsensor
- Algorithmus, der einen Zusammenhang der einzelnen Sensoren auf einem Muskelbauch zeigt.
- Messung mit verschiedenen Elektrodengrößen (in Bezug auf die Motorpoints)
- Weitere Untersuchungen der FES - Wave weiterer Muskelgruppen, z. B. M. biceps brachii
- Auswirkung der Stimulation im Bereich der FES - Wave auf andere physiologische Parameter, z. B. M - Wave.
- Auswirkung auf die Ermüdung der Muskulatur

Literaturverzeichnis

Albert, Thierry, et al. 2000. Anatomic Motor Point Localization for Partial Quadriceps Block in Spasticity. *American Congress of Rehabilitation Medicine and the American Academy of Phyical Medicine and Rehabilitation.* 2000, S. 285-287.

Bernmark, E. und Wiktorin, B. 2002. A triaxial accelerometer for measuring arm movements. *Applied Ergonomics.* 2002, 33(6), S. 541-547.

Bierl, Lutz. 2004. *Das große MSP430 Praxisbuch.* Poing : Franzis Verlag, 2004. ISBN 3-7723-4299-X.

Bijak, M., et al. 2002. The Vienna functional electrical stimulation system for restoration of walking functions in spastic paraplegia. *Artificial organs.* 2002, 9(1), S. 319-322.

Burbaum, Christoph. 1991. *Herstellung von mikromechanischen Beschleunigungssensoren in LIGA-Technik.* Dissertation : Universität Karlsruhe, 1991.

Cescon, C., et al. 2004. Effect of accelerometer location on mechanomyogram variables during voluntary constant-force contractions in three human muscles. *Medical and Biological Engineering and Computing.* 2004, 42(1), S. 121-127.

Chen, K. Y. und Bassett, D. R.Jr. 2005. The technology of accelerometry-based activity monitors: current and future. *Medicine and Science in Sports and Exercise.* 2005, 37(11).

Culhane, K. M., et al. 2005. Accelerometers in rehabilitation medicine for older adults. *Journal of Age Ageing.* 2005, 34(6), S. 556-560.

Datenblatt HWCN139. 2006. Avago Technologies. [Online] 01. Dezember 2006. [Zitat vom: 01. März 2007.] http://www.avagotech.com/assets/downloadDocument.do?id=1005.

Datenblatt LIS3L02AQ3. 2004. ST Microelectronics. [Online] November 2004. [Zitat vom: 20. Februar 2007.] http://www.st.com/LIS3L02AQ3.pdf.

Datenblatt MMA7260Q. 2005. Freescale Semiconductor. [Online] Juni 2005. [Zitat vom: 20. Februar 2007.] http://www.freescale.com/mma7260q.pdf.

Datenblatt MSP430F1611. 2006. Texas Instruments. [Online] August 2006. [Zitat vom: 01. März 2007.] http://focus.ti.com/lit/ds/symlink/msp430f1611.pdf.

Durfee, William K. und MacLean, Karon E. 1989. Methods for Estimating Isometric Recruitment Curves of Electrically Stimulated Muscle. *IEEE Transactions on Biomedical Engineering.* 1989, Vol.36, S. 654-666.

Fraden, Jacob. 1993. *AIP Handbook of Modern Sensors.* New York : American Institute of Physics, 1993. ISBN 1-56396-108-3.

Freilinger, G. und Mayr, W. 2002. Electrical stimulation as a countermeasure to muscle alteration in space. *Journal of gravitational physiology.* 2002, 9(1), S. 319-322.

Gallasch, E., et al. 1993. Akzelerometrische Messsysteme in der Bioomedizin, Anforderungen, Probleme und Perspektiven. *Jahrestagung der deutschen,östrerr. und schweiz. Biomedizinischen Gesellschaft.* 1993, Ergänzungsband 38, S. 333-334.

Gallasch, E., et al. 1994. Phyiological tremor and control of limb position in 1g and 0g. *Journal of Gravitational Physiology.* 1994, 1(1), S. 52-54.

