Steuerung einer Handprothese mit einem Mikrocontroller

Bachelor-Arbeit

zur Erlangung des akademischen Grades

Bachelor of
Science in Engineering

Eingereicht am

Studiengang Medizintechnik, Linz
Fachhochschul-Studiengänge Oberösterreich

von

Ingo Weigel

Linz, am 4. Mai 2015

Begutachter:

FH-Prof. Dipl.-Ing. Dr. Robert Merwa

Eidesstattliche Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Linz, am 4. Mai 2015

Ingo Weigel

Danksagung

Mein gebührender Dank gilt all jenen, die mich beim Erarbeiten und Verfassen dieser Bachelorarbeit unterstützt haben.

Insbesondere danken möchte ich FH-Prof. Dipl.-Ing. Dr. Robert Merwa, der mich im Projektteam "Controlling und Sensorik einer myoelektrisch gesteuerten Handprothese" betreute und mich in die Grundlagen der Mikrocontroller-Programmierung einführte.

Danke an Viktoria Spielmann und Georg Kehrer, die mich stets motivierten und motivieren, das Studium trotz meiner Erkrankung fortzuführen.

Für das Lektorat bedanke ich mich bei ebenfalls bei Viktoria Spielmann.

Kurzfassung

In dieser Arbeit wird Software für einen ATmega32 Mikrocontroller entwickelt, mit dem eine myoelektrische Handprothese mit Hilfe des Zustandsautomaten einer direkten Proportionalsteuerung gesteuert wird. Hierzu wurde eine modulare Software-Architektur entwickelt. Ein Modul das den Zustandsautomaten abbildet, ein Modul das EMG- und weitere Sensorsignale digitalisiert, und ein Modul das den Motor mittels PWM-Schnittstelle ansteuert werden im Hauptprogramm integriert. Die Module wurden einzeln getestet, wobei alle Tests zufrieden stellen. Die Software ist Echtzeitfähig.

Abstract

This thesis shows the development of software for an ATmega32 microcontroller that controls a myoelectric hand prosthesis with the help of a state machine (direct and proportional control). Therefore, a modular software architecture was developed. Modules, that represent the state machine, perform analog digital conversion of EMG and different sensor signals, and a module that controls the motor with the help of a PWM interface are beeing integrated in the main programm. Each singe module was tested, all tests passed. The software works in realtime.

Executive Summary

Im Rahmen eines Projektstudiums an der FH Oberösterreich soll eine myoelektrische Handprothese entwickelt werden. Diese Bachelorarbeit dokumentiert die Entwicklung von Software für einen *ATmega32* Mikrocontroller, der den Zustandsautmaten einer direkten proportionalen Steuerung betreibt.

Dabei wurde eine modulare Softwarearchitektur mit folgenden Modulen entwickelt:

- pwm: Erzeugung eines PWM-Signals zur Regelung des Motors
- motor: Ansteuerung des Motors in verschiedenen Betriebsmodi
- statmachine: Zustandsautomat einer proportionalen Direktsteuerung
- adc: Digitalisierung von EMG- und anderen Sensorsignalen
- prosthesiscontrol: Integrationsmodul, das die anderen Module verzahnt und die Steuerung im Hauptprogramm betreibt

Die Module wurden zunächst voneinander getrennt entwickelt und getestet. Alle einzelnen Modul-Tests stellten zufrieden.

Auch der abschließende Systemtest fiel positiv aus. Die Software erfüllt alle gestellten Spezifikationen und wurde erfolgreich auf Echtzeitfähigkeit geprüft.

Inhaltsverzeichnis

1 Einleitung **7**
1.1 Problemstellung und Motivation 7
1.2 Ziele und Vorgehensweise 7
1.3 Struktur und Gliederung der Arbeit 9

2 Definition und Spezifikation der Module **10**
2.1 Definition der Module 10
2.2 Spezifikation der Module 11
2.2.1 pwm 11
2.2.2 motor 13
2.2.3 adc 14
2.2.4 statemachine 15
2.2.5 prosthesiscontrol 19

3 Programmierung, Integration und Test der Module **21**
3.1 pwm 21
3.1.1 Auswahl eines PWM Modus 21
3.1.2 Implementierung des Moduls 24
3.1.3 Modultest 27
3.2 motor 29
3.2.1 Implementierung des Moduls 29
3.2.2 Modultest 31
3.3 adc 32
3.3.1 Implementierung des Moduls 32
3.3.2 Modultest 38
3.4 statemachine 40
3.4.1 Implementierung des Moduls 40
3.4.2 Modultest 43
3.5 prosthesiscontrol 46
3.5.1 Implementierung des Moduls 46
3.5.2 Modultest 50

4 Verzeichnisse **52**

1 Einleitung

Handprothesen sind der Versuch der Medizintechnik, eine menschliche Hand zu ersetzen, wenn diese durch einen Unfall oder eine Erkrankung verloren gegangen ist. Heutzutage gelten myoelektrische Handprothesen als Stand der Technik. Diese werden durch Muskelkontraktion angesteuert und besitzen Fremdenergiequellen zur Durchführung von Bewegungen. Große Herausforderungen bei der Entwicklung stellen aber nicht nur das Ersetzen der verloren gegangenen mechanischen Funktionen, sondern auch das optische Nachempfinden der menschlichen Hand und der Tragekomfort der Prothese dar. Die Prothesen sollen zudem geräuscharm und nicht schwerer als die verlorene Hand sein. Auch an Griffpräzision und an ein Feedback, das den verloren gegangenen Tastsinn ersetzt, werden hohe Erwartungen gestellt. Insgesamt sollen die Einschränkungen, die durch eine Amputation der Hand entstehen, möglichst aufgehoben werden.

1.1 Problemstellung und Motivation

Im Studiengang Medizintechnik der FH Oberösterreich soll von Studierenden über mehrere Jahre hinweg eine Handprothese entwickelt werden. Dieses Projekt wurde 2013 begonnen und nun zum ersten Mal an ein Nachfolgeteam übergeben. Um Kompetenzen aufzubauen, wird zunächst die Handprothese "Sensorhand Speed" der Firma Ottobock (Abb. 1.1) in Betrieb genommen.

Die Aufgabe des gegenständlichen Projekts besteht darin, Software zur Steuerung der Handprothese mit einem Mikrocontroller zu entwickeln. Dieser soll aus mit Elektroden erfassten Muskelkontraktion am Stumpf oder an der Schulter eine Ansteuerung der Prothese ableiten und ein entsprechendes Ansteuerungssignal für eine Motorsteuerungsschaltung erzeugen.

1.2 Ziele und Vorgehensweise

Es soll Software für einen Mikrocontroller entwickelt werden. Als Mikrocontroller wird ein *ATmega32* der Firma *Atmel* eingesetzt, der auf einem Entwicklungsboard zur Verfügung steht. Als Programmer wird ein *mySmartUSB MK2* eingesetzt. Als Entwicklungsumgebung wird *Geany* auf *Xubuntu Linux 12.4* (32 bit CPU-Architektur) verwendet. Diese nutzt *avr-gcc* als Compiler und überträgt den ausführbaren Code mit Hilfe des Tools *avrdude* auf den Mikrocontroller. Die makefiles werden stets manuell auf Grundlage einer Vorlage angefertigt. Der Mikrocontroller wird mit einem CPU-Takt von 8 MHz betrieben. Die Programmiersprache ist C.

Der Programmcode wird auf Englisch entwickelt. Dabei wird eine Nummerierung der Entwicklungsstufen des Programmcodes, eine Versionierung, eingeführt.

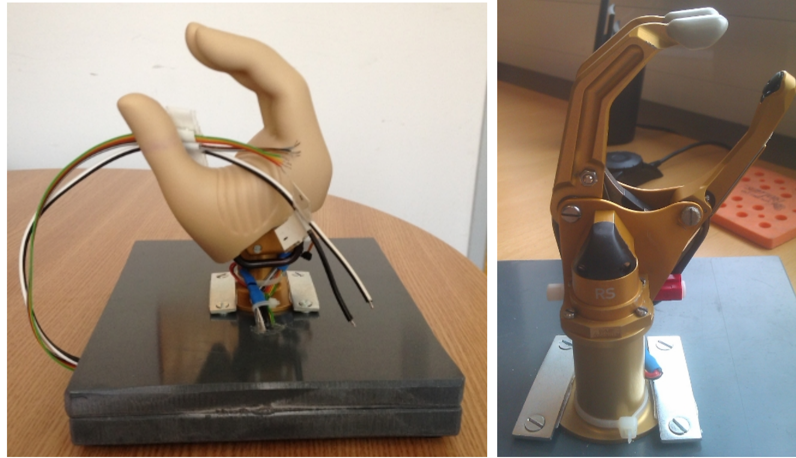


Abbildung 1.1: "Sensorhand Speed" der Firma Ottobock

Als Version 1.0 wird die Version des Programmcodes bezeichnet, die bei Abschluss des gegenständlichen Projekts (nach einem erfolgreichen Systemtest am Menschen) freigegeben wird. Frühere Implementierungen werden von 0.1 beginennd hinter dem Komma nummeriert. Studierende, welche die Software im Anschluß an das gegenständliche Projekt übernehmen und erweitern, können die Versionierung mit 1.1 fortsetzen. In dieser Arbeit wird vordergründig der Code in Version 1.0 behandelt.

Als Literatur wird das Datenblatt von Atmel für den ATmega32 [1] herangezogen. Aus diesem werden die Funktion der Hardware sowie die Bezeichnungen der Register und Bits des Mikrocontrollers entnommen.

Der Mikrocontroller soll ein Elektromyogramm (EMG), welches mit Hilfe zur Verfügung stehender Elektroden von Ottobock bereits analog aufbereitet zur Verfügung steht, digitalisieren.

Mit einem Zustandsautomaten, der auf dem Mikrocontroller implementiert wird, sollen die EMG-Signale ausgewertet und der Motor in der Prothese angesteuert werden. Mit Hilfe des Zustandsautomaten soll eine direkte Proportionalsteuerung als Kontrollstrategie für die Prothese implementiert werden. Das heißt, der Motor soll in Öffnungs- bzw. Schließrichtung proportional zur Intensität der Muskelkontraktion von Agonist bzw. Antagonist geregelt werden. Eine Kontraktion soll eindeutig festgestellt werden, ehe der Motor tatsächlich eingeschaltet wird. Das heißt ihre Intensität soll einerseits einen im Programmcode einstellbaren Schwellwert (**Threshold**) überschreiten und andererseits mindestens eine im Programmcode einstellbare Zeit andauern. Ist keiner der Muskeln kontrahiert, soll der Motor abgeschaltet werden. Der Motor soll auch abgeschaltet werden, wenn ein Gegenstand gegriffen oder die Prothese maximal geöffnet bzw. ganz geschlossen ist.

Als Grundlage zur Implementierung dieser Kontrollstrategie dient der Zustandsautomat einer direkten Proportionalsteuerung, der in der ersten Bachelor-Arbeit entwickelt wurde [2]. Dieser wird für die Sensorhand Speed adaptiert und erweitert.

Der Mikrocontroller soll Signale für die Motoransteuerungsschaltung erzeugen. Eines davon ist ein pulsweitenmoduliertes Signal (**PWM-Signal**), das die Information über die gewünschte Motordrehzahl enthält. Die detaillierte Spezifikation dieser Signale wird von den Entwicklern der Motoransteuerungsschaltung vorgegeben.

Die Signale der Sensoren in der Prothese sollen mit dem Mikrocontroller digital erfasst und auf dem Display ausgegeben werden. Sie werden vorerst noch nicht zur Motorsteuerung herangezogen.

Zur Erfüllung dieser Anforderungen werden die Aufgaben der Software in Bereiche eingeteilt. Dann wird eine Software-Architektur entwickelt, in der jeder Aufgabenbereich von einem **Modul** abgedeckt wird. Die Module werden seperat entwickelt und getestet. Dann werden die Module über eine jeweilige Modul-Schnittstelle (fortan **Interface** genannt) in einer Applikation, die ***prosthesiscontrol*** genannt und als Hauptprogramm vom Mikrocontroller ausgeführt wird, integriert. In der Planung der Software-Entwicklung wird *prosthesiscontrol* ebenfalls als Modul betrachtet, dessen Aufgabe der Betrieb des Zustandsautomaten und damit die zentrale Verzahnung der Module untereinander ist. Der Modultest von *prosthesiscontrol* ist somit zugleich ein Integrationstest, wobei auch die Echtzeitfähigkeit des entwickelten Systems geprüft wird.

Abschließend werden das Resultat und die Ergebnisse der Softwaretests diskutiert. Dabei ist insbesondere auf die Echtzeitfähigkeit der Software einzugehen.

1.3 Struktur und Gliederung der Arbeit

In Kapitel 2 werden die Software-Anforderungen aus Kapitel 1.2 in Aufgabenbereiche eingeteilt. Es werden Module, die jeweils einen Aufgabenbereich abdecken, definiert. Für jedes Modul werden detaillierte Spezifikationen festgelegt.

Dieses Kapitel stellt die Programmierung der Software gemäß der Spezifikationen aus Kapitel 2 dar. Die Module werden für den Mikrocontroller zunächst isoliert entwickelt und dann im Hauptprogramm des Mikrocontrollers integriert. Das Resultat der Implementierung wird anschließend an die Tests diskutiert.

2 Definition und Spezifikation der Module

In diesem Kapitel werden die Software-Anforderungen aus Kapitel 1.2 in Aufgabenbereiche eingeteilt. Es werden Module, die jeweils einen Aufgabenbereich abdecken, definiert. Für jedes Modul werden detaillierte Spezifikationen festgelegt.

2.1 Definition der Module

Die Software-Anforderungen aus Kapitel 1.2 werden in fünf Aufgabenbereiche unterteilt, zu deren Abdeckung jeweils ein Modul definitert wird (Tabelle 2.1).

Bezeichnung des Moduls	**zugeordneter Aufgabenbereich**
pwm	Erzeugung eines PWM-Signals, das die Information über die gewünschte Motordrehzahl enthält
motor	Erzeugung von Signalen zur Motoransteuerung für die Motoransteuerungsplatine
statemachine	Ableitung der gewünschten Bewegung aus dem EMG mit Hilfe eines Zustandsautomaten für eine direkte Proportionalsteuerung
adc	Digitalisierung des EMG, Ansteuerung der Sensoren und Digitalisierung der Sensorsignale, Digitalisierung von Motorspannung und -strom (als Spannung von der Motoransteuerungsplatine zur Verfügung gestellt), Ausgabe der momentanen digitalen Werte auf dem Display
prosthesiscontrol	Betrieb des Zustandsautomaten und Ansteuerung der Hardware gemäß des aktuellen Zustands, stellt das Hauptprogramm (Applikation, die mit dem Mikrocontroller ausgeführt wird)

Tabelle 2.1: Definition von Software-Modulen und Zuordnung von Aufgaben

Die Module besitzen Interfaces, über die sie miteinander interagieren können. Bei der Integration der Module nimmt das Modul *prosthesiscontrol* eine zentrale Rolle ein, denn es stellt das Hauptprogramm des Mikrocontrollers, welches den Zustandsautomaten betreibt und die Hardware gemäß des aktuellen Zustands ansteuert. Abb.2.1 veranschaulicht diese modulare Software-Architektur und die Interface-Interaktionen der Module. Die Interfaces wären im Allgemeinen jedoch auch für eine

Interaktion der Module ohne die Beteiligung des zentralen Moduls *prosthesiscontrol* geeignet. Das Modul *pwm* interagiert in dieser Software-Architektur ausschließlich mit dem Modul *motor*.

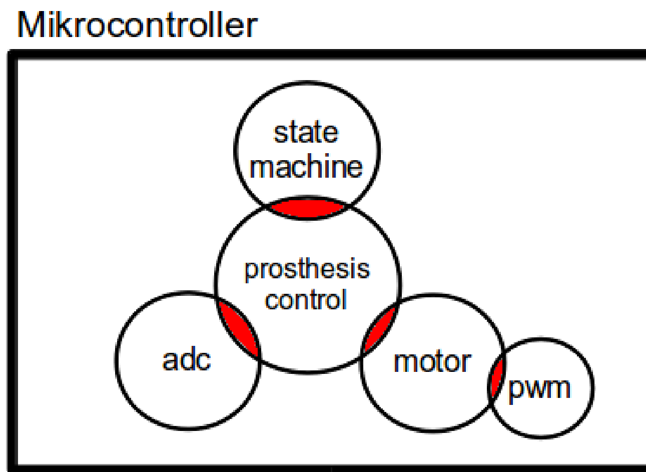


Abbildung 2.1: Software-Architektur mit Modulen (Kreise) und Interaktionen von Modulen über deren Interfaces (rot)

2.2 Spezifikation der Module

2.2.1 pwm

Das Modul *pwm* soll ein Interface bereitstellen, mit dem ein PWM-Signal erzeugt werden kann. Als **Pulsweitenmodulation (PWM)** bezeichnet man eine bestimmte Methode zur Codierung von Informationen in einem zeitkontinuierlichen Signal, das lediglich zwei diskrete (logische) Werte annehmen kann.

Die Form eines PWM-Signals ist in Abbildung 2.2 veranschaulicht.

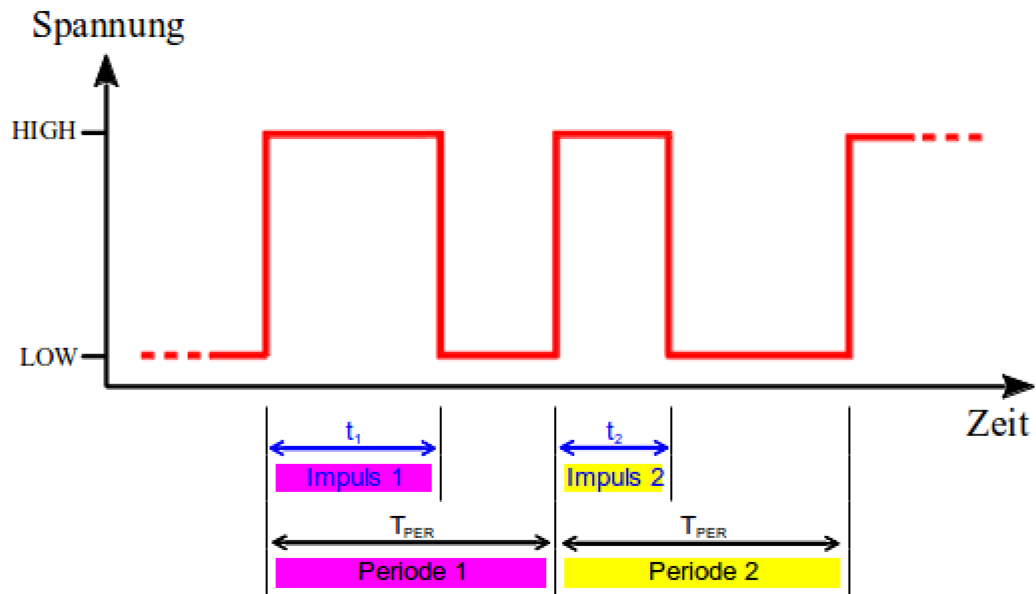


Abbildung 2.2: Form eines PWM-Signals (Duty Cycle 60% in Periode 1 und 40% in Periode 2)

Ein PWM-Signal ist typischerweise ein Rechtecksignal. Es hat eine konstante Frequenz f_{PWM} bzw. konstante Periodendauer T_{PER}. Zu Beginn der Periode i hat das PWM-Signal den logischen Pegel *HIGH*, bis es nach einer variablen Zeit t_i für die verbleibende Dauer der Periode den logischen Pegel *LOW* annimmt. So weist das PWM-Signal zu Beginn einer jeden Periode einen Impuls mit einer (aufgrund der variablen Pulsdauer t_i) von Periode zu Periode variablen Breite auf.

Die Pulsdauer t_i wird variabel in einem Bereich von 0 bis Periodendauer T_{PER} gestaltet. Ordnet man nun den möglichen Werten von t_i jeweils eine Bedeutung zu, so wird das momentane t_i zu einem Träger von Information, die mit Hilfe des PWM-Signals übertragen werden kann.

In der Regel wird t_i auf die Periodendauer T_{PER} bezogen. Die Information wird dabei nicht in der absoluten Pulsdauer t_i, sondern in deren prozentualem Anteil an der Periodendauer T_{PER} codiert. Diese Relation wird als **Tastgrad** D (Formel 2.1) bezeichnet, häufig wird auch der englische Terminus **Duty Cycle** verwendet.

$$D_i = \frac{t_i}{T_{PER}} \tag{2.1}$$

Der momentane Duty Cycle eines PWM-Signals bildet einen Wert in einem Bereich von 0% bis 100% ab.

Das Modul *pwm* soll ein Interface bereitstellen, mit dem ein PWM-Signal erzeugt und an einem Pin des Mikrocontrollers ausgegeben werden kann. Die Frequenz und der Duty Cycle des PWM-Signals sollen über das Interface einstellbar sein. Die Möglichkeit einer Einstellung der PWM-Frequenz über das Interface ist von Seiten der Entwickler der Motoransteuerungsplatine explizit gewünscht, da sich ihre Spezifikation der PWM-Frequenz während der Entwicklung der Platine in einem Bereich von 1 kHz bis 40 kHz verändern kann. Auch soll über das Interface einstellbar sein, ob

das PWM-Signal invertiert oder nicht-invertiert ausgegeben wird.

2.2.2 motor

Das Modul *motor* soll Signale für die Motoransteuerungsplatine des Motors in der Prothese erzeugen. Die Platine besitzt drei Anschlüsse (*PWM, Motor A* und *Motor B*), die vom Mikrocontroller angesteuert werden müssen.

Am Platinen-Anschluss *PWM* soll ein PWM-Signal anliegen, das im Duty Cyle die Information über die gewünschte Motordrehzahl enthält. Das PWM-Signal soll über das Interface des Moduls *pwm* erzeugt werden. Die Frequenz des PWM-Signals wurde von den Entwicklern der Platine vorläufig auf 20 kHz festgelegt. Sie soll jedoch in der Konfiguration des Moduls *motor* in einem Bereich von 1 kHz bis 40 kHz einstellbar sein, da sich die Spezifikation der PWM-Frequenz in der weiteren Entwicklung der Platine verändern kann. Ebenso verhält es sich mit der Festlegung, dass das PWM-Signal invertiert sein soll, weshalb auch diese in der Modul-Konfiguration einstellbar ist.

An den beiden anderen Platinen-Anschlüssen *Motor A* und *Motor B* sollen logische Spannungspegel (*HIGH* und *LOW*) anliegen. Dabei soll in der Konfiguration des Moduls *motor* einstellbar sein, an welchen Mikrocontroller-Pins die Anschlüsse *Motor A* und *Motor B* liegen.

Die Ansteuerung der drei Platinen-Anschlüsse bestimmt das Motorverhalten (siehe Tabelle 2.2).

Motor A	**Motor B**	**Duty Cycle**	**Motorverhalten**
LOW	LOW	0 %	Stand, Leerlauf (kein Bremseffekt)
		100 %	Stand, maximaler Bremseffekt
HIGH	LOW	> 0 %	Motorbetrieb, Hand wird geöffnet
		0 %	Messmodus für Schließrichtung
LOW	HIGH	> 0 %	Motorbetrieb, Hand wird geschlossen
		0 %	Messmodus für Öffnungsrichtung
HIGH	HIGH	beliebig	nicht in Verwendung (Motorkurzschluss)

Tabelle 2.2: Motorverhalten in Abhängigkeit von der Ansteuerung der Motorplatine

Das Motorverhalten *Messmodus* in Tabelle 2.2 bedeutet, dass der Motor nicht mit Spannung versorgt und somit nicht aktiv betrieben wird. Hat der Motor zuvor eine Bewegung ausgeführt, so dreht sich der Rotor aufgrund seiner Trägheit weiter. Dabei induziert er eine zur Drehzahl proportionale Spannung, die vom Modul *adc* digitalisiert wird. Diese Drehzahlmessung ist nur möglich, wenn der Motor nicht mit Spannung versorgt wird. Außerdem müssen *Motor A* und *Motor B* in Abhängigkeit von der Drehrichtung des Rotors angesteuert werden, damit die Platine dem Mikrocontroller die induzierte Spannung bereitstellt. Wenn der Motor eine Bewegung ausführt und nur kurz für die Drehzahlmessung auf das Motorverhalten *Messmodus* umgeschalten wird, so ist die zwischenzeitliche Passivität des Motors in der Praxis nicht bemerkbar.

Das Modul *motor* soll ein Interface bereitstellen, mit dem das Verhalten des Motors in der Handprothese über die Angabe der Parameter *Betriebsmodus* und *Auslastung* eingestellt werden kann (Tabelle 2.3). Es soll die Signale an den Anschlüssen der Motoransteuerungsplatine entsprechend einstellen.

Betriebsmodus	**Auslastung**	**Verhalten des Motors in der Prothese**
MOTOR_STOP	0% - 100%	Hand nicht bewegen, gem. *Auslastung* bremsen
MOTOR_OPEN	0% - 100%	Hand öffnen, Geschwindigkeit gem. *Auslastung*
MOTOR_CLOSE	0% - 100%	Hand schließen, Geschwindigkeit gem. *Auslastung*
MOTOR_MEASURE	0% - 100%	Griffgeschwindigkeit messen, *Auslastung* ignorieren

Tabelle 2.3: Einstellbare Motor-Betriebsmodi

2.2.3 adc

Das Modul *adc* soll eine Analog-Digital-Wandlung (**A/D-Wandlung**) der folgenden Signale vornehmen:

- 2 EMG-Signale
  (durch die Elektroden von *Ottobock* bereits aufbereitete Spannung)
- Signal des Bügelsensors im Handgelenk
  (Spannung von Sensorikplatine, Anschluss *BÜGEL-AUF* [3])
- 3 Signale des Rutschsensors im Daumen
  (Spannung von Sensorikplatine, Anschluss *COM*, Abb. 2.3)
- Motordrehzahl
  (Spannung von Motoransteuerungsplatine, Anschluss *MOTOR-REV*)
- Drehmoment
  (Spannung von Motoransteuerungsplatine, Anschluss *MOTOR-TORQUE*)

Das Interface des Moduls *adc* soll folgende Funktionalitäten bereitstellen:

- Initialisierung des Analog-Digital-Konverters des Mikrocontrollers,
- Durchführung der Abtastung und Digitalisierung der Signale nach Anweisung über das Interface des Moduls *adc* (das Modul *adc* soll eine regelmäßige A/D-Wandlung nicht selbstständig triggern),
- Bereitstellung der digitalisierten Werte zur Verarbeitung durch andere Module,
- Anzeige der digitalisierten Werte auf dem Display.

Der Rutschsensor im Daumen besteht aus drei Dehnungsmesstreifen und besitzt vier Anschlüsse *P0MUX*, *P1MUX*, *P2MUX* und *COM* (Abb. 2.3). Die Dehnungsmesstreifen sind in einer Sternschaltung angeordnet, wobei der Sensor-Anschluss *COM* am Sternpunkt und die Sensor-Anschlüsse *P0MUX*, *P1MUX* und *P2MUX*

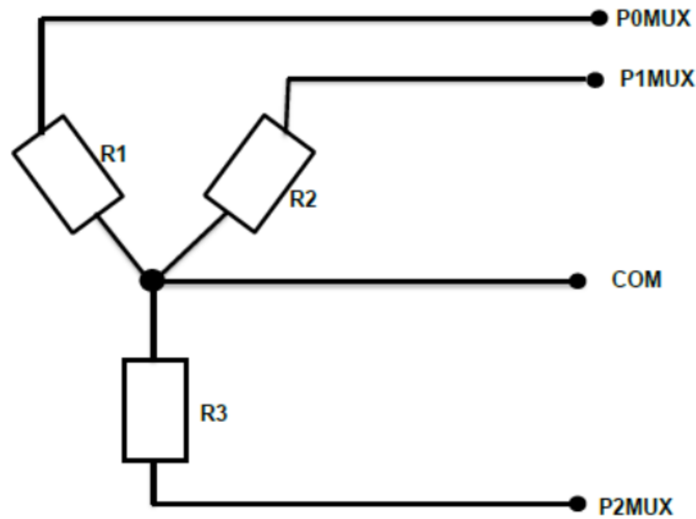


Abbildung 2.3: Aufbau und Anschlüsse des Rutschsensors im Daumen, *R1*, *R2* und *R3* stellen Dehnungsmesstreifen dar
(entnommen aus *Bauer: Sensorschaltung für myoelektrisch gesteuerte Handprothese* [3])

jeweils an einem der äußeren Enden der Sternschaltung liegen. Um einen bestimmten Dehnungsmesstreifen auszuwerten, muss der Mikrocontroller diesen Dehnungsmesstreifen aktiv selektieren, indem er dessen Anschluss *P0MUX*, *P1MUX* bzw. *P2MUX* mit dem logischen Spannungspegel *HIGH* ansteuert. Die beiden anderen *PxMUX*-Anschlüsse müssen dabei auf *LOW* liegen. Der Rutschsensor liefert dann am Anschluss *COM* das Signal des selektierten Dehnungsmesstreifens. Um alle Dehnungsmesstreifen des Rutschsensors auszuwerten, muss der Mikrocontroller also den Sensor auf insgesamt drei verschiedene Weisen ansteuern und dabei jeweils die Spannung am Anschluss *COM* digitalisieren. Daher liefert der Rutschsensor im Daumen in Summe nicht ein, sondern drei Signale, die digitalisiert werden müssen.

