Projektarbeit zum Kurs Computer- und Mikrocontrollertechnik WS21/22

Fakultät für Biologie und vorklinische Medizin

5DOF-Roboterarm

Markus Reichold

26.01.2022

Inhaltsverzeichnis

1. Einleitung

1.1 Hintergrund

Immer mehr industrielle Abläufe werden automatisiert von Robotern ausgeführt. Die Geometrie dieser Roboter variiert stark mit deren Aufgaben, weshalb die Kinetik nur schwer vereinheitlicht werden kann. Eine Möglichkeit der Standardisierung wurde im Jahr 1955 von Jacques Denavit und Richard S. Hartenberg entwickelt und gilt mittlerweile als Standardverfahren für die Vorwärtskinematik. Mit nur vier Parametern pro Gelenk (sog. DH-Parameter) kann damit der ganze Roboter bezüglich der Vorwärtskinematik beschrieben werden. Vereinfacht bedeutet das, dass neben den Armlängen noch die Winkel der Rotationsgelenke bekannt sein müssen, um die Position des Effektors berechnen zu können. Als Effektor wird das letzte Glied in der kinematischen Kette bezeichnet. Oft handelt es sich dabei um ein Werkzeug, wie einen Greifarm, eine Lackierdüse oder ein Schweißgerät. Der Mittelpunkt des Werkzeugs bzw. ein fest definierter Punk daran wird hingegen „Tool Center Point“ (TCP) genannt. Auf diesen Punkt werden normalerweise alle Bewegungen des Roboters bezogen.
Während die Vorwärtskinematik mathematisch einfach zu handhaben ist und eindeutige Ergebnisse liefert, ist die inverse Kinematik deutlich komplizierter. Inverse Kinematik bedeutet im Umkehrschluss, dass die Effektorposition (und die Armlängen) gegeben sind und daraus die benötigten Gelenkwinkel der Motoren berechnen werden sollen. Diese Art der Kinematik ist die deutlich interessantere und wichtigere für die Robotik, leider gib es dafür aber keine vereinheitlichten mathematischen Verfahren. Problematisch ist unter anderem, dass oft mehrere Gelenkwinkelkonfigurationen möglich sind, um die gewünschte Effektorposition zu erreichen. Bei einfachen Robotern werden zur Berechnung der inversen Kinematik häufig geometrische Ansätze gewählt. Weitergehende Methoden sind die Erstellung einer Pseudoinversen aus der Transformationsmatrix der Vorwärtskinematik und die Erstellung einer Jacobi-Matrix mit Gradientenabstieg. Diese Methoden haben den Nachteil, dass sie mathematisch sehr aufwendig sind und damit einen schnellen Prozessor benötigen, wenn die Bewegungen in Echtzeit berechnet werden sollen.
Ziel dieser Arbeit war es, einen einfachen 5DOF-Roboterarm (DOF = „degrees of freedom“, Freiheitsgrade) zu bauen und dessen Vorwärtskinematik mit Hilfe der DH-Parameter zu implementieren. Darüber hinaus sollte eine rudimentäre Form der inversen Kinematik (geometrischer Ansatz für Gelenk 1-3) integriert werden. Neben dieser automatisierten Form der Positionierung sollte zudem die Möglichkeit geschaffen werden, die Gelenkmotoren manuell über Joysticks zu steuern.

1.2 Kurzbeschreibung

Für die Bewegung der Glieder wurden fünf Servo-Motoren verwendet, 3x MG996R und 2x MG90S. Die Armglieder inklusive des Greifarms und der Roboterbasis wurden aus Lego Technik gefertigt. Für die Steuerung wurde ein ATmega328p Microcontroller verwendet (Arduino Uno).
Die Positionierung des Roboters kann über zwei verschiedene Modi erfolgen, den manuellen und den automatischen Modus. Im manuellen Modus werden die Servomotoren über drei (Daumen-)Joysticks („PS2 Joystick Game Controller XY Dual-Achsen Joystick“) angesteuert. Die Richtung der Joystick-Auslenkung bestimmt die Bewegungsrichtung und die Weite der Auslenkung die Motorgeschwindigkeit.
Im automatischen Modus hingegen muss der Microcontroller mit einem Computer verbunden sein. Über den seriellen Monitor können Koordinaten (x, y und z) vorgegeben werden, die der Effektor daraufhin ansteuert. Um eine mehr oder minder präzise Positionierung des Effektors vornehmen zu können, müssen die Servomotoren kalibriert werden. Die inverse Kinematik wurde mit Hilfe von analytischer Geometrie für die ersten drei Motoren gelöst. Es gibt immer zwei mögliche Gelenkstellungen, um das Ziel zu erreichen. In der Software kann eingestellt werden, welche der