Gerrits, H.L., et al. 2002. Effects of Training on Contractile Properties of Paralyzed Quadriceps Muscle. *Muscle & Nerve.* 2002, 25, S. 559-567.

Glück, Markus. 2005. *MEMS in der Mikrosystemtechnik.* Wiesbaden : B.G.Teubner Verlag, 2005. ISBN 3-519-00520-4.

Hallett, M. 1998. Overview of human tremor physiology. *Movement Disorders.* 1998, 13(3), S. 43-48.

Hugh, H. S. Liu und Pang, K.H. 2001. Accelerometer for Mobile Robot Positioning. *IEEE Transactions on Industry Applications.* 2001, Bd. Vol.37, No.3, S. 812-819.

Jaakko, Malmivuo. 1995. *Bioelectromagnetism, Principles and Applications of Bioelectric and Biomagnetic Fields.* New York : Oxford University Press, 1995. ISBN 0-19-505823-2.

MacIntosh, Brian R. und Willis, Janine C. 2000. Force-frequency relationship and potentiation in mammalian skeletal muscle. *Journal of Applied Physiology.* 2000, 88, S. 2088-2096.

Mathie, M. J., et al. 2004. Accelerometry: providing an integrated, practical method for long-term, ambulatory monitoring of human movement. *Physiological Measurement.* 2004, 25(2).

Mayr, W., et al. 2001. Basic design and construction of the Vienna FES implants: existing solutions and prospects for new generations of implants. *Medical engineering & physics.* 2001, 23(1), S. 53-60.

Merill, Daniel R., Bikson, Marom and Jefferys, John G.R. 2005. Electrical stimulation of excitable tissue: design of efficacious and safe protocols. *Journal of Neuroscience Methods.* 2005, Vol. 141.

Miller, Aaron und Ford, Jerry Lee. 2006. *Microsoft Visual C++ 2005, Express Edition.* Boston : Thomson Course Technology, 2006. ISBN 1-59200-816-X.

MSP430, User's Guide. 2006. Texas Instruments. [Online] Juni 2006. [Zitat vom: 01. März 2007.] http://focus.ti.com/lit/ug/slau049f/slau049f.pdf.

Müller, Helmut und Walz, Lothar. 2005. *Mikroprozessortechnik.* Würzburg : Vogel Buchverlag, 2005. ISBN 3-8343-3023-9.

Nagy, Chris. 2003. *Embedded Systems Design unsing the TI MSP430 Series.* Boston : Elsevier, 2003. ISBN0-7506-7623-X.

Popovic, Milos R. und Keller, Thierry. 2005. Modular transcutaneous functional electrical stimulation system. *Medical Engineering & Physics.* 2005, 27, S. 81-92.

Quittan, M., et al. 2001. Improvement of thigh muscles by neuromuscular electrical stimulation in patients with refractory heart failure: a single-blind, randomized, controlled trial. *American journal of physical medicine and rehabilitation.* 2001, 80(3), S. 215-216.

Rafolt, D. und Gallasch, E. 2002. Surface myomechanical responses recorded on a scanner galvanometer. *Medical and Biological Engineering and Comuting.* 2002, Vol. 40, S. 594-599.

Rushton, D. N. 1997. Functional electrical stimulation, Topical Review. *Physiological Measurement.* 1997, 18, S. 241-275.

Schmidt, Wolf-Dieter. 2002. *Sensorschaltungstechnik.* 2.Aufl. Würzburg : Vogel Verlag, 2002. ISBN 3-8023-1896-X.

Schumann, Jörg und Dietel, Jürgen. 2006. C++/CLI. [Online] 10. November 2006. [Zitat vom: 02. April 2007.] http://www.rz.rwth-aachen.de/mata/dienste/seminar_dv/upload/Schumann_8555_skript.pdf.

Shieh, J., et al. 2001. The selection of sensors. *Progress in Materials Science.* 2001, 46, S. 461-504.

Siegl, Johann. 2005. *Schaltungstechnik - Analog und gemischt analog/digital.* 2.Aufl. Heidelberg : Springer-Verlag, 2005. ISBN 3-540-44230-8.