Für die Digitalisierung der Motordrehzahl muss der Motor im Messmodus (*MOTOR_MEASURE*, siehe Kapitel 2.2.2) betrieben werden. Das Modul *adc* übernimmt das Umschalten des Motors in den Messmodus nicht. Dieser Vorgang wird durch das Modul *prosthesisControl* vorgenommen, welches Timer/Counter betreibt und mit deren Hilfe das Umschalten in den Messmodus und die anschließende Digitalisierung des Motordrehzahl-Signals zeitlich aufeinander abstimmt. Aufgabe des Moduls *adc* ist es hierbei lediglich, die Spannung am Anschluss *MOTOR-REV* der Motoransteuerungsplatine zu digitalisieren.

2.2.4 statemachine

Das Modul *statemachine* soll die gewünschte Bewegung aus dem EMG mit Hilfe eines Zustandsautomaten für eine direkte Proportionalteuerung ableiten. Es soll ein Interface bieten, das den aktuellen Zustand zur Verfügung stellt und ihn zu Testzwecken auf dem Display anzeigt.

Als Grundlage dient der Zustandsautomat einer direkten Proportionalsteuerung,

der in der ersten Bachelor-Arbeit entworfen wurde [2]. Dieser ist in Abb. 2.4 dargestellt.

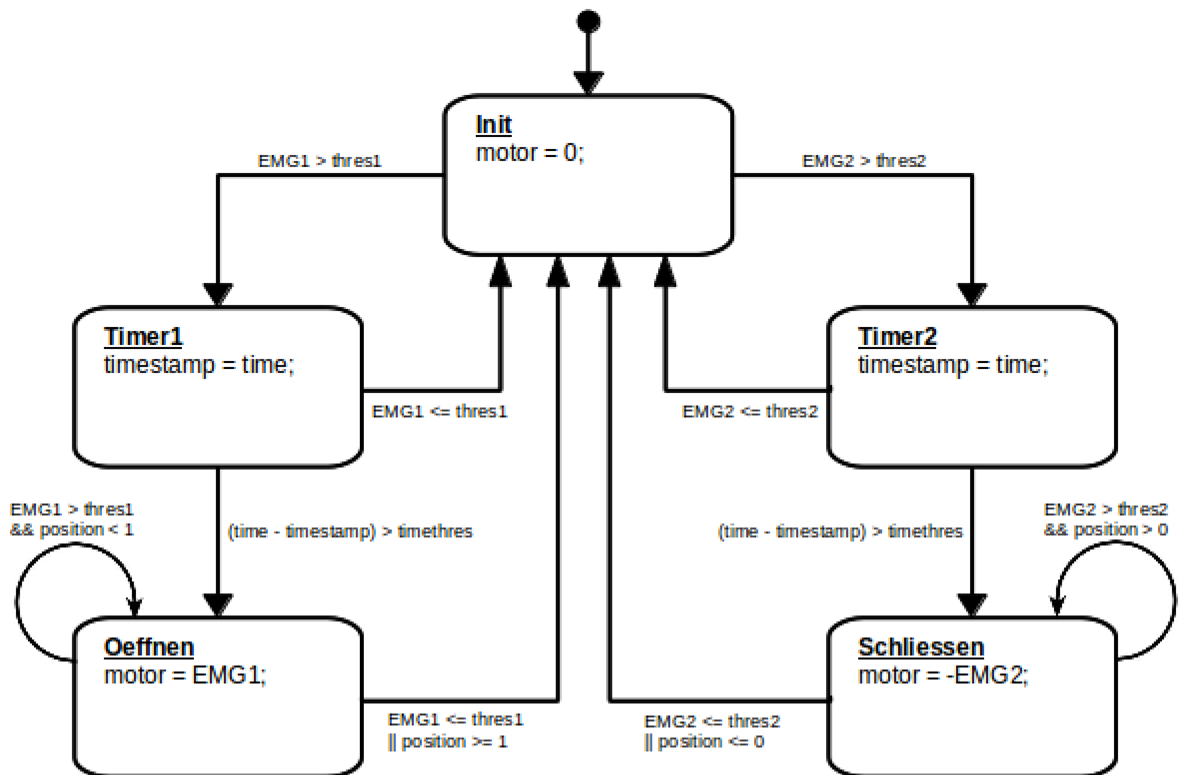


Abbildung 2.4: Zustandsautomat einer proportionalen Direktsteuerung, Entwurf aus der ersten Bachelorarbeit [2]

Wird dieser Entwurf eines Zustandsautomaten so wie in Abb. 2.5 adaptiert und erweitert, so kann er zur Steuerung der *Sensorhand Speed* mit dem *ATmega32* eingesetzt werden. Tabelle 2.4 fasst die im adaptierten Zustandsautomaten vorkommenden Größen zusammen. Das Modul *statemachine* soll den adaptierten Zustandsautomaten implementieren.

Im Vergleich zum Entwurf aus der ersten Bachelorarbeit (Abb. 2.4) werden alle Zustände, Zustandsgrößen, Verlaufsgrößen und Thresholds englisch benannt, da der Programmcode auf Englisch entwickelt werden soll.

Die im Entwurf vorgesehene Verlaufsgröße *position* (Öffnungsgrad der Prothese) liegt im gegenständlichen Projekt nicht vor. Die *Sensorhand Speed* besitzt keine Sensoren zur Positionsbestimmung der künstlichen Finger. Anstelle von *position* werden im überarbeiteten Zustandsautomaten die Verlaufsgrößen *motorREV* (Motordrehzahl) und *motorTorque* (Motormoment) eingeführt. Wenn die Prothese in den Zuständen *Open* bzw. *Close* mit dem Motor bewegt wird und anschlägt, hat sie entweder einen Gegegenstand gegriffen oder ist vollständig geöffnet bzw. geschlossen. Dabei sinkt *motorREV* und steigt *motorTorque*. Dementsprechend werden diese beiden Verlaufsgrößen im überarbeiteten Zustandsautomaten anhand der Thresholds *THRES_MOTOR_REV* und *THRES_MOTOR_TORQUE* beurteilt, um die Bewegung der Prothese abzubrechen, wenn diese anschlägt.

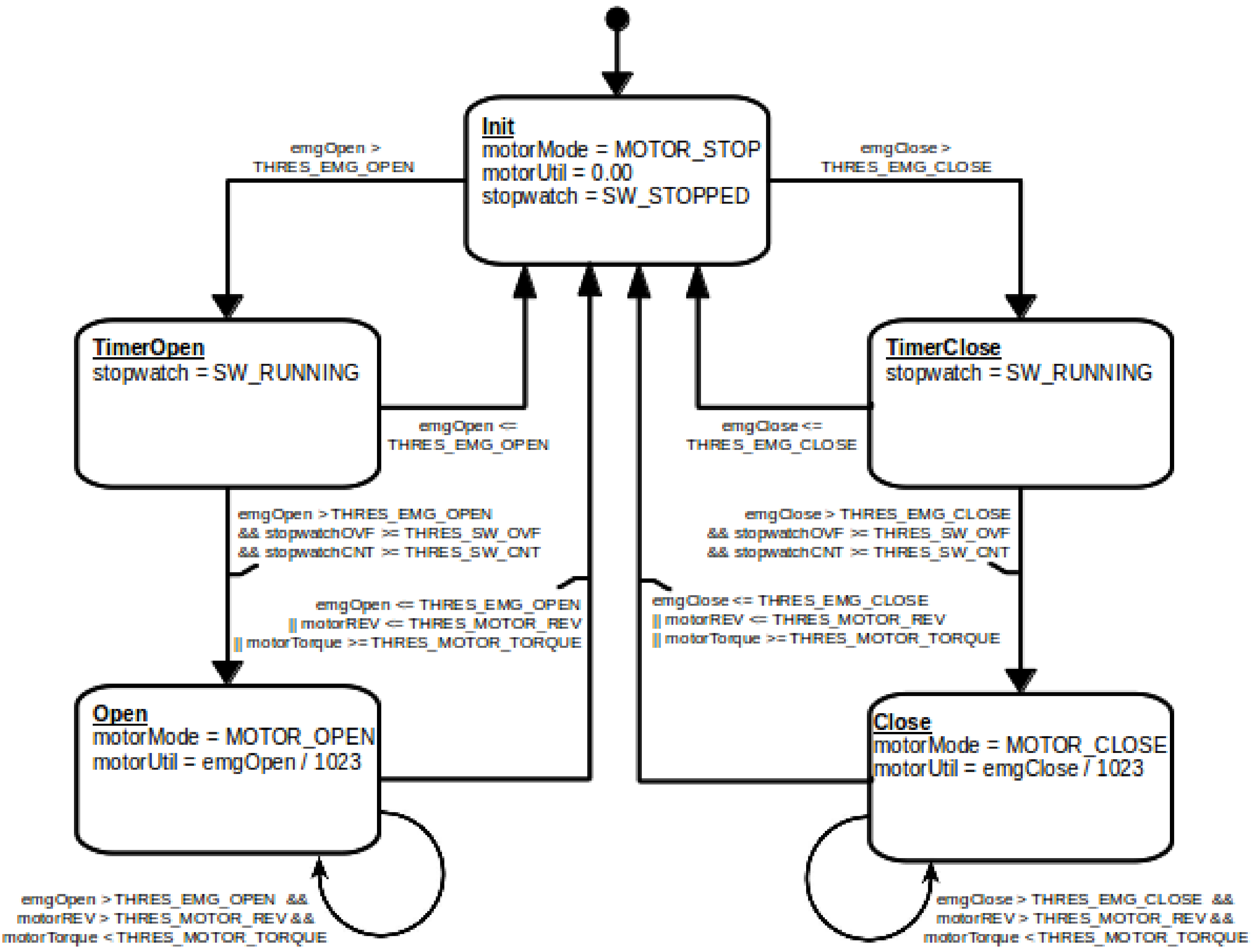


Abbildung 2.5: An das Projekt angepasster Zustandsautomat einer proportionalen Direktsteuerung

Art	Bezeichnung	Bedeutung
Zustandsgröße	motorMode	Betriebsmodus des Motors
	motorUtil	Motorauslastung in Prozent
	stopwatch	Betriebsmodus der Stoppuhr
Verlaufsgröße	emgOpen	Amplitude des EMG in Öffnungsrichtung
	emgClose	Amplitude des EMG in Schließrichtung
	stopwatchOVF	Stoppuhr: Anzahl der Überläufe seit Start
	stopwatchCNT	Stoppuhr: aktueller Zählerstand
	motorREV	Motordrehzahl
	motorTorque	Motormoment
Threshold	THRES_EMG_OPEN	Schwelle, die EMG überschreiten muss, um Hand zu öffnen
	THRES_EMG_CLOSE	Schwelle, die EMG überschreiten muss, um Hand zu schließen
	THRES_SW_OVF	Zeitdauer (Stoppuhr Überläufe), die EMG mindestens über Schwellwert sein muss, um Bewegung durchzuführen
	THRES_SW_CNT	Zeitdauer (Stoppuhr Zählwert), die EMG mindestens über Schwellwert sein muss, um Bewegung durchzuführen
	THRES_MOTOR_REV	Schwelle, die Drehzahl nicht unterschreiten darf, um eine Bewegung fortzusetzen
	THRES_MOTOR_TORQUE	Schwelle, die Motormoment nicht überschreiten darf, um eine Bewegung fortzusetzen

Tabelle 2.4: Zusammenfassung der Größen des adaptierten Zustandsautomaten

Der Entwurf sieht vor, dass Zeitpunkte von Ereignissen mit Hilfe der Zustandsgröße *timestamp* festgehalten werden, um dann bei der Prüfung von Transitionsbedingungen das Alter der Ereignisse bestimmen zu können. Der Entwurf berücksichtigt nicht, dass der für die Altersbestimmung benötigte Wert (Verlaufsgröße *time*, die sich proportional zur verstreichenden Zeit erhöht) in der Praxis der Wert im Zählregister eines Timer/Counters ist. Läuft der Timer/Counter über, zählt er wieder von 0 beginnend. Dann funktioniert die in den Transitionsbedingungen des entworfenen Zustandsautomaten vorgesehene Altersbestimmung nicht mehr. Der überarbeitete Zustandsautomat setzt stattdessen einen Timer/Counter als Stoppuhr ein, welche mit Hilfe der Zustandsgröße *stopwatch* gesteuert wird. Im Zustand *Init* ist die Stoppuhr gestoppt und auf Null zurückgesetzt. In den Zuständen *TimerOpen* und *TimerRun* läuft die Stoppuhr. Damit ist allgemein gültig festgelegt, dass die Stoppuhr beim Eintritt des Ereignisses *EMG überschreitet Threshold (im Zustand Init)* bei einem Stand von 0 gestartet wird. Zur Altersbestimmung steht der aktuelle Stand der Stoppuhr in den Verlaufsgrößen *stopwatchOVF* (Anzahl der Timer/Counter-Überläufe seit dem Starten der Stoppuhr) und *stopwatchCNT* (Zählerstand des Timer/Counters) zur Verfügung.

Der entworfene Zustandsautomat sieht die Zustandsgröße *motor* vor. Im Entwurf soll sie sowohl den vom System einzustellenden Betriebsmodus des Motors als auch

das Maß zur Regelung des Motors abbilden. Im überarbeiteten Zustandsautomaten wird die Zustandsgröße *motor* ersetzt durch die Zustandsgrößen *motorMode* und *motorUtil*. *motorMode* stellt den vom System einzustellenden Betriebsmodus des Motors (siehe Tab. 2.3) dar. *motorUtil* stellt die einzustellende Motorauslastung (in Prozent) dar. *motorUtil* wird beim Betreten der Zustände, in denen die Prothese eine Bewegung durchführen soll, auf den prozentualen Anteil der EMG Amplitude an der maximal möglichen EMG-Amplitude gesetzt. So wird die Motorauslastung proportional zur EMG-Amplitude geregelt.

2.2.5 prosthesiscontrol

Das Modul *prosthesiscontrol* stellt das Hauptprogramm, das auf dem Mikrocontroller ausgeführt wird. Darin betreibt es die proportionale Direktsteuerung der Handprothese, indem es in einem wiederkehrenden Ablauf die Interfaces der anderen Module anspricht. Es nimmt so innerhalb der modularen Software-Architektur eine zentrale und zwischen den anderen Modulen vermittelnde Rolle ein (Abb. 2.1).

Wird der Mikrocontroller eingeschaltet, initialisiert das Modul *prosthesiscontrol* zunächst alle Hardware-Komponenten und die peripheren Software-Module. Dann werden in einem alle 20 Millisekunden wiederkehrenden Vorgang alle EMG-, Sensor- und Motorsignale digitalisiert, der Steuerungs-Zustand aktualisiert und alle Hardware-Komponenten sowie Module dem aktuellen Zustand entsprechend angesteuert. Zusätzlich werden die digitalisierten Werte der Eingangssignale und der Name des aktuellen Steuerungs-Zustands auf dem Display ausgegegeben.

Die A/D-Wandlung wird somit alle 20 Millisekunden, das heisst mit einer Frequenz von 50 Hertz, durchgeführt. Gemäß der Spezifikationen von sowohl Sensor- als auch Motorplatine, die von anderen Studierenden entwickelt werden, ist diese Abtastfrequenz für alle Sensorsignale und Messignale des Motors geeignet. Auch aus den beiden EMG-Signalen wäre mit einer höheren Abtastfrequenz keine weitere Information zu gewinnen. Die EMG-Signale werden von aktiven Myoelektroden zur Verfügung gestellt, die das Signal bereits filtern, glätten, gleichrichten und die Einhüllende bilden.

Bevor das Modul *prosthesiscontrol* die A/D-Wandlung anstößt, schaltet es den Motor in den Messmodus. Nur in diesem Modus stellt die Motoransteuerungsplatine eine der Drehzahl entsprechende Spannung zur Verfügung. Gemäß der Spezifikation der Entwickler der Motorplatine wartet das Modul nach dem Umschalten in den Messmodus 2 Millisekunden, ehe es das Motordrehzahl-Signal tastet. Grund ist, dass zunächst der Strom, der durch die Induktivität im Motor vorhanden ist, abklingen muss. Erst dann liegt die vollständige Generatorspannung, welche der träge Rotor induziert und die als Maß für die aktuelle Drehzahl dient, am Platinenanschluss vor.

Um den Programmablauf mit dem Oszilloskop zeitlich beurteilen zu können, werden an relevanten Programmstellen Beobachtungs-Pins auf HIGH oder LOW geschaltet. Mit Hilfe dieser Pins kann insbesondere beobachtet werden, wann und wie oft der wiederkehrende Aktualisierungsvorgang stattfindet und wie lange er dauert. Ein anderes Beobachtungs-Pin gibt Aufschluss darüber, wie lange das Modul *prosthesiscontrol* den Motor im Messmodus betreibt, ehe es die A/D-Wandlung anstößt. Welches Pins als Beobachtungs-Pins verwendet werden, kann über die Konfiguration

des Moduls eingestellt werden.

3 Programmierung, Integration und Test der Module

Dieses Kapitel stellt die Programmierung der Software gemäß der Spezifikationen aus Kapitel 2 dar. Die Module werden für den Mikrocontroller zunächst isoliert entwickelt und dann im Hauptprogramm des Mikrocontrollers integriert.

3.1 pwm

3.1.1 Auswahl eines PWM Modus

Der *ATmega32* besitzt drei Timer/Counter, die selbstständig PWM-Signale erzeugen können. Sie können hierfür in unterschiedlichen PWM-Modi betrieben werden. Zunächst soll diskutiert werden, welcher dieser Modi sich für die Implementierung des Moduls *pwm* eignet.

Fast PWM Mode

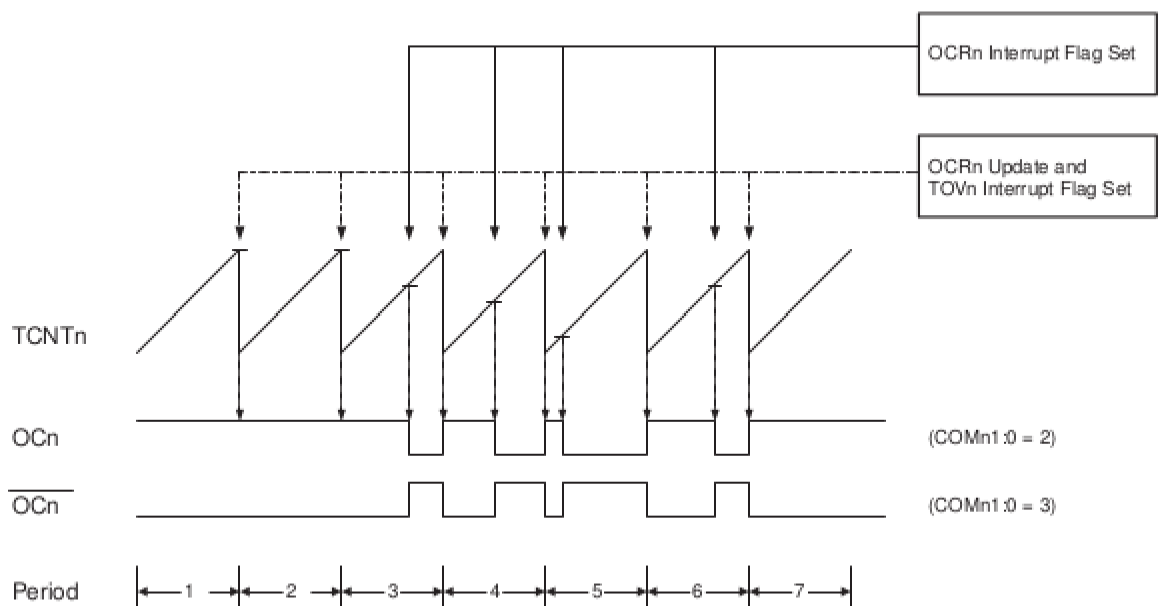


Abbildung 3.1: Fast PWM Mode des ATmega32
(entnommen aus *Atmel: ATmega32 Datasheet* [1])

Im *Fast PWM Mode* (Abbildung 3.1) inkrementiert der Timer/Counter den Wert in dessen Zählregister (*TCNTn*) von *BOTTOM* (0) bis *MAX* (maximal im Register darstellbarer Wert) [1]. Die Dauer eines solchen ganzen Durchlaufs des Timer/Counters bestimmt die Periodendauer T_{PER} und damit die Frequenz des PWM-Signals. Immer wenn der Timer/Counter beginnt, wieder von *BOTTOM* an zu inkrementieren, wird das Pin *OCn* des Mikrocontrollers [1] auf *HIGH* geschaltet. Sobald das Zählregister dann während des Inkrementierens des Timer/Counters den selben Wert wie das Vergleichsregister (*OCRn*) annimmt (**CompareMatch**) [1], wird Pin *OCn* auf *LOW* geschaltet. So kann durch Setzen des Vergleichsregisters die aktuelle Pulsdauer t_i und damit der Duty Cycle des PWM-Signals eingestellt werden.

Phase Correct PWM Mode

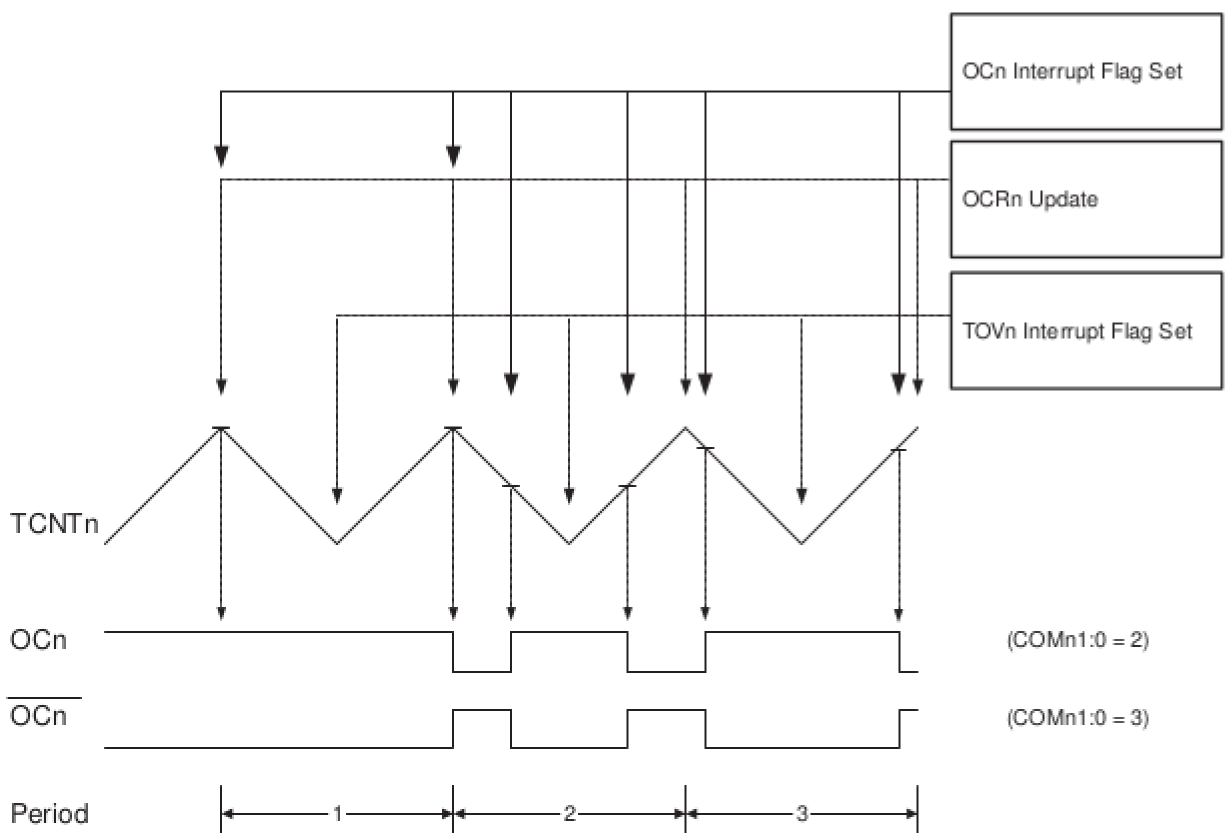


Abbildung 3.2: Phase Correct PWM Mode des ATmega32 (entnommen aus *Atmel: ATmega32 Datasheet* [1])

Im *Phase Correct PWM Mode* (Abbildung 3.2) inkrementiert der Timer/Counter den Wert in dessen Zählregister von *BOTTOM* bis *MAX* und dekrementiert dann wieder von *MAX* bis *BOTTOM*. Die Dauer eines solchen zweifachen Durchlaufs des Timer/Counters (*BOTTOM* → *MAX* → *BOTTOM*) bestimmt die Periodendauer T_{PER} und damit die Frequenz des PWM-Signals.

Immer wenn der Timer/Counter am Beginn einer PWM-Periode beginnt, von *BOTTOM* an zu inkrementieren, wird Pin *OCn* zunächst einmal auf *LOW* geschal-

tet. Erst sobald es dann während des Inkrementierens des Timer/Counters zu einem CompareMatch von Zähl- und Vergleichsregister kommt, wird Pin *OCn* auf *HIGH* geschaltet, bis es während des dann folgenden Dekrementierens des Timer/Counters erneut zu einem CompareMatch kommt und das Pin *OCn* wieder auf *LOW* geschaltet wird. So kann durch Setzen des Vergleichsregisters die aktuelle Pulsdauer t_i und damit der Duty Cycle des PWM-Signals eingestellt werden.

Im Vergleich zum *Fast PWM Mode* resultiert hier ein PWM-Signal, in dem der Impuls (*HIGH*-Phase) genau zentriert in der PWM-Signalperiode liegt. Im *Fast PWM Mode* hingegen tritt der Impuls nicht zentriert, sondern zu Beginn jeder PWM-Signalperiode auf. Das führt bei zeit-sensitiver Auswertung des PWM-Signals zu einer Verschiebung desjenigen Zeitpunktes (Phasenshift), welcher der im Duty Cycle enthaltenen Information zugeordnet ist. Der *Phase Correct PWM Mode* behebt dieses Problem.

Phase and Frequency Correct PWM Mode

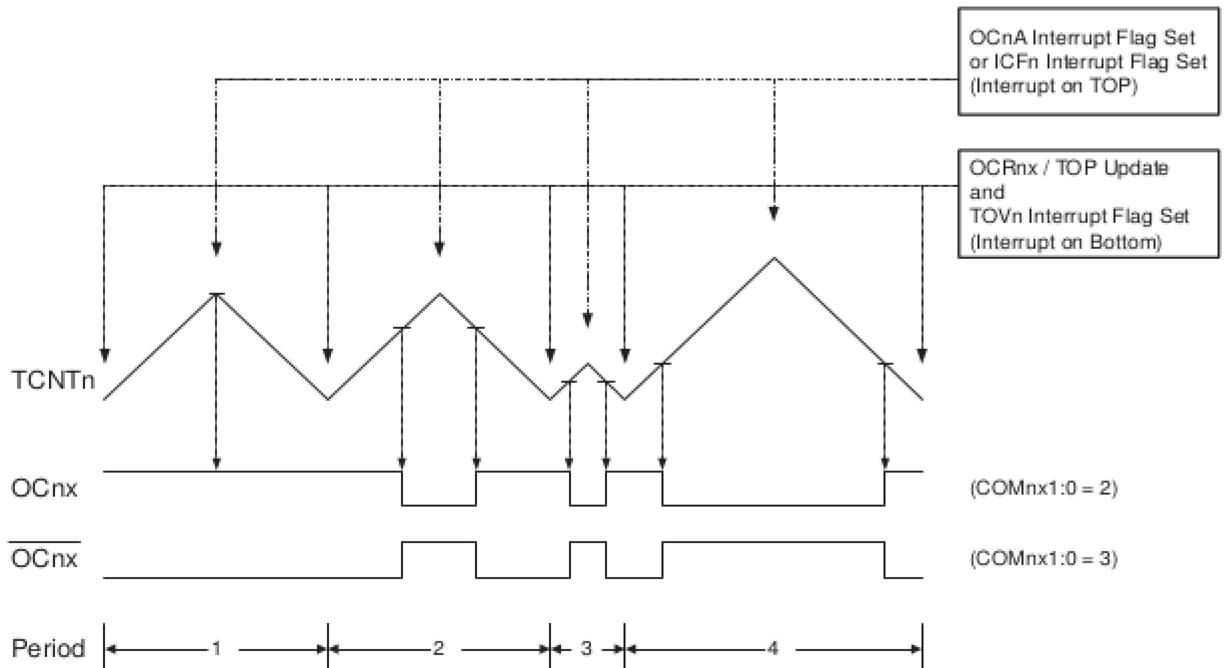


Abbildung 3.3: Phase and Frequency Correct PWM Mode des ATmega32 (entnommen aus *Atmel: ATmega32 Datasheet* [1])

Der *Phase and Frequency Correct Mode* (Abbildung 3.3), den im *ATmega32* lediglich Timer/Counter 1 beherrscht, funktioniert ähnlich wie der *Phase Correct PWM Mode*.

Wesentlicher Unterschied ist, dass der Timer/Counter in jeder Periode nicht bis *MAX*, sondern lediglich bis zu einem (über ein Register) einstellbaren Wert (*TOP* [1]) inkrementiert und von dort wieder bis *BOTTOM* dekrementiert (*BOTTOM* → *TOP* → *BOTTOM*). Die Wahl von *TOP* bestimmt also die Periodendauer T_{PER} und damit die Frequenz des erzeugten PWM-Signals.

