

**UMSETZUNG EINER
OBJEKTORIENTIERTEN
SOFTWARE-ARCHITEKTUR IN C++ ZUR
PLATTFORMUNABHÄNGIGEN
ANSTEUERUNG EINES
ROBOTERMANIPULATORS**

**Bachelorarbeit vorgelegt von
Willi Penner
Matrikelnummer: 1106136**

Angefertigt im Studiengang Mechatronik
(Bachelor of Science, B. Sc.)

Am Fachbereich Ingenierwissenschaften und Mathematik der Fachhochschule Bielefeld

Tag der Abgabe: 24.07.2020
Sommersemester 2020

Erstprüfer: Prof. Dr. rer. nat. Martin Hülse
Zweitprüfer: Prof. Dr. rer. nat. Axel Schneider

Inhaltsverzeichnis

1 Abstract	4
2 Abkürzungsverzeichnis	5
3 Einleitung	6
3.1 Ausgangssituation	6
3.2 Aufgabenstellung	7
3.3 Projektorganisation	8
3.4 Eingesetzte Software	8
4 Analyse der Software-Architektur	9
5 Vorgaben durch Hardware	11
5.1 Funktionsweise der Servomotoren	11
5.2 Befehle für den Controller	12
6 Implementierung der Software-Achritektur	14
6.1 ISerialCom/SerialCom	14
6.2 IPololu / Pololu	16
6.3 IServoMotor/ServoMotor	16
6.4 Testen der implementierten Klassen	19
7	21
8 Fazit und Ausblick	21
9 Abbildungsverzeichnis	22
10 Eidesstattliche Versicherung	23
A Anhang	24
A.1 Datenblatt MicroServo SG90	24
A.2 Datenblatt Servo K-Power DM1500	25
A.3 Pololu Maestro Servo Controller User's Guide (Seiten 54-58)	26
A.4 Quellcode: Headerfile von ISerialCom und SerialCom	31
A.5 Quellcode: Sourcefile von SerialCom	33
A.6 Quellcode: Headerfile von IPololu und Pololu	37
A.7 Quellcode: Sourcefile von Pololu	39
A.8 Quellcode: Headerfile von IServoMotor und ServoMotor	43
A.9 Quellcode: Sourcefile ServoMotor	45

A.10 Quellcode: Headerfile TestUnits.hpp 48

1 Abstract

Text

2 Abkürzungsverzeichnis

Abb.	Abbildung
ILIAS	Integriertes Lern-, Informations- und Arbeitskooperations-System
MEX	Modularer Experimentierbaukasten
μs	Mikrosekunden
ms	Millisekunden

3 Einleitung

Hintergrund dieser Bachelorarbeit ist ein Semesterprojekt aus dem Sommersemester 2019, welches dann im Wintersemester 2019/2020 weitergeführt wurde. Das Projekt mit dem Namen „modularer Experimentierbaukasten“ (im weiteren Verlauf mit MEX abgekürzt) hatte zum Ziel einen modularen Roboterbaukasten zu entwickeln, welcher im Rahmen des Moduls Robotik in den Praktika eingesetzt werden soll. Mithilfe des MEX sollen Studierende der Fachhochschule Bielefeld erste praktische Erfahrungen in der Roboterprogrammierung sammeln und gelernte Inhalte aus den Vorlesungen in der Praxis anwenden können.

Die Ergebnisse aus diesem Projekt sowie ihre Dokumentation dienen als Ausgangslage für diese Bachelorarbeit.

3.1 Ausgangssituation

Zum Beginn der Bachelorarbeit befand sich das zuvor genannte Semesterprojekt kurz vor dem Abschluss. Durch die Projektgruppe ist die Hardware beschafft und die ersten Prototypen aufgebaut worden. Nach der Übergabe der Hardware konnte sowohl der Robotermanipulator als auch eine Teststation für die Servomotoren aufgebaut werden. Im folgenden ist die für die Bachelorarbeit bereitgestellte Hardware gelistet:

Hardware der Teststation

- Controller: Mini Maestro 12-Channel USB Servo Controller (Abb. 1 auf Seite 6)
- Servomotoren: Zwei Micro Servo SG90 (Abb. 2 auf Seite 7)
- Spannungswandler: DC 7-32V to 0.8-28V Adjustable Step Down Power Supply Module Buck Converter (Abb. 1 auf Seite 6)

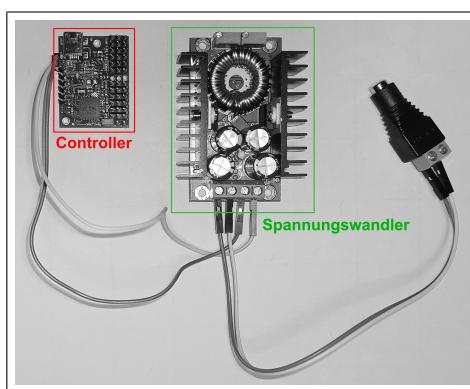


Abbildung 1: Controller und Spannungswandler (Quelle: eigene Aufnahme)

Hardware für den Robotermanipulator (3 auf Seite 7)

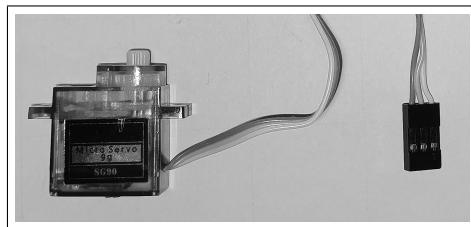


Abbildung 2: Servomotor SG90 (Quelle: eigene Aufnahme)

- Controller: Mini Maestro 12-Channel USB Servo Controller
- Servomotoren: Sechs K-Power DM1500 (Abb. 4 auf Seite 7)
- Diverse mechanische Bauteile für den Aufbau des Roboter manipulators

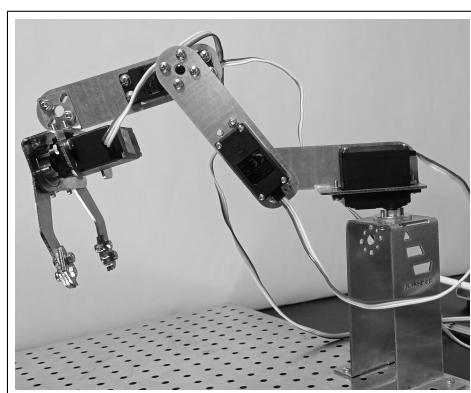


Abbildung 3: Robotomanipulator: Konfiguration mit Greifer (Quelle: eigene Aufnahme)



Abbildung 4: Servomotor K-Power DM1500 (Quelle: eigene Aufnahme)

3.2 Aufgabenstellung

Aufgabenstellung der Bachelorarbeit ist die „Umsetzung einer objektorientierten Software-Architektur in C++ zur plattformunabhängigen Ansteuerung eines Robotermanipulators“ für die bereits angeschaffte Hardware. Dabei umfasst die Aufgabenstellung:

- die Analyse und Erweiterung der durch die Projektgruppe erstellte Software-Architektur
- die Implementieren der Schnittstelle zur Ansteuerung des Robotermanipulators unter Windows
- die Portierung der Schnittstelle nach Linux
- das Testen der Schnittstelle an der Teststation und am Robotormanipulator sowohl unter Windows als auch unter Linux

3.3 Projektorganisation

Für die Projektbetreuung ist zu Beginn ein wöchentliches Projektmeeting festgelegt worden. Die Meetings dienten dem Projekt zugrunde liegenden interativen Entwicklungsprozess. Das bedeutet das sich die Aufgabenstellung von Meeting zu Meeting weiterentwickelt hat.

Um die sich entwickelnde Aufgabenstellung zu dokumentieren ist im ILIAS ein Forum eingerichtet worden. Zusätzlich existiert dort ein Ordner um Projektdateien hochladen zu können.

Unter GitHub ist für die programmierten Sourcefiles ein Repository eingerichtet. Dies dient sowohl dem verfolgen des Projektfortschritts durch den Projektbetreuer als auch der Datensicherung des Quellcodes. Die Kommentierung im Quellcode ist Doxygen-komform (Software-Dokumentationswerkzeug) erfolgt, so kann jederzeit mit Hilfe von Doxygen eine Quellcode-Dokumentation generiert werden.

3.4 Eingesetzte Software

Für die Durchführung des Projektes und die Erstellung der Bachelorarbeit ist folgende Software eingesetzt worden:

- **Microsoft Visio:** Erstellung des UML-Diagramms
- **Eclipse IDE for C/C++ Developers:** Programmierung in C++
- **Eclipse TeXlipse 2.0.2:** Erweiterung für Eclipse um LaTeX-Dokumente innerhalb von Eclipse zu erstellen.
- **eclox 0.12.1:** Erweiterung für Eclipse für Doxygen
- **Pololu Maestro Control Center:** Software um die Grundeinstellungen am Microcontroller und den angeschlossenen Servomotoren durchführen zu können.
- **Betriebssysteme:** Windows 10 und Ubuntu 18.04.4
- **VMware Workstation 15 Player:** Virtualisierung von Linux auf dem Windows-PC

4 Analyse der Software-Architektur

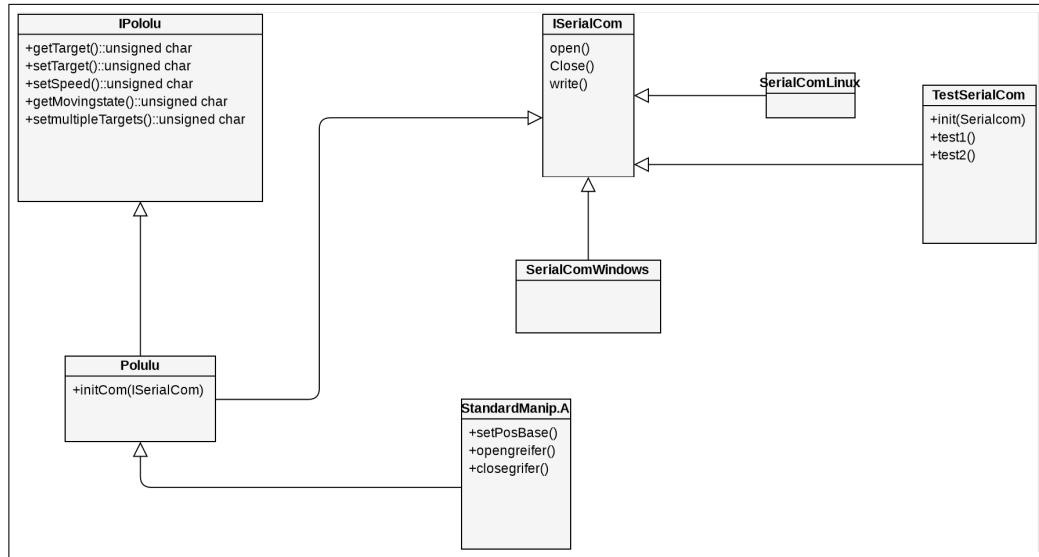


Abbildung 5: UML-Diagramm der Software-Architektur (Quelle: Projektdokumentation MEX-Projekt)

Der Abbildung 5 auf Seite 9 zeigt das UML-Diagramm der Software-Architektur, welches die Projektgruppe aus dem Wintersemester 2019/2020 erstellt hat. Bei dem Vergleich des UML-Diagramms und des Quellcodes ist festgestellt worden, dass diese sich signifikant unterscheiden.

Das UML-Diagramm sieht eine Klasse „`Pololu`“ vor, die durch Vererbung die Methoden der Klassen „`IPololu`“ und „`ISerialCom`“ erhält. Diese werden dann wiederum an die Klasse „`StandardManip.A`“ vererbt. Die Klasse „`StandardManip.A`“ bietet dem Nutzer die Funktionen einer seriellen Verbindung zu öffnen, zu schließen und Positionswerte für einen Servomotor zu setzen.

Die Namen „`IPololu`“ und „`ISerialCom`“ suggerieren, dass es sich um Interface-Klassen handelt, dabei handelt es sich um einfache Klassen. Aus dem UML-Diagramm ist ebenfalls nicht ersichtlich welche Klassen Attribute enthalten bzw. welche Übergabeparameter die einzelnen Funktionen haben. Im Quellcode ist lediglich die Klasse „`IPololu`“ umgesetzt, sie enthält die Funktionen um Positionswerte von angeschlossenen Servomotoren zu verändern:

Listing 1: Auszug aus dem Quellcode der Projektgruppe - Klasse: `IPololu`

```

unsigned char setTarget(HANDLE port, unsigned char channel,unsigned short target)
unsigned char setMultipleTargets(HANDLE port, unsigned char numberoftargets, unsigned char channel, unsigned short target)
unsigned char setSpeed(HANDLE port,unsigned char channel,unsigned short speed)
unsigned char getPosition(HANDLE port ,unsigned char channel, unsigned int* position)
unsigned char getmovingstate(HANDLE port)
  
```

Die umgesetzten Funktionen sind durch die Nutzung der WindowsAPI nur unter Windows kompilier- und nutzbar. Für die Herstellung der seriellen Verbindung wird eine Variable vom Typ `HANDLE` aus der `<windows.h>` genutzt. Die Handhabung der seriellen

Schnittstelle erfolgt lediglich in einer Funktion. Für die serielle Verbindung ist keine Klasse erstellt worden. Der Quellcode, der von der Projektgruppe erstellt wurde, funktioniert zwar um die Servomotoren zu testen und die Funktionsweise des Robotermanipulators vorzuführen, entspicht jedoch nicht den gewünschten Anforderungen einer objektorientierten Umsetzung.