Skibo, Craig, Young, Marc Young und Johnson, Brian. 2006. *Arbeiten mit Microsoft Visual Studio 2005.* s.l. : Microsoft Press, 2006. ISBN 3-86645-400-7.

Stein, Richard B., Momose, Kimito und Bobet, Jacques. 1999. Biomechanics of human quadriceps muscles during electrical stimulation. *Journal of Biomechanics.* 1999, 32, S. 347-357.

Tillmann, Bernhard N. 2005. *Atlas der Anatomie des Menschen.* Berlin Heidelberg : Springer-Verlag, 2005. ISBN 3-540-66651-6.

Trost, S. G., McIver, K. L. und Pate, R. R. 2005. Conducting accelerometer-based activity assessments in field-based research. *Medicine and Science in Sports and Exercise.* 2005, 37(11), S. 531-543.

Underwood, Steve. 2003. Manual - MSPGCC. [Online] 27. November 2003. [Zitat vom: 01. März 2007.] http://mspgcc.sourceforge.net.

W., Mayr. 2001. *Functional Electrical Stimulation (FES) as a Countermeasure against Muscular Atrophy in Long-term Spaceflights - First Application on Board of MIR-Station.* Wien : Facultas Verlag, 2001. ISBN 3-85076-572.

Watakabe, M., et al. 2003. Reliability of the mechanomyogram detected with an accelerometer during voluntary contractions. *Medical and Biological Engineering and Computing.* 2003, 41(2), S. 198-202.

Wüstling, Sascha. 1997. *Hochintegriertes triaxiales Beschleunigungssensorsystem.* Dissertation : Universisät Karlsruhe, 1997.

Zagar, T. und Krizaj, D. 2005. Validation of an accelerometer for determination of muscle belly radial displacement. *Medical and biological engineering and computing.* 2005, 43, S. 78-84.