Die Frequenz des PWM-Signals kann also im *Phase and Frequency Correct Mode* durch die Wahl von sowohl *Prescaler* [1] als auch *TOP* deutlich feiner eingestellt werden als im *Fast PWM Mode oder im Phase Correct PWM Mode*. Denn in diesen Modi ist die erzeugte Frequenz lediglich durch den *Prescaler* beeinflussbar. Die Anzahl der einstellbaren Frequenzen ist so auf die Anzahl der möglichen *Prescaler*-Konfigurationen begrenzt.

Gemäß der Modul-Spezifikation aus Kapitel 2.2.1 soll es möglich sein, die Frequenz des PWM-Signals über das Interface des Moduls *pwm* zu einzustellen. Aus diesem Grund fällt die Entscheidung auf den *Phase and Frequency Correct PWM Mode*. Denn er ermöglicht es als einziger unter den drei PWM-Modi, die PWM-Frequenz in einem durch den Prescaler vorgegebenen Bereich (quasi) beliebig einzustellen.

3.1.2 Implementierung des Moduls

Das Modul *pwm* wird als Funktionsbibliothek entwickelt. Im Folgenden werden essentielle Ausschnitte des entwickelten Programmcodes dargestellt und erläutert. Der gesamte Programmcode ist im Anhang dokumentiert.

Codeausschnitt 3.1 zeigt einen Ausschnitt der Header-Datei *pwm.h*.

```
1  #define CPU_FREQ 8000000 // CPU frequency (for calculation of PWM frequency)
2
3  // preprocessor directives PWM inversion modes
4  #define PWM_NON_INVERTED 0 // PWM signal not inverted
5  #define PWM_INVERTED     1 // PWM signal inverted
6
7  #include <avr/io.h> // AVR device-specific IO definitions
8
9  // pwm interface functions family
10 uint8_t pwmInit (double, uint8_t);
11 uint8_t pwmDuty (double);
12 void pwmTerminate (void);
```

Codeausschnitt 3.1: pwm.h

In Zeile 1 des Codeausschnitts 3.1 wird die CPU-Taktfrequenz von 8 MHz, mit welcher der Mikrocontroller betrieben wird, als Präprozessorvariable *CPU_FREQ* festgelegt. Dieses Makro wird von den Funktionen des Moduls *pwm* herangezogen, um die Frequenz des zu erzeugenden PWM-Signals einzustellen. In den Zeilen 4 und 5 werden Bezeichnungen für den invertierenden und nicht-invertierenden PWM-Modus als Präprozessorvariablen festgelegt. Eine derartige Benennung wird fortan als **Identifier** bezeichnet. In Zeile 7 werden mit der Bibliothek *avr/io.h* die AVR-Input/Output-Definitionen eingebunden. In Zeilen 10 bis 12 werden drei Funktionen deklariert, die in der Datei *pwm.c* definiert sind.

Codeausschnitt 3.2 aus der Datei *pwm.c* zeigt die Definition der Funktion *pwmInit()*. Diese Funktion initialisiert das PWM-Signal und setzt dessen Duty Cycle auf 0%. Die Funktionsargumente geben die einzustellende Frequenz des PWM-Signals und den Invertierungsmodus wieder. *pwmInit()* gibt einen Wert des Typs 8-bit Integer zurück, welcher der aufrufenden Instanz Aufschluss darüber gibt, ob während der Ausführung der Funktion Fehler auftraten. Ein derartiger Rückgabewert wird fortan als **Errorlevel** bezeichnet.

```
1 uint8_t pwmInit(double pwmFrequency, uint8_t inversionMode)
2 {
```

```
3      // [Step 1]
4
5      uint8_t errorlevel=0;
6
7      if ( pwmFrequency > 1330000.00 || pwmFrequency < 61.04 )
8      {
9        errorlevel = 1;
10    }
11
12     if ( inversionMode!=PWM_NON_INVERTED && inversionMode!=PWM_INVERTED )
13     {
14       errorlevel = 1;
15    }
16
17     if ( errorlevel == 0 )
18     {
19       // [Step 2]
20
21       TCCR1B &=~ ( (1<<CS12) | (1<<CS11) );
22       TCCR1B |=  (1<<CS10);
23
24       // [Step 3]
25
26       TCCR1A &=~ ( (1<<FOC1A) | (1<<FOC1B) );
27
28       // [Step 4]
29
30       TCCR1B |=  (1<<WGM13);
31       TCCR1B &=~ (1<<WGM12);
32       TCCR1A |=  (1<<WGM11);
33       TCCR1A &=~ (1<<WGM10);
34
35       // [Step 5]
36
37       TCCR1A |=  (1<<COM1A1);
38
39       switch (inversionMode)
40       {
41          case PWM_NON_INVERTED: TCCR1A &=~ (1<<COM1A0); break;
42          case PWM_INVERTED:     TCCR1A |=  (1<<COM1A0); break;
43       }
44
45       DDRD   |=  (1<<DDD5);
46
47       // [Step 6]
48
49       pwmDuty(0.0);
50
51       // [Step 7]
52
53       ICR1 = (uint16_t) ( CPU_FREQ/(2*1*pwmFrequency) );
54    }
55
56    // [Step 8]
57
58    return errorlevel;
59 }
```

Codeausschnitt 3.2: Funktion pwmInit() in pwm.c

In den Zeilen 7 bis 15 des Codeausschnitts 3.2 werden die übergebenen Funktionsargumente überprüft. Wie Tabelle 3.1 zeigt, sind beim gewählten Prescaler von 1 (siehe unten) nur PWM-Frequenzen im Bereich von 61,04 Hz bis 1,33 MHz einstellbar. Ist das Funktionsargument *pwmFrequency* ausserhalb dieses Bereiches, wird der Errorlevel auf 1 gesetzt und die Initialisierung des PWM-Signals nicht durchgeführt. Das Argument *inversionMode* muss einem der beiden Identifier für den Invertierungsmodus aus *pwm.h* entsprechen. Da es sich bei den Identifiern um

Prescaler	**minimale PWM-Frequenz**	**maximale PWM-Frequenz**	**maximale Duty Cycle Auflösung**
1	61,04 Hz	1,33 MHz	750 ns
8	7,63 Hz	166,66 kHz	6,00 µs
64	0,96 Hz	20,83 kHz	48,00 µs
256	0,24 Hz	5,20 kHz	192,00 µs
1024	0,06 Hz	1,30 kHz	768,00 µs

Tabelle 3.1: einstellbare PWM-Frequenzen und Duty Cycle Auflösung mit Timer/-Counter 1 im *Phase and Frequency Correct PWM Mode*

Präprozessorvariablen handelt, kann als Funktionsargument auch der Zahlenwert 0 oder 1 übergeben werden. Wurde ein ungültiges Argument *inversionMode* übergeben, wird der Errorlevel auf 1 gesetzt und die Initialisierung des PWM-Signals nicht durchgeführt.

In den Zeilen 21 und 22 wird der Timer/Counter 1 des *ATmega32* aktiviert. Dieser ist der einzige der drei vorhandenen Timer/Counter, welcher im gewählten *Phase and Frequency Correct PWM Mode* betrieben werden kann. Durch die Konfiguration der Bits *CS12*, *CS11* und *CS10* wird festgelegt, mit welchem Vorteiler (**Prescaler**) Timer/Counter 1 inkrementiert [1]. Mit einem Prescaler von 1 inkrementieren der Timer/Counter den Wert im Zählregister mit jedem CPU-Takt, hingegen mit einem Prescaler von zum Beispiel 8 nur mit jedem achten CPU-Takt. Tabelle 3.1 zeigt, wie im Falle von Timer/Counter 1 im *Phase and Frequency Correct PWM Mode* der Bereich der einstellbaren PWM-Frequenzen und die Auflösung des Duty Cycles von der Wahl des Prescalers abhängt. Da die PWM-Frequenz gemäß der Modul-Spezifikationen (Kapitel 2.2.1) in einem Bereich von 1 kHz bis 40 kHz einstellbar sein soll, kämen im Allgemeinen die Prescaler 1, 8, 64 und 256 in Frage. Da hierbei der Prescaler 1 die beste Auflösung des Duty Cycles bietet, wird dieser in Zeilen 21 und 22 des Codeausschnitts 3.2 eingestellt.

Mit den Zeilen 30 bis 33 wird der Timer/Counter 1 in den *Phase and Frequency Correct PWM Mode* geschaltet, wobei *TOP* (siehe Kapitel 3.1.1) mit dem Wert im Register *ICR1* festgelegt wird [1].

In Zeilen 37 bis 45 wird das Pin *OC1A* als Ausgabe-Pin für das erzeugte PWM-Signal definiert. In der switch-case-Anweisung in den Zeilen 39 bis 43 wird in Abhängigkeit vom Funktionsargument *inversionMode* der Invertierungsmodus des ausgegebenen PWM-Signals definiert [1].

In Zeile 53 wird *TOP* in Abhängigkeit vom Argument *pwmFrequency* berechnet und in das Register *ICR1* geschrieben werden. Die Berechnung von *TOP* erfolgt gemäß Formel 3.2, die sich durch Äquivalenzumformung der im Datenblatt des Mikrocontrollers angegebenen Formel 3.1 ergibt.

$$f_{PWM} = \frac{f_{CPU}}{2 \cdot N \cdot TOP} \tag{3.1}$$

$$TOP = \frac{f_{CPU}}{2 \cdot N \cdot f_{PWM}} \tag{3.2}$$

Codeausschnitt 3.3 aus der Datei *pwm.c* zeigt die Definition der Funktion *pwmDuty()*. Diese Funktion stellt den Duty Cycle des PWM-Signals auf einen im Funktionsargument angegebenen Wert ein. *pwmDuty()* gibt einen Errorlevel des Typs 8-bit Integer zurück.

```
uint8_t pwmDuty(double pwmDutycycle)
{
   uint8_t errorlevel=0; // declare errorlevel, initialize it with 0

   // check argument: is pwmDutycycle within [0.0;1.0] ?
   if ( pwmDutycycle < 0.0 || pwmDutycycle > 1.0 )
   {
      // invalid argument (value not between 0.0 and 1.0)
      errorlevel = 1;
   }

   // if argument valid
   if ( errorlevel == 0 )
   {
      // set PWM duty cyle corresponding to argument
      OCR1A = pwmDutycycle * ICR1;
   }

   // if argument invalid
   else
   {
      // set PWM duty cyle to zero
      OCR1A = 0;
   }

   // quit by returning errorlevel
   return errorlevel;
}
```

Codeausschnitt 3.3: Funktion pwmDuty() in pwm.c

In den Zeilen 6 bis 10 des Codeausschnitts 3.3 wird das Funktionsargument *pwmDutycycle* geprüft. Dieses muss in einem Bereich von 0,0 bis 1,0 liegen. Ist das nicht der Fall, wird der Errorlevel auf 1 gesetzt und der Duty Cycle in Zeile 23 auf 0 gesetzt. Ist das Argument im gültigen Bereich, wird in Zeile 16 der Duty Cycle durch Setzen des Vergleichswertes im Register *OCR1A* [1] gesetzt. So wie der Duty Cycle der prozentuale Anteil der PWM-Pulsdauer an der PWM-Periodendauer ist (siehe Kapitel 2.2.1), so ist der Vergleichswert (*OCR1A*) der selbe prozentuale Anteil an *TOP* (*ICR1A*).

3.1.3 Modultest

Der Test des Moduls *pwm* wird mit Hilfe eines auf dem Mikrocontroller ausgeführten Hauptprogramms, das verschiedene Anwendungsfälle des Moduls simuliert, vorgenommen. Codeausschnitt 3.4 zeigt einen Ausschnitt dieses Programms, welches in voller Länge im Anhang dokumentiert ist.

```
int main(void)
{
   // software test: case 1
   pwmInit ( 100, PWM_NON_INVERTED);        // PWM 100 Hz, non-inverted
   pwmDuty ( 0.75 );                        // PWM duty cycle 75%

   // software test: case 2
   // pwmInit ( 500, PWM_NON_INVERTED);        // PWM 500 Hz, non-inverted
   // pwmDuty ( 0.75 );                        // PWM duty cycle 75%
```

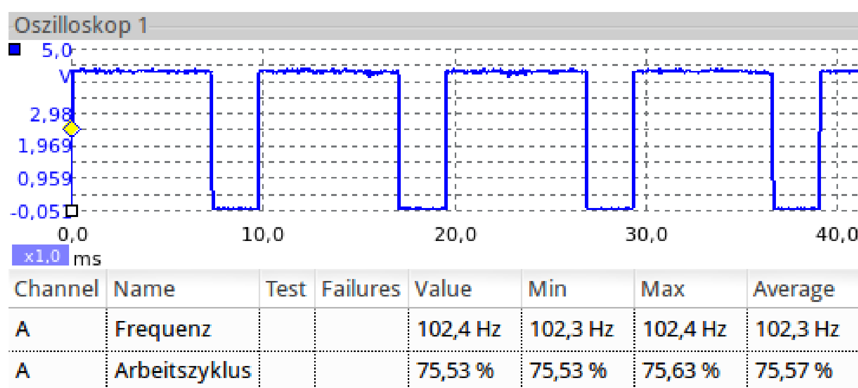
```
10
11      // software test: case 3
12      // (...)
13
14      _delay_ms(15000);                    // wait 15 seconds
15
16      pwmTerminate();                      // terminate PWM
17
18      while(1);
19
20      return 0;
21  }
```

Codeausschnitt 3.4: Hauptprogramm zur Simulation von Anwendungfällen des Moduls pwm

Codeausschnitt 3.4 zeigt den Ablauf des Hauptprograms für den Modultest. Zu Beginn initialisiert das Programm das PWM-Signal und stellt einen Duty Cycle ein. Die Frequenz, der Invertierungsmodus und der Duty Cycle sind dabei vom gewählten Anwendungsfall abhängig. Abschließend erzeugt das Programm das eingestellte PWM-Signal für eine Dauer von 15 Sekunden und terminiert dann das Modul.

Zur Testdurchführung wird der jeweilige Anwendungsfall simuliert, indem er nicht kommentiert wird und alle anderen Anwendungsfälle auskommentiert werden. Das am Pin *OC1A* ausgegebene Signal wird mit dem Oszilloskop dargestellt. Die Testparameter (Frequenz, der Invertierungsmodus und der Duty Cycle) des erzeugten Signals werden notiert, ebenso ob das PWM-Signal nach 15 Sekunden verschwindet. Der Test des jeweiligen Anwedungsfalles gilt als bestanden, wenn die notierten Testparameter mit der Definition des Anwedungsfalles übereinstimmen und das Signal nach 15 Sekunden verschwindet.

Abb. 3.4 zeigt den bestandenen Anwendungsfall 1 (PWM-Signal 100 Hz, nicht invertiert, Duty Cycle 75%) auf dem Oszilloskop. Das im Anhang in voller Länge dokumentierte Hauptprogramm für diesen Modultest beinhaltet im Kommentar eine Tabelle aller getesteten Anwendungsfälle. Das Modul *pwm* bestand alle Tests dieser 16 Anwendungsfälle.



Channel	Name	Test	Failures	Value	Min	Max	Average
A	Frequenz			102,4 Hz	102,3 Hz	102,4 Hz	102,3 Hz
A	Arbeitszyklus			75,53 %	75,53 %	75,63 %	75,57 %

Abbildung 3.4: Darstellung des Anwendungsfalles 1 (Test des Moduls pwm) auf dem Oszilloskop

3.2 motor

3.2.1 Implementierung des Moduls

Das Modul *motor* wird als Funktionsbibliothek entwickelt. Im Folgenden werden essentielle Ausschnitte des entwickelten Programmcodes dargestellt und erläutert. Der gesamte Programmcode ist im Anhang dokumentiert.

Codeausschnitt 3.5 zeigt einen Ausschnitt der Header-Datei *motor.h*.

```
#include <pwm/pwm1.0/pwm.h>  // include PWM interface (version 1.0)

#define MOTOR_PWM_FREQ       20000          // motor PWM frequency (Hz)
#define MOTOR_PWM_INV_MODE   PWM_INVERTED   // motor PWM inversion mode

// DDRs, Ports and Pins of Motor A and Motor B (output to motor circuit)
#define MOTOR_A_DDR   DDRB    // Data Direction Register for Motor A
#define MOTOR_A_PORT  PORTB   // Port Register for Motor A
#define MOTOR_A_PIN   0x06    // Pin for Motor A
#define MOTOR_B_DDR   DDRB    // Data Direction Register for Motor B
#define MOTOR_B_PORT  PORTB   // Port Register for Motor B
#define MOTOR_B_PIN   0x07    // Pin for Motor B

// preprocessor directives for motor operation modes
#define MOTOR_STOP     0  // no motor operation (free wheel or brake)
#define MOTOR_OPEN     1  // motor operation: open hand
#define MOTOR_CLOSE    2  // motor operation: close hand
#define MOTOR_MEASURE 3  // motor measure mode

// motor interface functions family
uint8_t motorInit (void);
uint8_t motorOperation (uint8_t, double);
uint8_t motorTerminate (void);
```

Codeausschnitt 3.5: motor.h

Um den Spezifikationen des Moduls *motor* (siehe Kapitel 2.2.2) zu entsprechen, werden in Codeausschnitt 3.5 in den Zeilen 2 bis 12 Präprozessorvariablen definiert, mit die das Modul konfiguriert werden kann. In den Zeilen 2 und 3 können die Frequenz des PWM-Signals, das an die Motoransteuerungsplatine übergeben wird, eingestellt werden. In der vorliegenden Version wird die PWM-Frequenz gemäß der vorläufigen Spezifikation (Kapitel 2.2.2) auf 20 kHz und invertiert eingestellt. Die Präprozessoranweisung in Zeile 3 verwendet den Indentifier für den invertierten Modus aus *pwm.h* (3.1). In den Zeilen 7 bis 12 kann eingestellt werden, welche Pins des Mikrocontrollers an die Anschlüsse *Motor A* und *Motor B* der Motoransteuerungsplatine angeschlossen werden. In den Zeilen 15 bis 18 werden Identifier für die vier in den Spezifikationen definierten Motormodi (Kapitel 2.2.2) vergeben. In den Zeilen 21 bis 23 werden drei Funktionen deklariert, die in der Datei *motor.c* definiert sind.

Codeausschnitt 3.6 aus der Datei *motor.c* zeigt die Definition der Funktion *motorInit()*. Diese Funktion initialisiert die Schnittstelle des Mikrocontrollers zur Motoransteuerungsplatine und setzt den Motorbetriebsmodus auf *MOTOR_STOP* und die Motorauslastung auf 0%. Sie gibt einen Errorlevel als 8bit Integer zurück.

```
uint8_t motorInit (void)
{
   uint8_t errorlevel=0; // declare errorlevel, initialize it with 0

   // initialize PWM signal
   if ( pwmInit(MOTOR_PWM_FREQ, MOTOR_PWM_INV_MODE) )
   {
```

```
8          // an error occured in pwmInit()
9          errorlevel = 1;
10     }
11
12     else
13     {
14         // set Data Direction Registers to Output for pins Motor A and B
15         MOTOR_A_DDR |= (1<<MOTOR_A_PIN);
16         MOTOR_B_DDR |= (1<<MOTOR_B_PIN);
17
18         // set motor mode to STOP (no operation)
19         if ( motorOperation (MOTOR_STOP, 0.0) )
20         {
21             // an error occured in motorOperation()
22             errorlevel = 1;
23         }
24     }
25
26     // quit by returning errorlevel
27     return errorlevel;
28 }
```

Codeausschnitt 3.6: Funktion motorInit() in der Datei motor.c

In den Zeilen 5 bis 10 des Codeausschnitts 3.6 initialisiert das Modul *motor* über das Interface des Moduls *pwm* mit Hilfe der Funktion *pwmInit()* ein PWM-Signal. Dabei stellt es die PWM-Frequenz und den Invertierungsmodus mit Hilfe der Präprozessorvariablen aus der Datei *motor.h* (Codeausschnitt 3.5) ein. Kommt es dabei zu einem Fehler, wird Errorlevel auf 1 gesetzt und die Schnittstelle des *ATmega32* zur Motoransteuerungsplatine nicht initialisiert. In den Zeilen 15 und 16 werden die Pins, die in *pwm.h* als Anschlüsse für *Motor A* und *Motor B* eingestellt sind, im Datenrichtungsregister als Ausgabe-Pins konfiguriert. Dann setzt das Modul mit Hilfe der Funktion *motorOperation()* (siehe unten) den Motorbetriebsmodus auf *MOTOR_STOP* und die Motoruslastung auf 0%. Kommt es hierbei zu einem Fehler, wird der Errorlevel auf 1 gesetzt.

Die Definition der oben bereits erwähnten Funktion *motorOperation()* ist im Codeausschnitt 3.7 aus der Datei *motor.h* dargestellt. Sie stellt den Motorbetriebsmodus und die Motorauslastung gemäß der übergebenen Funktionsargumente ein. Sie gibt einen Errorlevel als 8bit Integer zurück.

```
1  uint8_t motorOperation (uint8_t motorMode, double motorUtilization)
2  {
3      uint8_t errorlevel=0; // declare errorlevel, initialize it with 0
4
5      // set PWM duty cycle and pins Motor A and Motor B
6      switch (motorMode)
7      {
8          case MOTOR_STOP:
9
10             errorlevel = pwmDuty(motorUtilization); // set duty cycle
11             MOTOR_A_PORT &=~ (1<<MOTOR_A_PIN); // set Motor A to LOW
12             MOTOR_B_PORT &=~ (1<<MOTOR_B_PIN); // set Motor B to LOW
13
14         break;
15
16         case MOTOR_OPEN:
17
18             errorlevel = pwmDuty(motorUtilization); // set duty cycle
19             MOTOR_A_PORT |=  (1<<MOTOR_A_PIN); // set Motor A to HIGH
20             MOTOR_B_PORT &=~ (1<<MOTOR_B_PIN); // set Motor B to LOW
21
22         break;
23
```

```
24        case MOTOR_CLOSE:
25
26            errorlevel = pwmDuty(motorUtilization); // set duty cycle
27            MOTOR_A_PORT &=~ (1<<MOTOR_A_PIN); // set Motor A to LOW
28            MOTOR_B_PORT |=  (1<<MOTOR_B_PIN); // set Motor B to HIGH
29
30        break;
31
32        case MOTOR_MEASURE:
33
34            errorlevel = pwmDuty(0.00); // force duty cycle 0%
35
36            // toggle Motor A and Motor B (only if both are not LOW)
37            if (    ( MOTOR_A_PORT & (1<<MOTOR_A_PIN) )
38                || ( MOTOR_B_PORT & (1<<MOTOR_B_PIN) )    )
39            {
40                MOTOR_A_PORT ^=  (1<<MOTOR_A_PIN); // toggle Motor A
41                MOTOR_B_PORT ^=  (1<<MOTOR_B_PIN); // toggle Motor B
42            }
43
44        break;
45
46        default:
47
48            // invalid argument motorMode
49            errorlevel = 1;
50
51        break;
52    }
53
54    // quit by returning errorlevel
55    return errorlevel;
56 }
```

Codeausschnitt 3.7: Funktion motorInit() in der Datei motor.c

Die Funktion *motorOperation()* (Codeausschnitt 3.7) untersucht das Argument *motorMode* mit Hilfe einer switch-case-Anweisung. Es kann einer der Indentifier für Motorbetriebsmodi (Zeilen 15 bis 18 in 3.5) als Argument *motorMode* übergeben werden. Soll einer der Modi *MOTOR_STOP*, *MOTOR_OPEN* oder *MOTOR_CLOSE* eingestellt werden, so setzt die Funktion *motorOperation* die Pins *Motor A* und *Motor B* entsprechend der Spezifikationen in Kapitel 2.2.2 und die Motorauslatung durch Einstellung des PWM-Duty Cycles entsprechend des Funktionsargumentes *motorUtilization*.

Soll der Betriebsmodus *MEASURE_MODE* eingestellt werden, wird zuerst erst geprüft, ob eines der beiden Pins auf *HIGH* geschalten ist (Zeilen 37 und 38). Ist das der Fall, dann werden *Motor A* und *Motor B* schlichtweg getoggled, sodass der Motor entsprechend der Spezifikationen aus Kapitel 2.2.2 in seiner momentanen Drehrichtung ausgemessen werden kann. Ist das hingegen nicht der Fall, waren beide Pins *Motor A* und *Motor B* vor dem Funktionsaufruf auf *LOW* (Modus *MOTOR_STOP*) geschalten. Sie werden daher nicht getoggled und beide auf *LOW* belassen. Ein Togglen der Pins würde nicht den *MEASURE_MODE* herstellen, sondern einen per Spezifikation (Kapitel 2.2.2) nicht verwendeten Zustand (beide Pins *Motor A* und *Motor B* auf *HIGH*) herbeiführen.

3.2.2 Modultest

Der Test des Moduls *motor* wird mit Hilfe eines auf dem Mikrocontroller ausgeführten Hauptprogramms, das verschiedene Anwendungsfälle des Moduls simuliert,

vorgenommen. Codeausschnitt 3.8 zeigt einen Ausschnitt dieses Programms, welches in voller Länge im Anhang dokumentiert ist.

```
int main(void)
{
    motorInit();                          // initialize motor control

    //software test: use case 1
    motorOperation(MOTOR_STOP,0.50);      // motor stopped, break 50%

    // software test: use case 2
    // motorOperation(MOTOR_OPEN,0.50);      // open hand, motor 50%

    // software test: use case 3
    // ...

    _delay_ms(15000);                     // wait 15 seconds

    motorTerminate();                     // terminate motor control

    while(1);

    return 0;
}
```

Codeausschnitt 3.8: Hauptprogramm zur Simulation von Anwendungfällen des Moduls motor

Codeausschnitt 3.8 zeigt den Ablauf des Hauptprograms für den Modultest. Zu Beginn initialisiert das Programm die Schnittstelle des Mikrocontrollers zur Motoransteuerungsplatine. Dann stellt es einen Motorbetriebsmodus ein. Der Motormodus und die Motorauslastung sind dabei vom gewählten Anwendungsfall abhängig. Anschließend erzeugt das Programm dem eingestellten Modus entsprechende Motoransteuerungssignale. Nach einer Dauer von 60 Sekunden und terminiert das Testprogramm das Modul.

Zur Testdurchführung wird der jeweilige Anwendungsfall simuliert, indem er nicht kommentiert wird und alle anderen Anwendungsfälle auskommentiert werden. Das am Pin *OC1A* ausgegebene Signal wird mit dem Oszilloskop dargestellt, die Pins *Motor A* und *Motor B* mit Hilfe eines Multimeters geprüft. Die Testparameter (Frequenz, der Invertierungsmodus und der Duty Cycle, logische Spannungslevels an den Pins *Motor A* und *Motor B*) der erzeugten Signale werden notiert, ebenso ob sie nach 60 Sekunden verschwinden. Der Test des jeweiligen Anwedungsfalles gilt als bestanden, wenn die notierten Testparameter mit der Definition des Anwedungsfalles übereinstimmen und das Signal nach 60 Sekunden verschwindet.

Das im Anhang in voller Länge dokumentierte Hauptprogramm für diesen Modultest beinhaltet im Kommentar eine Tabelle aller getesteten Anwendungsfälle. Das Modul *motor* bestand alle Tests dieser 8 Anwendungsfälle.

3.3 adc

3.3.1 Implementierung des Moduls

Das Modul *adc* wird als Funktionsbibliothek entwickelt. Im Folgenden werden essentielle Ausschnitte des entwickelten Programmcodes dargestellt und erläutert. Der gesamte Programmcode ist im Anhang dokumentiert.