Aus diesem Grund ist die Software-Architektur überarbeitet worden. In Abb. 6 auf Seite 10 ist das neue Konzept zu sehen. Die Implementierung und die Zusammenhänge werden im nachfolgenden Kapitel behandelt.

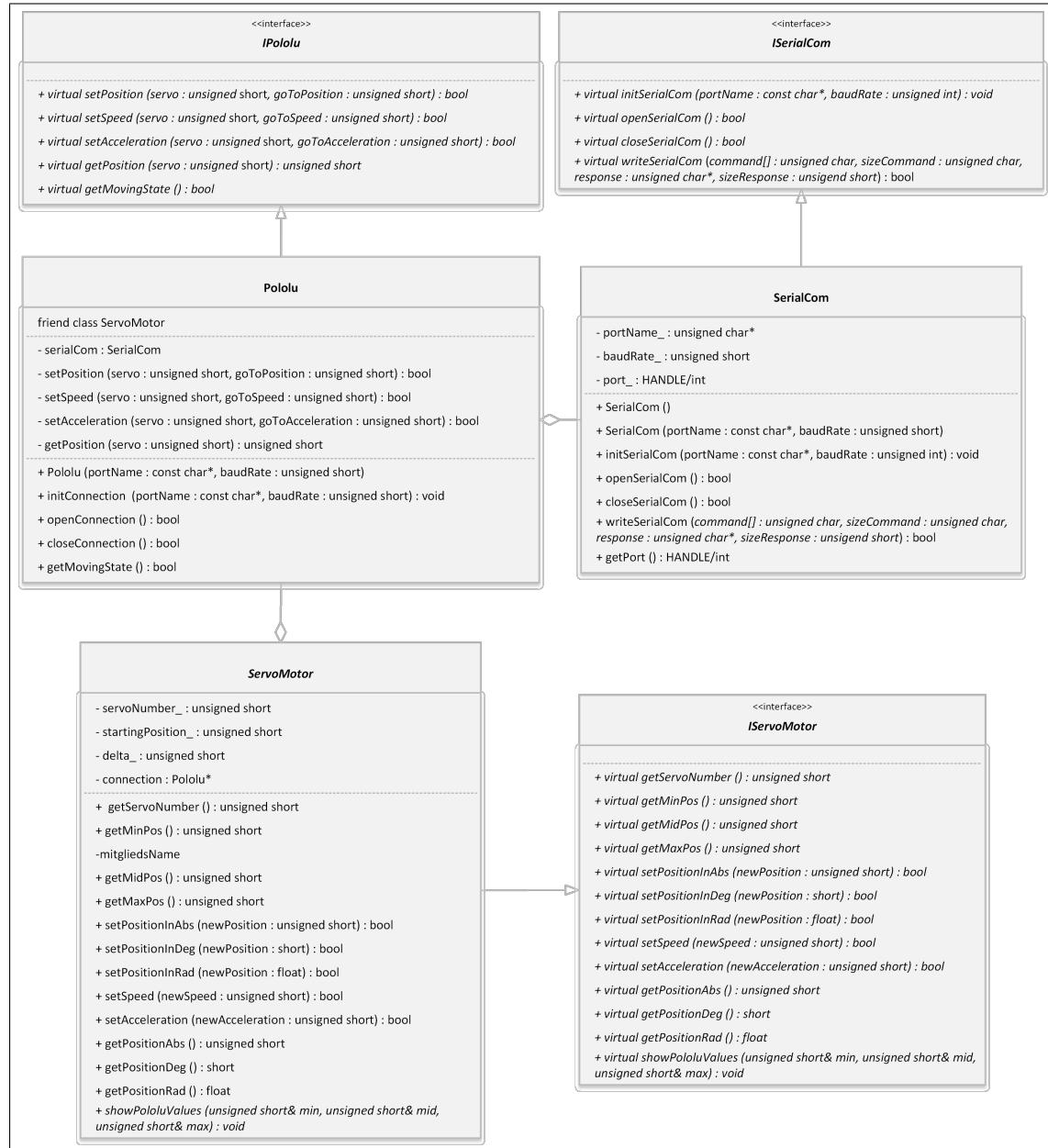


Abbildung 6: UML-Diagramm der neuen Software-Architektur (Quelle: eigene Aufnahme)

5 Vorgaben durch Hardware

5.1 Funktionsweise der Servomotoren

An dieser Stelle soll nicht die generelle Funktionsweise eines Servomotors erläutert werden, sondern es wird dargestellt, welche Positionsdaten die in diesem Projekt verwendeten Servomotoren für das Anfahren einer bestimmten Position benötigt und wie der Zusammenhang zwischen Positionsdaten des Servomotors und der Winkeleinheiten Grad und Radian ist.

Die Servomotoren werden über ein PWM¹ angesteuert. Der Winkel eines Servomotor wird über die Breite des Pulses bestimmt. Gängig ist ein 50Hz Signal, das entspricht einer Periodenlänge von 20ms). Diese Einstellungen können im Maestro Control Center für die Servomotoren vorgenommen werden. Daraus ergeben sich Positionsweite zwischen $500\mu s$ (linker Anschlag, -90°) und $2500\mu s$ (rechter Anschlag, $+90^\circ$). Dadurch ergibt sich auch die mittlere Position des Servomotors von $1500\mu s$ (entspricht der Position 0°). Einige Hersteller schränken diesen Bereich weiter ein, daher ist beim Kauf eines Servomotors darauf zu achten, welche Pulseweite sie abdecken. Jedoch ist es ratsam, bei jedem Servomotortyp diese Grenzen durch experemeteren zur ermitteln. So gibt der Hersteller des K-Power DM1500 über die Pulsweite keine Angaben preis, Händler bewerben eine Pulsweite von $1000-1500\mu s$. Durch Austesten des Servomotors K-Power DM1500 am Maestro Control Center hat sich gezeigt, dass der Servomotor die $500-2500\mu s$ beherrscht. Dies gilt auch für den Testservomotor SG90. Damit kann sich jeder Servomotor am Robotermanipulator um 180° drehen (jeweils $\pm 90^\circ$ von der zentralen Position von $1500\mu s$ aus). Die Tabelle 1 und die Formeln zeigen den Zusammenhang zwischen μs , Grad und Radian.

Einheit	$500\mu s$	$1500\mu s$	$2500\mu s$
Grad	-90°	0°	$+90^\circ$
Radian	$-\frac{\pi}{2}$	0	$+\frac{\pi}{2}$
Absolutwert	$6000 - 3600$	6000	$6000 + 3600$

Tabelle 1: Zusammenhang zwischen μs , Grad und Radian

Somit entsprechen $2000\mu s$ einem Bereich von 180° . Dadurch ergibt sich ein Umrechnungsfaktor von μs in Grad (conFactorPosToDeg) von $\frac{2000\mu s}{180^\circ} = 11.\overline{1}$. Jedoch hat sich in der Praxis gezeigt, dass sich ein Servomotor, vor allem in einer Preiskategorie 8-10 €, nicht die theoretischen Anschläge hat, die sich aus der Frequenz und der Periodenlänge des Signals errechnen lassen. Der K-Power DM1500 stoppt bereits am rechten Anschlag bei $2400\mu s$ und am linken Anschlag bei $600\mu s$. Dadurch ergibt sich

¹Pulsweitenmodulation: Modulationsverfahren, bei dem eine rechteckige Spannung einer gleichbleibenden Frequenz übermittelt wird und die Informationen in den verschiedenen Breiten der rechteckigen Pulse codiert ist. (Quelle: Onlinelixikon, <https://www.bet.de/lexikon/pulsweitenmodulation/>)

jedoch ein wesentlich komfortablerer Umrechnungsfaktor (`conFactorPosToDeg`) von $\frac{1800\mu s}{180^\circ} = 10$. Mit diesem Umrechnungsfaktor können Servopositionen in μs sowohl in Grad als auch in Radian umgerechnet werden. Der Umrechnungsfaktor kann als Konstante in der „`ServoMotor`“-Klasse unter dem Namen „`conFactorPosToDeg`“ angepasst werden.

Umrechnung von Gradmass in Bogenmass und umgekehrt:

$$Gradmass = \frac{180^\circ}{\pi} \times Bogenmass \quad (1)$$

$$Bogenmass = \frac{\pi}{180^\circ} \times Gradmass \quad (2)$$

Umrechnung von Servomotorposition in Grad und umgekehrt:

$$Grad = \frac{\mu s}{conFactorPosToDeg} \quad (3)$$

$$\mu s = Grad \times conFactorPosToDeg \quad (4)$$

Umrechnung von Servomotorposition in Radian und umgekehrt:

$$Radian = \frac{\mu s}{conFactorPosToDeg} \times \frac{\pi}{180^\circ} \quad (5)$$

$$\mu s = Radian \times \frac{180^\circ}{\pi} \times conFactorPosToDeg \quad (6)$$

5.2 Befehle für den Controller

Durch die aktuelle Festlegung auf den Mini Maestro 12-Channel USB Servo Controller, sind auch durch diese Hardware bestimmte Vorgaben in der Befehlsnutzung gegeben. Die dem Benutzer zur Verfügung gestellten manipulationsmöglichkeiten könne dem Pololu Maestro Servo Controller User's Guide¹ entnommen werden. Als Anlage A.3 auf Seite 26 dieser Bachelorarbeit ist lediglich das Kapitel „5.e Serial Servo Commands“ angefügt.

Ein Befehl, der an den Controller übertragen wird besteht, je nach Befehl, aus 1, 2 bzw. 4 Byte. Folgende Befehle bietet der Controller:

Set Target (in hex: 0x84) Der Befehl „Set Target“ dient dem setzen eines Positionswertes für den Servomotor, der dann direkt angefahren wird. Der Befehl setzt sich aus 4 Bytes (1 Befehlsbyte und 3 Datenbytes) zusammen. Das erste Byte identifiziert das Kommando, welches der Controller ausführen soll, das zweite Byte gibt das Ziel für den Befehl vor, also welcher angeschlossene Servomotor angesprochen werden soll. Das dritte und vierte Byte enthält die Position, die der Servomotor einnehmen soll (aufgeteilt in 0-6 Bits in

¹<https://www.pololu.com/docs/0J40> Stand: 13.07.2020

dritten Byte und die Bits 7-13 im vierten Byte).

Listing 2: Befehlsaufbau für Set Target

```
serialBytes[0] = 0x84; // Command byte: Set Target.  
serialBytes[1] = channel; // First data byte holds channel number.  
serialBytes[2] = target & 0x7F; // Second byte holds the lower 7 bits of target.  
serialBytes[3] = (target >> 7) & 0x7F; // Third data byte holds the bits 7-13 of target.
```

Dabei erwartet der Controller den Positionswert nicht in μs sondern als Absolutwert. Dieser errechnet sich wie folgt:

$$\text{Position} = \mu s \times \text{conFactorMyToPos} \quad (7)$$

Der Umrechnungsfaktor von Positionswert in μs nach Positionswert als Absolutwert ist durch den Controller vorgegeben und ist vom Betrag 4 (hier und im Quellcode mit `conFactorMyToPos` bezeichnet). Das bedeutet wenn ein Nutzer will, dass der Servomotor seine mittlere Position von $1500\mu s$ anfahren soll, muss über den Befehl Set Target der Wert 6000 ($1500\mu s \times 4$) übergeben werden.

Set Speed (in hex: 0x87) und Set Acceleration (in hex: 0x89) Die Befehle Set Speed und Set Acceleration sind ebenfalls wie Set Target aufgebaut. Der Befehl ist 4 Byte groß. Die Informationen in den Bytes entsprechen dem Befehl von Set Target, jedoch erwartet der Controller als Geschwindigkeits- bzw. Beschleunigungswert einen Wert zwischen 0-255.

Dabei steht 1 als Geschwindigkeitswert für die langsamste Geschwindigkeit des Servomotors und 255 für die Höchste. Ausnahme bildet die 0, sie steht für „unlimited speed“, also das Maximum, das der Servomotor leisten kann. Einen sichtbaren bzw. mit einfachen Mitteln meßbaren Unterschied zwischen den Werten 0 und 255 gibt es nicht. Daher ist in der Umsetzung der Befehle in C++ auf die 0 verzichtet worden. Es werden nur Werte von 1-255 zugelassen und es kommt auch nicht zu Geschwindigkeitssprüngen, wenn z.B. der Wert innerhalb einer Schleife erhöht bzw. verringert wird. Gleiches gilt für den Befehl der Beschleunigung.