beiden Möglichkeiten verwendet werden soll. Kann ein Punkt nicht angefahren werden, wird eine entsprechende Fehlermeldung am seriellen Monitor des PCs ausgegeben. Darüber hinaus wird die berechnete Effektorposition aus der inversen Kinematik zur Kontrolle über Vorwärtskinematik rückgerechnet. Dafür wurde ein Modell des Roboters nach Denavit-Hartenberg (DH-Parameter) erstellt und eine Transformationsmatrix angefertigt.
Als letztes wurden Info-LEDs in das Projekt integriert. Jeweils eine grüne, eine gelbe und eine rote LED pro Motor geben zu Laufzeit Informationen über die Gelenkstellungen und über mögliche Probleme aus. Die 15 LEDs werden mit zwei hintereinandergeschalteten Schieberegistern (74HC595) angesteuert, wodurch nur drei Pins am Arduino belegt werden. Die grüne LED leuchtet, wenn es keine Probleme gibt. Die gelbe LED leuchtet, wenn der Servo eine vorgegeben Minimal- bzw. Maximalposition erreicht hat („Anschlagsbremse“). Die roten LEDs sind vor allem für den automatischen Modus wichtig. Sie geben unter anderem an, ob die gewünschte Effektorposition (mathematisch) überhaupt erreicht werden kann.

2. Hauptteil

2.1 Erklärung des Aufbaus

2.1.1 Schema

Abbildung 1 zeigt das Schema des Roboterarms mit Gelenken (G1-G5), relevanten Längenangaben (L1-L3), Gelenkwinkeln (Φ1-Φ5, positive Drehrichtung +) und die (rechtsdrehenden) Koordinatensysteme für jedes Gelenk. Der Roboterarm befindet sich in der Neutralstellung, in der die Gelenkwinkel Φ1-Φ4 definitionsgemäß Null betragen. Der Servomotor für den Greifarm (G5) wurde nicht in die DH-Parameter einbezogen. Als „Tool Center Point“ wurde das Ende des geschlossenen Greifarms gewählt. Mit Hilfe von Abb. 1 wurden die DH-Parameter in Tabelle 1 bestimmt.

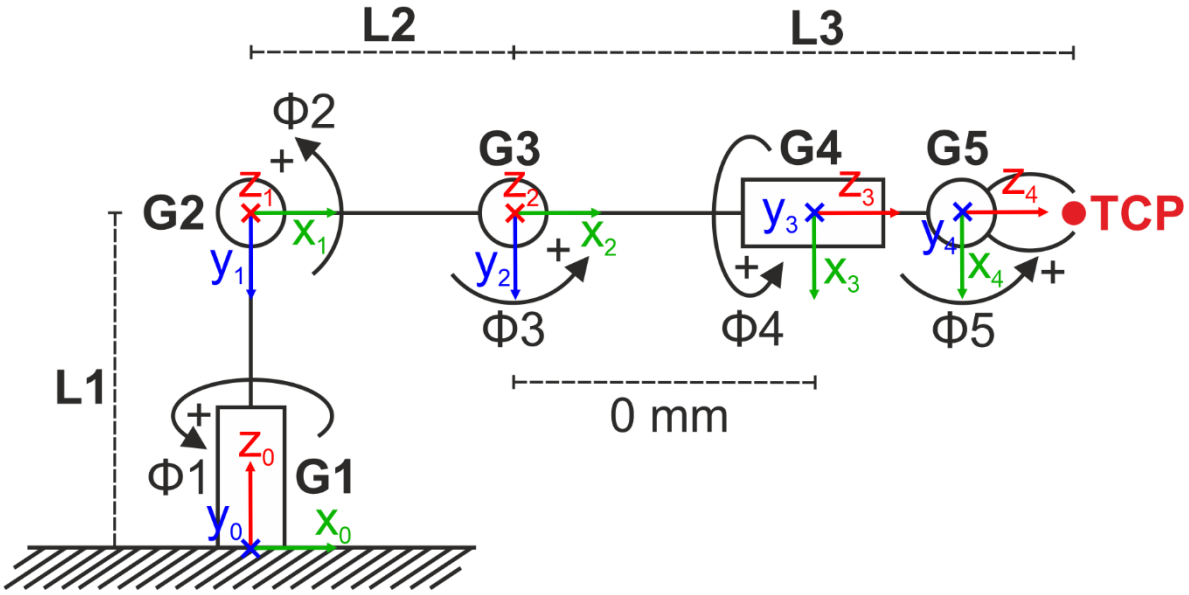

Abbildung 1: Schema des Roboterarms in Neutralstellung. G1-5: Gelenke in Form von Servo-Motoren, Φ1-Φ5: Gelenkwinkel in den Gelenken G1-G5 („+“ bedeutet positive Drehrichtung), L1-3: Längen der relevanten Armglieder, TCP: Lage des „Tool Center Points“. Zudem sind die Koordinatensysteme für jedes Gelenk eingezeichnet.