Codeausschnitt 3.9 zeigt einen Ausschnitt der Header-Datei *adc.h*.

```
#include <lcd/mylcd.h>   // interface for display control
#include <string.h>      // library for string operations
#include <stdlib.h>     // required for conversion integer to string
#include <lcd/font5x8.h> // font for display

// prepropcessor directives as identifiers for ADC channels
#define ADC0 0  // ADC channel 0 (EMG signal open direction)
#define ADC1 1  // ADC channel 1 (EMG signal close direction)
#define ADC2 2  // ADC channel 2 (motor voltage)
#define ADC3 3  // ADC channel 3 (motor current)
#define ADC4 4  // ADC channel 4 (film pressure sensor in thumb)
#define ADC5 5  // ADC channel 5 (clip strain sensor in wrist)

// DDRs, Ports and Pins of P0MUX, P1MUX and P2MUX
// (output to sensor in thumb)
#define P0MUX_DDR  DDRB   // Data Direction Register for P0MUX
#define P0MUX_PORT PORTB  // Port Register for P0MUX
#define P0MUX_PIN  0x00   // Pin for P0MUX
#define P1MUX_DDR  DDRB   // Data Direction Register for P1MUX
#define P1MUX_PORT PORTB  // Port Register for P1MUX
#define P1MUX_PIN  0x01   // Pin for P1MUX
#define P2MUX_DDR  DDRB   // Data Direction Register for P2MUX
#define P2MUX_PORT PORTB  // Port Register for P2MUX
#define P2MUX_PIN  0x02   // Pin for P2MUX

// preprocessor directives as identifiers
// for P0MUX, P1MUX and P2MUX pins
#define P0MUX 0
#define P1MUX 1
#define P2MUX 2

// volatile global variables for ADC results
volatile uint16_t adc_emgOpen;         // EMG signal (open direction)
volatile uint16_t adc_emgClose;       // EMG signal (close direction)
volatile uint16_t adc_motorREV;        // motor voltage (revolutions)
volatile uint16_t adc_motorTorque;     // motor current (torque)
volatile uint16_t adc_sensorThumb[3]; // sensor in thumb
volatile uint16_t adc_sensorWrist;     // clip strain sensor in wrist

// ADC functions family
void adcInit (void);
uint8_t adc (void);
uint8_t adcChannel (uint8_t);
uint8_t adcThumbSensorMUX (uint8_t);
void adcDisplay (void);
```

Codeausschnitt 3.9: adc.h

In Zeilen 1 bis 4 in Codeausschnitt 3.9 werden vier Bibliotheken eingebunden, die für die Ausgabe der digitalisierten Werte auf dem Display benötigt werden. Die Bibliothek *stdlib.h* ist eine C-Standard-Bibliothek und stellt die Funktion *itoa()* zur Verfügung. Mit dieser Funktion können die im Datentyp Integer erfassten digitalisierten Werte in eine Zeichenkette gewandelt werden um diese dann auf dem Display auszugeben. Die Bibliothek *string.h* ist ebenfalls C-Standard und stellt Funktionen zur vereinfachten Bearbeitung von Zeichenkeiten (wie etwa kopieren oder verketten) zur Verfügung. Diese Funktionen unterstützen das Modul *adc* im Aufbau der am Display ausgegebenen Zeichenketten. Die Bibliothek *mylcd.h* wurde von Andre Fabricius unter der GNU General Public License veröffentlicht. Sie stellt Funktionen zur Verfügung, mit denen das Display angesteuert werden kann. Die Bibliothek *font5x8.h* wurde von Dr. Merwa, dem Projektbetreuer, entwickelt und zur Vefügung gestellt. Sie stellt eine Schriftart zur Anzeige von Text auf dem Display zur Verfügung.

In den Zeilen 7 bis 12 werden Identifiers für die ADC-Kanäle 0 bis 5 (Port A Pins 0 bis 5) vergeben. In den Zeilen 16 bis 24 kann mit Hilfe von Präprozessorvariablen eingestellt werden, welche Pins des Mikrocontrollers als *P0MUX*, *P1MUX* und *P2MUX* (s. Kapitel 2.2.3) fungieren sollen. Diese Pins erhalten in den Zeilen 28 bis 30 Identifier.

In den Zeilen 33 bis 38 werden sechs Variablen (eine davon als Array) deklariert, in denen die Resultate der A/D-Wandlung gespeichert werden. Es handelt sich um globale und volatile Variablen. So wird erreicht, dass die digitalisierten Werte pauschal jedem Software-Modul und auch jeder Interrupt-Service-Routine zur Verfügung stehen. In den Zeilen 41 bis 45 werden 5 Funktionen deklariert, die in der Datei *adc.c* definiert sind.

Codeausschnitt 3.10 aus *adc.c* zeigt die Funktion *adcInit()*. Diese initialisiert den Analog-Digital-Konverter des Mikrocontrollers und die Pins zur Ansteuerung des Daumen-Rutschsensors.

```
1  void adcInit (void)
2  {
3     // initialize ADC unit
4
5        // configure pin AVCC as source for VREF (ADC reference voltage)
6        ADMUX &=~ (1<<REFS1);
7        ADMUX |=  (1<<REFS0);
8
9        // configure ADC prescaler: 64
10       ADCSRA |=  (1<<ADPS2) | (1<<ADPS1);
11       ADCSRA &=~ (1<<ADPS0);
12
13       // enable ADC unit
14       ADCSRA |= (1<<ADEN);
15
16       // perform dummy conversion
17       ADCSRA |= (1<<ADSC);            // start dummy conversion
18       while ( ADCSRA & (1<<ADSC) ); // wait until conversion is finished
19
20    // initialize PxMUX pins as output pins (output to thumb sensor)
21
22    P0MUX_DDR |=  (1<<P0MUX_PIN); // set data direction register for P0MUX
23    P1MUX_DDR |=  (1<<P1MUX_PIN); // set data direction register for P1MUX
24    P2MUX_DDR |=  (1<<P2MUX_PIN); // set data direction register for P2MUX
25 }
```

Codeausschnitt 3.10: Funktion adcInit() in der Datei adc.c

Zu Beginn initialisiert die Funktion *adcInit()* (Codeausschnitt 3.10) den Analog-Digital-Konverter in den Zeilen 6 bis 18. Gemäß der Dokumentation im Datenblatt des ATmega32 [1] werden Bits so gesetzt, dass als Referenzspannung die Spannung am Pin *AVCC* (auf dem Entwicklungsboard kurzgeschlossen mit Betriebsspannung *VCC*, ca. 5 Volt) verwendet wird. Als Prescaler für den Takt des sukzessiven Approximationsregisters wird 64 eingestellt. Denn der Takt des Approximationsregisters soll für gute Ergebnisse zwischen 50 kHz und 200 kHz liegen, um gut aufgelöste Resultate (10 bit) zu erhalten. Mit einem Prescaler von 64 und einem CPU-Takt von 8 MHz stellt sich so ein Takt von 125 kHz ein. Der nächst niedrigere Prescaler 32 reicht nicht aus, um unter 200 kHz zu bleiben. Nach der Initialisierung wird eine dummy-Wandlung vorgenommen (Zeilen 21 und 22), denn nach dem Initialisieren des A/D-Wandlers ist laut Datenblatt des ATmega die erste Wandlung zu verwerfen. Abschließend setzt die Funktion adcInit die Pins *P0MUX*, *P1MUX* und *P2MUX* im Datenrichtungsregister auf Ausgang.

Codeausschnitt 3.11 zeigt die Funktion *adc()*. Diese digitalisiert (pro Ausführung einmal) alle zu erfassenden Signale. Dabei steuert es auch den Rutschsensor (wie in Kapitel 2.2.3 beschrieben) auf drei verschiedene Weisen an und schreibt die drei digitalisierten Werte in ein dem Rutschsensor zugeordneten Array.

```
1  void adc (void)
2  {
3     uint8_t errorlevel = 0; // declare errorlevel, initialize it with 0
4
5        // acquire motor revolutions (motor voltage) on ADC2
6     adcChannel(ADC2);                  // select pin ADC2 as ADC-channel
7     ADCSRA |= (1<<ADSC);               // start conversion
8     while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
9     adc_motorREV = ADCW;               // write result to global variable
10
11    // acquire EMG signal (open direction) on ADC0
12    adcChannel(ADC0);                  // select pin ADC0 as ADC-channel
13    ADCSRA |= (1<<ADSC);               // start conversion
14    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
15    adc_emgOpen = ADCW;                // write result to global variable
16
17    // acquire EMG signal (close direction) on ADC1
18    adcChannel(ADC1);                  // select pin ADC1 as ADC-channel
19    ADCSRA |= (1<<ADSC);               // start conversion
20    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
21    adc_emgClose = ADCW;               // write result to global variable
22
23    // acquire motor torque (motor current) on ADC3
24    adcChannel(ADC3);                  // select pin ADC3 as ADC-channel
25    ADCSRA |= (1<<ADSC);               // start conversion
26    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
27    adc_motorTorque = ADCW;            // write result to global variable
28
29    // acquire thumb pressure (film pressure sensor) on ADC4
30
31    adcChannel(ADC4);                  // select pin ADC4 as ADC-channel
32
33    // read star-connected resistor 0 (P0MUX) in thumb pressure sensor
34    adcThumbSensorMUX(P0MUX);          // set P0MUX to HIGH
35    ADCSRA |= (1<<ADSC);               // start conversion
36    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
37    adc_sensorThumb[0] = ADCW;         // write result to global variable
38
39    // read star-connected resistor 1 (P1MUX) in thumb pressure sensor
40    adcThumbSensorMUX(P1MUX);          // set P1MUX to HIGH
41    ADCSRA |= (1<<ADSC);               // start conversion
42    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
43    adc_sensorThumb[1] = ADCW;         // write result to global variable
44
45    // read star-connected resistor 2 (P2MUX) in thumb pressure sensor
46    adcThumbSensorMUX(P2MUX);          // set P2MUX to HIGH
47    ADCSRA |= (1<<ADSC);               // start conversion
48    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
49    adc_sensorThumb[2] = ADCW;         // write result to global variable
50
51    // acquire wrist strain (clip strain sensor) on ADC5
52    adcChannel(ADC5);                  // select pin ADC5 as ADC-channel
53    ADCSRA |= (1<<ADSC);               // start conversion
54    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
55    adc_sensorWrist = ADCW;            // write result to global variable
56 }
```

Codeausschnitt 3.11: Funktion adc() in der Datei adc.c

Dabei ist der Vorgang der A/D-Wandlung eines einzelnen Signals immer der selbe: Zuerst wird der jeweilige ADC-Kanal, an dem das Signal anliegt, ausgewählt. Dann wird das “start conversion”-Bit *ADSC* gesetzt. Mit einer while-Schleife wird

gewartet, bis dieses Bit erlöscht. Dann ist die Konvertierung beendet und kann mit Hilfe des Makros *ADCW* ausgelesen und in die jeweilige globale Variable geschrieben werden.

Der ADC-Konverter liefert Resultate mit 10 bit, das heisst, die digitalisierten Werte bewegen sich linear in einem Bereich von 0 (0 Volt) bis 1023 (Referenzspannung *VREF*). Formel 3.3 gibt an, welches Resultat *ADCW* der A/D-Wandler in Abhängigkeit von der digitalisierten Spannung und der Referenzspannung liefert.

$$ADCW = \frac{V_{Signal}}{VREF} \cdot 1023 \tag{3.3}$$

Ausserdem ist zu erkennen, dass die Funktion *adc()* immer zuerst das Signal der Motordrehzahl tastet. Dies vereinfacht es dem Modul *prosthesiscontrol*, die Zeit zwischen Umschalten des Motors in den Messmodus und Abtastung des Drehzahlsignals auf exakt 2 Millisekunden einzurichten.

Codeausschnitt 3.12 zeigt die Funktion *adcChannel()*. Diese stellt einen ADC Kanal ein, sodass der A/D-Konverter dann das Signal auf diesem Kanal tasten kann. Dabei kann als Argument der Indentifier des ADC-Kanals übergeben werden.

```
1  uint8_t adcChannel (uint8_t adc_channel)
2  {
3     uint8_t errorlevel = 0; // declare errorlevel, initialize it with 0
4
5      /* selected channel -> required MUX4:0 (in register ADMUX)
6       * ADC0 -> 00000
7       * ADC1 -> 00001
8       * ADC2 -> 00010
9       * ADC3 -> 00011
10      * ADC4 -> 00100
11      * ADC5 -> 00101
12      */
13
14    // select channel (set MUX bits in ADMUX)
15    switch (adc_channel)
16    {
17       case ADC0:
18          ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX2) | (1<<MUX1) | (1<<MUX0) )
            ;
19       break;
20
21       case ADC1:
22          ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX2) | (1<<MUX1) );
23          ADMUX |=  (1<<MUX0);
24       break;
25
26       case ADC2:
27          ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX2) | (1<<MUX0) );
28          ADMUX |=  (1<<MUX1);
29       break;
30
31       case ADC3:
32          ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX2) );
33          ADMUX |=  (1<<MUX1) | (1<<MUX0);
34       break;
35
36       case ADC4:
37          ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX1) | (1<<MUX0) );
38          ADMUX |=  (1<<MUX2);
39       break;
40
41       case ADC5:
42          ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX1) );
43          ADMUX |=  (1<<MUX2) | (1<<MUX0);
```

```
        break;

        // for invalid argument: set errorlevel to 1
        default: errorlevel = 1; break;
    }

    // quit by returning errorlevel
    return errorlevel;
}
```

Codeausschnitt 3.12: Funktion adcChannel() in der Datei adc.c

Codeausschnitt 3.13 zeigt die Funktion *adcThumbSensorMux()*. Diese unterstützt die Funktion *adc()* beim Auslesen des Daumen-Rutschsensors, indem sie die Pins P0MUX, *P1MUX* und *P2MUX* so schaltet, dass der gewünschte Dehnungsmessstreifen selektiert wird.

```
uint8_t adcThumbSensorMUX (uint8_t PxMUX)
{
    uint8_t errorlevel = 0; // declare errorlevel, initialize it with 0

    switch (PxMUX)
    {
        case P0MUX:

            P0MUX_PORT |=  (1<<P0MUX_PIN); // set P0MUX to HIGH
            P1MUX_PORT &=~ (1<<P1MUX_PIN); // set P1MUX to LOW
            P2MUX_PORT &=~ (1<<P2MUX_PIN); // set P2MUX to LOW

        break;

        case P1MUX:

            P0MUX_PORT &=~ (1<<P0MUX_PIN); // set P0MUX to LOW
            P1MUX_PORT |=  (1<<P1MUX_PIN); // set P1MUX to HIGH
            P2MUX_PORT &=~ (1<<P2MUX_PIN); // set P2MUX to LOW

        break;

        case P2MUX:

            P0MUX_PORT &=~ (1<<P0MUX_PIN); // set P0MUX to LOW
            P1MUX_PORT &=~ (1<<P1MUX_PIN); // set P1MUX to LOW
            P2MUX_PORT |=  (1<<P2MUX_PIN); // set P2MUX to HIGH

        break;

        // for invalid argument: set errorlevel to 1
        default: errorlevel = 1; break;
    }

    // quit by returning errorlevel
    return errorlevel;
}
```

Codeausschnitt 3.13: Funktion adcThumbSensorMUX() in der Datei adc.c

Codeausschnitt 3.14 zeigt die Funktion *adcDisplay()*. Diese zeigt die zuletzt digitalisierten Werte auf dem Display an. Hierfür wird die Funktion *itoa()* zur Wandlung von Integer-Werten in Zeichenketten, die Funktionen *strcpy()* und *strcat()* zum Kopieren und Verketten von Zeichenketten und die Funktionen *lcd_set_cursor()* sowie *lcd_puts()* zur Ausgabe von Zeichenketten auf dem Display eingesetzt.

```
void adcDisplay (void)
{
    char s_emgOpen[25], s_emgClose[25], s_motorREV[25], s_motorTorque[25],
```

```
4         s_sensorThumb[25], s_sensorWrist[25];
5     char s_itoa_buffer[5]; // string used to buffer results of itoa()
6 
7     lcd_clear(); // clear display
8 
9     strcpy(s_emgOpen, "EMG open: ");
10    strcpy(s_emgClose, "EMG close: ");
11    strcpy(s_motorREV, "revolutions: ");
12    strcpy(s_motorTorque, "torque: ");
13    strcpy(s_sensorThumb, "thumb: ");
14    strcpy(s_sensorWrist, "wrist: ");
15 
16    itoa(adc_emgOpen,s_itoa_buffer,10);
17    strcat(s_emgOpen,s_itoa_buffer);
18 
19    itoa(adc_emgClose,s_itoa_buffer,10);
20    strcat(s_emgClose,s_itoa_buffer);
21 
22    itoa(adc_motorREV,s_itoa_buffer,10);
23    strcat(s_motorREV,s_itoa_buffer);
24 
25    itoa(adc_motorTorque,s_itoa_buffer,10);
26    strcat(s_motorTorque,s_itoa_buffer);
27 
28    itoa(adc_sensorThumb[0],s_itoa_buffer,10);
29    strcat(s_sensorThumb,s_itoa_buffer);
30    strcat(s_sensorThumb," ");
31    itoa(adc_sensorThumb[1],s_itoa_buffer,10);
32    strcat(s_sensorThumb,s_itoa_buffer);
33    strcat(s_sensorThumb," ");
34    itoa(adc_sensorThumb[2],s_itoa_buffer,10);
35    strcat(s_sensorThumb,s_itoa_buffer);
36 
37    itoa(adc_sensorWrist,s_itoa_buffer,10);
38    strcat(s_sensorWrist,s_itoa_buffer);
39 
40    // write values to display
41    lcd_set_cursor(0,0);
42    lcd_puts(font5x8,s_emgOpen);
43    lcd_set_cursor(0,9);
44    lcd_puts(font5x8,s_emgClose);
45    lcd_set_cursor(0,18);
46    lcd_puts(font5x8,s_motorREV);
47    lcd_set_cursor(0,27);
48    lcd_puts(font5x8,s_motorTorque);
49    lcd_set_cursor(0,36);
50    lcd_puts(font5x8,s_sensorThumb);
51    lcd_set_cursor(0,45);
52    lcd_puts(font5x8,s_sensorWrist);
53 }
```

Codeausschnitt 3.14: Funktion adcDisplay() in der Datei adc.c

3.3.2 Modultest

Der Test des Moduls *adc* wird mit Hilfe eines auf dem Mikrocontroller ausgeführten Hauptprogramms (Codeausschnitt 3.15) vorgenommen. Dieses initialisiert das Modul *adc* und das Display. In einer Endlosschleife igitalisiert es dann mit der Funktion *adc()* alle Werte, zeigt diese auf dem Display und und wartet 200 Millisekunden.

```
1 #define F_CPU 8000000          // CPU frequency (macro for delay.h)
2 
3 #include <avr/io.h>            // AVR device-specific IO definitions
4 #include <util/delay.h>        // functions for delay loops
5 
6 #include <adc/adc1.0/adc.h>    // include adc module (version 1.0)
```

```
7
8  int main (void)
9  {
10     adcInit();
11
12     lcd_init();
13
14     while (1)
15     {
16         adc();
17
18         adcDisplay();
19
20         _delay_ms (200);
21     }
22
23     return 0;
24 }
```

Codeausschnitt 3.15: Hauptprogramm für den Modultest von adc

Während das Hauptprogramm für den Test des Moduls *adc* vom Mikrocontroller ausgeführt wird, werden unterschiedliche Spannungen an die ADC-Kanäle (Pins ADC0 bis ADC5) angelegt und das Resultat auf dem Display mit dem erwarteten Wert abgeglichen. Ausserdem wird überprüft, ob die Werte auch korrekt gewandelt werden, wenn an mehreren Kanälen gleichzeitig unterschiedliche Spannungen anliegen. Tabelle 3.2 dokumentiert einige Messwerte aus der Testdurchführung.

ADC-Kanal	VREF	angelegte Spannung	erwarteter Wert	abgelesener Wert
ADC0	4,7 V	0,0 V	0	1
ADC0	4,7 V	2,0 V	435	438
ADC0	4,7 V	4,7 V	1023	1020
ADC1	4,7 V	0,0 V	0	2
ADC1	4,7 V	2,0 V	435	438
ADC1	4,7 V	4,7 V	1023	1022
ADC2	4,7 V	0,0 V	0	0
ADC2	4,7 V	2,0 V	435	436
ADC2	4,7 V	4,7 V	1023	1021
ADC3	4,7 V	0,0 V	0	0
ADC3	4,7 V	2,0 V	435	435
ADC3	4,7 V	4,7 V	1023	1020
ADC4	4,7 V	0,0 V	0	2
ADC4	4,7 V	2,0 V	435	436
ADC4	4,7 V	4,7 V	1023	1021
ADC5	4,7 V	0,0 V	0	1
ADC5	4,7 V	2,0 V	435	438
ADC5	4,7 V	4,7 V	1023	1023

Tabelle 3.2: Testresultat des Modultests adc

Ausserdem wurde das Verhalten der Pins *P0MUX*, *P1MUX* und *P2MUX* mit dem Oszilloskop überprüft. Abb. 3.5 zeigt die Spannung an *P0MUX* und P1MUX. Nicht

abgebildet ist die Betrachtung des Pins *P2MUX*. Dieses war immer im Anschluss an *P0MUX* und *P1MUX* auf dem Pegel *HIGH*.

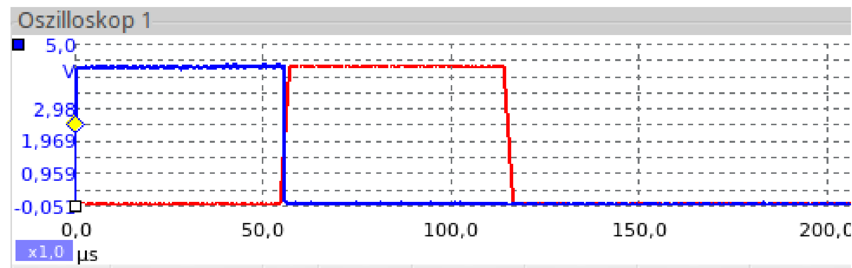


Abbildung 3.5: Spannungsverlauf der Pins P0MUX und P1MUX im Modultest adc

3.4 statemachine

3.4.1 Implementierung des Moduls

Das Modul statemachine implementiert den adaptierten Zustandsautomaten aus der Modul-Spezifikation in Kapitel 2.2.4. Da es ebenfalls ein Interface bestehend aus einer Funktionsfamilie sowie auf eine Vielzahl von einstellbaren Parametern angewiesen ist, wurde auch dieses Modul als Funktionsbibliothek entwickelt. Codeausschnitt 3.16 zeigt die Header-Datei *statemachine.h*.

```
1  #include <lcd/mylcd.h>    // interface for display control
2  #include <string.h>       // library for string operations
3  #include <stdlib.h>       // required for conversion integer to string
4  #include <lcd/font5x8.h> // font for display
5
6  // include motor module (version 1.0) for motor operation modes
7  #include <motor/motor1.0/motor.h>
8
9  // thresholds for state machine transition conditions
10 #define THRES_EMG_OPEN       700  // threshold for emg in open direction
11 #define THRES_EMG_CLOSE      700  // threshold for emg in close direction
12 #define THRES_SW_OVF         61   // 61 x 8.192 ms = 499.712 ms
13 #define THRES_SW_CNT         9    //  9 x 0.032 ms =   0.288 ms
14 #define THRES_MOTOR_REV      200  // threshold for motor revolutions
15 #define THRES_MOTOR_TORQUE   900  // threshold for motor torque
16
17 // preprocessor directives for stopwatch modes
18 #define SW_STOPPED 0  // stopwatch stopped
19 #define SW_RUNNING 1  // stopwatch running
20
21 // enumeration of state names
22 enum stateName { Init, TimerOpen, TimerClose, Open, Close };
23
24 // bundle state variables in a struct
25 struct state
26 {
27    enum stateName name;
28    uint8_t stopwatch;
29    uint8_t motorMode;
30    double motorUtil;
31 };
32
33 // statemachine functions family
34 struct state statemachineRefresh
35               (struct state, uint16_t, uint16_t, uint8_t,
```

```
36                  uint8_t, uint16_t, uint16_t);
37  void statemachineDisplay (struct state);
```

Codeausschnitt 3.16: statemachine.h

In den Zeilen 1 bis 4 bedient sich das Modul *statemachine* denselben Bibliotheken, wie das Modul *adc*, um Zeichenketten auf dem Display auszugeben. Da die Zustandsvariable *motorMode* des Zustandsautomaten mit Identifiern der Motorbetriebsmodi arbeitet, wird das Modul motor in Zeile 7 eingebunden. In den Zeilen 10 bis 15 werden die Thresholds des Zustandsautomaten als Präprozessorvariable definiert. Darauf folgen in den Zeilen 18 und 19 die zwei Indentifier für den Betriebsmodus der Stoppuhr. Wie bereits in der ersten Bachelor-Arbeit [2] beschrieben werden in der Zeile 22 alle Zustandsnamen in einer Enumeration und alle Zustandsvariablen in einem Struct zusammengefasst (Zeilen 25 bis 31). Ebenso den Ausarbeitungen aus der ersten Bachelor-Arbeit folgend wird eine Funktion zur Aktualisierung des Zustands deklariert (Zeilen 34 bis 36), welcher der aktuelle Zustand sowie alle Verlaufsvariablen übergeben werden. Darüber hinaus wird in Zeile 37 die Funktion, welche den als Argument übergebenen Zustand auf dem Display anzeigt, deklariert.

Da sie sehr lang ist, ist die Funktion *statemachineRefresh()* in Codeausschnitt 3.17 nur in Auszügen dargestellt. Die volle Version ist im Anhang zu finden.

```
1   struct state statemachineRefresh
2                (struct state currentState, uint16_t emgOpen,
3                 uint16_t emgClose, uint8_t stopwatchOVF,
4                 uint8_t stopwatchCNT, uint16_t motorREV,
5                 uint16_t motorTorque)
6   {
7      switch (currentState.name)
8      {
9         case Init:
10
11           if ( emgOpen > THRES_EMG_OPEN )
12           {
13              // enter state TimerOpen
14              currentState.name = TimerOpen;
15              currentState.stopwatch = SW_RUNNING;
16           }
17
18           else if ( emgClose > THRES_EMG_CLOSE )
19           {
20              // enter state TimerClose
21              currentState.name = TimerClose;
22              currentState.stopwatch = SW_RUNNING;
23           }
24
25        break;
26
27        /* --- gekürzt --- */
28
29        case Close:
30
31           if ( emgClose <= THRES_EMG_CLOSE || motorREV <= THRES_MOTOR_REV ||
               motorTorque >= THRES_MOTOR_TORQUE )
32           {
33              // enter state Init
34              currentState.name = Init;
35              currentState.motorMode = MOTOR_STOP;
36              currentState.motorUtil = 0.0;
37              currentState.stopwatch = SW_STOPPED;
38           }
39
40           else
41           {
```

```
42              // enter state Close (again)
43              currentState.name = Close;
44              currentState.motorMode = MOTOR_CLOSE;
45              currentState.motorUtil = (double) (emgClose/1023.0);
46           }
47
48        break;
49
50        default:
51
52           // enter state Init
53           currentState.name = Init;
54           currentState.motorMode = MOTOR_STOP;
55           currentState.motorUtil = 0.0;
56           currentState.stopwatch = SW_STOPPED;
57
58        break;
59     }
60
61     return currentState;
62  }
```

Codeausschnitt 3.17: statemachineRefresh() in statemachine.c

Der Ablauf der Funktion *refreshDisplay()* ist so gestaltet, wie in der ersten Bachelorarbeit schematisch vorgezeichnet. Mit Hilfe einer switch-case-Anweisung werden die Transitionsbedingungen des aktuellen Zustands selektiert und anhand der übergebenen Verlaufsgrößen geprüft. Sind sie erfüllt, wird der neue Zustand betreten, indem die Zustandsvariablen neu gesetzt und der gesamte Zustand abschließend als aktueller Zustand zurückgegeben wird.

Codeausschnitt 3.18 zeigt die Funktion *statemachineDisplay()*, mit welcher der als Argument übergebene Zustand auf dem Display angezeigt wird.

```
1  void statemachineDisplay (struct state currentState)
2  {
3     char s_name[25], s_stopwatch[25], s_motorMode[25], s_motorUtil[25];
4     char s_itoa_buffer[4]; // string used to buffer results of itoa()
5     uint8_t i_motorUtil;
6
7     lcd_clear(); // clear display
8
9     strcpy(s_name,       "name:  ");
10    strcpy(s_stopwatch,  "stopw: ");
11    strcpy(s_motorMode,  "mMode: ");
12    strcpy(s_motorUtil,  "mUtil: ");
13
14    switch (currentState.name)
15    {
16       case Init:        strcat(s_name,"Init");       break;
17       case TimerOpen:   strcat(s_name,"TimerOpen");  break;
18       case TimerClose:  strcat(s_name,"TimerClose"); break;
19       case Open:        strcat(s_name,"Open");       break;
20       case Close:       strcat(s_name,"Close");      break;
21       default:          strcat(s_name,"INVALID");    break;
22    }
23
24    switch (currentState.stopwatch)
25    {
26       case SW_RUNNING: strcat(s_stopwatch,"SW_RUNNING"); break;
27       case SW_STOPPED: strcat(s_stopwatch,"SW_STOPPED"); break;
28       default:         strcat(s_stopwatch,"INVALID");    break;
29    }
30
31    switch (currentState.motorMode)
32    {
33       case MOTOR_STOP:    strcat(s_motorMode,"MOTOR_STOP");    break;
```

```
34        case MOTOR_OPEN:     strcat(s_motorMode,"MOTOR_OPEN");     break;
35        case MOTOR_CLOSE:    strcat(s_motorMode,"MOTOR_CLOSE");    break;
36        case MOTOR_MEASURE:  strcat(s_motorMode,"MOTOR_MEASURE");  break;
37        default:             strcat(s_motorMode,"INVALID");        break;
38    }
39
40    i_motorUtil = (uint8_t) ( currentState.motorUtil * 100.0 );
41    itoa(i_motorUtil,s_itoa_buffer,10);
42    strcat(s_motorUtil,s_itoa_buffer);
43    strcat(s_motorUtil," %");
44
45    // write values to display
46    lcd_set_cursor(0,0);
47    lcd_puts(font5x8,"current state:");
48    lcd_set_cursor(3,10);
49    lcd_puts(font5x8,s_name);
50    lcd_set_cursor(3,20);
51    lcd_puts(font5x8,s_stopwatch);
52    lcd_set_cursor(3,30);
53    lcd_puts(font5x8,s_motorMode);
54    lcd_set_cursor(3,40);
55    lcd_puts(font5x8,s_motorUtil);
56 }
```

Codeausschnitt 3.18: statemachineDisplay() in statemachine.c

3.4.2 Modultest

Das Modul statemachine wird getestet, indem der Zustandsautomat in einem Testprogramm betrieben wird. Dabei werden die Verlaufsvariablen gezielt simuliert. Über die Anzeige des Zustands auf dem Display wird beobachtet, ob der Zustandsautomat zu jedem Simulationsschritt den Zustand betritt, der gemäß der Spezifikation betreten werden soll.