Get Position (in hex: 0x90) Der Befehl Get Position veranlasst den Controller einen Rückantwort bereitzustellen, die von der seriellen Schnittstelle gelesen werden kann. Der Befehl Get Position besteht nur aus 2 Bytes (1 Befehlsbyte und 1 Datenbyte).

Listing 3: Befehlsaufbau für Get Target

```
serialBytes[0] = 0x90; // Command byte: Get Target.  
serialBytes[1] = channel; // First data byte holds channel number.
```

Damit weis der Controller von welchem Servomotor er den Positionswert bereitstellen soll. Der Postitions Wert der vom Controller zurückgeliefert wird, ist im gegensatz zum Set

Target nicht als Absolutwert, sondern kommt in μs direkt vom Servomotor. Das bedeutet, um in der Programmierung nicht in den Einheiten durcheinander zu geraten, muss der, von der Funktion Get Position, zurückgelieferte Wert mit dem Umrechnungsfaktor (conFactorMyToPos) multipliziert werden.

Get Moving State (in hex: 0x93) Dieser Befehl ist nur 1 Byte groß und enthält damit nur das Befehlskommando an den Controller. Der Controller antwortet dann mit einer 1 oder 0. Solange sich irgend ein angeschlossener Servomotor noch bewegt, liefert dieser Befehl als Antwort eine 1, stehen alle Servomotoren still, dann liefert der Befehl als Antwort die 0.

Wie die Abfrage der Rückantwort erfolgt wird im Abschnitt 6.1 auf Seite 14 näher erläutert.

6 Implementierung der Software-Achritektur

Für die Implementierung der Schnittstelle ist entschieden worden unter Windows die WindowsAPI zu nutzen (#include <windows.h>) und unter Linux die File Control Options in der „#include <fcntl.h>“. Die Entscheidung fiel nach Einarbeitung in das User’s Guide von Pololu, welches auch Beispielcode für die Nutzung der Serial Commands sowohl unter Windows als auch unter Linux bot. Der Quellcode aller implementierten Header- und Sourcefile sind dem Anhang bzw. der beiliegenden DVD zu entnehmen.

6.1 ISerialCom/SerialCom

Für das Handling der seriellen Schnittstelle ist ein Interface „ISerialCom“ definiert und die zugehörige Klasse „SerialCom“ implementiert worden (Quellcode siehe Anhang A.4 auf Seite 31). Das Interface setzt vier grundlegende Funktionen voraus, die in einer von ISerialCom abgeleiteten Klasse umgesetzt werden müssen. Dabei handelt es sich um folgende funktionen:

Listing 4: Auszug aus dem Headerfile SerialCom.hpp: ISerialCom

```
class ISerialCom {
public:
    virtual ~ISerialCom() {};
    virtual void initSerialCom(const char* portName, unsigned short baudRate) = 0;
    virtual bool openSerialCom() = 0;
    virtual bool closeSerialCom() = 0;
    virtual bool writeSerialCom(unsigned char command[], unsigned short sizeCommand, unsigned char *response,
        unsigned short sizeResponse) = 0;
};
```

Die Funktion „iniSerialCom“ dient dem initialisieren einer seriellen Verbindung. Dazu werden der Funktion der Portname (z.B. „COM4“ unter Windows oder „/dev/ttyACM0“) und die Baudrate mit der die serielle Verbindung arbeiten soll übergeben. Mit den Funk-

tionen „openSerialCom“ und „closeSerialCom“ lässt sich die serielle Verbindung öffnen bzw. schliessen. Zusätzlich bekommt der Nutzer von den Funktionen ein Feedback, ob der gewünschte Vorgang erfolgreich war, oder ein Fehler aufgetreten ist.

Die Funktion „writeSerialCom“ dient dem Übertragen von Befehlen an den angeschlossenen Controller (in diesem Fall ein Mini Maestro 12-Channel USB Servo Controller). Auch hier bekommt der Nutzer das Feedback über erfolgreiches oder fehlerhaftes Übertragen von Befehlen. Als Übergabeparameter bekommt die Funktion den Schreibbefehl für den Controller (command) und die Größe dieses Befehls in Bytes (sizeCommand). Handelt es sich um reine Schreibbefehle ist „response“ gleich NULL und die Größe von „sizeResponse“ ist gleich Null, da keine Antwort vom Controller erwartet wird. Bei Befehlen, die eine Antwort erwarten wird „response“ als Zeiger auf ein Feld vom Typ „unsigned char“ übergeben und die „sizeResponse“ ist dann, je nach Antwortgröße, 1 bzw. 2 Byte gross. Die Übertragung eines zusammengesetzten Befehls, wie in Kapitel 5.2 auf Seite 12 beschrieben, erfolgt unter Windows mit dem Befehl „WriteFile“ (unter Linux mit „write“). Ist die Variable „sizeResponse“ gleich 0, wird keine Antwort vom Controller erwartet. Ist der Wert 1 bzw. 2 wird eine Antwort erwartet. Der Controller stellt aufgrund des Befehl, der über das „WriteFile“ übertragen wurde, eine Antwortbereit, die jetzt noch über „ReadFile“ bzw. „read“ über die serielle Schnittstelle gelesen werden muss. Im Falle einer Abgefragten Position ist die Antwort 2 Byte groß und im Falle einer Abfrage des Bewegungsstatus ist die Antwort 1 Byte groß.

Von diesem Interface ist die Klasse „SerialCom“ abgeleitet. In dieser sind die Funktionen vom Interface implementiert. Die Klasse „SerialCom“ erhält die privaten Member für den Portnamen und die Baudrate. Als dritten Member hat die Klasse die serielle Verbindung „port_“ selbst. Um den Quellcode von Vorherein sowohl für Windows als auch für Linux komplizierbar zu halten, ist mit dem Kontrukt

Listing 5: #ifdef

```
#ifdef _WIN32
    // Quellcode für Windows
    HANDLE port_;
#else
    // Quellcode für Linux
    int port_;
#endif
```

sowohl der Quellcode für Windows als auch für Linux in der selben Klasse und den selben Funktionen untergebracht. Über „#ifdef“ wird sichergestellt, dass während der Kompilierung nur der für das aktuelle Betriebssystem relevante Code übersetzt wird. Für die serielle Verbindung wird „port_“ als HANDLE definiert und unter Linux als Integer. Mit diesem Interface und der dazugehörigen Klasse kann eine serielle Schnittstelle etabliert werden und damit eine Verbindung zum Servocontroller hergestellt werden.

6.2 IPololu / Pololu

Um die Servomotoren ansteuern zu können ist ein weiteres Interface „IPololu“ definiert und die Klasse „Pololu“ implementiert worden (Quellcode siehe Anhang A.6 auf Seite 37).

Listing 6: Auszug aus dem Headerfile Pololu.hpp: IPololu

```
class IPololu {  
protected:  
    virtual bool setPosition(unsigned short servo, unsigned short goToPosition) = 0;  
    virtual bool setSpeed(unsigned short servo, unsigned short goToSpeed) = 0;  
    virtual bool setAcceleration(unsigned short servo, unsigned short goToAcceleration) = 0;  
    virtual unsigned short getPosition(unsigned short servo) = 0;  
public:  
    virtual ~IPololu(){};  
    virtual bool getMovingState() = 0;  
};
```

Das Interface „IPololu“ definiert die Funktionen, die es dem Nutzer erlauben, Servomotoren am Controller anzusteuern. Dabei können die Position, Geschwindigkeit und Beschleunigung eines Servomotors gesetzt werden. Des Weiteren bietet das Interface eine Funktion, die die aktuelle Position des Servomotor liefert. Diese genannten Funktionen sind im Interface und auch in der Klasse „Pololu“ als „protected“ gesetzt. „Public“ ist lediglich die Funktion „getMovingState“, diese liefert den Bewegungsstatus der angeschlossenen Servomotoren. Stehen alle Servomotoren still, liefert die Funktion den Wert „0“, befindet sich noch auch nur ein Servomotor in Bewegung, liefert die Funktion den Wert „1“.

Die vom Interface abgeleitete Klasse „Pololu“ steht mit der Klasse „SerialCom“ in einer Aggregationsbeziehung. Das bedeutet, dass ein Objekt vom Typ „SerialCom“ als „protected Member“ teil der Klasse Pololu ist. Damit ist es möglich mit einem erzeugten „Pololu“-Objekt eine serielle Verbindung zu etablieren. Dazu ist die Klasse „Pololu“ mit „public“-Funktionen erweitert, die es ermöglichen eine serielle Verbindung zu initialisieren und diese dann öffnen und schließen zu können. Die Funktionen „initConnection“, „openConnection“ und „closeConnection“ rufen die entsprechenden Funktionen des Objekts „serialCom“ auf. Somit kann zuerst von außerhalb der Klasse „Pololu“ nur auf die zum Verbindungsauflauf notwendigen Funktionen zugegriffen werden. Im nächsten Abschnitt wird darauf näher eingegangen. Für die Ansteuerung eines Controllers ist damit ein Pololu-Objekt notwendig.

6.3 IServoMotor/ServoMotor

Die dritte Interface-Klasse-Kombination dient dem Erzeugen von Servomotor-Objekten (Quellcode siehe Anhang A.8 auf Seite 43). Jeder angeschlossene Servomotor wird in C++ durch ein Objekt der Klasse „ServoMotor“ representiert.

Listing 7: Auszug aus dem Headerfile ServoMotor.hpp: IServoMotor und ServoMotor

```
class IServoMotor {
public:
    virtual ~IServoMotor(){}; 
    virtual unsigned short getServoNumber() = 0;
    virtual unsigned short getMinPos() = 0;
    virtual unsigned short getMidPos() = 0;
    virtual unsigned short getMaxPos() = 0;
    virtual bool setPositionInAbs(unsigned short newPosition) = 0;
    virtual bool setPositionInDeg(short newPosition) = 0;
    virtual bool setPositionInRad(float newPosition) = 0;
    virtual bool setSpeed(unsigned short newSpeed) = 0;
    virtual bool setAcceleration(unsigned short newAcceleration) = 0;
    virtual unsigned short getPositionInAbs() = 0;
    virtual short getPositionInDeg() = 0;
    virtual float getPositionInRad() = 0;
    virtual void showPololuValues (unsigned short& min, unsigned short& mid, unsigned short& max) = 0;
};

class ServoMotor : public IServoMotor{
private:
    const short maxDeg = 90;      //maximum degree allowed
    const float maxRad = M_PI/2;   //maximum radiant allowed, M_PI is the constant from <cmath> for the number Pi
    const short maxSpeed = 255;    //maximum value for the speed
    const short maxAcceleration = 255; //maximum value for the acceleration
    const short minSpeed = 1;      //minimum value for the speed
    const short minAcceleration = 1; //minimum value for the acceleration
    const short confactorDegToPos = 10; //conversion factor from degrees to position
    const short confactorMyToPos = 4; //conversion factor to convert microseconds (position value of a servo) to
                                    //position values
    unsigned short servoNumber_;
    unsigned short startingPosition_; //startPosition is the center position of a servo, in most cases it is
                                    //value of 6000 (1500microseconds * 4)
    unsigned short delta_;
    Pololu *connection_ = NULL;
public:
    ServoMotor(unsigned short servo, unsigned short startingPosition, unsigned short delta, Pololu *connection);
```

Bei dem Erzeugen eines Objektes vom Typ „ServoMotor“ muss dieser zwingend über den Konstruktor mit Übergabeparametern initialisiert werden. Über den Konstruktor wird dem Objekt, die Servonummer (Anschlussport auf dem Controller. Für den hier genutzten Mini Maestro 12-Channel USB Servo Controller sind es die Anschlüsse 1-12), die Startposition (damit ist die mittlere Position des Servomotors gemeint, die als Ausgangslage für Bewegungen dient), ein Delta (gibt den Positionsreich an, den der Servomotor nach Links und Rechts von der Startposition aus drehen kann) und als Zeiger wird dem Servomotor-Objekt die serielle Verbindung in Form des Pololu-Objektes, mitgegeben.

Das Servomotor-Objekt bietet dem Nutzer alle Manipulationsmöglichkeiten die der Servo-Controller unterstützt. Die Klasse „ServoMotor“ greift dazu auf die Funktionen des Pololu-Objektes zurück. Allerdings wäre das nicht ohne weiteres möglich, da die benötigten Funktionen in der Klasse „Pololu“ „protected“ sind. Damit die Klasse „ServoMotor“ die Funktionen dennoch nutzen kann ist in der Klasse „Pololu“ die Klasse „ServoMotor“ als „friend class“ definiert, damit kann aus der Klasse „ServoMotor“ auf die „protected“ Funktionen der Klasse „Pololu“ zugegriffen werden.

Damit ist sichergestellt, dass aus der Main-Klasse oder einer anderen Stelle des Programmes wo ein Objekt vom Typ „Pololu“ erzeugt wird, zwar auf die Funktionen zugegriffen werden kann um eine serielle Verbindung zu etablieren, die angeschlossenen Servomotoren jedoch

nur von einem „ServoMotor“-Objekt manipuliert werden können.