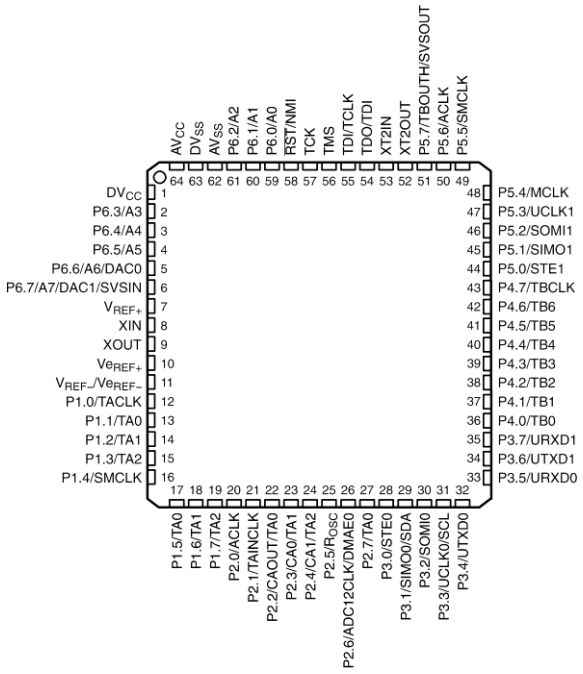
Abbildungsverzeichnis

Tabellenverzeichnis

KAPITEL 5 Anhang

Anhang A: Hardware-Stimulator

Pinbelegung des Mikrokontrollers MSP430F1611



1 : DV_{cc}
2 : ADC - Eingang f. Amplitudeneinstellung Ch2
3 : NC[1)]
4 : NC
5 : DAC - Ausgang f. Amplitudenspannung Ch1
6 : DAC - Ausgang f. Amplitudenspannung Ch2
7 : Referenzspannung für ADC-Konvertierung
8 : NC
9 : NC
10 : NC
11 : NC
12 : Ausgang für LED Ansteuerung
13 : NC
14 : Eingang Auswahlschalter - Taster
15 : NC
16 : NC
17 : NC
18 : NC
19 : Eingang für externer Trigger
20 : Eingang Auswahlschalter - Drehschalter A
21 : Eingang Auswahlschalter - Drehschalter A
22 : Eingang Auswahlschalter - Drehschalter B
23 : Eingang Auswahlschalter - Drehschalter B
24 : NC
25 : NC
26 : NC
27 : NC
28 : NC
29 : Ausgang - positiver Rechteckimpuls Ch1
30 : Ausgang - negativer Rechteckimpuls Ch1
31 : Ausgang - positiver Rechteckimpuls Ch2
32 : Ausgang - negativer Rechteckimpuls Ch2
33 : NC
34 : NC (optional Trigger Ausgang)
35 : UART1 - Empfangsleitung (RX1)
36 : UART1 - Sendeleitung (TX1)
37 : NC
38 : LCD - ENA
39 : LCD - RS
40 : LCD - D4
41 : LCD - D5
42 : LCD - D6
43 : LCD - D7
44 : NC
45 : NC
46 : NC
47 : NC
48 : NC
49 : NC
50 : NC
51 : NC
52 : XT2OUT - 8MHz
53 : XT1IN - 8MHz
54 : JTAG - TDO
55 : JTAG - TDI
56 : JTAG - TMS
57 : JTAG - TCK
58 : JTAG - RSTNMI
59 : ADC0 (wird nicht verwendet)
60 : ADC - Eingang f. Amplitudeneinstellung Ch1
61 : ADC2 (wird nicht verwendet)
62 : AV_{ss}
63 : DV_{ss}
64 : AV_{cc}

Quelle: (Datenblatt MSP430F1611, 2006)

[1)] NC … nicht verbunden (not connected)

Pinbelegung der Schnittstellen

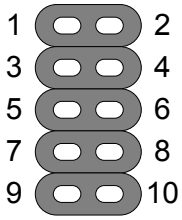


Abbildung 5.1: 10-polige Steckerleiste

1 :	positiver Rechteckimpuls (low aktiv)
2 :	+5 V - Versorgungsspannung
3 :	negativer Rechteck impuls (low aktive)
4 :	Amplitudenwert in einem Bereich von 0 bis 5 V
5 :	kompinierter Rechteckimpulse (high aktive)
6 :	NC
7 :	+ U_B … Versorgung direkt von dem Akkublock
8 :	+ U_B … Versorgung direkt von dem Akkublock
9 :	GND
10 :	GND

Tabelle 5.1: Pinbelegung Schnittstelle <1>, <2> (siehe Abbildung 2.6)

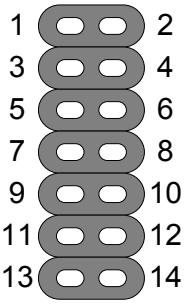


Abbildung 5.2: 14-polige Steckerleiste

1 :	1. Kanal - positiver Rechteckimpuls (high aktiv)
2 :	1. Kanal - negativer Rechteckimpuls (hight aktive)
3 :	2. Kanal - positiver Rechteckimpuls (high aktiv)
4 :	2. Kanal - negativer Rechteckimpuls (hight aktive)
5 :	DAC0 - Ausgang: Amplitudenwert für Kanal 1
6 :	DAC1 - Ausgang: Amplitudenwert für Kanal 2
7 :	ADC1 - Eingang:
8 :	ADC3 - Eingang:
9 :	+5 V - Versorgung für serielle Kommunikation zum PC
10 :	-5 V - Versorgung: Spannungsregler auf der Hautplatine
11 :	+ U_B … Versorgung direkt von dem Akkublock
12 :	+ U_B … Versorgung direkt von dem Akkublock
13 :	GND
14 :	GND

Tabelle 5.2: Pinbelegung Schnittstelle <3> (siehe Abbildung 2.6)

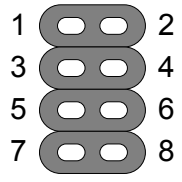


Abbildung 5.3: 8-polige Steckerleiste

1 :	TDO
2 :	TDI
3 :	TMS
4 :	TCK
5 :	RSTNMI
6 :	DV_{cc}
7 :	DV_{cc}
8 :	DV_{cc}

Tabelle 5.3: Pinbelegung Schnittstelle <4> (siehe Abbildung 2.6)