Codeausschnitt 3.19 zeigt das Testprogramm.

```
1  #define F_CPU 8000000      // CPU frequency (macro for delay.h)
2
3  #include <avr/io.h>       // AVR device-specific IO definitions
4  #include <util/delay.h>   // functions for delay loops
5
6  // include state machine module (version 1.0)
7  #include <statemachine/statemachine1.0/statemachine.h>
8
9  // define environment simulation
10
11 #define SIM_STEPS 27   // number of environment simulation steps
12
13 uint16_t simulated_emgOpen[SIM_STEPS] =
14 { 0,    // simulation step 0
15   500,  // simulation step 1
16   0,    // simulation step 2
17   800,  // and so on ...
18   800, 500, 0, 0, 0, 1000, 1000, 1000, 1000, 150, 500, 500, 500, 500,
19   1000, 1000, 1000, 800, 800, 500, 500, 500, 500 };
20
21 uint16_t simulated_emgClose[SIM_STEPS] =
22 { 0, 0, 500, 0, 0, 0, 800, 800, 500, 500, 500, 500, 500, 1000, 1000,
23   1000, 1000, 1000, 150, 500, 500, 500, 500, 1000, 1000, 800, 800 };
24
25 uint8_t  simulated_stopwatchOVF[SIM_STEPS] =
26 { 0, 0, 0, 0, 0, 100, 0, 0, 100, 0, 30, 100, 123, 123, 0, 30, 100, 123,
27   123, 0, 100, 123, 123, 0, 100, 123, 123 };
28
29 uint16_t simulated_stopwatchCNT[SIM_STEPS] =
30 { 0, 0, 0, 0, 0, 1000, 0, 0, 1000, 0, 55, 100, 123, 123, 0, 55, 100,
```

```
31    123, 123, 0, 100, 123, 123, 0, 100, 123, 123 };
32
33  uint16_t simulated_motorREV[SIM_STEPS] =
34  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 850, 850, 0, 0, 0, 850, 850, 0,
35    0, 600, 10, 0, 0, 600, 10 };
36
37  uint16_t simulated_motorTorque[SIM_STEPS] =
38  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 300, 300, 0, 0, 0, 300, 300, 0,
39    0, 200, 1000, 0, 0, 200, 1000 };
40
41  int main(void)
42  {
43      uint8_t simStep=0;     // step of environment variables simulation
44      char s_simStep[25];     // string to present simStep on display
45      char s_itoa_buffer[5]; // string to buffer results of itoa()
46
47      // declare a test state
48
49      struct state testState;
50
51      // declare environment variables
52
53      uint16_t emgOpen;       // current EMG level for opening direction
54      uint16_t emgClose;      // current EMG level for closing direction
55      uint8_t  stopwatchOVF; // timer overflows since T/C1 has been reset
56      uint16_t stopwatchCNT; // current counter value of T/C1
57      uint16_t motorREV;      // current motor voltage (motor revolutions)
58      uint16_t motorTorque;  // current motor current (motor torque)
59
60      // initialize system components
61
62      /* In practice, all system components will be initialized here.
63       * In this module test, the only system component is the display.
64       */
65
66      lcd_init(); // initialize display
67
68      // initialize state machine with state Init
69
70      testState.name = Init;
71      testState.motorMode = MOTOR_STOP;
72      testState.motorUtil = 0.0;
73      testState.stopwatch = SW_STOPPED;
74
75      // run state machine in endless loop
76
77      while (1)
78      {
79          // acquire current environment variables
80
81          /* In practice, the environment variables will be delivered by
82           * the system components.
83           * In this module test, the environment variables are simulated.
84           * Therefore, the values for the current simulation simulation
85           * step are used.
86           */
87
88          emgOpen      = simulated_emgOpen      [simStep];
89          emgClose     = simulated_emgClose     [simStep];
90          stopwatchOVF = simulated_stopwatchOVF [simStep];
91          stopwatchCNT = simulated_stopwatchCNT [simStep];
92          motorREV     = simulated_motorREV     [simStep];
93          motorTorque  = simulated_motorTorque  [simStep];
94
95          // refresh state with current environment variables
96
97          testState = statemachineRefresh (testState, emgOpen, emgClose,
98                                            stopwatchOVF, stopwatchCNT,
99                                            motorREV, motorTorque);
100
```

```
101         // set system components according to current state
102
103         /* In practice, all system components will be set here
104          * (according to current state).
105          * In this module test, the only system component is the
106          * display.
107          */
108
109         statemachineDisplay(testState); // present current state on display
110
111         // for module test only: present simStep on display and wait
112
113         itoa(simStep,s_itoa_buffer,10); // convert simStep to string
114
115         strcpy(s_simStep,"SIMULATION STEP: "); // assemble s_simStep
116         strcat(s_simStep,s_itoa_buffer);        // assemble s_simStep
117
118         lcd_set_cursor(7,53);         // set cursor on display
119         lcd_puts(font5x8,s_simStep); // present s_simStep on display
120
121         _delay_ms(5000); // wait 5 seconds
122
123         // for module test only: go to next environment simulation step
124
125         simStep++;
126
127         if ( simStep >= SIM_STEPS )
128         {
129             simStep = 0;
130         }
131     }
132
133     return 0;
134 }
```

Codeausschnitt 3.19: Testprogramm für das Modul statemachine

In Zeilen 13 bis 39 werden Arrays definiert, welche die simulierten Werte der Verlaufsvariablen bereitstellen. Jede Spalte entspricht einem Simulationsschritt. Im Hauptprogramm wird der Zustandsautomat initialisiert. Dann wird der Zustandsautomat alle fünf Sekunden mit dem jeweils nächsten Simulationsschritt aktualisiert und der aktuelle Zustand auf dem Display (incl. Nummer des Simulationsschrittes) ausgegeben.

Das Software-Modul zeigte für jeden der 27 Simulationsschritte den erwarteten Zustand auf dem Display an. Auch die einzelnen Zustandsgrößen entsprachen ohne Ausnahme den zuvor festgelegten Erwartungen. Abbildung zeigt die Anzeige eines Simulationsschrittes auf dem Display.

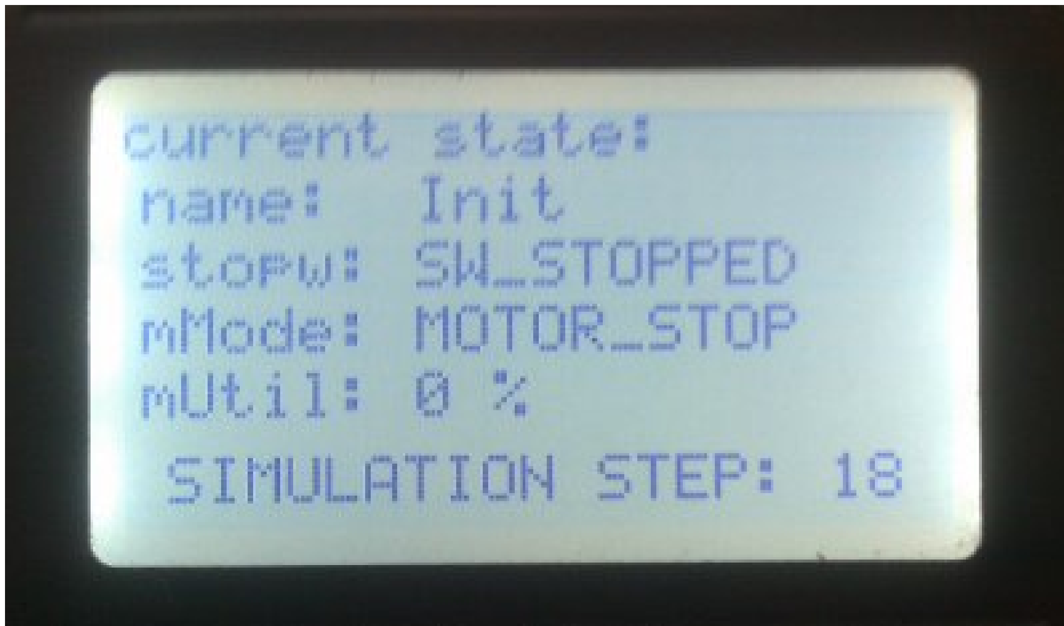


Abbildung 3.6: Display während der Simulation im Modultest statemachine

3.5 prosthesiscontrol

3.5.1 Implementierung des Moduls

Dieses Modul wurde nicht, wie die anderen, als Funktionsbibliothek entwickelt. Es stellt das Hauptprogramm, welches vom Mikrocontroller ausgeführt wird und die Prothesensteuerung betreibt. Es betreibt den Zustandsautomaten, stößt die Erfassung der Verlaufsgrößen an und steuert alle Systemkomponenten in Abhängigkeit des aktuellen Zustands an. Auf dem Display zeigt es, gemäß Spezifikation, die aktuellen digitalisierten Werte sowie den Namen des aktuellen Zustands.

Programmausschnitt 3.20 zeigt Definitionen und Deklarationen, die dem Hauptprogramm in der Datei *main.c* vorangehen.

```
1  #include <avr/io.h>          // AVR device-specific IO definitions
2  #include <avr/interrupt.h>   // library for interrupt handling
3
4  #include <lcd/mylcd.h>       // interface for display control
5  #include <lcd/font5x8.h>     // font for display
6
7  #include <avr/io.h>          // AVR device-specific IO definitions
8  #define F_CPU 8000000        // CPU frequency (macro for delay.h)
9  #include <util/delay.h>      // functions for delay loops
10
11 // include ADC module, motor module and state machine module
12 #include <adc/adc1.0/adc.h>
13 #include <motor/motor1.0/motor.h>
14 #include <statemachine/statemachine1.0/statemachine.h>
15
16 // DDRs, Ports and Pins of pins for observing ISR execution
17 #define ISR_REFR_DDR   DDRB  // Data Direction Register of pin ISR_REFR
18 #define ISR_REFR_PORT  PORTB // Port Register of pin ISR_REFR
19 #define ISR_REFR_PIN   0x03  // Pin of pin ISR_REFR
20 #define ISR_SWOVF_DDR  DDRB  // Data Direction Register of pin ISR_SWOVF
21 #define ISR_SWOVF_PORT PORTB // Port Register of pin ISR_SWOVF
22 #define ISR_SWOVF_PIN  0x04  // Pin of pin ISR_SWOVF
```

```
23
24 // DDR, Port and Pin for observing duration: motor measure mode <-> ADC
25 #define MOTOR_MEASURE_MODE_DDR  DDRB  // Data Direction Register
26 #define MOTOR_MEASURE_MODE_PORT PORTB // Port Register
27 #define MOTOR_MEASURE_MODE_PIN  0x05  // Pin
28
29 volatile struct  state prosthesisState;  // system state of prosthesis
30 volatile uint8_t stopwatch_overflows;    // stopwatch overflow counter
```

Codeausschnitt 3.20: Definition und Deklarationen des Moduls prosthesiscontrol in main.c

In den Zeilen 12 bis 14 des Codeausschnitts 3.20 werden alle anderen Module eingebunden (*pwm.h* ist in *motor.h* enthalten). In den Zeilen 17 bis 27 werden können Pins eingestellt werden, an denen die Durchführungsdauern der Interrupt-Service-Routinen und die Zeit zwischen Schalten in den Motormessmodus und Abtasten der Motorspannung abgelesen werden können. Der Zustand *prosthesisState*, mit dem die Prothesensteuerung vorgenommen wird, ist in Zeile 29 global und volatile deklariert. So können auch Interrupt-Service-Routinen auf den Zustand zugreifen.

Codeausschnitt 3.21 zeigt das Hauptprogramm des Moduls *prosthesiscontrol*.

```
1  int main (void)
2  {
3     // declare string used to present current prosthesis state on display
4     char s_name[25];
5
6     // initialize system components
7
8        // initialize display
9        lcd_init();
10       lcd_clear();
11
12       // initialize ADC module
13       adcInit();
14
15       // initialize motor module
16       motorInit();
17
18       // initialize system state as Init
19       prosthesisState.name      = Init;
20       prosthesisState.motorMode = MOTOR_STOP;
21       prosthesisState.motorUtil = 0.0;
22       prosthesisState.stopwatch = SW_STOPPED;
23
24       // initialize timer used as stopwatch by state machine module
25       // (T/C2, normal mode, stopped, local interrupt on OVF)
26       TIMSK |= (1<<TOIE2);      // enable local overflow interrupt
27       TCNT2 = 0;                // reset counter
28       stopwatch_overflows = 0; // reset stopwatch overflow counter
29
30       // initialize timer that triggers state refresh every 20 ms
31       // (T/C0, CTC mode, prescaler 1024, interrupt on Compare Match)
32       TCCR0 |= (1<<WGM01);  // CTC mode
33       TCCR0 |= (1<<CS02) | (1<<CS00);  // start T/C0, prescaler 1024
34       OCR0   = 156;  // compare value, Compare Match every 20 ms
35       TIMSK |= (1<<OCIE0);  // enable local Compare Match interrupt
36
37       // initialize pins for observation issues (set DDR to output)
38
39          // pins for observation of ISR execution
40          ISR_REFR_DDR  |= (1<<ISR_REFR_PIN);
41          ISR_SWOVF_DDR |= (1<<ISR_SWOVF_PIN);
42
43          // pin for observation of duration motor measure mode <-> ADC
44          MOTOR_MEASURE_MODE_DDR |= (1<<MOTOR_MEASURE_MODE_PIN);
45
```

```
46     // enable global interrupt to start interrupt handled prosthesis control
47     sei();
48
49
50     // refresh display in endless loop
51     while (1)
52     {
53        // show ADC conversion results on display
54        adcDisplay();
55
56        // show name of current prosthesis state on display
57
58        strcpy(s_name, "state: ");
59
60        switch (prosthesisState.name)
61        {
62           case Init:       strcat(s_name,"Init");       break;
63           case TimerOpen:  strcat(s_name,"TimerOpen");  break;
64           case TimerClose: strcat(s_name,"TimerClose"); break;
65           case Open:       strcat(s_name,"Open");       break;
66           case Close:      strcat(s_name,"Close");      break;
67           default:         strcat(s_name,"INVALID");    break;
68        }
69
70        lcd_set_cursor(10,54);
71        lcd_puts(font5x8,s_name);
72
73        // delay display refresh
74        _delay_ms(400);
75     }
76
77     return 0;
78  }
```

Codeausschnitt 3.21: Hauptprogramm main() von prosthesiscontrol in main.c

So wie schematisch bereits in der ersten Bachelor-Arbeit vorgezeichnet, initialisiert das Hauptprogramm, das den Zustandsautomaten betreibt, zunächst alle Systemkomponenten (Zeile 9 bis 44). Darunter auch den ersten Zustand. Darüber hinaus stellt das Modul *prosthesiscontrol* dem Zustandsautomaten die benötigte Stoppuhr zur Verfügung, indem Timer/Counter 2 initialisiert und dessen Overflow-Interrupt aktiviert wird. Nachdem in Zeile 47 die Interrupts global aktiviert werden, folgt jeden Overflow von Timer/Counter 2 folgt Interrupt-Service-Routine, welche die Overflows zählt. Der Zustand der gesamten Prothese wird mit Hilfe von Timer/Counter 0 aktualisiert. Er hat alle 20 Millisekunden einen CompareMatch. Der dabei fallende Interrupt löst die Interrupt-Service-Routine in Codeausschnitt 3.22 aus. Diese aktualisiert alle Verlaufsgrößen (stößt die A/D-Wandlung aller Signale an), aktualisiert dann den Zustand und steuert abschließend alle Systemkomponenten gemäß aktuellem Zustand an.

```
1  ISR (TIMER0_COMP_vect)
2  {
3     // set pin for observation of this ISR execution to HIGH
4     ISR_REFR_PORT |= (1<<ISR_REFR_PIN);
5
6     // save AVR status register to protect I-bit
7     uint8_t sreg = SREG;
8
9     uint8_t timestamp;
10
11    // refresh ADC values
12
13       // switch motor to measure mode, observation pin to HIGH
14       motorOperation(MOTOR_MEASURE,0.0);
```

```
15        MOTOR_MEASURE_MODE_PORT |= (1<<MOTOR_MEASURE_MODE_PIN);
16
17        // wait 2 ms (until motor voltage is available on ADC channel)
18        timestamp = TCNT0;
19        while ( (TCNT0-timestamp) < 16 );
20
21        // observ. pin to LOW (since adc of revolutions is the very next step)
22        MOTOR_MEASURE_MODE_PORT &=~ (1<<MOTOR_MEASURE_MODE_PIN);
23
24        // perform ADC for all external signals
25        adc();
26
27        // switch motor back to current operation mode
28        motorOperation(prosthesisState.motorMode, prosthesisState.motorUtil);
29
30     // refresh system state of prosthesis
31     prosthesisState = statemachineRefresh
32                       ( prosthesisState, adc_emgOpen, adc_emgClose,
33                         stopwatch_overflows, TCNT2, adc_motorREV,
34                         adc_motorTorque );
35
36     // set system components according to refreshed system state
37
38        // set motor operation
39        motorOperation (prosthesisState.motorMode, prosthesisState.motorUtil);
40
41        // enable or stop/reset stopwatch (T/C2)
42        if (prosthesisState.stopwatch == SW_STOPPED)
43        {
44           // stop T/C2
45           TCCR2 &=~ ( (1<<CS22) | (1<<CS21) | (1<<CS20) );
46
47           // reset T/C2 counter
48           TCNT2 = 0;
49        }
50        else if (prosthesisState.stopwatch == SW_RUNNING)
51        {
52           // enable T/C2 (prescaler 256)
53           TCCR2 |=  (1<<CS22) | (1<<CS21);
54           TCCR2 &=~ (1<<CS20);
55        }
56
57     // restore AVR status register to protect I-bit
58     SREG = sreg;
59
60     // set pin for observation of this ISR execution to LOW
61     ISR_REFR_PORT &=~ (1<<ISR_REFR_PIN);
62  }
63
64  ISR (TIMER2_OVF_vect)
65  {
66     // set pin for observation of this ISR execution to HIGH
67     ISR_SWOVF_PORT |= (1<<ISR_SWOVF_PIN);
68
69     // save AVR status register to protect I-bit
70     uint8_t sreg = SREG;
71
72     // raise stopwatch counter by 1
73     if (stopwatch_overflows < 255)
74        stopwatch_overflows++;
75     else
76        stopwatch_overflows = 0;
77
78     // restore AVR status register to protect I-bit
79     SREG = sreg;
80
81     // set pin for observation of this ISR execution to LOW
82     ISR_SWOVF_PORT &=~ (1<<ISR_SWOVF_PIN);
83  }
```

Codeausschnitt 3.22: Interrupt-Service-Routinen des Moduls prosthesiscontrol in main.c

3.5.2 Modultest

Das Modul wird auf mehrere Varianten getestet.

Zunächst wird überprüft, ob das entwickelte Softwaresystem echtzeitfähig ist. Hierzu werden die Spannungsverläufe an den Pins, welche während der Durchführung der Interrupt-Service-Routinen *HIGH* sind, auf dem Oszilloskop dargestellt (Abb. 3.7).

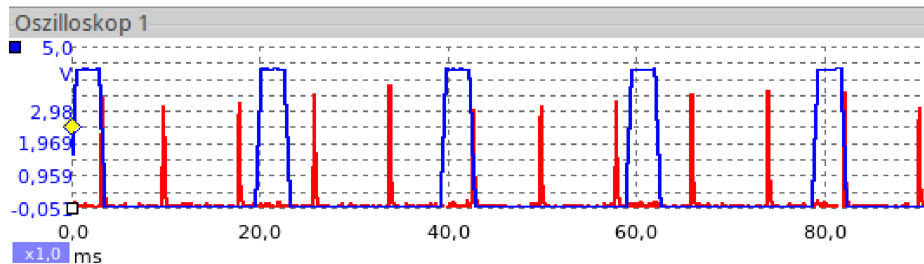
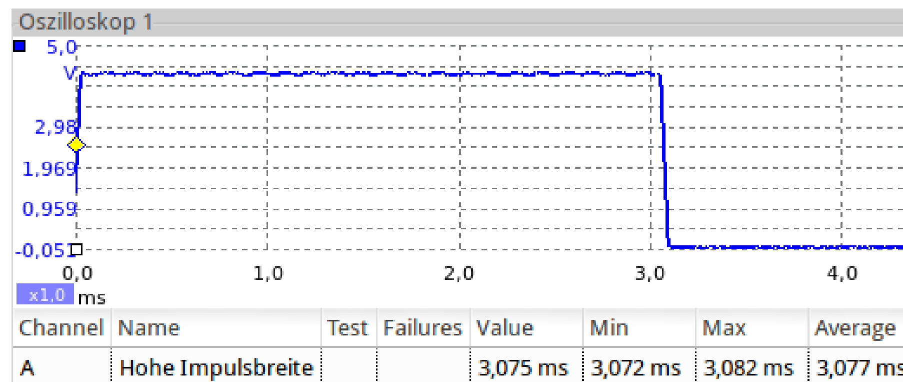


Abbildung 3.7: Durchführung der refresh-Interrupt-Service-Route (blau) und der Stoppuhr-Interrupt-Service-Route (rot)

Es ist in 3.7 deutlich erkennbar, dass die Interrupt-Service-Routine, die das gesamte System aktualisiert, wie gefordert alle 20 Millisekunden durchgeführt wird. Ausserdem ist sie schnell genug vor dem nächsten Interrupt abgearbeitet. Die Stoppuhr-Service-Route kommt ebenfalls in den erwarteten Abständen und mit einer sehr kurzen Durchführungsdauer. Es ist jedoch möglich, dass sie manchmal nicht ausgeführt wird, wenn gerade die refresh-Interrupt-Service-Route ausgeführt wird. Dies stört jedoch die Funktion der Prothese nicht erheblich, im Zweifel wird lediglich etwas länger geprüft, ob das EMG-Signal über dem Threshold ist.



Channel	Name	Test	Failures	Value	Min	Max	Average
A	Hohe Impulsbreite			3,075 ms	3,072 ms	3,082 ms	3,077 ms

Abbildung 3.8: Dauer der refresh-Interrupt-Service-Routine (blau)

Abbildung 3.8zeigt, dass die Durchführung der Interrupt-Service-Routine, die das gesamte System aktualisiert, zwischen 3,0 und 3,1 Millisekunden dauert.

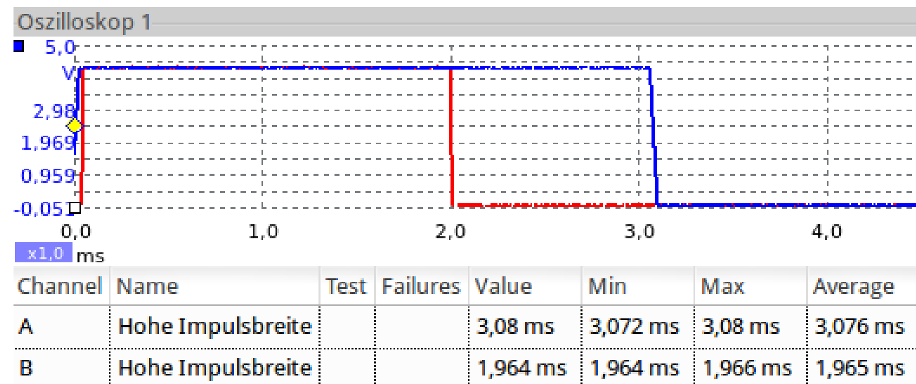


Abbildung 3.9: Dauer der refresh-Interrupt-Service-Routine (blau) und Dauer zwischen Umschalten und Motormessmodus und Abtastung der Motorspannung (rot)

Abb. 3.9zeigt, dass die Zeit zwischen Umschaltung des Motors in den Messmodus und Abtastung der Motorspannung wie gefordert bei 2 Millisekunden liegt.

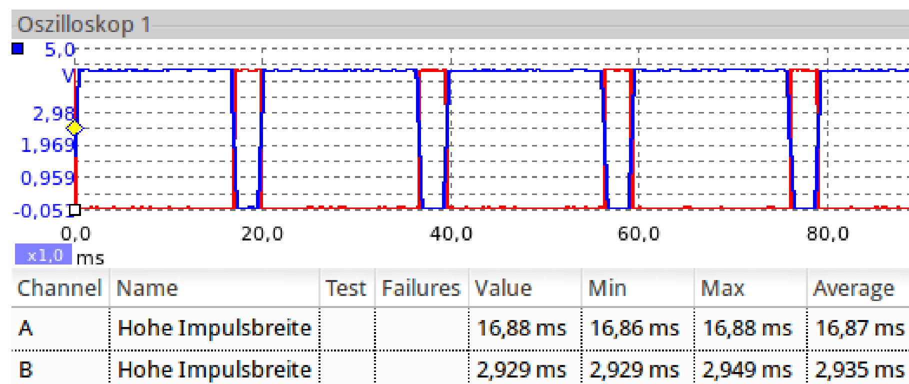


Abbildung 3.10: Verhalten der Pins Motor A (blau) und Motor B (rot) im Zustand Open

4 Verzeichnisse

Literaturverzeichnis

Fußnote	Referenz
[1]	"ATmega32 Data Sheet" Atmel, 2008
[2]	"Kontrollstrategien für myoelektrisch gesteuerte Handprothesen" Ingo Weigel, 2014
[3]	"Sensorschaltung für myoelektrisch gesteuerte Handprothese" Angelika Bauer, 2014

Abbildungsverzeichnis

1.1 "Sensorhand Speed" der Firma Ottobock 8

2.1 Software-Architektur mit Modulen (Kreise) und Interaktionen von Modulen über deren Interfaces (rot) 11
2.2 Form eines PWM-Signals (Duty Cycle 60% in Periode 1 und 40% in Periode 2) 12
2.3 Aufbau und Anschlüsse des Rutschsensors im Daumen, *R1*, *R2* und *R3* stellen Dehnungsmesstreifen dar (entnommen aus *Bauer: Sensorschaltung für myoelektrisch gesteuerte Handprothese* [3]) 15
2.4 Zustandsautomat einer proportionalen Direktsteuerung, Entwurf aus der ersten Bachelorarbeit [2] 16
2.5 An das Projekt angepasster Zustandsautomat einer proportionalen Direktsteuerung 17

3.1 Fast PWM Mode des ATmega32 (entnommen aus *Atmel: ATmega32 Datasheet* [1]) 21
3.2 Phase Correct PWM Mode des ATmega32 (entnommen aus *Atmel: ATmega32 Datasheet* [1]) 22
3.3 Phase and Frequency Correct PWM Mode des ATmega32 (entnommen aus *Atmel: ATmega32 Datasheet* [1]) 23
3.4 Darstellung des Anwendungsfalles 1 (Test des Moduls pwm) auf dem Oszilloskop 28
3.5 Spannungsverlauf der Pins P0MUX und P1MUX im Modultest adc . 40
3.6 Display während der Simulation im Modultest statemachine 46
3.7 Durchführung der refresh-Interrupt-Service-Route (blau) und der Stoppuhr-Interrupt-Service-Route (rot) 50
3.8 Dauer der refresh-Interrupt-Service-Routine (blau) 50
3.9 Dauer der refresh-Interrupt-Service-Routine (blau) und Dauer zwischen Umschalten und Motormessmodus und Abtastung der Motorspannung (rot) 51
3.10 Verhalten der Pins Motor A (blau) und Motor B (rot) im Zustand Open 51