Desweiteren enthält die Klasse „ServoMotor“ notwendige Parameter für die Festlegung von Grenzen und Umrechnung von Positions値en. Wie bereits in Kapitel 5.1 auf Seite 11 erwähnt, arbeitet der Servomotor intern mit Positionsangaben in μs . Dem Nutzer werden durch die Klasse „ServoMotor“ drei Möglichkeiten für die Positionsangabe geboten.

setPositionInAbs: Mit dieser Funktion ist es möglich die Positionsangabe für den Servomotor in absoluten Werten anzugeben. Durch die Vorgabe des Controller bedeutet dies, wenn z.B. der Servomotor seine mittlere Position von $1500\mu s$ anfahren soll, erwartet der Controller vom Benutzer einen Positions値 von 6000 ($1500\mu s \times 4$). Dadurch ergibt sich ein Arbeitsbereich für den Servomotor von 6000 (als mittlere Position) ± 3600 .

setPositionInDeg: Eine zweite Möglichkeit ist das Setzen der Position durch Angabe der Position in Grad. Durch die Konstante „maxDeg“ wird der Bereich der möglichen Eingaben auf einen Bereich von $\pm 90^\circ$ eingeschränkt. Die Einschränkungen ergeben sich durch die Bauweise der Servomotoren. Dadurch dass der Controller einen Positions値 als Absolutwert zwischen 2400 und 9600 erwartet muss innerhalb der Funktion zuerst die Gradzahl in den Absolutwert umgerechnet werden. Der Factor für die Umrechnung ist 10 (conFactorDegToPos). Zuerst wird die Gradzahl in Absolutwert umgerechnet und das Ergebnis im Anschluss zur mittleren Position addiert bzw. substrahiert (je nach Vorzeichen der Gradzahl).

setPositionInRad: Die dritte Möglichkeit ist die Eingabe einer Position in Radian. Durch die Konstante „maxRad“ wird der Bereich der möglichen Eingaben auf einen Bereich von $\pm \frac{\pi}{2}$ eingeschränkt. Hier muss zuerst der Radian in Grad und im Anschluss in den Positions値 umgerechnet werden. Die Formeln dafür sind auf der Seite 12 zu sehen.

Mit Hilfe der Funktion „getPosition“ kann der aktuelle Positions値 eines Servomotors erfragt werden. Auch in diesem Fall kann sich der Nutzer den aktuellen Positions値 als Absolutwert, als Gradmass bzw. als Bogenmass geben lassen. Hier nochmal der Hinweis, dass der Positions値, der vom Controller als Rückgabewert geliefert wird, in μs ist und erst noch in den Absolutwert umgerechnet werden muss. Die Besonderheit des Controllers ist, er liefert den Positions値 in μs , erwartet aber zum setzen einer neuen Position den Wert als Absolutwert. Hier ist Vorsicht geboten. Daher hat der Nutzer nur die drei Möglichkeiten Positions値e anzugeben (Absolutwert, Grad, Radian). Positions値e in μs kann der Nutzer nicht verwenden.

Lediglich über die Funktion „showPololuValues“ kann der Nutzer sich vom „ServoMotor“-Objekt die Minimum-, Mittel- und Maximalposition als Wert in μs anzeigen lassen. Dies dient der Überprüfung der Einstellungen im Maestro Control Center. Für jeden neuen

Servomotor müssen zuerst Einstellungen im Maestro Control Center getätigt werden. Sie erfolgen ausschließlich in μs . Damit der Nutzer überprüfen kann, ob die Einstellungen die er dem „ServoMotor“-Objekt mit den Einstellungen im Maestro Control Center übereinstimmen, steht ihm die Funktion „showPololuValues“ zur Verfügung.

Die Methoden „getMinPos“, „getMidPos“ und „getMaxPos“ liefern dem Nutzer Informationen darüber, wie die Einstellungen für ein bestimmtes „ServoMotor“-Objekt sind. Die Positionswerte, die diese Funktionen liefert sind Absolutwerte. Sie errechnen sich aus den bei der Initialisierung des Objektes angegebenen Werten für die Mittlere-/Ausgangsposition mit dem dazugehörigen Delta (Schwänkbereich nach Links und Rechts).

Die Set- und Get-Methoden der „ServoMotor“-Klasse können aufgrund ihrer „friend class“-Beziehung zur „Pololu“-Klasse auf dessen Private bzw. durch „protected“ geschützte Methoden zugreifen.

6.4 Testen der implementierten Klassen

Um die Funktionalität der implementierenden Klassen „SerialCom“, „Pololu“ und „ServoMotor“ zu testen, ist ein weiteres Header- und Sourcefile angelegt worden (TestUnits.hpp, Quellcode siehe Anhang A.10 auf Seite 48). Das Sourcefile „TestUnits.cpp“ enthält aktuell 7 Funktionen um die Funktionalität der Schnittstelle zu testen.

1. Eine Funktion Namens „wait“. Bei dieser Funktion handelt es sich um eine Hilffunktion, die bei der Nutzung der Servomotoren dafür genutzt wird um das ablaufende Programm zu stoppen und eine bestimmte Zeit abzuwarten. Damit Servomotoren die Chance haben Bewegungen zu Ende auszuführen, bevor sie einen neuen Befehl erhalten. Ähnlicher Effekt wird mit folgender Codezeile erreicht:

Listing 8: Aufforderung zum Warten

```
while(conn.getMovingState());
```

Diese Zeile sorgt im Quellcode dafür, dass der nächste Befehl erst aufgeführt wird, wenn alle Servomotoren wieder stillstehen. Solange dies nicht der Fall ist, befindet sich das Programm in einer Schleife in der nichts getan wird. „conn“ ist dabei das „Pololu“-Objekt, welches die serielle Verbindung zum physikalischen Controller ist.

2. Die Funktion „testOpenClose“ dient dem Test der seriellen Verbindung selbst. In dieser Funktion werden unterschiedliche Tests mit der Verbindung durchgeführt:
 - Definieren eines „Pololu“-Objektes
 - Öffnen der definierten Verbindung
 - Öffnen der selben bereits offenen Verbindung
 - Schließen der Verbindung

- Schließen derselben bereits geschlossenen Verbindung
- Neu initialisieren des „Pololu“-Objektes mit einem unbekannten Portnamen und öffnen dieser Verbindung
- Neu initialisieren des „Pololu“-Objektes und mehrfachen Öffnen der Verbindung hintereinander (100x)

Mit diesen Test wird das Verhalten des „Pololu“-Objektes getestet und überprüft, ob es dem gewünschten Verhalten entspricht. Als ein Beispiel sei hier das Öffnen einer Verbindung genannt. Eine bereits geöffnete Verbindung zu einem COM-Port kann nicht erneut geöffnet werden. Um sicherzustellen, dass es nicht dazu kommen kann, wird zu Beginn der „openSerialCom“-Funktion die Verbindung pauschal geschlossen. Damit ist sichergestellt, dass sich das Öffnen nicht doppelt.

3. In der Funktion „testSetGetMethods“ wird das Schreiben und Lesen auf den Controller durch die Klassen „ServoMotor“, „Pololu“ und „SerialCom“ überprüft. **HINWEIS:** Diese Funktion darf nur in der Testumgebung ohne zusammengebaute Roboter manipulator getestet werden. Dieser Hinweis ist auch im Quellcode und der Doxygen-Dokumentation vermerkt.
4. Des Weiteren existieren zwei Funktionen (testMEXMovementSetting1 und testMEX-MovementSetting2), die dem Test am Roboter manipulator dienen. **HINWEIS:** Gründliches lesen der Doxygen-Dokumentation, da vor der Verwendung dieser Funktion der Roboter manipulator auf eine bestimmte Weise zusammengesetzt sein muss und Einstellungen im Maestro Control Center vorgenommen werden müssen. Die erste Version dient dem Test des Roboter manipulators mit angebautem Greifer. Die Funktion lässt den Roboter manipulator eine Box auf dem Untergrund ansteuern, hebt diese dann an und platziert sie an einer anderen Stelle. Die Version 2 ist mit dem Stifthalter ausgestattet und fährt mit dem Stift 5 unterschiedliche Punkte auf dem Papier an. **HINWEIS:** Auch für diesen Test sind Umbauten und Anpassungen der Einstellungen notwendig.
5. Für den Einzeltest der Klasse „SerialCom“ ist die Funktion „testSerialCom“ vorhanden. Diese Testet die Funktionen der Klasse, initialisieren einer seriellen Verbindung mit und ohne Nutzung des Konstruktors, Öffnen und Schließen der Verbindung, sowie das Schreiben und Lesen.
6. Die Funktion „testPololu“ dient dem Test der Klasse „Pololu“ vor allem, ob der Zugriff auf die „protected“-Funktionen möglich ist.

7

Text

8 Fazit und Ausblick

Text

9 Abbildungsverzeichnis

1	Controller und Spannungswandler (Quelle: eigene Aufnahme)	6
2	Servomotor SG90 (Quelle: eigene Aufnahme)	7
3	Robotormanipulator: Konfiguration mit Greifer (Quelle: eigene Aufnahme)	7
4	Servomoter K-Power DM1500 (Quelle: eigene Aufnahme)	7
5	UML-Diagramm der Software-Architektur (Quelle: Projektdokumentation MEX-Projekt)	9
6	UML-Diagramm der neuen Software-Architektur (Quelle: eigene Aufnahme)	10

10 Eidesstattliche Versicherung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

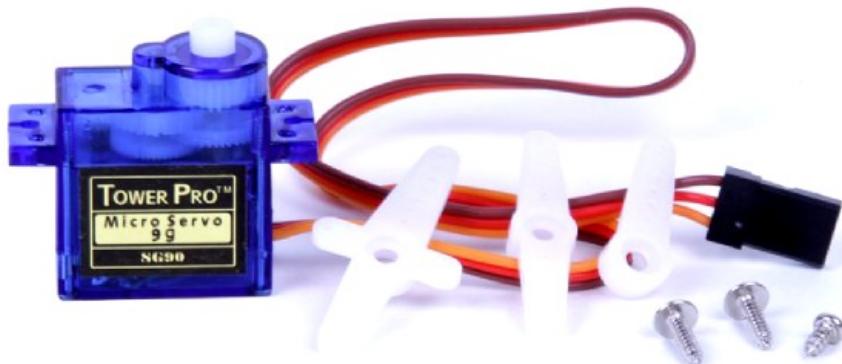
(Willi Penner)

A Anhang

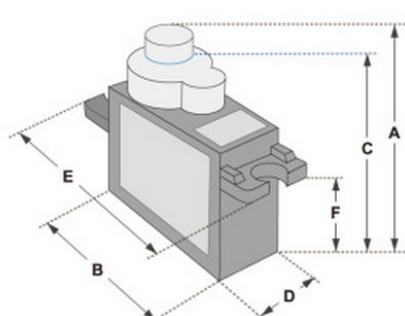
A.1 Datenblatt MicroServo SG90

SERVO MOTOR SG90

DATA SHEET



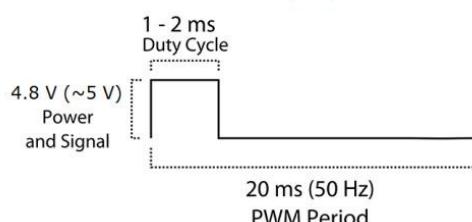
Tiny and lightweight with high output power. Servo can rotate approximately 180 degrees (90 in each direction), and works just like the standard kinds but smaller. You can use any servo code, hardware or library to control these servos. Good for beginners who want to make stuff move without building a motor controller with feedback & gear box, especially since it will fit in small places. It comes with a 3 horns (arms) and hardware.



Dimensions & Specifications	
A (mm)	: 32
B (mm)	: 23
C (mm)	: 28.5
D (mm)	: 12
E (mm)	: 32
F (mm)	: 19.5
Speed (sec)	: 0.1
Torque (kg-cm)	: 2.5
Weight (g)	: 14.7
Voltage	: 4.8 - 6

Position "0" (1.5 ms pulse) is middle, "90" (~2ms pulse) is middle, is all the way to the right, "-90" (~1ms pulse) is all the way to the left.