Layout: Platine zur Pegelanpassung

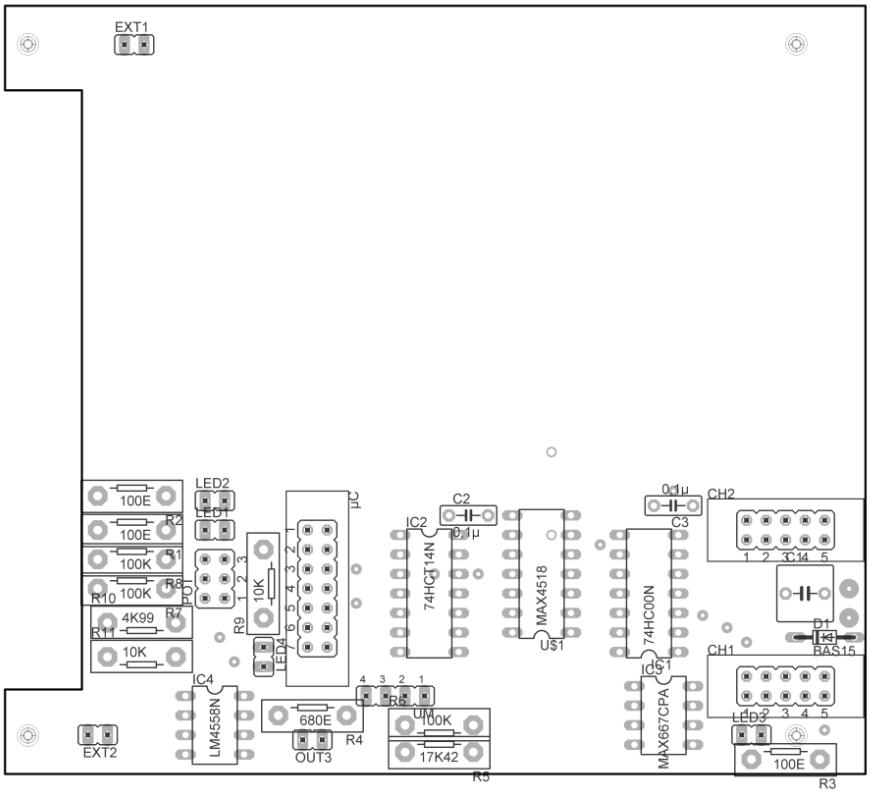


Abbildung 5.4: Bestückungsplan - Pegelanpassungsplatine

Anhang B: Beschleunigungssensoren

LIS3L02AQ3

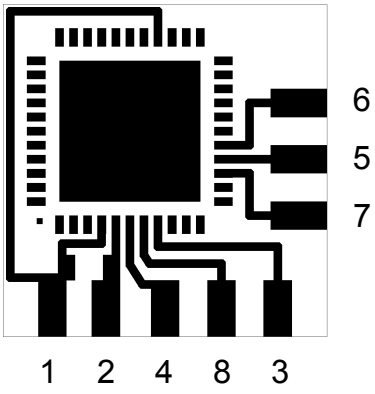


Abbildung 5.5: Adapterplatine LIS3L02AQ

1:	GND	blau	Masse
2:	VDD	grün	Versorgungsspg. 3.3V
3:	V_{X_OUT}	grau	Ausgangsspg. X-Richtung
4:	V_{Y_OUT}	braun	Ausgangsspg. Y-Richtung
5:	V_{Z_OUT}	gelb	Ausgangsspg. Z-Richtung
6:	FS	rosa	g-Bereich (0: ±2g | 1: ±6g)
7:	PD	NC[1)]	Power Down (intern auf 0)
8:	ST	NC	Selbsttest (intern auf 0)

(Datenblatt LIS3L02AQ3, 2004)