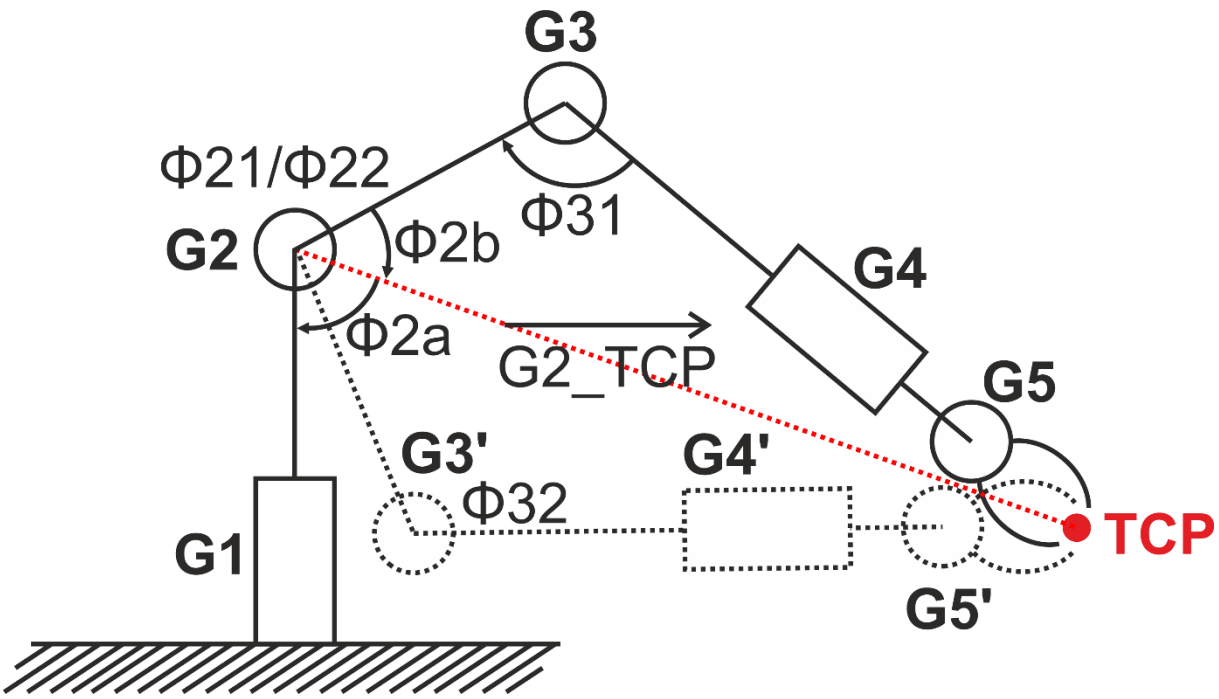

Abbildung 2: Hilfswinkel, Vektoren und alternative Gelenkposition. Gestrichelt ist die alternative Gelenkposition eingezeichnet, die für (fast) jeden TCP eingenommen werden kann (G3‘ – G5‘). Für diese Position muss Φ2 unterschiedlich berechnet und das Vorzeichen von Φ3 umgekehrt werden. Die Berechnung von Φ2 erfolgt in zwei Schritten. Φ2a ist der Winkel zwischen den Vektoren G2_0 (0 = Ursprung des Welt-Koordinatensystems) und G2_TCP. Φ2b hingegen kann über den Kosinussatz berechnet werden, genau wie Φ3. Die obere Gelenkkonfiguration (Φ21) wird erreicht, wenn Φ21 = Φ2a + Φ2b, die untere (Φ22), wenn Φ22 = Φ2a - Φ2b. Die erhaltenen Winkel müssen noch auf die Neutralstellung des Roboters bezogen werden: Φ21/Φ22 = (Φ21/Φ22 – 90°) * -1 und: Φ31 = 180° - Φ31; Φ32 = Φ31 * -1.