PWM=Orange (⊿⊿)
Vcc = Red (+)
Ground=Brown (-)



A.2 Datenblatt Servo K-Power DM1500

Ein Datenblatt für den Servomotor K-Power DM1500 ist im Netz nicht auffindbar gewesen. Die Informationen in nachfolgender Tabelle entstammen der Webseite des Herstellers: https://m.kpower.com/product/products_rc_servo_airplane_servos/DM1500.html (Stand:

Item No.	DM1500	
Torque	15.5kg-cm/4.8V	16.3kg-cm/6.0V
Speed	0.19sec/60°/4.8V	0.18sec/60°/6.0V
Weight	58g	
Size (LxWxH)	40mm*20mm*37.5mm	
Motor Type	DC Motor	
Gear	Metal	
Ball Bearing	Dual BB	
Feature	Digital metal gear	

10.07.2020)

A.3 Pololu Maestro Servo Controller User's Guide (Seiten 54-58)

Pololu Maestro Servo Controller User's Guide

© 2001–2019 Pololu Corporation

```

add 7 zeros to the end of the message):
CRC-7 Polynomial = [1 0 0 0 1 0 0 1]
message = [1 1 0 0 0 0 0 0 1] [1 0 0 0 0 0 0 0 0] 0 0 0 0 0 0 0 0
Steps 3, 4, & 5:
      1 0 0 0 1 0 0 1 ) 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      XOR 1 0 0 0 1 0 0 1
      -----
      1 0 0 1 0 0 0 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
shift  ----> 1 0 0 0 1 0 0 1 | | | | | | | | | | | | | | | | | | | | | | | | | | |
      -----
      1 1 0 0 0 0 0 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      1 0 0 0 1 0 0 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      -----
      1 0 0 1 0 0 1 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      1 0 0 0 1 0 0 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      -----
      1 1 0 1 1 0 0 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      1 0 0 0 1 0 0 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      -----
      1 0 1 0 0 0 1 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      1 0 0 0 1 0 0 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      -----
      1 0 0 1 0 1 1 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      1 0 0 0 1 0 0 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
      -----
      1 1 1 0 1 0 0 = 0x17

```

So the full command packet we would send with CRC enabled is: **0x83, 0x01, 0x17**.

5.e. Serial Servo Commands

The Maestro has several serial commands for setting the target of a channel, getting its current position, and setting its speed and acceleration limits.

Set Target (Pololu/Compact protocol)

Compact protocol: **0x84, channel number, target low bits, target high bits**

Pololu protocol: **0xAA, device number, 0x04, channel number, target low bits, target high bits**

The lower 7 bits of the third data byte represent bits 0–6 of the target (the lower 7 bits), while the lower 7 bits of the fourth data byte represent bits 7–13 of the target. The target is a non-negative integer.

If the channel is configured as a servo, then the target represents the pulse width to transmit in units of quarter-microseconds. A target value of 0 tells the Maestro to stop sending pulses to the servo.

If the channel is configured as a digital output, values less than 6000 tell the Maestro to drive the line low, while values of 6000 or greater tell the Maestro to drive the line high.

For example, if channel 2 is configured as a servo and you want to set its target to 1500 µs ($1500 \times 4 = 6000$ = **01011101110000** in binary), you could send the following byte sequence:

in binary: 10000100, 00000010, 0**1110000**, 0**0101110**
 in hex: 0x84, 0x02, 0x70, 0x2E
 in decimal: 132, 2, 112, 46

Here is some example C code that will generate the correct serial bytes, given an integer “channel” that holds the channel number, an integer “target” that holds the desired target (in units of quarter microseconds if this is a servo channel) and an array called serialBytes:

```
1 | serialBytes[0] = 0x84; // Command byte: Set Target.
2 | serialBytes[1] = channel; // First data byte holds channel number.
3 | serialBytes[2] = target & 0x7F; // Second byte holds the lower 7 bits of target.
4 | serialBytes[3] = (target >> 7) & 0x7F; // Third data byte holds the bits 7-13 of target.
```

Many servo control applications do not need quarter-microsecond target resolution. If you want a shorter and lower-resolution set of commands for setting the target you can use the Mini-SSC command below.

Set Target (Mini SSC protocol)

Mini-SSC protocol: **0xFF**, channel address, 8-bit target

This command sets the target of a channel to a value specified by an 8-bit target value from 0 to 254. The 8-bit target value is converted to a full-resolution *target* value according to the *range* and *neutral* settings stored on the Maestro for that channel. Specifically, an 8-bit target of 127 corresponds to the neutral setting for that channel, while 0 or 254 correspond to the neutral setting minus or plus the range setting. These settings can be useful for calibrating motion without changing the program sending serial commands.

The channel address is a value in the range 0–254. By default, the channel address is equal to the channel number, so it should be from 0 to 23. To allow multiple Maestros to be controlled on the same serial line, set the Mini SSC Offset parameter to different values for each Maestro. The Mini SSC Offset is added to the channel number to compute the correct channel address to use with this command. For example, a Micro Maestro 6-channel servo controller with a Mini SSC Offset of 12 will obey Mini-SSC commands whose address is within 12–17.

Set Multiple Targets (Mini Maestro 12, 18, and 24 only)

Compact protocol: **0x9F**, number of targets, first channel number, first target low bits, first target high bits, second target low bits, second target high bits, ...

Pololu protocol: **0xAA**, device number, **0x1F**, number of targets, first channel number, first target low bits, first target high bits, second target low bits, second target high bits, ...

This command simultaneously sets the targets for a contiguous block of channels. The first byte specifies how many channels are in the contiguous block; this is the number of target values you will

need to send. The second byte specifies the lowest channel number in the block. The subsequent bytes contain the target values for each of the channels, in order by channel number, in the same format as the Set Target command above. For example, to set channel 3 to 0 (off) and channel 4 to 6000 (neutral), you would send the following bytes:

0x9F, 0x02, 0x03, 0x00, 0x00, 0x70, 0x2E

The Set Multiple Targets command allows high-speed updates to your Maestro, which is especially useful when controlling a large number of servos in a chained configuration. For example, using the Pololu protocol at 115.2 kbps, sending the Set Multiple Targets command lets you set the targets of 24 servos in 4.6 ms, while sending 24 individual Set Target commands would take 12.5 ms.

Set Speed

Compact protocol: **0x87, channel number, speed low bits, speed high bits**

Pololu protocol: **0xAA, device number, 0x07, channel number, speed low bits, speed high bits**

This command limits the *speed* at which a servo channel's output value changes. The speed limit is given in units of $(0.25 \mu\text{s})/(10 \text{ ms})$, except in special cases (see **Section 4.b**). For example, the command 0x87, 0x05, 0x0C, 0x01 sets the speed of servo channel 5 to a value of 140, which corresponds to a speed of $3.5 \mu\text{s}/\text{ms}$. What this means is that if you send a **Set Target** command to adjust the target from, say, $1000 \mu\text{s}$ to $1350 \mu\text{s}$, it will take 100 ms to make that adjustment. A speed of 0 makes the speed unlimited. Setting the *target* of a channel that has a speed of 0 and an acceleration 0 will immediately affect the channel's *position*. Note that the actual speed at which your servo moves is also limited by the design of the servo itself, the supply voltage, and mechanical loads; this parameter will not help your servo go faster than what it is physically capable of.

At the minimum speed setting of 1, the servo output takes 40 seconds to move from 1 to 2 ms.

The speed setting has no effect on channels configured as inputs or digital outputs.

Set Acceleration

Compact protocol: **0x89, channel number, acceleration low bits, acceleration high bits**

Pololu protocol: **0xAA, device number, 0x09, channel number, acceleration low bits, acceleration high bits**

This command limits the *acceleration* of a servo channel's output. The acceleration limit is a value from 0 to 255 in units of $(0.25 \mu\text{s})/(80 \text{ ms})$, except in special cases (see **Section 4.b**). A value of 0 corresponds to no acceleration limit. An acceleration limit causes the speed of a servo to slowly ramp up until it reaches the maximum speed, then to ramp down again as *position* approaches *target*, resulting in a relatively smooth motion from one point to another. With acceleration and speed limits, only a few target settings are required to make natural-looking motions that would otherwise be quite

complicated to produce.

At the minimum acceleration setting of 1, the servo output takes about 3 seconds to move smoothly from a target of 1 ms to a target of 2 ms.

The acceleration setting has no effect on channels configured as inputs or digital outputs.

Set PWM (Mini Maestro 12, 18, and 24 only)

Compact protocol: **0x8A, on time low bits, on time high bits, period low bits, period high bits**

Pololu protocol: **0xAA, device number, 0x0A, on time low bits, on time high bits, period low bits, period high bits**

This command sets the PWM output to the specified on time and period, in units of 1/48 µs. The on time and period are both encoded with 7 bits per byte in the same way as the target in command 0x84, above. For more information on PWM, see [Section 4.a](#). The PWM output is not available on the Micro Maestro.

Get Position

Compact protocol: **0x90, channel number**

Pololu protocol: **0xAA, device number, 0x10, channel number**

Response: position low 8 bits, position high 8 bits

This command allows the device communicating with the Maestro to get the *position* value of a channel. The position is sent as a two-byte response immediately after the command is received.

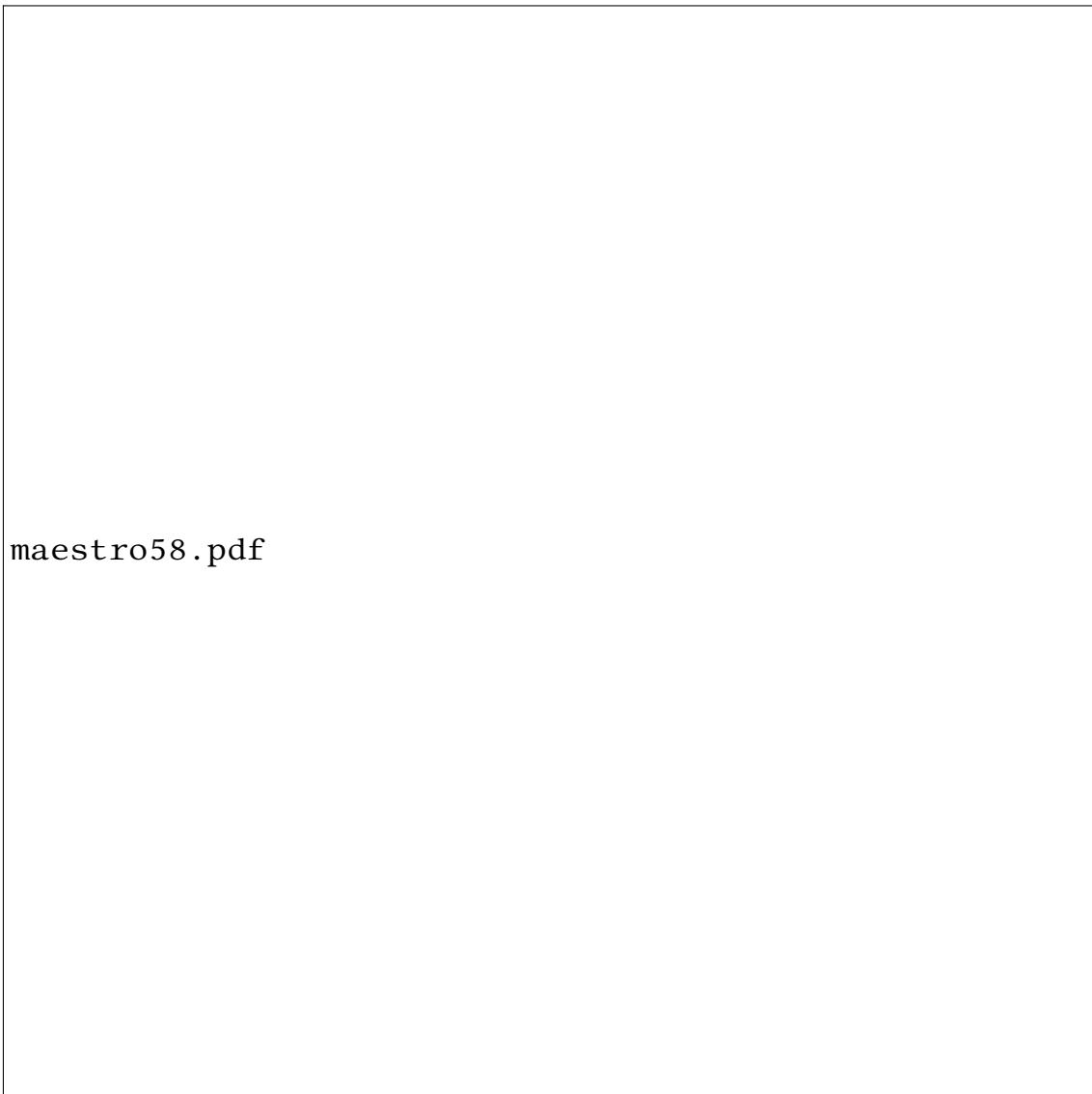
If the specified channel is configured as a servo, this position value represents the current pulse width that the Maestro is transmitting on the channel, reflecting the effects of any previous commands, speed and acceleration limits, or scripts running on the Maestro.

If the channel is configured as a digital output, a position value less than 6000 means the Maestro is driving the line low, while a position value of 6000 or greater means the Maestro is driving the line high.

If the channel is configured as an input, the position represents the voltage measured on the channel. The inputs on channels 0–11 are analog: their values range from 0 to 1023, representing voltages from 0 to 5 V. The inputs on channels 12–23 are digital: their values are either exactly 0 or exactly 1023.

Note that the formatting of the position in this command differs from the target/speed/acceleration formatting in the other commands. Since there is no restriction on the high bit, the position is formatted as a standard little-endian two-byte unsigned integer. For example, a position of 2567 corresponds to a response 0x07, 0x0A.