MMA7260Q

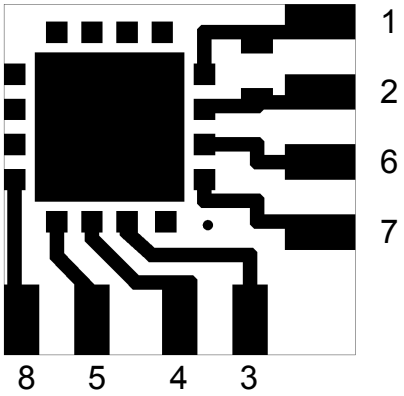


Abbildung 5.6: Adapterplatine MMA7260Q

1:	GND	blau	Masse
2:	VDD	rot	Versorgungsspg. 3.3V
3:	V_{X_OUT}	braun	Ausgangsspg. X-Richtung
4:	V_{Y_OUT}	grün	Ausgangsspg. Y-Richtung
5:	V_{Z_OUT}	gelb	Ausgangsspg. Z-Richtung
6:	G2	grau	Setzen des g – Levels : G2 (logischer Eingang)
7:	G1	rosa	Setzen des g – Levels: G1 (logischer Eingang)
8:	SP	NC	Sleep Mode – mit VDD verbunden (Drahtbrücke)

(Datenblatt MMA7260Q, 2005)

Pinbelegung – Redelstecker/-Buchse:

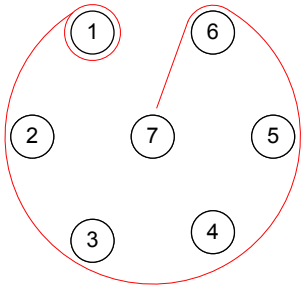


Abbildung 5.7: Redelkontakt

no.	Farbschema der Kabellitzen		Beschreibung
	MMA7260Q	LIS3L02Q	
1 :	blau	blau	VDD (3,3V)
2 :	rot	grün	GND
3 :	braun	grau	V_X_out
4 :	grün	braun	V_Y_out
5 :	gelb	gelb	V_Z_out
6 :	grau	rosa	(LIS3 L02Q) Full Scale (MMA7260Q) G2
7 :	rosa	weiß	(LIS3 L02Q) NC (MMA7260Q) G1

[1] NC … not connected (nicht verbunden)

Anhang C: Hardware Filter - Array

Pinbelegung Ni-DAQ Stecke

Eingangsspannung	NiDAQ	Pin	Eingangsspannung	NiDAQ	Pin
U_{X1} / U_{X6}	AI0	68	U_{X5} / U_{X10}	AI12	61
U_{Y1} / U_{Y6}	AI1	33	U_{Y5} / U_{Y10}	AI13	26
U_{Z1} / U_{Z6}	AI2	65	U_{Z5} / U_{Z10}	AI14	58
U_{X2} / U_{X7}	AI3	30	Takt	AI15	23
U_{Y2} / U_{Y7}	AI4	28	GND	AGND	32
U_{Z2} / U_{Z7}	AI5	60	GND	DGND	50
U_{X3} / U_{X8}	AI6	25	Takt (optional)	P0.0	52
U_{Y3} / U_{Y8}	AI7	57			
U_{Z3} / U_{Z8}	AI8	34			
U_{X4} / U_{X9}	AI9	66			
U_{Y4} / U_{Y9}	AI10	31			
U_{Z4} / U_{Z9}	AI11	63			