Anhang

Eingesetztes makefile

```
# Hey Emacs, this is a -*- makefile -*-
#
# WinAVR makefile written by Eric B. Weddington, Jörg Wunsch, et al.
# Released to the Public Domain
# Please read the make user manual!
#
# Additional material for this makefile was submitted by:
#  Tim Henigan
#  Peter Fleury
#  Reiner Patommel
#  Sander Pool
#  Frederik Rouleau
#  Markus Pfaff
#
# On command line:
#
# make all = Make software.
#
# make clean = Clean out built project files.
#
# make coff = Convert ELF to AVR COFF (for use with AVR Studio 3.x or VMLAB).
#
# make extcoff = Convert ELF to AVR Extended COFF (for use with AVR Studio
#                4.07 or greater).
#
# make program = Download the hex file to the device, using avrdude.  Please
#                customize the avrdude settings below first!
#
# make filename.s = Just compile filename.c into the assembler code only
#
# To rebuild project do "make clean" then "make all".
#

# mth 2004/09
# Differences from WinAVR 20040720 sample:
# - DEPFLAGS according to Eric Weddingtion's fix (avrfreaks/gcc-forum)
# - F_OSC Define in CFLAGS and AFLAGS


# MCU name
MCU = atmega32

# Main Oscillator Frequency
# This is only used to define F_OSC in all assembler and c-sources.
#F_OSC = 3686400
F_OSC = 8000000

# Output format. (can be srec, ihex, binary)
FORMAT = ihex

# Target file name (without extension).
TARGET = main


# List C source files here. (C dependencies are automatically generated.)
SRC  = $(TARGET).c
```

```
57 SRC += /usr/local/avr/lcd/mylcd.c
58 SRC += /usr/local/avr/adc/adc1.0/adc.c
59 SRC += /usr/local/avr/motor/motor1.0/motor.c
60 SRC += /usr/local/avr/pwm/pwm1.0/pwm.c
61 SRC += /usr/local/avr/statemachine/statemachine1.0/statemachine.c
62
63
64 # List Assembler source files here.
65 # Make them always end in a capital .S.  Files ending in a lowercase .s
66 # will not be considered source files but generated files (assembler
67 # output from the compiler), and will be deleted upon "make clean"!
68 # Even though the DOS/Win* filesystem matches both .s and .S the same,
69 # it will preserve the spelling of the filenames, and gcc itself does
70 # care about how the name is spelled on its command-line.
71 ASRC =
72
73
74
75 # Optimization level, can be [0, 1, 2, 3, s].
76 # 0 = turn off optimization. s = optimize for size.
77 # (Note: 3 is not always the best optimization level. See avr-libc FAQ.)
78 OPT = s
79
80 # Debugging format.
81 # Native formats for AVR-GCC's -g are stabs [default], or dwarf-2.
82 # AVR (extended) COFF requires stabs, plus an avr-objcopy run.
83 #DEBUG = stabs
84 DEBUG = dwarf-2
85
86 # List any extra directories to look for include files here.
87 #     Each directory must be seperated by a space.
88 EXTRAINCDIRS = /usr/local/avr
89
90
91 # Compiler flag to set the C Standard level.
92 # c89   - "ANSI" C
93 # gnu89 - c89 plus GCC extensions
94 # c99   - ISO C99 standard (not yet fully implemented)
95 # gnu99 - c99 plus GCC extensions
96 CSTANDARD = -std=gnu99
97
98 # Place -D or -U options here
99 CDEFS =
100
101 # Place -I options here
102 CINCS =
103
104
105 # Compiler flags.
106 #  -g*:          generate debugging information
107 #  -O*:          optimization level
108 #  -f...:        tuning, see GCC manual and avr-libc documentation
109 #  -Wall...:     warning level
110 #  -Wa,...:      tell GCC to pass this to the assembler.
111 #    -adhlns...: create assembler listing
112 CFLAGS = -g$(DEBUG)
113 CFLAGS += $(CDEFS) $(CINCS)
114 CFLAGS += -O$(OPT)
115 CFLAGS += -funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums
116 CFLAGS += -Wall -Wstrict-prototypes
117 CFLAGS += -Wa,-adhlns=$(<:.c=.lst)
118 CFLAGS += $(patsubst %,-I%,$(EXTRAINCDIRS))
119 CFLAGS += $(CSTANDARD)
120 CFLAGS += -DF_OSC=$(F_OSC)
121
122
123
124 # Assembler flags.
125 #  -Wa,...:   tell GCC to pass this to the assembler.
126 #  -ahlms:    create listing
```

```
127 #   -gstabs:    have the assembler create line number information; note that
128 #               for use in COFF files, additional information about filenames
129 #               and function names needs to be present in the assembler source
130 #               files -- see avr-libc docs [FIXME: not yet described there]
131 ASFLAGS = -Wa,-adhlns=$(<:.S=.lst),-gstabs
132 ASFLAGS += -DF_OSC=$(F_OSC)
133
134
135 #Additional libraries.
136
137 # Minimalistic printf version
138 PRINTF_LIB_MIN = -Wl,-u,vfprintf -lprintf_min
139
140 # Floating point printf version (requires MATH_LIB = -lm below)
141 PRINTF_LIB_FLOAT = -Wl,-u,vfprintf -lprintf_flt
142
143 PRINTF_LIB =
144
145 # Minimalistic scanf version
146 SCANF_LIB_MIN = -Wl,-u,vfscanf -lscanf_min
147
148 # Floating point + %[ scanf version (requires MATH_LIB = -lm below)
149 SCANF_LIB_FLOAT = -Wl,-u,vfscanf -lscanf_flt
150
151 SCANF_LIB =
152
153 MATH_LIB = -lm
154
155 # External memory options
156
157 # 64 KB of external RAM, starting after internal RAM (ATmega128!),
158 # used for variables (.data/.bss) and heap (malloc()).
159 #EXTMEMOPTS = -Wl,-Tdata=0x801100,--defsym=__heap_end=0x80ffff
160
161 # 64 KB of external RAM, starting after internal RAM (ATmega128!),
162 # only used for heap (malloc()).
163 #EXTMEMOPTS = -Wl,--defsym=__heap_start=0x801100,--defsym=__heap_end=0x80ffff
164
165 EXTMEMOPTS =
166
167 # Linker flags.
168 #   -Wl,...:     tell GCC to pass this to linker.
169 #     -Map:      create map file
170 #     --cref:    add cross reference to  map file
171 LDFLAGS = -Wl,-Map=$(TARGET).map,--cref
172 LDFLAGS += $(EXTMEMOPTS)
173 LDFLAGS += $(PRINTF_LIB) $(SCANF_LIB) $(MATH_LIB)
174
175
176
177
178 # Programming support using avrdude. Settings and variables.
179
180 # Programming hardware: alf avr910 avrisp bascom bsd
181 # dt006 pavr picoweb pony-stk200 sp12 stk200 stk500
182 #
183 # Type: avrdude -c ?
184 # to get a full listing.
185 #
186 AVRDUDE_PROGRAMMER = avr910
187
188 # com1 = serial port. Use lpt1 to connect to parallel port.
189 AVRDUDE_PORT = /dev/ttyUSB0    # programmer connected to serial device
190
191 AVRDUDE_WRITE_FLASH = -U flash:w:$(TARGET).hex
192 #AVRDUDE_WRITE_EEPROM = -U eeprom:w:$(TARGET).eep
193
194
195 # Uncomment the following if you want avrdude's erase cycle counter.
196 # Note that this counter needs to be initialized first using -Yn,
```

```
197 # see avrdude manual.
198 #AVRDUDE_ERASE_COUNTER = -y
199
200 # Uncomment the following if you do /not/ wish a verification to be
201 # performed after programming the device.
202 #AVRDUDE_NO_VERIFY = -V
203
204 # Increase verbosity level.  Please use this when submitting bug
205 # reports about avrdude. See <http://savannah.nongnu.org/projects/avrdude>
206 # to submit bug reports.
207 #AVRDUDE_VERBOSE = -v -v
208
209 AVRDUDE_FLAGS = -p $(MCU) -P $(AVRDUDE_PORT) -c $(AVRDUDE_PROGRAMMER)
210 AVRDUDE_FLAGS += $(AVRDUDE_NO_VERIFY)
211 AVRDUDE_FLAGS += $(AVRDUDE_VERBOSE)
212 AVRDUDE_FLAGS += $(AVRDUDE_ERASE_COUNTER)
213
214
215
216 # ---------------------------------------------------------------------------
217
218 # Define directories, if needed.
219 DIRAVR = c:/winavr
220 DIRAVRBIN = $(DIRAVR)/bin
221 DIRAVRUTILS = $(DIRAVR)/utils/bin
222 DIRINC = .
223 DIRLIB = $(DIRAVR)/avr/lib
224
225
226 # Define programs and commands.
227 SHELL = sh
228 CC = avr-gcc
229 OBJCOPY = avr-objcopy
230 OBJDUMP = avr-objdump
231 SIZE = avr-size
232 NM = avr-nm
233 AVRDUDE = avrdude
234 REMOVE = rm -f
235 COPY = cp
236
237
238
239
240 # Define Messages
241 # English
242 MSG_ERRORS_NONE = Errors: none
243 MSG_BEGIN = -------- begin --------
244 MSG_END = --------  end  --------
245 MSG_SIZE_BEFORE = Size before:
246 MSG_SIZE_AFTER = Size after:
247 MSG_COFF = Converting to AVR COFF:
248 MSG_EXTENDED_COFF = Converting to AVR Extended COFF:
249 MSG_FLASH = Creating load file for Flash:
250 MSG_EEPROM = Creating load file for EEPROM:
251 MSG_EXTENDED_LISTING = Creating Extended Listing:
252 MSG_SYMBOL_TABLE = Creating Symbol Table:
253 MSG_LINKING = Linking:
254 MSG_COMPILING = Compiling:
255 MSG_ASSEMBLING = Assembling:
256 MSG_CLEANING = Cleaning project:
257
258
259
260
261 # Define all object files.
262 OBJ = $(SRC:.c=.o) $(ASRC:.S=.o)
263
264 # Define all listing files.
265 LST = $(ASRC:.S=.lst) $(SRC:.c=.lst)
266
```

```
267
268 # Compiler flags to generate dependency files.
269 ### GENDEPFLAGS = -Wp,-M,-MP,-MT,$(*F).o,-MF,.dep/$(@F).d
270 GENDEPFLAGS = -MD -MP -MF .dep/$(@F).d
271
272 # Combine all necessary flags and optional flags.
273 # Add target processor to flags.
274 ALL_CFLAGS = -mmcu=$(MCU) -I. $(CFLAGS) $(GENDEPFLAGS)
275 ALL_ASFLAGS = -mmcu=$(MCU) -I. -x assembler-with-cpp $(ASFLAGS)
276
277
278
279
280
281 # Default target.
282 all: begin gccversion sizebefore build sizeafter finished end
283
284 build: elf hex eep lss sym
285
286 elf: $(TARGET).elf
287 hex: $(TARGET).hex
288 eep: $(TARGET).eep
289 lss: $(TARGET).lss
290 sym: $(TARGET).sym
291
292
293
294 # Eye candy.
295 # AVR Studio 3.x does not check make's exit code but relies on
296 # the following magic strings to be generated by the compile job.
297 begin:
298     @echo
299     @echo $(MSG_BEGIN)
300
301 finished:
302     @echo $(MSG_ERRORS_NONE)
303
304 end:
305     @echo $(MSG_END)
306     @echo
307
308
309 # Display size of file.
310 HEXSIZE = $(SIZE) --target=$(FORMAT) $(TARGET).hex
311 ELFSIZE = $(SIZE) -A $(TARGET).elf
312 sizebefore:
313     @if [ -f $(TARGET).elf ]; then echo; echo $(MSG_SIZE_BEFORE); $(ELFSIZE);
        echo; fi
314
315 sizeafter:
316     @if [ -f $(TARGET).elf ]; then echo; echo $(MSG_SIZE_AFTER); $(ELFSIZE); echo
        ; fi
317
318
319
320 # Display compiler version information.
321 gccversion :
322     @$(CC) --version
323
324
325
326 # Program the device.
327 program: $(TARGET).hex $(TARGET).eep
328     $(AVRDUDE) $(AVRDUDE_FLAGS) $(AVRDUDE_WRITE_FLASH) $(AVRDUDE_WRITE_EEPROM)
329
330
331
332
333 # Convert ELF to COFF for use in debugging / simulating in AVR Studio or VMLAB.
334 COFFCONVERT=$(OBJCOPY) --debugging \
```

```
335 --change-section-address .data-0x800000 \
336 --change-section-address .bss-0x800000 \
337 --change-section-address .noinit-0x800000 \
338 --change-section-address .eeprom-0x810000
339
340
341 coff: $(TARGET).elf
342     @echo
343     @echo $(MSG_COFF) $(TARGET).cof
344     $(COFFCONVERT) -O coff-avr $< $(TARGET).cof
345
346
347 extcoff: $(TARGET).elf
348     @echo
349     @echo $(MSG_EXTENDED_COFF) $(TARGET).cof
350     $(COFFCONVERT) -O coff-ext-avr $< $(TARGET).cof
351
352
353
354 # Create final output files (.hex, .eep) from ELF output file.
355 %.hex: %.elf
356     @echo
357     @echo $(MSG_FLASH) $@
358     $(OBJCOPY) -O $(FORMAT) -R .eeprom $< $@
359
360 %.eep: %.elf
361     @echo
362     @echo $(MSG_EEPROM) $@
363     -$(OBJCOPY) -j .eeprom --set-section-flags=.eeprom="alloc,load" \
364     --change-section-lma .eeprom=0 -O $(FORMAT) $< $@
365
366 # Create extended listing file from ELF output file.
367 %.lss: %.elf
368     @echo
369     @echo $(MSG_EXTENDED_LISTING) $@
370     $(OBJDUMP) -h -S $< > $@
371
372 # Create a symbol table from ELF output file.
373 %.sym: %.elf
374     @echo
375     @echo $(MSG_SYMBOL_TABLE) $@
376     $(NM) -n $< > $@
377
378
379
380 # Link: create ELF output file from object files.
381 .SECONDARY : $(TARGET).elf
382 .PRECIOUS : $(OBJ)
383 %.elf: $(OBJ)
384     @echo
385     @echo $(MSG_LINKING) $@
386     $(CC) $(ALL_CFLAGS) $(OBJ) --output $@ $(LDFLAGS)
387
388
389 # Compile: create object files from C source files.
390 %.o : %.c
391     @echo
392     @echo $(MSG_COMPILING) $<
393     $(CC) -c $(ALL_CFLAGS) $< -o $@
394
395
396 # Compile: create assembler files from C source files.
397 %.s : %.c
398     $(CC) -S $(ALL_CFLAGS) $< -o $@
399
400
401 # Assemble: create object files from assembler source files.
402 %.o : %.S
403     @echo
404     @echo $(MSG_ASSEMBLING) $<
```

```
405     $(CC) -c $(ALL_ASFLAGS) $< -o $@
406 
407 
408 
409 # Target: clean project.
410 clean: begin clean_list finished end
411 
412 clean_list :
413     @echo
414     @echo $(MSG_CLEANING)
415     $(REMOVE) $(TARGET).hex
416     $(REMOVE) $(TARGET).eep
417     $(REMOVE) $(TARGET).obj
418     $(REMOVE) $(TARGET).cof
419     $(REMOVE) $(TARGET).elf
420     $(REMOVE) $(TARGET).map
421     $(REMOVE) $(TARGET).obj
422     $(REMOVE) $(TARGET).a90
423     $(REMOVE) $(TARGET).sym
424     $(REMOVE) $(TARGET).lnk
425     $(REMOVE) $(TARGET).lss
426     $(REMOVE) $(OBJ)
427     $(REMOVE) $(LST)
428     $(REMOVE) $(SRC:.c=.s)
429     $(REMOVE) $(SRC:.c=.d)
430     $(REMOVE) .dep/*
431 
432 
433 
434 # Include the dependency files.
435 -include $(shell mkdir .dep 2>/dev/null) $(wildcard .dep/*)
436 
437 
438 # Listing of phony targets.
439 .PHONY : all begin finish end sizebefore sizeafter gccversion \
440 build elf hex eep lss sym coff extcoff \
441 clean clean_list program
```

Programmcode Modul pwm

pwm.h

```
1  /* PWM Interface for ATmega32
2   * Version 1.0
3   *
4   * Author:        Ingo Weigel
5   * Filename:      pwm.h
6   * Software test: 2014-11-07
7   * Release:       2014-11-07
8   *
9   * Description:
10  *
11  * These functions form an interface for the generation of a PWM signal
12  * with an ATmega32 microcontroller. The PWM is used for motor control
13  * in an "Ottobock Sensorhand Speed" hand prosthesis.
14  *
15  * This syntax is suitable for the Timer/Counter 1 unit on the ATmega32,
16  * using the Phase and Frequency Correct PWM Mode. Using this mode with
17  * T/C1 Prescaler 1 it is possible to set the PWM frequency to any value
18  * within 61,04 Hz and 1.33 MHz (resolution: 750 ns PWM period time),
19  * which would not be possible with different (more simple) PWM modes.
20  *
21  * The interface contains three functions:
22  *
23  *
24  * pwmInit()
```

```
 *
 * - description:
 *   Initializes the PWM and sets the PWM duty cycle to zero.
 *   The PWM frequency is set by the argument pwmFrequency. The PWM
 *   inversion mode is set by the argument inversionMode. The PWM
 *   signal will be visible on the OC1A pin of the ATmega32.
 *
 * - return value:
 *   type: uint_8t
 *   errorlevel 0 -> PWM initialization successful
 *   errorlevel 1 -> invalid argument
 *
 * - argument: pwmFrequency
 *   type: double
 *   description: Frequency (in Hz) of the PWM signal
 *
 * - argument: inversionMode
 *   type: uint8_t
 *   description: inversion mode of the PWM signal (use preprocessor
 *                directives PWM_NON_INVERTED and PWM_INVERTED of pwm.h)
 *
 *
 * pwmDuty()
 *
 * - description:
 *   Sets duty cycle of the PWM signal. The argument pwmDutycycle
 *   specifies the PWM duty cycle as a percentage (range 0.0 - 1.0) of
 *   the PWM period.
 *
 * - return value:
 *   type: uint_8t
 *   errorlevel 0 -> PWM duty cycle set
 *   errorlevel 1 -> invalid argument (value not between 0.0 and 1.0)
 *
 * - argument: pwmDutycycle
 *   type: double
 *   description: PWM duty cycle as percentage of PWM period
 *
 *
 * pwmTerminate()
 *
 * - description
 *   Sets the PWM duty cycle to zero and then resets the hardware so it
 *   will be free again for different purposes.
 *
 * - return value:
 *   type: void
 *
 * - arguments: none
 */

#define CPU_FREQ 8000000 // CPU frequency (for calculation of PWM frequency)

// preprocessor directives PWM inversion modes
#define PWM_NON_INVERTED 0 // PWM signal not inverted
#define PWM_INVERTED     1 // PWM signal inverted

#include <avr/io.h> // AVR device-specific IO definitions

// pwm interface functions family
uint8_t pwmInit (double, uint8_t);
uint8_t pwmDuty (double);
void pwmTerminate (void);
```

pwm.c

```
/* PWM Interface for ATmega32
```

```
2   * Version 1.0
3   *
4   * Author:         Ingo Weigel
5   * Filename:       pwm.c
6   * Software test:  2014-11-07
7   * Release:        2014-11-07
8   *
9   * Description:
10  *
11  * These functions form an interface for the generation of a PWM signal
12  * with an ATmega32 microcontroller. The PWM is used for motor control
13  * in an "Ottobock Sensorhand Speed" hand prosthesis.
14  *
15  * This syntax is suitable for the Timer/Counter 1 unit on the ATmega32,
16  * using the Phase and Frequency Correct PWM Mode. Using this mode with
17  * T/C1 Prescaler 1 it is possible to set the PWM frequency to any value
18  * within 61,04 Hz and 1.33 MHz (resolution: 750 ns PWM period time),
19  * which would not be possible with different (more simple) PWM modes.
20  *
21  * The interface contains three functions:
22  *
23  *
24  * pwmInit()
25  *
26  * - description:
27  *   Initializes the PWM and sets the PWM duty cycle to zero.
28  *   The PWM frequency is set by the argument pwmFrequency. The PWM
29  *   inversion mode is set by the argument inversionMode. The PWM
30  *   signal will be visible on the OC1A pin of the ATmega32.
31  *
32  * - return value:
33  *   type: uint_8t
34  *   errorlevel 0 -> PWM initialization successful
35  *   errorlevel 1 -> invalid argument
36  *
37  * - argument: pwmFrequency
38  *   type: double
39  *   description: Frequency (in Hz) of the PWM signal
40  *
41  * - argument: inversionMode
42  *   type: uint8_t
43  *   description: Inversion mode of the PWM signal (use preprocessor
44  *                directives PWM_NON_INVERTED and PWM_INVERTED of pwm.h)
45  *
46  *
47  * pwmDuty()
48  *
49  * - description:
50  *   Sets duty cycle of the PWM signal. The argument pwmDutycycle
51  *   specifies the PWM duty cycle as a percentage (range 0.0 - 1.0) of
52  *   the PWM period.
53  *
54  * - return value:
55  *   type: uint_8t
56  *   errorlevel 0 -> PWM duty cycle set
57  *   errorlevel 1 -> invalid argument (value not between 0.0 and 1.0)
58  *
59  * - argument: pwmDutycycle
60  *   type: double
61  *   description: PWM duty cycle as percentage of PWM period
62  *
63  *
64  * pwmTerminate()
65  *
66  * - description
67  *   Sets the PWM duty cycle to zero and then resets the hardware so it
68  *   will be free again for different purposes.
69  *
70  * - return value:
71  *   type: void
```

```
72   *
73   * - arguments: none
74   */
75 
76 #include <pwm/pwm1.0/pwm.h>  // include header file (of version 1.0)
77 
78 /* Sequence for pwmDuty()
79  * Sets the the PWM duty cycle.
80  *
81  * For the single steps, see comments in the definition of pwmDuty().
82  */
83 
84 uint8_t pwmDuty(double pwmDutycycle)
85 {
86    uint8_t errorlevel=0; // declare errorlevel, initialize it with 0
87 
88    // check argument: is pwmDutycycle within [0.0;1.0] ?
89    if ( pwmDutycycle < 0.0 || pwmDutycycle > 1.0 )
90    {
91       // invalid argument (value not between 0.0 and 1.0)
92       errorlevel = 1;
93    }
94 
95    // if argument valid
96    if ( errorlevel == 0 )
97    {
98       // set PWM duty cyle corresponding to argument
99       OCR1A = pwmDutycycle * ICR1;
100    }
101 
102    // if argument invalid
103    else
104    {
105       // set PWM duty cyle to zero
106       OCR1A = 0;
107    }
108 
109    // quit by returning errorlevel
110    return errorlevel;
111 }
112 
113 /* Sequence for pwmInit()
114  * Initialize Phase and Frequency Correct PWM Mode
115  *
116  * [Step 1] Initialize errorlevel and check argument values
117  *
118  * Declare errorlevel as a uint8_t und initialize it with 0.
119  * If argument pwmFrequency is invalid (i.e. pwmFrequency is NOT within
120  * [ 61.04 Hz ; 1.33 MHz ]), set errorlevel to 1.
121  * If argument inversionMode is invalid (i.e. inversionMode is neither
122  * PWM_NON_INVERTED nor PWM_INVERTED), set errorlevel to 1.
123  *
124  * If errorlevel is 0, take [Step 2] to [Step 8].
125  * Otherwise, take [Step 8] only.
126  *
127  * [Step 2] T/C 1 clock select and prescaler configuration
128  *
129  * Clock: System clock
130  * Prescaler: 1
131  *
132  *     TCCR1B:CS12 0
133  *     TCCR1B:CS11 0
134  *     TCCR1B:CS10 1
135  *
136  * [Step 3] Set bits FOC1A and FOC1B to zero
137  *
138  * Recommended by datasheet for PWM-modes to ensure compatibility
139  * with future devices.
140  *
141  *     TCCR1A:FOC1A 0
```

```
142  *     TCCR1A:FOC1B 0
143  *
144  * [Step 4] Configure Waveform Generation Mode
145  *
146  * Mode: PWM, Phase and Frequency Correct (using ICR1 as TOP)
147  *
148  *     TCCR1B:WGM13 1
149  *     TCCR1B:WGM12 0
150  *     TCCR1A:WGM11 1
151  *     TCCR1A:WGM10 0
152  *
153  * [Step 5] Configure PWM output
154  *
155  * PWM destination: Pin OC1A (i.e. pin 19, port D pin 5)
156  * Inversion mode: set COM1A0 corresponding to argument inversionMode
157  *
158  *     TCCR1A:COM1A1 1
159  *     TCCR1A:COM1A0 0 (for non-inverted PWM)
160  *     TCCR1A:COM1A0 1 (for inverted PWM)
161  *     DDRD:DDD5     1
162  *
163  * [Step 6] Set PWM dutycycle to zero
164  *
165  * Use pwmDuty().
166  *
167  * [Step 7] Set PWM frequency
168  *
169  * Calculate TOP and write it to ICR1. A uint16_t cast is required.
170  *
171  *     since: PWM frequency is CPU_FREQ / (2 * Prescaler * TOP)
172  *     then:  TOP is CPU _FREQ / (2 * Prescaler * PWM frequency)
173  *
174  * [Step 8] quit by returning errorlevel
175  */
176 
177 uint8_t pwmInit(double pwmFrequency, uint8_t inversionMode)
178 {
179     // [Step 1]
180 
181     uint8_t errorlevel=0;
182 
183     if ( pwmFrequency > 1330000.00 || pwmFrequency < 61.04 )
184     {
185       errorlevel = 1;
186    }
187 
188     if ( inversionMode!=PWM_NON_INVERTED && inversionMode!=PWM_INVERTED )
189     {
190       errorlevel = 1;
191    }
192 
193     if ( errorlevel == 0 )
194     {
195       // [Step 2]
196 
197       TCCR1B &=~ ( (1<<CS12) | (1<<CS11) );
198       TCCR1B |=  (1<<CS10);
199 
200       // [Step 3]
201 
202       TCCR1A &=~ ( (1<<FOC1A) | (1<<FOC1B) );
203 
204       // [Step 4]
205 
206       TCCR1B |=  (1<<WGM13);
207       TCCR1B &=~ (1<<WGM12);
208       TCCR1A |=  (1<<WGM11);
209       TCCR1A &=~ (1<<WGM10);
210 
211       // [Step 5]
```

```
212
213        TCCR1A  |=   (1<<COM1A1);
214
215        switch (inversionMode)
216        {
217           case PWM_NON_INVERTED: TCCR1A &=~ (1<<COM1A0); break;
218           case PWM_INVERTED:     TCCR1A |=  (1<<COM1A0); break;
219        }
220
221        DDRD    |=   (1<<DDD5);
222
223        // [Step 6]
224
225        pwmDuty(0.0);
226
227        // [Step 7]
228
229        ICR1 = (uint16_t) ( CPU_FREQ/(2*1*pwmFrequency) );
230     }
231
232     // [Step 8]
233
234     return errorlevel;
235  }
236
237  /* Sequence for pwmTerminate()
238   * Terminate Phase and Frequency Correct PWM Mode
239   *
240   * [Step 1] Set PWM dutycyle to zero
241   *
242   * Use pwmDuty().
243   *
244   * [Step 2] Stop T/C unit
245   *
246   *     TCCR1B:CS12 0
247   *     TCCR1B:CS11 0
248   *     TCCR1B:CS10 0
249   *
250   * [Step 3] Disconnect pin OC1A
251   *
252   * Reset to normal port operation on this pin.
253   *
254   *     TCCR1A:COM1A1 0
255   *     TCCR1A:COM1A0 0
256   *     DDRD:DDD5     0
257   */
258
259  void pwmTerminate(void)
260  {
261     // [Step 1]
262     pwmDuty(0.0);
263
264     // [Step 2]
265     TCCR1B &=~ ( (1<<CS12) | (1<<CS11) | (1<<CS10) );
266
267     // [Step 3]
268     TCCR1A &=~ ( (1<<COM1A1) | (1<<COM1A0) );
269     DDRD   &=~ (1<<DDD5);
270  }
```