A ANHANG



maestro58.pdf

A.4 Quellcode: Headerfile von ISerialCom und SerialCom

Listing 9: Interface: ISerialCom - Klasse: SerialCom

```

//=====
// Name      : SerialCom.hpp
// Author    : Willi Penner
//
// Description : SerialCom header file. It contains the
//                declaration of the ISerialCom interface
//                and the SerialCom class
//=====

#ifndef SERIALCOM_HPP_INCLUDED
#define SERIALCOM_HPP_INCLUDED

#ifndef _WIN32
#include <windows.h>
#else
#include <iostream>
#endif

/** \brief Interface for a serial connection via a COM port
 * Prescribes at least the included pure virtual functions
 * for initiating, opening, closing and writing a serial
 * connection.
 */
class ISerialCom {
public:
    virtual ~ISerialCom() {};
    virtual void initSerialCom(const char* portName, unsigned short baudRate) = 0;
    virtual bool openSerialCom() = 0;
    virtual bool closeSerialCom() = 0;
    virtual bool writeSerialCom(unsigned char command[], unsigned short sizeCommand, unsigned char *response,
                               unsigned short sizeResponse) = 0;
};

/** \brief SerialCom is a class that inherits the
 * functions from the ISerialCom interface. The
 * functionality of the interface is expanded by
 * the function getPort().
 *
 * \param portName_ : The port name is used to open
 * a serial connection via the port name for the
 * controller specified by the operating system.
 * \param baudRate_ : The baud rate determines the
 * transmission speed at which communication between
 * the PC and controller takes place.
 * \param port_ : If a serial connection is opened
 * successfully, depending on the operating system,
 * a HANDLE (under Windows) or an integer value
 * (under Linux) is defined, via which the communication
 * takes place.
 */
class SerialCom : public ISerialCom {
private:
    const char* portName_ = NULL;
    unsigned short baudRate_ = 0;
    /**< depending on the operating system, a HANDLE
     * (under Windows) or an integer value (under Linux)
     * for an open connection */
#ifndef _WIN32
    HANDLE port_;
#else
    int port_;
#endif
public:
    /* Constructors: A constructor without
     * transfer parameters, so that when used in the
     * Pololu class or another class, an object of
     * type SerialCom can be created without having
     * to pass values to it. And a constructor for
     * creating an object with start parameters for
     * the port name and the baud rate.
     * portName : The port name is used to
     * open a serial connection via the port name
     * for the controller specified by the operating
     * system.

```

```
baudRate : The baud rate determines
           the transmission speed at which communication
           between the PC and controller takes place.
*/
SerialCom();
SerialCom(const char* portName, unsigned short baudRate);

/** \brief "initSerialCom" is used to initiate
 *  the SerialCom object with port name and baud rate.
 *  The function puts the object in the same state
 *  as the constructor with transfer parameters.
 *  It can be used to change the port name and the
 *  baud rate of a SerialCom object.
 *
 *  \param portName : The port name is used to
 *  open a serial connection via the port name
 *  for the controller specified by the operating
 *  system.
 *  \param baudRate : The baud rate determines
 *  the transmission speed at which communication
 *  between the PC and controller takes place.
 */
void initSerialCom(const char* portName, unsigned short baudRate);

/** \brief Uses the set port name and the baud
 *  rate of the class to open a serial connection.
 *  If the opening is successful, the value in
 *  "port_" stands for the open connection and
 *  can be used for communication.
 *
 *  \return Returns TRUE on successful opening
 *  of a serial connection, otherwise it returns
 *  FALSE.
 */
bool openSerialCom();

/** \brief Closes a serial connection.
 *
 *  \return Returns TRUE on successful closing
 *  of a serial connection, otherwise it returns
 *  FALSE.
 */
bool closeSerialCom();

/** \brief "writeSerialCom" is used to write
 *  commands to the controller via the open serial
 *  connection.
 *
 *  \param command[] : Contains the command to be
 *  sent (size of 1, 2 or 4 bytes, depending on
 *  the command).
 *  \param sizeCommand : Contains the size of the
 *  command (1, 2 or 4).
 *  \param response : If the command to be sent
 *  expects a return value from the controller, the
 *  writeSerialCom is given a pointer to a response
 *  array. This can be 1 or 2 bytes in size. If no
 *  return value is expected, the pointer is NULL.
 *  \param sizeResponse : Contains the size of the
 *  command (1 or 2, in case of no expected return
 *  value it is 0).
 *
 *  \return Returns TRUE on successful writing to
 *  a serial connection, otherwise it returns FALSE.
 */
bool writeSerialCom(unsigned char command[], unsigned short sizeCommand, unsigned char *response, unsigned short sizeResponse);

/** \brief Returns the port HANDLE under Windows
 *  or the integer value for the connection under Linux.
 *
 *  \return port_
 */
#ifdef _WIN32
    HANDLE getPort();
#else
    int getPort();
#endif
};



---


```

A.5 Quellcode: Sourcefile von SerialCom

Listing 10: Klasse: SerialCom

```

//=====
// Name      : SerialCom.cpp
// Author    : Willi Penner
//
// Description : SerialCom source file. It contains the definition of the
//                functions of the SerialCom class.
//=====

#include "SerialCom.hpp"
#include <stdio.h>
#include <string>
#include <iostream>

#ifndef _WIN32
    #include <windows.h>
#else
    #include <fcntl.h>
    #include <unistd.h>
    #include <stdint.h>
    #include <termios.h>
    #include <stdbool.h>
#endif

/** \brief Constructors: A constructor without transfer parameters, so that when used in the Pololu class
 * or another class, an object of type SerialCom can be created without having to pass values to it.
 * And a constructor for creating an object with start parameters for the port name and the baud rate.
 *
 * \param portName : The port name is used to open a serial connection via the port name for the controller
 *                  specified by the operating system.
 * \param baudRate : The baud rate determines the transmission speed at which communication between the PC and
 *                  controller takes place.
 */
SerialCom::SerialCom(){
    portName_ = "";
    baudRate_ = 0;
#ifdef _WIN32
    port_ = NULL;
#else
    port_ = 0;
#endif
}

SerialCom::SerialCom(const char* portName, unsigned short baudRate){
    portName_ = portName;
    baudRate_ = baudRate;
#ifdef _WIN32
    port_ = NULL;
#else
    port_ = 0;
#endif
}

/** \brief "initSerialCom" is used to initiate the SerialCom object with port name and baud rate.
 * The function puts the object in the same state as the constructor with transfer parameters.
 * It can be used to change the port name and the baud rate of a SerialCom object.
 *
 * \param portName : The port name is used to open a serial connection via the port name for the controller
 *                  specified by the operating system.
 * \param baudRate : The baud rate determines the transmission speed at which communication between the PC and
 *                  controller takes place.
 */
void SerialCom::initSerialCom(const char* portName, unsigned short baudRate){
    /**< Before a serial connection is reinitialized, a possible open connection is closed. */
    #ifdef _WIN32
        CloseHandle(port_);
    #else
        close(port_);
    #endif
    portName_ = portName;
    baudRate_ = baudRate;
}

/** \brief Uses the set port name and the baud rate of the class to open a serial connection.
 * If the opening is successful, the value in "port_" stands for the open connection and
 * can be used for communication.

```

```

/*
 * \return Returns TRUE on successful opening of a serial connection, otherwise it returns FALSE.
 */
bool SerialCom::openSerialCom(){
    #ifdef _WIN32
        /*< Opening a serial connection in windows. */
        bool success = FALSE;
        DCB state;

        /*< If there is still an open connection, it will be closed before opening it again. */
        CloseHandle(port_);
        /*< Opens a serial connection using the CreateFileA function from <windows.h>. Port_ is
        opened with read and write access. */
        port_ = CreateFileA(portName_, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
                            FILE_ATTRIBUTE_NORMAL, NULL);
        if (port_ == INVALID_HANDLE_VALUE){
            throw std::string("SerialCom::openSerialCom: Failed to open port.\n");
            return 0;
        }
        /*< Flushes the file buffer of the opened connection. */
        success = FlushFileBuffers(port_);
        if (!success)
        {
            throw std::string("SerialCom::openSerialCom: Failed to flush file buffer.\n");
            CloseHandle(port_);
            return 0;
        }
        /*< Configure read and write operations to time out after 100 ms. */
        COMTIMEOUTS timeouts = { 0 };
        timeouts.ReadIntervalTimeout = 0;
        timeouts.ReadTotalTimeoutConstant = 100;
        timeouts.ReadTotalTimeoutMultiplier = 0;
        timeouts.WriteTotalTimeoutConstant = 100;
        timeouts.WriteTotalTimeoutMultiplier = 0;
        success = SetCommTimeouts(port_, &timeouts);
        if (!success)
        {
            throw std::string("SerialCom::openSerialCom: Failed to set serial timeouts.\n");
            CloseHandle(port_);
            return 0;
        }
        /*< Reads the connection status of the serial connection. If the reading of
        the connection status was successful, the baud rate for the communication is set. */
        state.DCBlength = sizeof(DCB);
        success = GetCommState(port_, &state);
        if (!success)
        {
            throw std::string("SerialCom::openSerialCom: Failed to get serial settings.\n");
            CloseHandle(port_);
            return 0;
        }
        state.BaudRate = baudRate_;
        success = SetCommState(port_, &state);
        if (!success)
        {
            throw std::string("SerialCom::openSerialCom: Failed to set serial settings.\n");
            CloseHandle(port_);
            return 0;
        }
        return 1;
    #else
        bool success = false;

        /*< If there is still an open connection, it will be closed before opening it again. */
        close(port_);
        /*< Opens a serial connection using the CreateFileA function from <windows.h>. Port_ is
        opened with read and write access. */
        port_ = open(portName_, O_RDWR | O_NOCTTY); //you have to set the permission for the /dev/ttyACM0
        if (port_ == -1){
            throw std::string("SerialCom::openSerialCom: Failed to open port.\n");
            return 0;
        }
        /*< Flushes the file buffer of the opened connection. */
        success = tcflush(port_, TCIOFLUSH);
        if (success){
            throw std::string("SerialCom::openSerialCom: Failed to flush file buffer.\n");
            close(port_);
            return 0;
        }
    #endif
}
```

```

// Get the current configuration of the serial port.
struct termios options;
success = tcgetattr(port_, &options);
if (success){
    throw std::string("SerialCom::openSerialCom: Failed to get serial settings.\n");
    close(port_);
    return 0;
}
// Turn off any options that might interfere with our ability to send and
// receive raw binary bytes.
options.c_iflag &= ~(INLCR | IGNCR | ICRNL | IXON | IXOFF);
options.c_oflag &= ~(ONLCR | OCRNL);
options.c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);

// Set up timeouts: Calls to read() will return as soon as there is
// at least one byte available or when 100 ms has passed.
options.c_cc[VTIME] = 1;
options.c_cc[VMIN] = 0;

// This code only supports certain standard baud rates. Supporting
// non-standard baud rates should be possible but takes more work.
cfsetospeed(&options, B9600);
cfsetispeed(&options, cfgetospeed(&options));
success = tcsetattr(port_, TCSANOW, &options);
if (success){
    throw std::string("SerialCom::openSerialCom: Failed to set serial settings.\n");
    close(port_);
    return 0;
}
return 1;
#endif
}

/** \brief Closes a serial connection.
 *
 * \return Returns TRUE on successful closing of a serial connection, otherwise it returns FALSE.
 */
bool SerialCom::closeSerialCom(){
#ifdef _WIN32
if (CloseHandle(port_) == 0){
    throw std::string("SerialCom::closeSerialCom: Failed to close port.\n");
    return 0;
}
return 1;
#else
return close(port_);
#endif
}

/** \brief "writeSerialCom" is used to write commands to the controller via the open serial connection.
 *
 * \param command[] : Contains the command to be sent (size of 1, 2 or 4 bytes, depending on the command).
 * \param sizeCommand : Contains the size of the command (1, 2 or 4).
 * \param response : If the command to be sent expects a return value from the controller, the writeSerialCom is
     given a pointer to a response array. This can be 1 or 2 bytes in size. If no return value is expected,
     the pointer is NULL.
 * \param sizeResponse : Contains the size of the command (1 or 2, in case of no expected return value it is 0).
 *
 * \return Returns TRUE on successful writing to a serial connection, otherwise it returns FALSE.
 */
bool SerialCom::writeSerialCom(unsigned char command[], unsigned short sizeCommand, unsigned char *response,
    unsigned short sizeResponse){
/** Check the length of the command */
if ((sizeCommand != 1) && (sizeCommand != 2) && (sizeCommand != 4)){
    throw std::string("SerialCom::writeSerialCom: wrong parameter sizeCommand, allowed parameter 1,2 or 4.");
}

#ifndef _WIN32
DWORD bytesTrasfered; //Is given to the write or read command as a pointer. After executing the
//WriteFile or ReadFile, bytesTranferred contains the number of bytes transmitted or received.
bool success = 0;

/** Sending the command to the controller via port_. */
success = WriteFile(port_, command, sizeCommand, &bytesTrasfered, NULL);
if (!success){
    throw std::string("SerialCom::writeSerialCom: Failed to write to port.");
    return 0;
}