Layout: Filter/Multiplexer - Platine

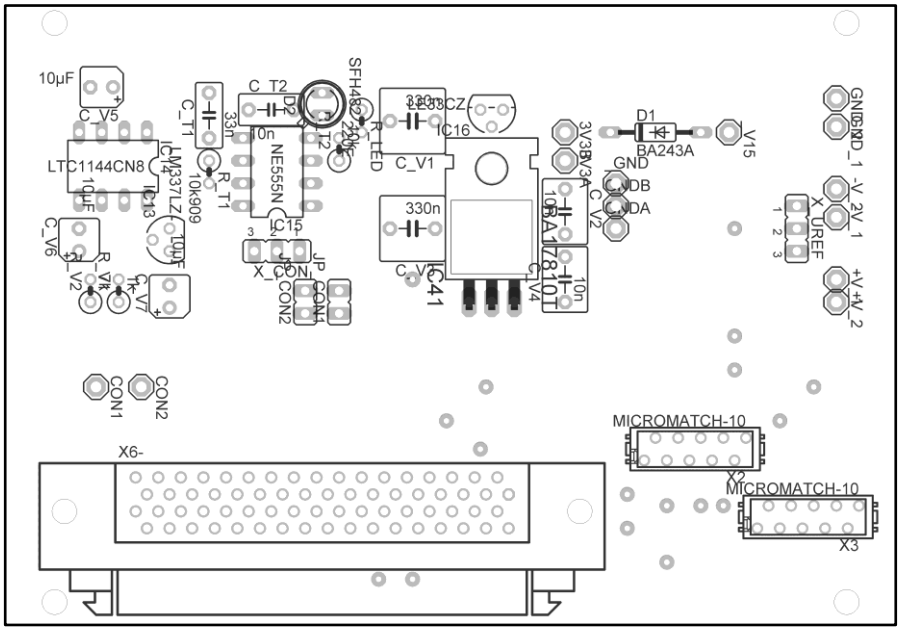


Abbildung 5.8: Bestückungsplan – Versorungteil des Filter Array‘s

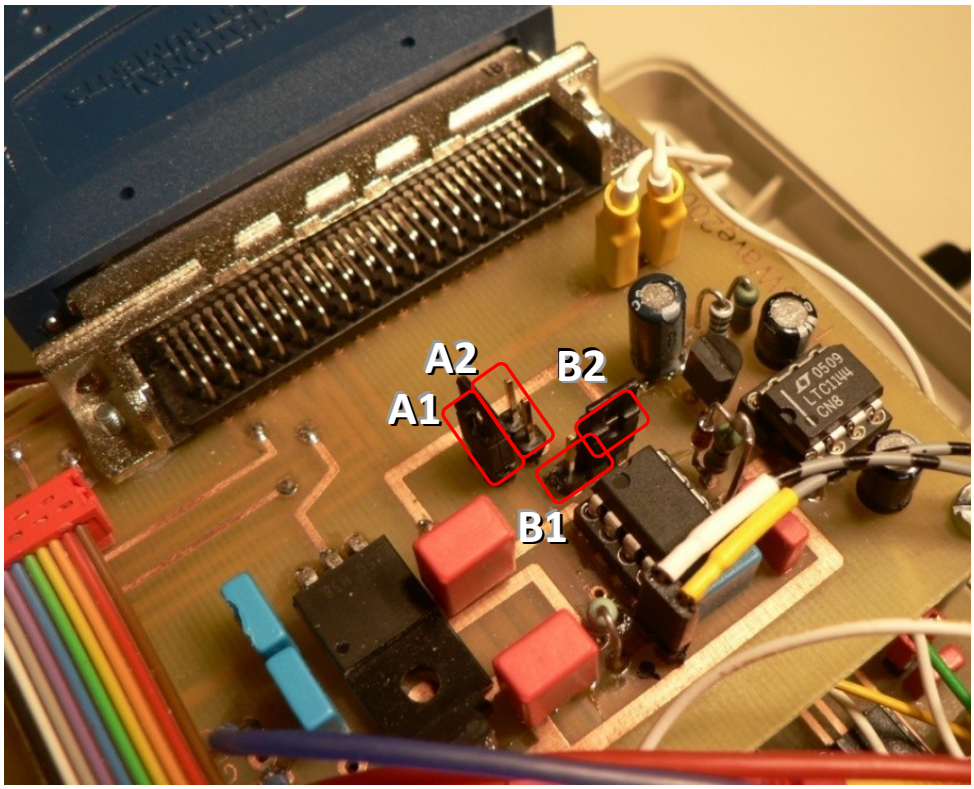


Abbildung 5.9: Foto – Filter-Array: Jumperstellungen

Jumperfunktion:

A1 – 2 kHz Taktrate
A2 – Auswahl einer fixen Filterbank
↳ **B1** – obere Filterbank
↳ **B2** – untere Filterbank