main.c (Modultest)

```
1  /* Module test: PWM interface version 1.0 (pwm1.0)
2   *
3   * Author:         Ingo Weigel
4   * Filename:       main.c
5   * Release:        2014-11-07
```

```
 *
 * Description:
 *
 * This main program tests the PWM interface (version 1.0). Each of the
 * 16 use cases can be tested by uncommenting the respective lines.
 *
 * - Header and source files covered by the test:
 *
 *   pwm/pwm1.0/pwm.h
 *   pwm/pwm1.0/pwm.c
 *
 * - Functions covered by the test:
 *
 *   pwmInit()
 *   pwmDuty()
 *   pwmTerminate()
 *
 * - This test is designed for various parameter configurations of:
 *
 *   PWM signal frequency
 *   PWM signal inversion mde
 *   PWM duty cycle
 *
 * - Use case design:
 *
 *   +------+---------------+--------------------+----------------+
 *   | case | PWM frequency | PWM inversion mode | PWM duty cycle |
 *   +------+---------------+--------------------+----------------+
 *   |  1   |    100 Hz     |    non-inverted    |      75%       |
 *   +------+---------------+--------------------+----------------+
 *   |  2   |    500 Hz     |    non-inverted    |      75%       |
 *   +------+---------------+--------------------+----------------+
 *   |  3   |      1 kHz    |    non-inverted    |      75%       |
 *   +------+---------------+--------------------+----------------+
 *   |  4   |     20 kHz    |    non-inverted    |      75%       |
 *   +------+---------------+--------------------+----------------+
 *   |  5   |    100 kHz    |    non-inverted    |      75%       |
 *   +------+---------------+--------------------+----------------+
 *   |  6   |    500 kHz    |    non-inverted    |      75%       |
 *   +------+---------------+--------------------+----------------+
 *   |  7   |      1 MHz    |    non-inverted    |      75%       |
 *   +------+---------------+--------------------+----------------+
 *   |  8   |     20 kHz    |    non-inverted    |       0%       |
 *   +------+---------------+--------------------+----------------+
 *   |  9   |     20 kHz    |    non-inverted    |       5%       |
 *   +------+---------------+--------------------+----------------+
 *   | 10   |     20 kHz    |    non-inverted    |      20%       |
 *   +------+---------------+--------------------+----------------+
 *   | 11   |     20 kHz    |    non-inverted    |      50%       |
 *   +------+---------------+--------------------+----------------+
 *   | 12   |     20 kHz    |    non-inverted    |      75%       |
 *   +------+---------------+--------------------+----------------+
 *   | 13   |     20 kHz    |    non-inverted    |      95%       |
 *   +------+---------------+--------------------+----------------+
 *   | 14   |     20 kHz    |    non-inverted    |     100%       |
 *   +------+---------------+--------------------+----------------+
 *   | 15   |     20 kHz    |      inverted      |      40%       |
 *   +------+---------------+--------------------+----------------+
 *   | 16   |     20 kHz    |      inverted      |      80%       |
 *   +------+---------------+--------------------+----------------+
 *
 * - Test sequence (technical test design):
 *
 *   Step 1: initialize PWM signal (corresponding to use case)
 *   Step 2: set PWM duty cycle (corresponding to use case)
 *   Step 3: wait 15 seconds
 *   Step 4: terminate PWM signal
 *
 * - Test procedure (tester's tasks):
 *
```

```
76   *   Preparation:
77   *   Step 1.1: connect controller, programmer and computer
78   *   Step 1.2: connect pin OC1A to oscilloscope
79   *
80   *   For each use case:
81   *   Step 2.1: uncomment actual use case in main(), comment the others
82   *   Step 2.2: compile main program and execute it on microcontroller
83   *   Step 2.3: capture PWM signal on oscilloscope with useful scaling
84   *             (you may press STOP-button to freeze the graph on screen)
85   *   Step 2.4: note down frequency, inversion and duty cycle
86   *             (you may use oscilloscope measure tools)
87   *   Step 2.5: observe and note down whether PWM signal disappears after
88   *             15 seconds (unfreeze the graph with STOP-button)
89   *
90   * - Evaluation of test result:
91   *
92   *   For each use case, the module passed the software test, if:
93   *   condition 1: each measured parameter matches the use case design
94   *   condition 2: PWM signal disappeared after 15 seconds
95   */
96
97  #define F_CPU 8000000         // CPU frequency (macro for delay.h)
98
99  #include <avr/io.h>           // AVR device-specific IO definitions
100 #include <util/delay.h>       // functions for delay loops
101
102 #include <pwm/pwm1.0/pwm.h>  // include PWM interface (version 1.0)
103
104 int main(void)
105 {
106     // software test: case 1
107     pwmInit ( 100, PWM_NON_INVERTED);        // PWM 100 Hz, non-inverted
108     pwmDuty ( 0.75 );                        // PWM duty cycle 75%
109
110     // software test: case 2
111     // pwmInit ( 500, PWM_NON_INVERTED);        // PWM 500 Hz, non-inverted
112     // pwmDuty ( 0.75 );                        // PWM duty cycle 75%
113
114     // software test: case 3
115     // pwmInit ( 1000, PWM_NON_INVERTED);       // PWM 1 kHz, non-inverted
116     // pwmDuty ( 0.75 );                        // PWM duty cycle 75%
117
118     // software test: case 4
119     // pwmInit ( 20000, PWM_NON_INVERTED);      // PWM 20 kHz, non-inverted
120     // pwmDuty ( 0.75 );                        // PWM duty cycle 75%
121
122     // software test: case 5
123     // pwmInit ( 100000, PWM_NON_INVERTED);     // PWM 100 kHz, non-inverted
124     // pwmDuty ( 0.75 );                        // PWM duty cycle 75%
125
126     // software test: case 6
127     // pwmInit ( 500000, PWM_NON_INVERTED);     // PWM 500 kHz, non-inverted
128     // pwmDuty ( 0.75 );                        // PWM duty cycle 75%
129
130     // software test: case 7
131     // pwmInit ( 1000000, PWM_NON_INVERTED);    // PWM 1 MHz, non-inverted
132     // pwmDuty ( 0.75 );                        // PWM duty cycle 75%
133
134     // software test: case 8
135     // pwmInit ( 20000, PWM_NON_INVERTED);      // PWM 20 kHz, non-inverted
136     // pwmDuty ( 0.00 );                        // PWM duty cycle 0%
137
138     // software test: case 9
139     // pwmInit ( 20000, PWM_NON_INVERTED);      // PWM 20 kHz, non-inverted
140     // pwmDuty ( 0.05 );                        // PWM duty cycle 5%
141
142     // software test: case 10
143     // pwmInit ( 20000, PWM_NON_INVERTED);      // PWM 20 kHz, non-inverted
144     // pwmDuty ( 0.20 );                        // PWM duty cycle 20%
145
```

```
146     // software test: case 11
147     // pwmInit ( 20000, PWM_NON_INVERTED);      // PWM 20 kHz, non-inverted
148     // pwmDuty ( 0.50 );                        // PWM duty cycle 50%
149
150     // software test: case 12
151     // pwmInit ( 20000, PWM_NON_INVERTED);      // PWM 20 kHz, non-inverted
152     // pwmDuty ( 0.75 );                        // PWM duty cycle 75%
153
154     // software test: case 13
155     // pwmInit ( 20000, PWM_NON_INVERTED);      // PWM 20 kHz, non-inverted
156     // pwmDuty ( 0.95 );                        // PWM duty cycle 95%
157
158     // software test: case 14
159     // pwmInit ( 20000, PWM_NON_INVERTED);      // PWM 20 kHz, non-inverted
160     // pwmDuty ( 1.00 );                        // PWM duty cycle 100%
161
162     // software test: case 15
163     // pwmInit ( 20000, PWM_INVERTED);          // PWM 20 kHz, inverted
164     // pwmDuty ( 0.40 );                        // PWM duty cycle 40%
165
166     // software test: case 16
167     // pwmInit ( 20000, PWM_INVERTED);          // PWM 20 kHz, inverted
168     // pwmDuty ( 0.80 );                        // PWM duty cycle 80%
169
170     _delay_ms(15000);                       // wait 15 seconds
171
172     pwmTerminate();                         // terminate PWM
173
174     while(1);
175
176     return 0;
177 }
```

Programmcode Modul motor

motor.h

```
1  /* Motor module for prosthesis control
2   * Version 1.0
3   *
4   * Author:        Ingo Weigel
5   * Filename:      motor.h
6   * Software test: 2014-11-10
7   * Release:       2014-11-10
8   *
9   * Description:
10  *
11  * This software module forms an interface for motor control in an
12  * "Ottobock Sensorhand Speed" hand prosthesis with an ATmega32
13  * microcontroller.
14  *
15  * The motor module generates signals for the microcontroller interface
16  * of the motor control circuit. The motor control circuit expects a
17  * PWM signal as well as logical levels on two pins (named Motor A and
18  * Motor B). These signals determine the motor behaviour:
19  *
20  * +-------+-------+----------+------------------------------------+
21  * | Motor | Motor | PWM duty |                motor               |
22  * |   A   |   B   |  cycle   |              behaviour             |
23  * +-------+-------+----------+------------------------------------+
24  * |       |       |     0%   | free wheel (no active brake)       |
25  * |  LOW  |  LOW  +----------+------------------------------------+
26  * |       |       |   100%   | active brake (max. brake effect)   |
27  * +-------+-------+----------+------------------------------------+
28  * |       |       |   > 0%   | motor operation: open hand         |
29  * |  HIGH |  LOW  +----------+------------------------------------+
```

```
30  * |       |       |     0%    | motor measure mode when closing hand |
31  * +-------+-------+-----------+--------------------------------------+
32  * |       |       |   > 0%    | motor operation: close hand          |
33  * |  LOW  |  HIGH +-----------+--------------------------------------+
34  * |       |       |     0%    | motor measure mode when opening hand |
35  * +-------+-------+-----------+--------------------------------------+
36  * |  HIGH |  HIGH |           |  - not in use -                      |
37  * +-------+-------+-----------+--------------------------------------+
38  *
39  * For setting the PWM duty cycle, Motor A and Motor B, the motor module
40  * deploys four motor operation modes:
41  *
42  *  MOTOR_STOP     (Motor A set to LOW,  Motor B set to LOW)
43  *  MOTOR_OPEN     (Motor A set to HIGH, Motor B set to LOW)
44  *  MOTOR_CLOSE    (Motor A set to LOW,  Motor B set to HIGH)
45  *  MOTOR_MEASURE  (toggle Motor A and Motor B, force PWM duty cycle 0%)
46  *
47  * For generating the PWM signal, the motor module uses the PWM
48  * interface (pwm.h). The PWM frequency and PWM inversion mode are
49  * specified by the developers of the motor circuit. Since these
50  * specifications may change from time to time, these parameters can
51  * easily be modified with the macros MOTOR_PWM_INV_MODE and
52  * MOTOR_PWM_FREQ.
53  *
54  * The motor module consists of three functions:
55  *
56  *
57  * motorInit()
58  *
59  * - description:
60  *   Initializes the motor control interface with motor operation mode
61  *   MOTOR_STOP and PWM duty cycle 0%.
62  *
63  * - return value:
64  *   type: uint_8t
65  *   errorlevel 0 -> motor interface initilization successful
66  *   errorlevel 1 -> error (motor interface NOT initialized properly)
67  *
68  * - arguments: none
69  *
70  *
71  * motorOperation()
72  *
73  * - description:
74  *   Sets motor operation mode and PWM duty cycle. The argument
75  *   motorMode specifies the applied motor operation mode. The argument
76  *   motorUtilization specifies the motor utilization (for MOTOR_OPEN
77  *   and MOTOR_CLOSE mode) and the brake utilization (for MOTOR_STOP
78  *   mode). For MOTOR_MEASURE mode, the argument motorUtilization will
79  *   be ignored (force PWM duty cycle 0%).
80  *
81  * - return value:
82  *   type: uint_8t
83  *   errorlevel 0 -> motor operation mode set
84  *   errorlevel 1 -> error (motor operation mode NOT set properly)
85  *
86  * - argument: motorMode
87  *   type: uint8_t
88  *   description: applied motor operation mode (use preprocessor
89  *                directives MOTOR_STOP, MOTOR_OPEN, MOTOR_CLOSE and
90  *                MOTOR_MEASURE of motor.h)
91  *
92  * - argument: motorUitilization
93  *   type: double
94  *   description: percentage of motor or brake utiliziation (ignored in
95  *                motor measure mode)
96  *
97  *
98  * motorTerminate()
99  *
```

```
100  * - description
101  *   Sets the motor operation mode to MOTOR_STOP and free wheel (no
102  *   brake). Then terminates PWM signal and resets pins Motor A and
103  *   Motor B so that Timer/Counter and registers will be free again for
104  *   different purposes.
105  *
106  * - return value:
107  *   type: uint_8t
108  *   errorlevel 0 -> motor interface terminated
109  *   errorlevel 1 -> error (motor interface NOT terminated properly)
110  *
111  * - arguments: none
112  */
113
114 #include <pwm/pwm1.0/pwm.h>  // include PWM interface (version 1.0)
115
116 #define MOTOR_PWM_FREQ      20000         // motor PWM frequency (Hz)
117 #define MOTOR_PWM_INV_MODE  PWM_INVERTED  // motor PWM inversion mode
118
119 // DDRs, Ports and Pins of Motor A and Motor B (output to motor circuit)
120 #define MOTOR_A_DDR  DDRB   // Data Direction Register for Motor A
121 #define MOTOR_A_PORT PORTB  // Port Register for Motor A
122 #define MOTOR_A_PIN  0x06   // Pin for Motor A
123 #define MOTOR_B_DDR  DDRB   // Data Direction Register for Motor B
124 #define MOTOR_B_PORT PORTB  // Port Register for Motor B
125 #define MOTOR_B_PIN  0x07   // Pin for Motor B
126
127 // preprocessor directives for motor operation modes
128 #define MOTOR_STOP    0  // no motor operation (free wheel or brake)
129 #define MOTOR_OPEN    1  // motor operation: open hand
130 #define MOTOR_CLOSE   2  // motor operation: close hand
131 #define MOTOR_MEASURE 3  // motor measure mode
132
133 // motor interface functions family
134 uint8_t motorInit (void);
135 uint8_t motorOperation (uint8_t, double);
136 uint8_t motorTerminate (void);
```

motor.c

```
1  /* Motor module for prosthesis control
2   * Version 1.0
3   *
4   * Author:        Ingo Weigel
5   * Filename:      motor.
6   * Software test: 2014-11-10
7   * Release:       2014-11-10
8   *
9   * Description:
10  *
11  * This software module forms an interface for motor control in an
12  * "Ottobock Sensorhand Speed" hand prosthesis with an ATmega32
13  * microcontroller.
14  *
15  * The motor module generates signals for the microcontroller interface
16  * of the motor control circuit. The motor control circuit expects a
17  * PWM signal as well as logical levels on two pins (named Motor A and
18  * Motor B). These signals determine the motor behaviour:
19  *
20  * +-------+-------+----------+-------------------------------------+
21  * | Motor | Motor | PWM duty |                motor                |
22  * |   A   |   B   |  cycle   |              behaviour              |
23  * +-------+-------+----------+-------------------------------------+
24  * |       |       |      0%  | free wheel (no active brake)        |
25  * |  LOW  |  LOW  +----------+-------------------------------------+
26  * |       |       |    100%  | active brake (max. brake effect)    |
27  * +-------+-------+----------+-------------------------------------+
```

```
28  * |       |       |    > 0%    | motor operation: open hand            |
29  * |  HIGH |  LOW  +-----------+---------------------------------------+
30  * |       |       |      0%    | motor measure mode when closing hand |
31  * +-------+-------+-----------+---------------------------------------+
32  * |       |       |    > 0%    | motor operation: close hand           |
33  * |  LOW  |  HIGH +-----------+---------------------------------------+
34  * |       |       |      0%    | motor measure mode when opening hand |
35  * +-------+-------+-----------+---------------------------------------+
36  * |  HIGH |  HIGH |           |   - not in use -                      |
37  * +-------+-------+-----------+---------------------------------------+
38  *
39  * For setting the PWM duty cycle, Motor A and Motor B, the motor module
40  * deploys four motor operation modes:
41  *
42  *  MOTOR_STOP    (Motor A set to LOW,  Motor B set to LOW)
43  *  MOTOR_OPEN    (Motor A set to HIGH, Motor B set to LOW)
44  *  MOTOR_CLOSE   (Motor A set to LOW,  Motor B set to HIGH)
45  *  MOTOR_MEASURE (toggle Motor A and Motor B, force PWM duty cycle 0%)
46  *
47  * For generating the PWM signal, the motor module uses the PWM
48  * interface (pwm.h). The PWM frequency and PWM inversion mode are
49  * specified by the developers of the motor circuit. Since these
50  * specifications may change from time to time, these parameters can
51  * easily be modified with the macros MOTOR_PWM_INV_MODE and
52  * MOTOR_PWM_FREQ.
53  *
54  * The motor module consists of three functions:
55  *
56  *
57  * motorInit()
58  *
59  * - description:
60  *   Initializes the motor control interface with motor operation mode
61  *   MOTOR_STOP and PWM duty cycle 0%.
62  *
63  * - return value:
64  *   type: uint_8t
65  *   errorlevel 0 -> motor interface initilization successful
66  *   errorlevel 1 -> error (motor interface NOT initialized properly)
67  *
68  * - arguments: none
69  *
70  *
71  * motorOperation()
72  *
73  * - description:
74  *   Sets motor operation mode and PWM duty cycle. The argument
75  *   motorMode specifies the applied motor operation mode. The argument
76  *   motorUtilization specifies the motor utilization (for MOTOR_OPEN
77  *   and MOTOR_CLOSE mode) and the brake utilization (for MOTOR_STOP
78  *   mode). For MOTOR_MEASURE mode, the argument motorUtilization will
79  *   be ignored (force PWM duty cycle 0%).
80  *
81  * - return value:
82  *   type: uint_8t
83  *   errorlevel 0 -> motor operation mode set
84  *   errorlevel 1 -> error (motor operation mode NOT set properly)
85  *
86  * - argument: motorMode
87  *   type: uint8_t
88  *   description: applied motor operation mode (use preprocessor
89  *                directives MOTOR_STOP, MOTOR_OPEN, MOTOR_CLOSE and
90  *                MOTOR_MEASURE of motor.h)
91  *
92  * - argument: motorUitilization
93  *   type: double
94  *   description: percentage of motor or brake utiliziation (ignored in
95  *                motor measure mode)
96  *
97  *
```

```
98   * motorTerminate()
99   *
100  * - description
101  *   Sets the motor operation mode to MOTOR_STOP and free wheel (no
102  *   brake). Then terminates PWM signal and resets pins Motor A and
103  *   Motor B so that Timer/Counter and registers will be free again for
104  *   different purposes.
105  *
106  * - return value:
107  *   type: uint_8t
108  *   errorlevel 0 -> motor interface terminated
109  *   errorlevel 1 -> error (motor interface NOT terminated properly)
110  *
111  * - arguments: none
112  */
113
114 #include <motor/motor1.0/motor.h>  // include header (of version 1.0)
115
116 uint8_t motorInit (void)
117 {
118    uint8_t errorlevel=0; // declare errorlevel, initialize it with 0
119
120    // initialize PWM signal
121    if ( pwmInit(MOTOR_PWM_FREQ, MOTOR_PWM_INV_MODE) )
122    {
123       // an error occured in pwmInit()
124       errorlevel = 1;
125    }
126
127    else
128    {
129       // set Data Direction Registers to Output for pins Motor A and B
130       MOTOR_A_DDR |= (1<<MOTOR_A_PIN);
131       MOTOR_B_DDR |= (1<<MOTOR_B_PIN);
132
133       // set motor mode to STOP (no operation)
134       if ( motorOperation (MOTOR_STOP, 0.0) )
135       {
136          // an error occured in motorOperation()
137          errorlevel = 1;
138       }
139    }
140
141    // quit by returning errorlevel
142    return errorlevel;
143 }
144
145 uint8_t motorOperation (uint8_t motorMode, double motorUtilization)
146 {
147    uint8_t errorlevel=0; // declare errorlevel, initialize it with 0
148
149    // set PWM duty cycle and pins Motor A and Motor B
150    switch (motorMode)
151    {
152       case MOTOR_STOP:
153
154          errorlevel = pwmDuty(motorUtilization); // set duty cycle
155          MOTOR_A_PORT &=~ (1<<MOTOR_A_PIN); // set Motor A to LOW
156          MOTOR_B_PORT &=~ (1<<MOTOR_B_PIN); // set Motor B to LOW
157
158       break;
159
160       case MOTOR_OPEN:
161
162          errorlevel = pwmDuty(motorUtilization); // set duty cycle
163          MOTOR_A_PORT |=  (1<<MOTOR_A_PIN); // set Motor A to HIGH
164          MOTOR_B_PORT &=~ (1<<MOTOR_B_PIN); // set Motor B to LOW
165
166       break;
167
```

```
168        case MOTOR_CLOSE:
169
170            errorlevel = pwmDuty(motorUtilization); // set duty cycle
171            MOTOR_A_PORT &=~ (1<<MOTOR_A_PIN); // set Motor A to LOW
172            MOTOR_B_PORT |=  (1<<MOTOR_B_PIN); // set Motor B to HIGH
173
174        break;
175
176        case MOTOR_MEASURE:
177
178            errorlevel = pwmDuty(0.00); // force duty cycle 0%
179
180            // toggle Motor A and Motor B (only if both are not LOW)
181            if (    ( MOTOR_A_PORT & (1<<MOTOR_A_PIN) )
182                 || ( MOTOR_B_PORT & (1<<MOTOR_B_PIN) )    )
183            {
184                MOTOR_A_PORT ^=  (1<<MOTOR_A_PIN); // toggle Motor A
185                MOTOR_B_PORT ^=  (1<<MOTOR_B_PIN); // toggle Motor B
186            }
187
188        break;
189
190        default:
191
192            // invalid argument motorMode
193            errorlevel = 1;
194
195        break;
196    }
197
198    // quit by returning errorlevel
199    return errorlevel;
200 }
201
202 uint8_t motorTerminate (void)
203 {
204    uint8_t errorlevel=0; // declare errorlevel, initialize it with 0
205
206    // set motor operation to MOTOR_STOP and free wheel (no brake)
207    errorlevel = motorOperation (MOTOR_STOP, 0.0);
208
209    // terminate PWM signal
210    pwmTerminate(); // (does not return any errorlevel)
211
212    // reset Data Direction Registers for pins Motor A and B
213    MOTOR_A_DDR &=~ (1<<MOTOR_A_PIN);
214    MOTOR_B_DDR &=~ (1<<MOTOR_B_PIN);
215
216    // quit by returning errorlevel
217    return errorlevel;
218 }
```

main.c (Modultest)

```
1  /* Module test: motor controle module version 1.0 (motor1.0)
2   *
3   * Author:          Ingo Weigel
4   * Filename:        main.c
5   * Release:         2014-11-10
6   *
7   * Description:
8   *
9   * This main program tests the motor control module (version 1.0). Each
10  * of the 8 use cases can be tested by uncommenting the respective
11  * lines.
12  *
13  * - Header and source files covered by the test:
```

```
14  *
15  *   motor/motor1.0/motor.h
16  *   motor/motor1.0/motor.c
17  *
18  * - Functions covered by the test:
19  *
20  *   motorInit()
21  *   motorOperation()
22  *   motorTerminate()
23  *
24  * - This test is designed for various parameter configurations of:
25  *
26  *   motor operation mode
27  *   motor/brake utilization
28  *
29  * - Use case design:
30  *
31  *   +------+----------------------+-------------------------+
32  *   | case | motor operation mode | motor/brake utilization |
33  *   +------+----------------------+-------------------------+
34  *   |  1   |      MOTOR_STOP      |           50%           |
35  *   +------+----------------------+-------------------------+
36  *   |  2   |      MOTOR_OPEN      |           50%           |
37  *   +------+----------------------+-------------------------+
38  *   |  3   |      MOTOR_CLOSE     |           50%           |
39  *   +------+----------------------+-------------------------+
40  *   |  4   | (1)  MOTOR_CLOSE     |           50%           |
41  *   |      | (2) MOTOR_MEASURE    |           50%           |
42  *   +------+----------------------+-------------------------+
43  *   |  5   |      MOTOR_OPEN      |            0%           |
44  *   +------+----------------------+-------------------------+
45  *   |  6   |      MOTOR_OPEN      |           25%           |
46  *   +------+----------------------+-------------------------+
47  *   |  7   |      MOTOR_OPEN      |           75%           |
48  *   +------+----------------------+-------------------------+
49  *   |  8   |      MOTOR_OPEN      |          100%           |
50  *   +------+----------------------+-------------------------+
51  *
52  *   Use case 4 is to test the MOTOR_MEASURE mode (step 2). Therefore,
53  *   a defined mode must be provided first (step 1).
54  *
55  * - Test sequence (technical test design):
56  *
57  *   Step 1: initialize motor control interface
58  *   Step 2: set motor operation mode and motor/brake utiliziation
59  *           (corresponding to use case)
60  *   Step 3: wait 60 seconds
61  *   Step 4: terminate motor control interface
62  *
63  * - Test procedure (tester's tasks):
64  *
65  *   Preparation:
66  *   Step 1.1: connect controller, programmer and computer
67  *   Step 1.2: connect pin OC1A to oscilloscope
68  *   Step 1.3: connect pins Motor A and Motor B each to a multimeter
69  *
70  *   For each use case:
71  *   Step 2.1: uncomment actual use case in main(), comment the others
72  *   Step 2.2: compile main program and execute it on microcontroller
73  *   Step 2.3: capture PWM signal on oscilloscope with useful scaling
74  *             (you may press STOP-button to freeze the graph on screen)
75  *   Step 2.4: note down frequency, inversion and duty cycle
76  *             (you may use oscilloscope measure tools)
77  *   Step 2.5: note down logical levels of pins Motor A and Motor B
78  *   Step 2.6: observe and note down whether PWM signal and logical
79  *             levels on Motor A and Motor B disappear after 15 seconds
80  *
81  * - Evaluation of test result:
82  *
83  *   For each use case, the module passed the software test, if:
```

```
84   *     condition 1: each measured parameter matches the use case design
85   *                  (for expected Motor A and Motor B, see table below)
86   *     condition 2: all signals disappeared after 15 seconds
87   *
88   *        +----------+-----------------+-----------------+
89   *        | use case | expected Motor A | expected Motor B |
90   *        +----------+-----------------+-----------------+
91   *        |        1 |       LOW       |       LOW       |
92   *        +----------+-----------------+-----------------+
93   *        |        2 |       HIGH      |       LOW       |
94   *        +----------+-----------------+-----------------+
95   *        |        3 |       LOW       |       HIGH      |
96   *        +----------+-----------------+-----------------+
97   *        |        4 |       HIGH      |       LOW       |
98   *        +----------+-----------------+-----------------+
99   *        |        5 |       HIGH      |       LOW       |
100  *        +----------+-----------------+-----------------+
101  *        |        6 |       HIGH      |       LOW       |
102  *        +----------+-----------------+-----------------+
103  *        |        7 |       HIGH      |       LOW       |
104  *        +----------+-----------------+-----------------+
105  *        |        8 |       HIGH      |       LOW       |
106  *        +----------+-----------------+-----------------+
107  */
108
109 #define F_CPU 8000000          // CPU frequency (macro for delay.h)
110
111 #include <avr/io.h>            // AVR device-specific IO definitions
112 #include <util/delay.h>        // functions for delay loops
113
114 #include <motor/motor1.0/motor.h>  // include motor module (version 1.0)
115
116 int main(void)
117 {
118    motorInit();                          // initialize motor control
119
120    //software test: use case 1
121    motorOperation(MOTOR_STOP,0.50);      // motor stopped, break 50%
122
123    // software test: use case 2
124    // motorOperation(MOTOR_OPEN,0.50);      // open hand, motor 50%
125
126    // software test: use case 3
127    // motorOperation(MOTOR_CLOSE,0.50);     // close hand, motor 50%
128
129    // software test: use case 4
130    // motorOperation(MOTOR_CLOSE,0.50);     // close hand, motor 50%
131    // motorOperation(MOTOR_MEASURE,0.50);   // measure mode, motor 50%
132
133    // software test: use ase 5
134    // motorOperation(MOTOR_OPEN,0.00);      // open hand, motor 0%
135
136    // software test: use case 6
137    // motorOperation(MOTOR_OPEN,0.25);      // open hand, motor 25%
138
139    // software test: use case 7
140    // motorOperation(MOTOR_OPEN,0.75);      // open hand, motor 75%
141
142    // software test: use case 8
143    // motorOperation(MOTOR_OPEN,1.00);      // open hand, motor 100%
144
145    _delay_ms(60000);                     // wait 60 seconds
146
147    motorTerminate();                     // terminate motor control
148
149    while(1);
150
151    return 0;
152 }
```