```

```
/** Check whether data needs to be read. */
if (sizeResponse > 0){
    success = ReadFile(port_, (void *)response, sizeResponse, &bytesTransferred, NULL);
} else{
    return 1; //No need to read data. Confirmation of the successful writing of the data.
}
if (!success){
    throw std::string("SerialCom::writeSerialCom: Failed to read from port.");
    return 0;
}
return 1; //Confirmation that data was written and read data were saved in the response array.
#else
/** Sending the command to the controller via port_. */
if(write(port_, command, sizeCommand) == -1){
    throw std::string("SerialCom::writeSerialCom: Failed to write to port.");
    return 0;
}
/** Check whether data needs to be read. */
if (sizeResponse > 0){
    if(read(port_, (void *)response, sizeResponse) != sizeResponse)
    {
        throw std::string("SerialCom::writeSerialCom: Failed to read from port.");
        return 0;
    }
    else{
        return 1;
    }
    return 1;
#endif
}

/** \brief Returns the port HANDLE under Windows or the integer value for the connection under Linux.
 */
* \return port_
*/
#ifndef _WIN32
    HANDLE SerialCom::getPort(){
#else
    int SerialCom::getPort(){
#endif
    return port_;
}
```

A.6 Quellcode: Headerfile von IPololu und Pololu

Listing 11: Interface: IPololu - Klasse: Pololu

```

//=====
// Name      : Pololu.hpp
// Author    : Willi Penner
// 
// Description : Pololu header file. It contains the declaration of the
//                IPololu interface and the Pololu class
//=====

#ifndef POLOLU_HPP_INCLUDED
#define POLOLU_HPP_INCLUDED

#include "SerialCom.hpp"

/** \brief Interface to control a Pololu controller. The interface
 * provides the basic functions for the control of servo motors.
 */
class IPololu {
protected:
    virtual bool setPosition(unsigned short servo, unsigned short goToPosition) = 0;
    virtual bool setSpeed(unsigned short servo, unsigned short goToSpeed) = 0;
    virtual bool setAcceleration(unsigned short servo, unsigned short goToAcceleration) = 0;
    virtual unsigned short getPosition(unsigned short servo) = 0;
public:
    virtual ~IPololu(){};
    virtual bool getMovingState() = 0;
};

/** \brief Class for a Pololu object that contains a serial connection and provides basic functions
 * for programming the controller.
 *
 * \param serialCom = Is an object of the SerialCom class without initialization of port name and baud rate,
 *                   the "port_" is NULL.
 */
class Pololu : public IPololu {
friend class ServoMotor;
protected:
    SerialCom serialCom;

    /** \brief Funktion is used to move a specific servo to a new position.
     *
     * \param servo = Servo to move
     * \param goToPosition = New position to be approached. The value for the new position is calculated from
     *                      the position in microseconds times 4 (e.g. new position should be 1500 microseconds, then 6000 must be
     *                      set in the function as new position)
     *
     * \return The return value of the function is 1 if the new position was successfully set and 0 if an error
     *        occurred.
     */
    bool setPosition(unsigned short servo, unsigned short goToPosition);

    /** \brief Function is used to set the speed for a servo with which it should move.
     *
     * \param servo = Servo to set speed
     * \param goToSpeed = Speed of the servo (speed value 1 = 0.25us / 10ms or speed value 100 = 25us / 10ms). A
     *                   speed value of 0 means infinite speed, i.e. the maximum speed of the servo.
     *
     * \return The return value of the function is 1 if the new speed was successfully set and 0 if an error
     *        occurred.
     */
    bool setSpeed(unsigned short servo, unsigned short goToSpeed);

    /** \brief Function is used to set the acceleration for a servo with which it should reach the set speed.
     *
     * \param servo = Servo to set acceleration
     * \param goToAcceleration = Acceleration of the servo (acceleration value 1 = 0.25us / 10ms / 80ms or speed
     *                           value 100 = 25us / 10ms / 80ms). A speed value of 0 means infinite acceleration, i.e. the maximum
     *                           acceleration of the servo.
     *
     * \return The return value of the function is 1 if the new acceleration was successfully set and 0 if an
     *        error occurred.
     */
}

```

```
/*
bool setAcceleration(unsigned short servo, unsigned short goToAcceleration);

/** \brief Function is used to read out the current position of a particular servo.
 *
 * \param servo = Servo whose current position is to be read out.
 *
 * \return The return value is the current position of the selected servo. The position value supplied by the
 * controller must still be multiplied by 4.
 *
 */
unsigned short getPosition(unsigned short servo);

public:
    /** \brief Constructor executes serialCom.initSerialCom to initialize the serial connection. An object of
     * the Pololu class must be
     * initialized with the port name and baud rate using the constructor.
     *
     * \param portName : The port name is used to open a serial connection via the port name for the controller
     * specified by the operating system.
     * \param baudRate : The baud rate determines the transmission speed at which communication between the PC
     * and controller takes place.
     *
     */
    Pololu(const char* portName, unsigned short baudRate);

    /** \brief Used to change the connection data. Sets the serial connection in the same state as the
     * constructor, but with a new port name and baud rate
     *
     * \param portName : The port name is used to open a serial connection via the port name for the controller
     * specified by the operating system.
     * \param baudRate : The baud rate determines the transmission speed at which communication between the PC
     * and controller takes place.
     *
     */
    void initConnection(const char* portName, unsigned short baudRate);

    /** \brief Functions are used to open and close the serial connection. Functions only call
     * the openSerialCom and closeSerialCom functions of the serialCom object.
     *
     * \return Value is 1 when opening or closing was successful and 0 when an error occurred.
     *
     */
    bool openConnection();
    bool closeConnection();

    /** \brief Function provides the movement status of all connected servos.
     *
     * \return The return value is 1 while a servo is still in motion and 0 when all servos are at a standstill.
     *
     */
    bool getMovingState();
};

#endif // POLOLU_HPP_INCLUDED
```

A.7 Quellcode: Sourcefile von Pololu

Listing 12: Klasse: Pololu

```

//=====
// Name      : Pololu.hpp
// Author    : Willi Penner
// 
// Description : Pololu source file. It contains the definition of the
//                functions of the Pololu class.
//=====

#include "Pololu.hpp"
#include "SerialCom.hpp"
#include <string>
#include <iostream>

/** \brief Constructor executes serialCom.initSerialCom to initialize the serial connection. An object of the
 *        Pololu class must be
 *        initialized with the port name and baud rate using the constructor.
 *
 *        \param portName : The port name is used to open a serial connection via the port name for the controller
 *                         specified by the operating system.
 *        \param baudRate : The baud rate determines the transmission speed at which communication between the PC and
 *                         controller takes place.
 *
 */
Pololu::Pololu(const char* portName, unsigned short baudRate){serialCom.initSerialCom(portName, baudRate);}

/** \brief Function is used to open the serial connection. Function only calls
 *        the openSerialCom function of the serialCom object.
 *
 *        \return The return value is 1 when opening the port was successful, 0 when an error occurred.
 *
 */
bool Pololu::openConnection(){
    try
    {
        serialCom.openSerialCom();
    }
    catch (std::string &errorMessage)
    {
        std::cout << errorMessage;
        return 0;
    }
    return 1;
}

/** \brief Function is used to close the serial connection. Function only calls
 *        the closeSerialCom function of the serialCom object.
 *
 *        \return The return value is 1 when closing the port was successful, 0 when an error occurred.
 *
 */
bool Pololu::closeConnection(){
    try
    {
        serialCom.closeSerialCom();
    }
    catch (std::string &errorMessage)
    {
        std::cout << errorMessage;
        return 0;
    }
    return 1;
}

/** \brief Used to change the connection data. Sets the serial connection in the same state as the constructor,
 *        but with a new port name and baud rate
 *
 *        \param portName : The port name is used to open a serial connection via the port name for the controller
 *                         specified by the operating system.
 *        \param baudRate : The baud rate determines the transmission speed at which communication between the PC and
 *                         controller takes place.
 *
 */
void Pololu::initConnection(const char* portName, unsigned short baudRate){serialCom.initSerialCom(portName,
    baudRate);}

```

```

/** \brief Funktion is used to move a specific servo to a new position.
 *
 * \param servo = Servo to move
 * \param goToPosition = New position to be approached. The value for the new position is calculated from the
 * position in microseconds times 4 (e.g. new position should be 1500 microseconds, then 6000 must be set in
 * the function as new position)
 *
 * \return The return value of the function is 1 if the new position was successfully set and 0 if an error
 * occurred.
 *
 */
bool Pololu::setPosition(unsigned short servo, unsigned short goToPosition){
    /* Generates the command for the controller.
     * 0x84 = Pololu command for setting the position
     * servo = servo to address as a transfer parameter
     * goToPosition = divided into 2 bytes, first the low bits, then the high bits
     */
    unsigned short sizeCommand = 4;
    unsigned char command[] = {0x84, (unsigned char)servo, (unsigned char)(goToPosition & 0x7F), (unsigned
        char)((goToPosition >> 7) & 0x7F)};
    try
    {
        serialCom.writeSerialCom(command, sizeCommand, NULL, 0);
    }
    catch (std::string &errorMessage)
    {
        throw std::string("Pololu::setPosition: " + errorMessage);
        return 0;
    }
    catch(...)
    {
        throw std::string("Pololu::setPosition: Unknown error, while writing to port.");
        return 0;
    }
    return 1;
}

/** \brief Function is used to set the speed for a servo with which it should move.
 *
 * \param servo = Servo to set speed
 * \param goToSpeed = Speed of the servo (speed value 1 = 0.25us / 10ms or speed value 100 = 25us / 10ms). A
 * speed value of 0 means infinite speed, i.e. the maximum speed of the servo.
 *
 * \return The return value of the function is 1 if the new speed was successfully set and 0 if an error
 * occurred.
 *
 */
bool Pololu::setSpeed(unsigned short servo, unsigned short goToSpeed){
    /* Generates the command for the controller.
     * 0x87 = Pololu command for setting the speed
     * servo = servo to address as a transfer parameter
     * goToSpeed = divided into 2 bytes, first the low bits, then the high bits
     */
    unsigned short sizeCommand = 4;
    unsigned char command[] = {0x87, (unsigned char)servo, (unsigned char)(goToSpeed & 0x7F), (unsigned
        char)((goToSpeed >> 7) & 0x7F)};
    try
    {
        serialCom.writeSerialCom(command, sizeCommand, NULL, 0);
    }
    catch (std::string &errorMessage)
    {
        throw std::string("Pololu::setSpeed: " + errorMessage);
        return 0;
    }
    catch(...)
    {
        throw std::string("Pololu::setSpeed: Unknown error, while writing to port.");
        return 0;
    }
    return 1;
}

/** \brief Function is used to set the acceleration for a servo with which it should reach the set speed.
 *
 * \param servo = Servo to set acceleration
 * \param goToAcceleration = Acceleration of the servo (acceleration value 1 = 0.25us / 10ms / 80ms or speed
 * value 100 = 25us / 10ms / 80ms). A speed value of 0 means infinite acceleration, i.e. the maximum

```

```

        acceleration of the servo.
*
* \return The return value of the function is 1 if the new acceleration was successfully set and 0 if an error
    occurred.
*
*/
bool Pololu::setAcceleration(unsigned short servo, unsigned short goToAcceleration){
    /* Generates the command for the controller.
     * 0x89 = Pololu command for setting the acceleration
     * servo = servo to address as a transfer parameter
     * goToAcceleration = divided into 2 bytes, first the low bits, then the high bits
     */
    unsigned short sizeCommand = 4;

    unsigned char command[] = {0x89, (unsigned char)servo, (unsigned char)(goToAcceleration & 0x7F), (unsigned
        char)((goToAcceleration >> 7) & 0x7F)};
    try
    {
        serialCom.writeSerialCom(command, sizeCommand, NULL, 0);
    }
    catch (std::string &errorMessage)
    {
        throw std::string("Pololu::setAcceleration: " + errorMessage);
        return 0;
    }
    catch(...)
    {
        throw std::string("Pololu::setAcceleration: Unknown error, while writing to port.");
        return 0;
    }
    return 1;
}

/** \brief Function is used to read out the current position of a particular servo.
*
* \param servo = Servo whose current position is to be read out.
*
* \return The return value is the current position of the selected servo. The position value supplied by the
    controller must still be multiplied by 4.
*
*/
unsigned short Pololu::getPosition(unsigned short servo){
    /* Generates the command for the controller.
     * 0x90 = Pololu command for reading out the position
     * servo = servo to address as a transfer parameter
     */
    unsigned short sizeCommand = 2;
    unsigned short sizeResponse = 2;
    unsigned char response[sizeResponse];

    unsigned char command[] = {0x90, (unsigned char)servo};
    try
    {
        serialCom.writeSerialCom(command, sizeCommand, response, sizeResponse);
    }
    catch (std::string &errorMessage)
    {
        throw std::string("Pololu::getPosition: " + errorMessage);
    }
    catch(...)
    {
        throw std::string("Pololu::getPosition: Unknown error, while writing/reading to port.");
    }

    return response[0] + 256 * response[1];
}

/** \brief Function provides the movement status of all connected servos.
*
* \return The return value is 1 while a servo is still in motion and 0 when all servos are at a standstill.
*
*/
bool Pololu::getMovingState(){
    /* Generates the command for the controller.
     * 0x93 = Pololu command for reading out the movement of all servos
     */
    unsigned short sizeCommand = 1;
    unsigned short sizeResponse = 1;
    unsigned char response[sizeResponse];