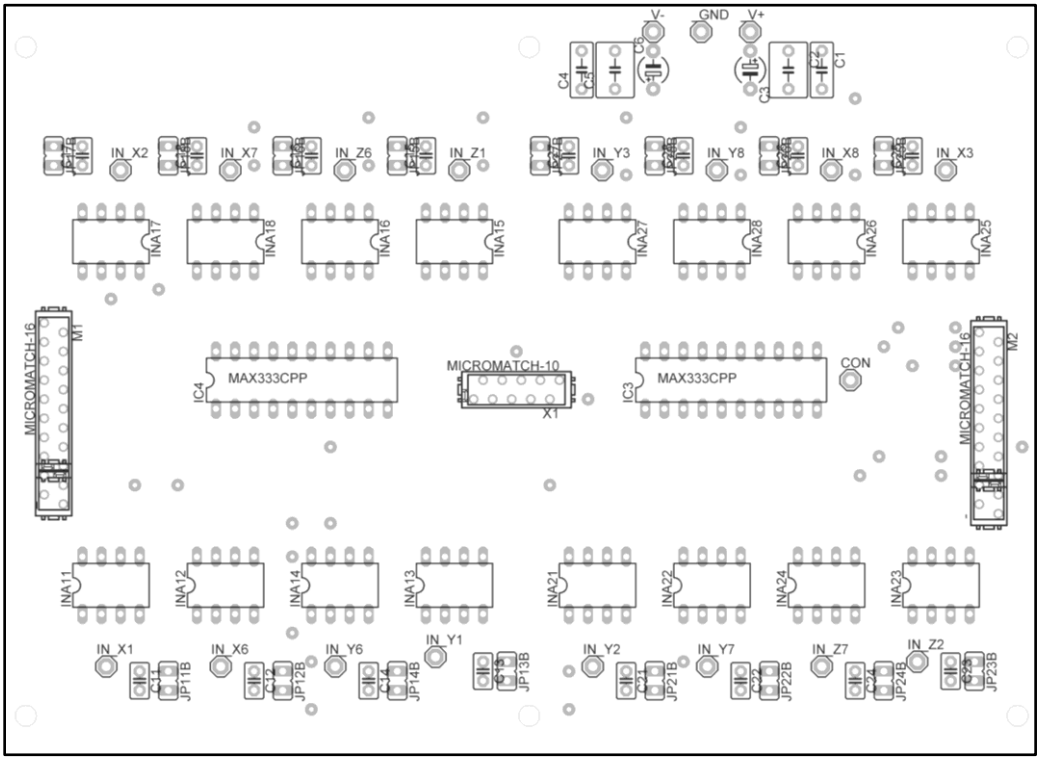


Abbildung 5.10: Bestückungsplan 32/16 - Tiefpass Array

Verstärkung : R_G - Widerstände

An den Steckerleisten M1 und M2 liegen die R_G - Ausgänge der Instrumenten-verstärker, deren Beschaltung in der folgenden Abbildung gezeigt wird. Obwohl die Stecker 20-polig sind, werden nur die oberen 16-Pins verwendet, welche in dem Bestückungsplan mit Micromatch-16 gekennzeichnet sind.

Vertärkung	R_G
1	NC
2	50,00 kΩ
5	12,50 kΩ
10	5,55 kΩ
20	2,63 kΩ
50	1,02 kΩ
100	505,20 Ω
500	100,20 Ω
1000	50,05 Ω
5000	10,00 Ω
10000	5,00 Ω

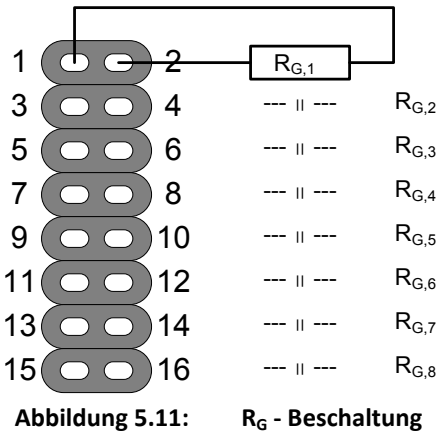


Abbildung 5.11: R_G - Beschaltung

Tabelle 5.4: R_G - Widerstandsliste