Programmcode Modul statemachine

statemachine.h

```c
#include <lcd/mylcd.h>    // interface for display control
#include <string.h>       // library for string operations
#include <stdlib.h>       // required for conversion integer to string
#include <lcd/font5x8.h>  // font for display

// include motor module (version 1.0) for motor operation modes
#include <motor/motor1.0/motor.h>

// thresholds for state machine transition conditions
#define THRES_EMG_OPEN       700   // threshold for emg in open direction
#define THRES_EMG_CLOSE      700   // threshold for emg in close direction
#define THRES_SW_OVF         61    // 61 x 8.192 ms = 499.712 ms
#define THRES_SW_CNT         9     //  9 x 0.032 ms =   0.288 ms
#define THRES_MOTOR_REV      200   // threshold for motor revolutions
#define THRES_MOTOR_TORQUE   900   // threshold for motor torque

// preprocessor directives for stopwatch modes
#define SW_STOPPED 0   // stopwatch stopped
#define SW_RUNNING 1   // stopwatch running

// enumeration of state names
enum stateName { Init, TimerOpen, TimerClose, Open, Close };

// bundle state variables in a struct
struct state
{
   enum stateName name;
   uint8_t stopwatch;
   uint8_t motorMode;
   double motorUtil;
};

// statemachine functions family
struct state statemachineRefresh
               (struct state, uint16_t, uint16_t, uint8_t,
                uint8_t, uint16_t, uint16_t);
void statemachineDisplay (struct state);
```

statemachine.c

```c
// include header file (of version 1.0)
#include <statemachine/statemachine1.0/statemachine.h>

struct state statemachineRefresh
               (struct state currentState, uint16_t emgOpen,
                uint16_t emgClose, uint8_t stopwatchOVF,
                uint8_t stopwatchCNT, uint16_t motorREV,
                uint16_t motorTorque)
{
   switch (currentState.name)
   {
      case Init:

         if ( emgOpen > THRES_EMG_OPEN )
         {
            // enter state TimerOpen
            currentState.name = TimerOpen;
            currentState.stopwatch = SW_RUNNING;
         }

         else if ( emgClose > THRES_EMG_CLOSE )
```

```
22            {
23               // enter state TimerClose
24               currentState.name = TimerClose;
25               currentState.stopwatch = SW_RUNNING;
26            }
27
28         break;
29
30         case TimerOpen:
31
32            if ( emgOpen <= THRES_EMG_OPEN )
33            {
34               // enter state Init
35               currentState.name = Init;
36               currentState.motorMode = MOTOR_STOP;
37               currentState.motorUtil = 0.0;
38               currentState.stopwatch = SW_STOPPED;
39            }
40
41            else if ( emgOpen > THRES_EMG_OPEN && stopwatchOVF >= THRES_SW_OVF &&
                stopwatchCNT >= THRES_SW_CNT )
42            {
43               // enter state Open
44               currentState.name = Open;
45               currentState.motorMode = MOTOR_OPEN;
46               currentState.motorUtil = (double) (emgOpen/1023.0);
47            }
48
49         break;
50
51         case TimerClose:
52
53            if ( emgClose <= THRES_EMG_CLOSE )
54            {
55               // enter state Init
56               currentState.name = Init;
57               currentState.motorMode = MOTOR_STOP;
58               currentState.motorUtil = 0.0;
59               currentState.stopwatch = SW_STOPPED;
60            }
61
62            else if ( emgClose > THRES_EMG_CLOSE && stopwatchOVF >= THRES_SW_OVF &&
                 stopwatchCNT >= THRES_SW_CNT )
63            {
64               // enter state Open
65               currentState.name = Close;
66               currentState.motorMode = MOTOR_CLOSE;
67               currentState.motorUtil = (double) (emgClose/1023.0);
68            }
69
70         break;
71
72         case Open:
73
74            if ( emgOpen <= THRES_EMG_OPEN || motorREV <= THRES_MOTOR_REV ||
                motorTorque >= THRES_MOTOR_TORQUE )
75            {
76               // enter state Init
77               currentState.name = Init;
78               currentState.motorMode = MOTOR_STOP;
79               currentState.motorUtil = 0.0;
80               currentState.stopwatch = SW_STOPPED;
81            }
82
83            else
84            {
85               // enter state Open (again)
86               currentState.name = Open;
87               currentState.motorMode = MOTOR_OPEN;
88               currentState.motorUtil = (double) (emgOpen/1023.0);
```

```
89          }
90
91       break;
92
93       case Close:
94
95          if ( emgClose <= THRES_EMG_CLOSE || motorREV <= THRES_MOTOR_REV ||
              motorTorque >= THRES_MOTOR_TORQUE )
96          {
97             // enter state Init
98             currentState.name = Init;
99             currentState.motorMode = MOTOR_STOP;
100            currentState.motorUtil = 0.0;
101            currentState.stopwatch = SW_STOPPED;
102         }
103
104         else
105         {
106            // enter state Close (again)
107            currentState.name = Close;
108            currentState.motorMode = MOTOR_CLOSE;
109            currentState.motorUtil = (double) (emgClose/1023.0);
110         }
111
112      break;
113
114      default:
115
116         // enter state Init
117         currentState.name = Init;
118         currentState.motorMode = MOTOR_STOP;
119         currentState.motorUtil = 0.0;
120         currentState.stopwatch = SW_STOPPED;
121
122      break;
123   }
124
125   return currentState;
126 }
127
128 void statemachineDisplay (struct state currentState)
129 {
130    char s_name[25], s_stopwatch[25], s_motorMode[25], s_motorUtil[25];
131    char s_itoa_buffer[4]; // string used to buffer results of itoa()
132    uint8_t i_motorUtil;
133
134    lcd_clear(); // clear display
135
136    strcpy(s_name,       "name:  ");
137    strcpy(s_stopwatch,  "stopw: ");
138    strcpy(s_motorMode,  "mMode: ");
139    strcpy(s_motorUtil,  "mUtil: ");
140
141    switch (currentState.name)
142    {
143       case Init:        strcat(s_name,"Init");        break;
144       case TimerOpen:   strcat(s_name,"TimerOpen");   break;
145       case TimerClose:  strcat(s_name,"TimerClose");  break;
146       case Open:        strcat(s_name,"Open");        break;
147       case Close:       strcat(s_name,"Close");       break;
148       default:          strcat(s_name,"INVALID");     break;
149    }
150
151    switch (currentState.stopwatch)
152    {
153       case SW_RUNNING: strcat(s_stopwatch,"SW_RUNNING"); break;
154       case SW_STOPPED: strcat(s_stopwatch,"SW_STOPPED"); break;
155       default:         strcat(s_stopwatch,"INVALID");    break;
156    }
157
```

```
158     switch (currentState.motorMode)
159     {
160        case MOTOR_STOP:     strcat(s_motorMode,"MOTOR_STOP");     break;
161        case MOTOR_OPEN:     strcat(s_motorMode,"MOTOR_OPEN");     break;
162        case MOTOR_CLOSE:    strcat(s_motorMode,"MOTOR_CLOSE");    break;
163        case MOTOR_MEASURE:  strcat(s_motorMode,"MOTOR_MEASURE");  break;
164        default:             strcat(s_motorMode,"INVALID");        break;
165     }
166
167     i_motorUtil = (uint8_t) ( currentState.motorUtil * 100.0 );
168     itoa(i_motorUtil,s_itoa_buffer,10);
169     strcat(s_motorUtil,s_itoa_buffer);
170     strcat(s_motorUtil," %");
171
172     // write values to display
173     lcd_set_cursor(0,0);
174     lcd_puts(font5x8,"current state:");
175     lcd_set_cursor(3,10);
176     lcd_puts(font5x8,s_name);
177     lcd_set_cursor(3,20);
178     lcd_puts(font5x8,s_stopwatch);
179     lcd_set_cursor(3,30);
180     lcd_puts(font5x8,s_motorMode);
181     lcd_set_cursor(3,40);
182     lcd_puts(font5x8,s_motorUtil);
183  }
```

main.c (Modultest)

```
1  #define F_CPU 8000000     // CPU frequency (macro for delay.h)
2
3  #include <avr/io.h>       // AVR device-specific IO definitions
4  #include <util/delay.h>   // functions for delay loops
5
6  // include state machine module (version 1.0)
7  #include <statemachine/statemachine1.0/statemachine.h>
8
9  // define environment simulation
10
11 #define SIM_STEPS 27  // number of environment simulation steps
12
13 uint16_t simulated_emgOpen[SIM_STEPS] =
14 { 0,   // simulation step 0
15   500, // simulation step 1
16   0,   // simulation step 2
17   800, // and so on ...
18   800, 500, 0, 0, 0, 1000, 1000, 1000, 1000, 150, 500, 500, 500, 500,
19   1000, 1000, 1000, 800, 800, 500, 500, 500, 500 };
20
21 uint16_t simulated_emgClose[SIM_STEPS] =
22 { 0, 0, 500, 0, 0, 0, 800, 800, 500, 500, 500, 500, 500, 1000, 1000,
23   1000, 1000, 1000, 150, 500, 500, 500, 500, 1000, 1000, 800, 800 };
24
25 uint8_t  simulated_stopwatchOVF[SIM_STEPS] =
26 { 0, 0, 0, 0, 0, 100, 0, 0, 100, 0, 30, 100, 123, 123, 0, 30, 100, 123,
27   123, 0, 100, 123, 123, 0, 100, 123, 123 };
28
29 uint16_t simulated_stopwatchCNT[SIM_STEPS] =
30 { 0, 0, 0, 0, 0, 1000, 0, 0, 1000, 0, 55, 100, 123, 123, 0, 55, 100,
31   123, 123, 0, 100, 123, 123, 0, 100, 123, 123 };
32
33 uint16_t simulated_motorREV[SIM_STEPS] =
34 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 850, 850, 0, 0, 0, 850, 850, 0,
35   0, 600, 10, 0, 0, 600, 10 };
36
37 uint16_t simulated_motorTorque[SIM_STEPS] =
38 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 300, 300, 0, 0, 0, 300, 300, 0,
```

```
39   0, 200, 1000, 0, 0, 200, 1000 };
40
41 int main(void)
42 {
43    uint8_t simStep=0;    // step of environment variables simulation
44    char s_simStep[25];    // string to present simStep on display
45    char s_itoa_buffer[5]; // string to buffer results of itoa()
46
47    // declare a test state
48
49    struct state testState;
50
51    // declare environment variables
52
53    uint16_t emgOpen;       // current EMG level for opening direction
54    uint16_t emgClose;      // current EMG level for closing direction
55    uint8_t  stopwatchOVF; // timer overflows since T/C1 has been reset
56    uint16_t stopwatchCNT; // current counter value of T/C1
57    uint16_t motorREV;      // current motor voltage (motor revolutions)
58    uint16_t motorTorque;   // current motor current (motor torque)
59
60    // initialize system components
61
62    /* In practice, all system components will be initialized here.
63     * In this module test, the only system component is the display.
64     */
65
66    lcd_init(); // initialize display
67
68    // initialize state machine with state Init
69
70    testState.name = Init;
71    testState.motorMode = MOTOR_STOP;
72    testState.motorUtil = 0.0;
73    testState.stopwatch = SW_STOPPED;
74
75    // run state machine in endless loop
76
77    while (1)
78    {
79       // acquire current environment variables
80
81       /* In practice, the environment variables will be delivered by
82        * the system components.
83        * In this module test, the environment variables are simulated.
84        * Therefore, the values for the current simulation simulation
85        * step are used.
86        */
87
88       emgOpen      = simulated_emgOpen      [simStep];
89       emgClose     = simulated_emgClose     [simStep];
90       stopwatchOVF = simulated_stopwatchOVF [simStep];
91       stopwatchCNT = simulated_stopwatchCNT [simStep];
92       motorREV     = simulated_motorREV     [simStep];
93       motorTorque  = simulated_motorTorque  [simStep];
94
95       // refresh state with current environment variables
96
97       testState = statemachineRefresh (testState, emgOpen, emgClose,
98                                       stopwatchOVF, stopwatchCNT,
99                                       motorREV, motorTorque);
100
101      // set system components according to current state
102
103      /* In practice, all system components will be set here
104       * (according to current state).
105       * In this module test, the only system component is the
106       * display.
107       */
108
```

```
      statemachineDisplay(testState); // present current state on display

      // for module test only: present simStep on display and wait

      itoa(simStep,s_itoa_buffer,10); // convert simStep to string

      strcpy(s_simStep,"SIMULATION STEP: "); // assemble s_simStep
      strcat(s_simStep,s_itoa_buffer);       // assemble s_simStep

      lcd_set_cursor(7,53);        // set cursor on display
      lcd_puts(font5x8,s_simStep); // present s_simStep on display

      _delay_ms(5000); // wait 5 seconds

      // for module test only: go to next environment simulation step

      simStep++;

      if ( simStep >= SIM_STEPS )
      {
         simStep = 0;
      }
   }

   return 0;
}
```

Programmcode Modul adc

adc.h

```
#include <lcd/mylcd.h>   // interface for display control
#include <string.h>      // library for string operations
#include <stdlib.h>      // required for conversion integer to string
#include <lcd/font5x8.h> // font for display

// prepropcessor directives as identifiers for ADC channels
#define ADC0 0  // ADC channel 0 (EMG signal open direction)
#define ADC1 1  // ADC channel 1 (EMG signal close direction)
#define ADC2 2  // ADC channel 2 (motor voltage)
#define ADC3 3  // ADC channel 3 (motor current)
#define ADC4 4  // ADC channel 4 (film pressure sensor in thumb)
#define ADC5 5  // ADC channel 5 (clip strain sensor in wrist)

// DDRs, Ports and Pins of P0MUX, P1MUX and P2MUX (output to sensor in thumb)
#define P0MUX_DDR  DDRB   // Data Direction Register for P0MUX
#define P0MUX_PORT PORTB  // Port Register for P0MUX
#define P0MUX_PIN  0x00   // Pin for P0MUX
#define P1MUX_DDR  DDRB   // Data Direction Register for P1MUX
#define P1MUX_PORT PORTB  // Port Register for P1MUX
#define P1MUX_PIN  0x01   // Pin for P1MUX
#define P2MUX_DDR  DDRB   // Data Direction Register for P2MUX
#define P2MUX_PORT PORTB  // Port Register for P2MUX
#define P2MUX_PIN  0x02   // Pin for P2MUX

// preprocessor directives as identifiers for P0MUX, P1MUX and P2MUX pins
#define P0MUX 0
#define P1MUX 1
#define P2MUX 2

// volatile global variables for ADC results
volatile uint16_t adc_emgOpen;        // EMG signal (open direction)
volatile uint16_t adc_emgClose;       // EMG signal (close direction)
volatile uint16_t adc_motorREV;       // motor voltage (revolutions)
volatile uint16_t adc_motorTorque;    // motor current (torque)
volatile uint16_t adc_sensorThumb[3]; // film pressure sensor in thumb
```

```
36 volatile uint16_t adc_sensorWrist;     // clip strain sensor in wrist
37
38 // ADC functions family
39 void adcInit (void);
40 void adc (void);
41 uint8_t adcChannel (uint8_t);
42 uint8_t adcThumbSensorMUX (uint8_t);
43 void adcDisplay (void);
```

adc.c

```
1  #include <adc/adc1.0/adc.h> // include header (of version 1.0)
2
3  void adcInit (void)
4  {
5     // initialize ADC unit
6
7        // configure pin AVCC as source for VREF (ADC reference voltage)
8        ADMUX &=~ (1<<REFS1);
9        ADMUX |=  (1<<REFS0);
10
11       // configure ADC prescaler: 64
12       ADCSRA |=  (1<<ADPS2) | (1<<ADPS1);
13       ADCSRA &=~ (1<<ADPS0);
14
15       // enable ADC unit
16       ADCSRA |= (1<<ADEN);
17
18       // perform dummy conversion
19       ADCSRA |= (1<<ADSC);             // start dummy conversion
20       while ( ADCSRA & (1<<ADSC) ); // wait until conversion is finished
21
22    // initialize PxMUX pins as output pins (output to thumb sensor)
23
24    P0MUX_DDR |=  (1<<P0MUX_PIN); // set data direction register for P0MUX
25    P1MUX_DDR |=  (1<<P1MUX_PIN); // set data direction register for P1MUX
26    P2MUX_DDR |=  (1<<P2MUX_PIN); // set data direction register for P2MUX
27 }
28
29 void adc (void)
30 {
31    uint8_t errorlevel = 0; // declare errorlevel, initialize it with 0
32
33       // acquire motor revolutions (motor voltage) on ADC2
34    adcChannel(ADC2);                  // select pin ADC2 as ADC-channel
35    ADCSRA |= (1<<ADSC);               // start conversion
36    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
37    adc_motorREV = ADCW;               // write result to global variable
38
39    // acquire EMG signal (open direction) on ADC0
40    adcChannel(ADC0);                  // select pin ADC0 as ADC-channel
41    ADCSRA |= (1<<ADSC);               // start conversion
42    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
43    adc_emgOpen = ADCW;                // write result to global variable
44
45    // acquire EMG signal (close direction) on ADC1
46    adcChannel(ADC1);                  // select pin ADC1 as ADC-channel
47    ADCSRA |= (1<<ADSC);               // start conversion
48    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
49    adc_emgClose = ADCW;               // write result to global variable
50
51    // acquire motor torque (motor current) on ADC3
52    adcChannel(ADC3);                  // select pin ADC3 as ADC-channel
53    ADCSRA |= (1<<ADSC);               // start conversion
54    while ( ADCSRA & (1<<ADSC) );     // wait until conversion is finished
55    adc_motorTorque = ADCW;            // write result to global variable
56
```

```
57     // acquire thumb pressure (film pressure sensor) on ADC4
58
59     adcChannel(ADC4);                 // select pin ADC4 as ADC-channel
60
61     // read star-connected resistor 0 (P0MUX) in thumb pressure sensor
62     adcThumbSensorMUX(P0MUX);         // set P0MUX to HIGH
63     ADCSRA |= (1<<ADSC);              // start conversion
64     while ( ADCSRA & (1<<ADSC) );    // wait until conversion is finished
65     adc_sensorThumb[0] = ADCW;        // write result to global variable
66
67     // read star-connected resistor 1 (P1MUX) in thumb pressure sensor
68     adcThumbSensorMUX(P1MUX);         // set P1MUX to HIGH
69     ADCSRA |= (1<<ADSC);              // start conversion
70     while ( ADCSRA & (1<<ADSC) );    // wait until conversion is finished
71     adc_sensorThumb[1] = ADCW;        // write result to global variable
72
73     // read star-connected resistor 2 (P2MUX) in thumb pressure sensor
74     adcThumbSensorMUX(P2MUX);         // set P2MUX to HIGH
75     ADCSRA |= (1<<ADSC);              // start conversion
76     while ( ADCSRA & (1<<ADSC) );    // wait until conversion is finished
77     adc_sensorThumb[2] = ADCW;        // write result to global variable
78
79     // acquire wrist strain (clip strain sensor) on ADC5
80     adcChannel(ADC5);                 // select pin ADC5 as ADC-channel
81     ADCSRA |= (1<<ADSC);              // start conversion
82     while ( ADCSRA & (1<<ADSC) );    // wait until conversion is finished
83     adc_sensorWrist = ADCW;           // write result to global variable
84 }
85
86 uint8_t adcChannel (uint8_t adc_channel)
87 {
88     uint8_t errorlevel = 0; // declare errorlevel, initialize it with 0
89
90      /* selected channel -> required MUX4:0 (in register ADMUX)
91       * ADC0 -> 00000
92       * ADC1 -> 00001
93       * ADC2 -> 00010
94       * ADC3 -> 00011
95       * ADC4 -> 00100
96       * ADC5 -> 00101
97       */
98
99     // select channel (set MUX bits in ADMUX)
100    switch (adc_channel)
101    {
102        case ADC0:
103            ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX2) | (1<<MUX1) | (1<<MUX0) )
               ;
104        break;
105
106        case ADC1:
107            ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX2) | (1<<MUX1) );
108            ADMUX |=  (1<<MUX0);
109        break;
110
111        case ADC2:
112            ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX2) | (1<<MUX0) );
113            ADMUX |=  (1<<MUX1);
114        break;
115
116        case ADC3:
117            ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX2) );
118            ADMUX |=  (1<<MUX1) | (1<<MUX0);
119        break;
120
121        case ADC4:
122            ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX1) | (1<<MUX0) );
123            ADMUX |=  (1<<MUX2);
124        break;
125
```

```
126        case ADC5:
127           ADMUX &=~ ( (1<<MUX4) | (1<<MUX3) | (1<<MUX1) );
128           ADMUX |=  (1<<MUX2) | (1<<MUX0);
129        break;
130
131        // for invalid argument: set errorlevel to 1
132        default: errorlevel = 1; break;
133     }
134
135     // quit by returning errorlevel
136     return errorlevel;
137  }
138
139  uint8_t adcThumbSensorMUX (uint8_t PxMUX)
140  {
141     uint8_t errorlevel = 0; // declare errorlevel, initialize it with 0
142
143     switch (PxMUX)
144     {
145        case P0MUX:
146
147           P0MUX_PORT |=  (1<<P0MUX_PIN); // set P0MUX to HIGH
148           P1MUX_PORT &=~ (1<<P1MUX_PIN); // set P1MUX to LOW
149           P2MUX_PORT &=~ (1<<P2MUX_PIN); // set P2MUX to LOW
150
151        break;
152
153        case P1MUX:
154
155           P0MUX_PORT &=~ (1<<P0MUX_PIN); // set P0MUX to LOW
156           P1MUX_PORT |=  (1<<P1MUX_PIN); // set P1MUX to HIGH
157           P2MUX_PORT &=~ (1<<P2MUX_PIN); // set P2MUX to LOW
158
159        break;
160
161        case P2MUX:
162
163           P0MUX_PORT &=~ (1<<P0MUX_PIN); // set P0MUX to LOW
164           P1MUX_PORT &=~ (1<<P1MUX_PIN); // set P1MUX to LOW
165           P2MUX_PORT |=  (1<<P2MUX_PIN); // set P2MUX to HIGH
166
167        break;
168
169        // for invalid argument: set errorlevel to 1
170        default: errorlevel = 1; break;
171     }
172
173     // quit by returning errorlevel
174     return errorlevel;
175  }
176
177  void adcDisplay (void)
178  {
179     char s_emgOpen[25], s_emgClose[25], s_motorREV[25], s_motorTorque[25],
180           s_sensorThumb[25], s_sensorWrist[25];
181     char s_itoa_buffer[5]; // string used to buffer results of itoa()
182
183     lcd_clear(); // clear display
184
185     strcpy(s_emgOpen, "EMG open: ");
186     strcpy(s_emgClose, "EMG close: ");
187     strcpy(s_motorREV, "revolutions: ");
188     strcpy(s_motorTorque, "torque: ");
189     strcpy(s_sensorThumb, "thumb: ");
190     strcpy(s_sensorWrist, "wrist: ");
191
192     itoa(adc_emgOpen,s_itoa_buffer,10);
193     strcat(s_emgOpen,s_itoa_buffer);
194
195     itoa(adc_emgClose,s_itoa_buffer,10);
```

```
196     strcat(s_emgClose,s_itoa_buffer);
197
198     itoa(adc_motorREV,s_itoa_buffer,10);
199     strcat(s_motorREV,s_itoa_buffer);
200
201     itoa(adc_motorTorque,s_itoa_buffer,10);
202     strcat(s_motorTorque,s_itoa_buffer);
203
204     itoa(adc_sensorThumb[0],s_itoa_buffer,10);
205     strcat(s_sensorThumb,s_itoa_buffer);
206     strcat(s_sensorThumb," ");
207     itoa(adc_sensorThumb[1],s_itoa_buffer,10);
208     strcat(s_sensorThumb,s_itoa_buffer);
209     strcat(s_sensorThumb," ");
210     itoa(adc_sensorThumb[2],s_itoa_buffer,10);
211     strcat(s_sensorThumb,s_itoa_buffer);
212
213     itoa(adc_sensorWrist,s_itoa_buffer,10);
214     strcat(s_sensorWrist,s_itoa_buffer);
215
216     // write values to display
217     lcd_set_cursor(0,0);
218     lcd_puts(font5x8,s_emgOpen);
219     lcd_set_cursor(0,9);
220     lcd_puts(font5x8,s_emgClose);
221     lcd_set_cursor(0,18);
222     lcd_puts(font5x8,s_motorREV);
223     lcd_set_cursor(0,27);
224     lcd_puts(font5x8,s_motorTorque);
225     lcd_set_cursor(0,36);
226     lcd_puts(font5x8,s_sensorThumb);
227     lcd_set_cursor(0,45);
228     lcd_puts(font5x8,s_sensorWrist);
229 }
```

main.c (Modultest)

```
1  #define F_CPU 8000000          // CPU frequency (macro for delay.h)
2
3  #include <avr/io.h>            // AVR device-specific IO definitions
4  #include <util/delay.h>        // functions for delay loops
5
6  #include <adc/adc1.0/adc.h>    // include adc module (version 1.0)
7
8  int main (void)
9  {
10     adcInit();
11
12     lcd_init();
13
14     while (1)
15     {
16         adc();
17
18         adcDisplay();
19
20         _delay_ms (200);
21     }
22
23     return 0;
24 }
```

Programmcode Modul prosthesiscontrol

main.c

```
1 #include <avr/io.h>          // AVR device-specific IO definitions
2 #include <avr/interrupt.h>   // library for interrupt handling
3
4 #include <lcd/mylcd.h>       // interface for display control
5 #include <lcd/font5x8.h>     // font for display
6
7 #include <avr/io.h>          // AVR device-specific IO definitions
8 #define F_CPU 8000000        // CPU frequency (macro for delay.h)
9 #include <util/delay.h>      // functions for delay loops
10
11 // include ADC module, motor module and state machine module
12 #include <adc/adc1.0/adc.h>
13 #include <motor/motor1.0/motor.h>
14 #include <statemachine/statemachine1.0/statemachine.h>
15
16 // DDRs, Ports and Pins of pins for observing ISR execution
17 #define ISR_REFR_DDR    DDRB  // Data Direction Register of pin ISR_REFR
18 #define ISR_REFR_PORT   PORTB // Port Register of pin ISR_REFR
19 #define ISR_REFR_PIN    0x03  // Pin of pin ISR_REFR
20 #define ISR_SWOVF_DDR   DDRB  // Data Direction Register of pin ISR_SWOVF
21 #define ISR_SWOVF_PORT  PORTB // Port Register of pin ISR_SWOVF
22 #define ISR_SWOVF_PIN   0x04  // Pin of pin ISR_SWOVF
23
24 // DDR, Port and Pin for observing duration: motor measure mode <-> ADC
25 #define MOTOR_MEASURE_MODE_DDR   DDRB  // Data Direction Register
26 #define MOTOR_MEASURE_MODE_PORT  PORTB // Port Register
27 #define MOTOR_MEASURE_MODE_PIN   0x05  // Pin
28
29 volatile struct  state prosthesisState;  // system state of prosthesis
30 volatile uint8_t stopwatch_overflows;    // stopwatch overflow counter
31
32 int main (void)
33 {
34    // declare string used to present current prosthesis state on display
35    char s_name[25];
36
37    // initialize system components
38
39       // initialize display
40       lcd_init();
41       lcd_clear();
42
43       // initialize ADC module
44       adcInit();
45
46       // initialize motor module
47       motorInit();
48
49       // initialize system state as Init
50       prosthesisState.name       = Init;
51       prosthesisState.motorMode = MOTOR_STOP;
52       prosthesisState.motorUtil = 0.0;
53       prosthesisState.stopwatch = SW_STOPPED;
54
55       // initialize timer used as stopwatch by state machine module
56       // (T/C2, normal mode, stopped, local interrupt on OVF)
57       TIMSK |= (1<<TOIE2);      // enable local overflow interrupt
58       TCNT2 = 0;                // reset counter
59       stopwatch_overflows = 0;  // reset stopwatch overflow counter
60
61       // initialize timer that triggers state refresh every 20 ms
62       // (T/C0, CTC mode, prescaler 1024, interrupt on Compare Match)
63       TCCR0 |= (1<<WGM01);  // CTC mode
64       TCCR0 |= (1<<CS02) | (1<<CS00);  // start T/C0, prescaler 1024
65       OCR0   = 156;  // compare value, Compare Match every 20 ms
```

```
66          TIMSK |= (1<<OCIE0);  // enable local Compare Match interrupt
67
68          // initialize pins for observation issues (set DDR to output)
69
70             // pins for observation of ISR execution
71             ISR_REFR_DDR  |= (1<<ISR_REFR_PIN);
72             ISR_SWOVF_DDR |= (1<<ISR_SWOVF_PIN);
73
74             // pin for observation of duration motor measure mode <-> ADC
75             MOTOR_MEASURE_MODE_DDR |= (1<<MOTOR_MEASURE_MODE_PIN);
76
77       // enable global interrupt to start interrupt handled prosthesis control
78       sei();
79
80
81       // refresh display in endless loop
82       while (1)
83       {
84          // show ADC conversion results on display
85          adcDisplay();
86
87          // show name of current prosthesis state on display
88
89          strcpy(s_name, "state: ");
90
91          switch (prosthesisState.name)
92          {
93             case Init:       strcat(s_name,"Init");       break;
94             case TimerOpen:  strcat(s_name,"TimerOpen");  break;
95             case TimerClose: strcat(s_name,"TimerClose"); break;
96             case Open:       strcat(s_name,"Open");       break;
97             case Close:      strcat(s_name,"Close");      break;
98             default:         strcat(s_name,"INVALID");    break;
99          }
100
101         lcd_set_cursor(10,54);
102         lcd_puts(font5x8,s_name);
103
104         // delay display refresh
105         _delay_ms(400);
106      }
107
108      return 0;
109   }
110
111   ISR (TIMER0_COMP_vect)
112   {
113      // set pin for observation of this ISR execution to HIGH
114      ISR_REFR_PORT |= (1<<ISR_REFR_PIN);
115
116      // save AVR status register to protect I-bit
117      uint8_t sreg = SREG;
118
119      uint8_t timestamp;
120
121      // refresh ADC values
122
123         // switch motor to measure mode, observation pin to HIGH
124         motorOperation(MOTOR_MEASURE,0.0);
125         MOTOR_MEASURE_MODE_PORT |= (1<<MOTOR_MEASURE_MODE_PIN);
126
127         // wait 2 ms (until motor voltage is available on ADC channel)
128         timestamp = TCNT0;
129         while ( (TCNT0-timestamp) < 16 );
130
131         // observ. pin to LOW (since adc of revolutions is the very next step)
132         MOTOR_MEASURE_MODE_PORT &=~ (1<<MOTOR_MEASURE_MODE_PIN);
133
134         // perform ADC for all external signals
135         adc();
```

```
136
137         // switch motor back to current operation mode
138         motorOperation(prosthesisState.motorMode, prosthesisState.motorUtil);
139
140     // refresh system state of prosthesis
141     prosthesisState = statemachineRefresh
142                       ( prosthesisState, adc_emgOpen, adc_emgClose,
143                         stopwatch_overflows, TCNT2, adc_motorREV,
144                         adc_motorTorque );
145
146     // set system components according to refreshed system state
147
148         // set motor operation
149         motorOperation (prosthesisState.motorMode, prosthesisState.motorUtil);
150
151         // enable or stop/reset stopwatch (T/C2)
152         if (prosthesisState.stopwatch == SW_STOPPED)
153         {
154             // stop T/C2
155             TCCR2 &=~ ( (1<<CS22) | (1<<CS21) | (1<<CS20) );
156
157             // reset T/C2 counter
158             TCNT2 = 0;
159         }
160         else if (prosthesisState.stopwatch == SW_RUNNING)
161         {
162             // enable T/C2 (prescaler 256)
163             TCCR2 |=  (1<<CS22) | (1<<CS21);
164             TCCR2 &=~ (1<<CS20);
165         }
166
167     // restore AVR status register to protect I-bit
168     SREG = sreg;
169
170     // set pin for observation of this ISR execution to LOW
171     ISR_REFR_PORT &=~ (1<<ISR_REFR_PIN);
172 }
173
174 ISR (TIMER2_OVF_vect)
175 {
176     // set pin for observation of this ISR execution to HIGH
177     ISR_SWOVF_PORT |= (1<<ISR_SWOVF_PIN);
178
179     // save AVR status register to protect I-bit
180     uint8_t sreg = SREG;
181
182     // raise stopwatch counter by 1
183     if (stopwatch_overflows < 255)
184         stopwatch_overflows++;
185     else
186         stopwatch_overflows = 0;
187
188     // restore AVR status register to protect I-bit
189     SREG = sreg;
190
191     // set pin for observation of this ISR execution to LOW
192     ISR_SWOVF_PORT &=~ (1<<ISR_SWOVF_PIN);
193 }
```