```

```
unsigned char command[] = {0x93};
try
{
    serialCom.writeSerialCom(command, sizeCommand, response, sizeResponse);
}
catch (std::string &errorMessage)
{
    throw std::string("Pololu::getMovingState: " + errorMessage);
}
catch(...)
{
    throw std::string("Pololu::getMovingState: Unknown error, while writing/reading to port.");
}
return response[0]; // The return value is 1 if a servo is still moving, 0 if there is no moving.
```

A.8 Quellcode: Headerfile von IServoMotor und ServoMotor

Listing 13: Interface: IServoMotor - Klasse: ServoMotor

```

//=====
// Name      : SerialCom.hpp
// Author    : Willi Penner
//
// Description : SerialCom header file. It contains the declaration of the
//                ISerialCom interface and the SerialCom class
//=====

#ifndef SERIALCOM_HPP_INCLUDED
#define SERIALCOM_HPP_INCLUDED

#ifdef _WIN32
    #include <windows.h>
#else
    #include <iostream>
#endif

/** \brief Interface for a serial connection via a COM port
 *  Prescribes at least the included pure virtual functions for initiating,
 *  opening, closing and writing a serial connection.
 */
class ISerialCom {
public:
    virtual ~ISerialCom(){};
    virtual void initSerialCom(const char* portName, unsigned short baudRate) = 0;
    virtual bool openSerialCom() = 0;
    virtual bool closeSerialCom() = 0;
    virtual bool writeSerialCom(unsigned char command[], unsigned short sizeCommand, unsigned char *response,
                               unsigned short sizeResponse) = 0;
};

/** \brief SerialCom is a class that inherits the functions from the ISerialCom interface.
 *  The functionality of the interface is expanded by the function getPort ().

 * \param portName_ : The port name is used to open a serial connection via the port name for the controller
 *                   specified by the operating system.
 * \param baudRate_ : The baud rate determines the transmission speed at which communication between the PC and
 *                   controller takes place.
 * \param port_ : If a serial connection is opened successfully, depending on the operating system, a HANDLE
 *               (under Windows) or an integer value (under Linux) is defined, via which the communication takes place.
 */
class SerialCom : public ISerialCom {
private:
    const char* portName_ = NULL;
    unsigned short baudRate_ = 0;
    /**< depending on the operating system, a HANDLE (under Windows) or an integer value (under Linux) for an
     * open connection */
    #ifdef _WIN32
        HANDLE port_;
    #else
        int port_;
    #endif
public:
    /** \brief Constructors: A constructor without transfer parameters, so that when used in the Pololu class
     *  or another class, an object of type SerialCom can be created without having to pass values to it.
     *  And a constructor for creating an object with start parameters for the port name and the baud rate.
     *
     * \param portName : The port name is used to open a serial connection via the port name for the controller
     *                  specified by the operating system.
     * \param baudRate : The baud rate determines the transmission speed at which communication between the PC
     *                  and controller takes place.
     */
    SerialCom();
    SerialCom(const char* portName, unsigned short baudRate);

    /** \brief "initSerialCom" is used to initiate the SerialCom object with port name and baud rate.
     *  The function puts the object in the same state as the constructor with transfer parameters.
     *  It can be used to change the port name and the baud rate of a SerialCom object.
     *
     * \param portName : The port name is used to open a serial connection via the port name for the controller
     *                  specified by the operating system.
     * \param baudRate : The baud rate determines the transmission speed at which communication between the PC
     *                  and controller takes place.
     */
}

```

```
void initSerialCom(const char* portName, unsigned short baudRate);

/** \brief Uses the set port name and the baud rate of the class to open a serial connection.
 * If the opening is successful, the value in "port_" stands for the open connection and
 * can be used for communication.
 *
 * \return Returns TRUE on successful opening of a serial connection, otherwise it returns FALSE.
 */
bool openSerialCom();

/** \brief Closes a serial connection.
 *
 * \return Returns TRUE on successful closing of a serial connection, otherwise it returns FALSE.
 */
bool closeSerialCom();

/** \brief "writeSerialCom" is used to write commands to the controller via the open serial connection.
 *
 * \param command[] : Contains the command to be sent (size of 1, 2 or 4 bytes, depending on the command).
 * \param sizeCommand : Contains the size of the command (1, 2 or 4).
 * \param response : If the command to be sent expects a return value from the controller, the
 * writeSerialCom is given a pointer to a response array. This can be 1 or 2 bytes in size. If no return
 * value is expected, the pointer is NULL.
 * \param sizeResponse : Contains the size of the command (1 or 2, in case of no expected return value it is
 * 0).
 *
 * \return Returns TRUE on successful writing to a serial connection, otherwise it returns FALSE.
 */
bool writeSerialCom(unsigned char command[], unsigned short sizeCommand, unsigned char *response, unsigned
short sizeResponse);

/** \brief Returns the port HANDLE under Windows or the integer value for the connection under Linux.
 *
 *
 * \return port_
 */
#ifdef _WIN32
    HANDLE getPort();
#else
    int getPort();
#endif
};

#endif // SERIALCOM_HPP_INCLUDED
```

A.9 Quellcode: Sourcefile ServoMotor

Listing 14: Auszug aus dem Sourcefile von ServoMotor

```

//=====
// Name      : ServoMotor.cpp
// Author    : Willi Penner
//
// Description : ServoMotor source file. It contains the definition of the
//                functions of the ServoMotor class.
//=====

#include "ServoMotor.hpp"
#include "Pololu.hpp"
#include <iostream>
#include <cmath>

/** \brief ServoMotor class constructor. An object of the ServoMotor type must be initiated via the constructor.
 *
 * \param servo = is the slot number on the controller board to which the servo is connected
 * \param startingPosition = is the center position of the servo motor (must be determined in the Pololu
 *                           Maestro Control Center)
 * \param delta = is the range of motion that the servo can reach from the center position
 * \param *connection = is a pointer to the Pololu object for the serial connection
 *
 */
ServoMotor::ServoMotor(unsigned short servo, unsigned short startingPosition, unsigned short delta, Pololu
                      *connection){
    servoNumber_ = servo-1;
    startingPosition_ = startingPosition;
    delta_ = delta;
    connection_ = connection;
}

/** \brief Returns the port to which the servo is connected to the controller board.
 *
 * \return The return value is the servoNumber+1 because the internal counting of the controller starts at 0.
 */
unsigned short ServoMotor::getServoNumber(){
    return servoNumber_+1;
}

/** \brief Returns the minimum position the servo is able to reach.
 *
 * \return The return value is the set startingPosition minus the set delta
 */
unsigned short ServoMotor::getMinPos (){
    return startingPosition_ - delta_ ;
}

/** \brief Returns the center position of the servo.
 *
 * \return The return value is the set startingPosition.
 */
unsigned short ServoMotor::getMidPos (){
    return startingPosition_ ;
}

/** \brief Returns the maximum position the servo is able to reach.
 *
 * \return The return value is the set startingPosition plus the set delta.
 */
unsigned short ServoMotor::getMaxPos (){
    return startingPosition_ + delta_ ;
}

/** \brief Function causes the servo to move to a specific new position (position
 *        is specified as an absolute value in a range from 5860 +- 3600).
 *
 * \param newPosition as an absolut value
 *
 * \return The return value is the return value of the setPosition funktion of the Pololu object.
 */
bool ServoMotor::setPositionInAbs(unsigned short newPosition){


```

```

if (newPosition > (startingPosition_ + delta_) || newPosition < (startingPosition_ - delta_)){
    throw std::string("ServoMotor::setPositionInAbs: Abs position is out of range (startingPosition +- delta_).");
} else{
    return connection_->setPosition(servoNumber_, newPosition);
}
}

/** \brief Function causes the servo to move to a specific new position (position is set with a value between
 * -90 and 90 degrees).
 *
 * \param newPosition in degrees
 *
 * \return The return value is the return value of the setPosition funktion of the Pololu object.
 */
bool ServoMotor::setPositionInDeg(short newPosition){
if (newPosition > maxDeg || newPosition < -maxDeg){
    throw std::string("ServoMotor::setPositionInDeg: Degree is out of range (-90 - 90).");
} else{
    return connection_->setPosition(servoNumber_, startingPosition_ + newPosition * conFactorDegToPos *
        conFactorMyToPos);
}
}

/** \brief Function causes the servo to move to a specific new position (position is set with a radiant between
 * -PI/2 and PI/2).
 *
 * \param newPosition as radiant
 *
 * \return The return value is the return value of the setPosition funktion of the Pololu object.
 */
bool ServoMotor::setPositionInRad(float newPosition){
// Since float numbers cannot be compared, the radians are converted to an int. The factor 100 sets the
// accuracy to 2 digits after the decimal point.
if ((int)(100 * newPosition) > (int)(100 * (maxRad)) || (int)(100 * newPosition) < (int)(100 * (-maxRad))){
    throw std::string("ServoMotor::setPositionInRad: Radian is out of range (-PI/2 - +PI/2).");
} else{
    // first the neu radiant position is converted to degree (newPosition * 180 / M_PI)
    return connection_->setPosition(servoNumber_, (startingPosition_ + (newPosition * 180 / M_PI) *
        conFactorDegToPos * conFactorMyToPos));
}
}

/** \brief Sets the speed at which the servo should move (speed range is between 1 and 255)
 *
 * \param newSpeed = Speed value (1 is (0.25 microseconds) / (10 milliseconds),
 * 255 is (63,75 microseconds) / (10 milliseconds))
 *
 * \return The return value is the return value of the setSpeed funktion of the Pololu object.
 */
bool ServoMotor::setSpeed(unsigned short newSpeed){
if (newSpeed > maxSpeed || newSpeed < minSpeed){
    throw std::string("ServoMotor::setSpeed: Speed is out of range (1 - 255).");
} else{
    return connection_->setSpeed(servoNumber_, newSpeed);
}
}

/** \brief Sets the acceleration with which the set speed should be reached (acceleration range is between 1 and
 * 255)
 *
 * \param newAcceleration = Acceleration value (1 is (0.25 microseconds) / (10 milliseconds) / (80
 * milliseconds),
 * 255 is (63,75 microseconds) / (10 milliseconds) / (80 milliseconds))
 *
 * \return The return value is the return value of the setAcceleration funktion of the Pololu object.
 */
bool ServoMotor::setAcceleration(unsigned short newAcceleration){
if (newAcceleration > maxAcceleration || newAcceleration < minAcceleration){
    throw std::string("ServoMotor::setAcceleration: Acceleration is out of range (1 - 255).");
} else{
    return connection_->setAcceleration(servoNumber_, newAcceleration);
}
}

/** \brief Returns the position of the servo as absolute value.
 *
 * \return The return value is the return value of the getPosition function of the Pololu object.
 */

```

```
unsigned short ServoMotor::getPositionInAbs(){
    return connection_->getPosition(servoNumber_);
}

/** \brief Returns the position of the servo in degrees.
 *
 * \return The return value is the return value of the getPosition function of the Pololu object converted into
 *         degrees.
 *
 */
short ServoMotor::getPositionInDeg(){
    return (connection_->getPosition(servoNumber_) / conFactorMyToPos - startingPosition_) / conFactorDegToPos;
}

/** \brief Returns the position of the servo as radiant.
 *
 * \return The return value is the return value of the getPosition function of the Pololu object converted into
 *         radiant.
 *
 */
float ServoMotor::getPositionInRad(){
    return (connection_->getPosition(servoNumber_) / conFactorMyToPos - startingPosition_) * M_PI /
        conFactorDegToPos / 180;
}

/** \brief Shows which settings have to be made in the Pololu Maestro Control Center for the servo,
 *        based on the starting position and the delta for the specific servo.
 */
void ServoMotor::showPololuValues (unsigned short& min, unsigned short& mid, unsigned short& max){
    min = (unsigned short)((startingPosition_ - delta_) / conFactorMyToPos);
    max = (startingPosition_ + delta_) / conFactorMyToPos;
    mid = (startingPosition_) / conFactorMyToPos;
}
```

A.10 Quellcode: Headerfile TestUnits.hpp

Listing 15: Auszug aus dem Headerfile von TestUnits

```
//=====
// Name      : TestUnits.hpp
// Author    : Willi Penner
//
// Description :
//=====
#ifndef TESTUNITS_HPP_
#define TESTUNITS_HPP_

void wait(unsigned long milliseconds);
void testOpenClose ();
void testSetGetMethods ();
void testMEXMovementSetting1 ();
void testMEXMovementSetting2 ();

#endif /* TESTUNITS_HPP_ */
```