Tabelle 1: DH-Parameter der Gelenke 1-4

	Φ [°]	**d [mm]**	**a [mm]**	**α [°]**
G1	Φ1	L1	0	-90°
G2	Φ2	0	L2	0°
G3	90°- Φ3	0	0	+90°
G4	Φ4	L3	0	0

Längenangaben:
L1 = 160 mm
L2 = 90 mm
L3 = 190 mm

Die Lage von G4 zwischen G3 und G5 ist nicht relevant. Obwohl beim echten Roboter G4 näher an G5 als an G3 liegt, wurde die Rotationsachse von G4 für die Bestimmung der DH-Parameter auf die Rotationsachse von G3 gelegt, d.h. zwischen G3 und G4 ist 0 mm Abstand.

2.1.2 Schaltplan

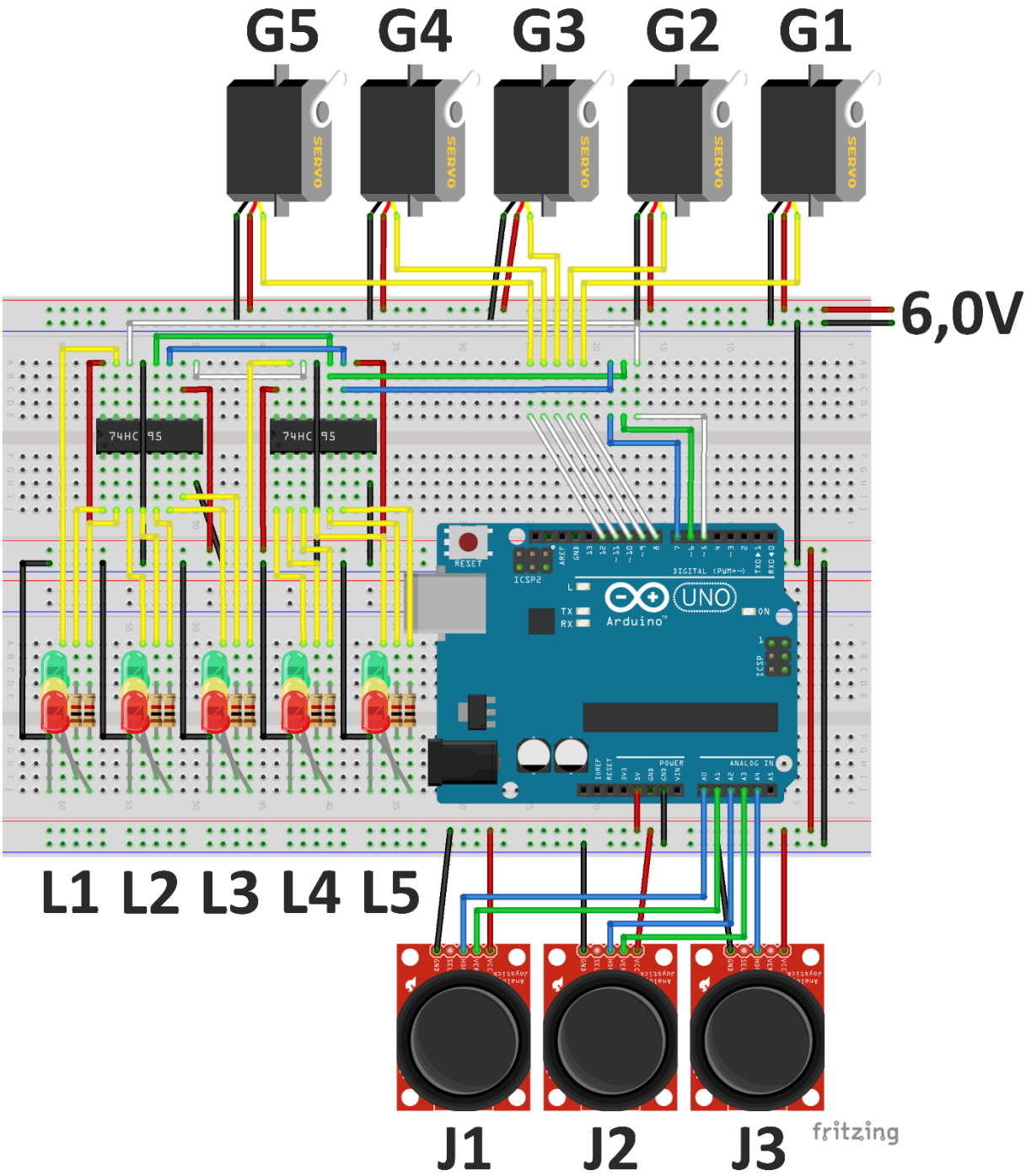

Abbildung 3: Schaltplan. G1-G5: Servomotoren der fünf Gelenke. Anders als in der Abbildung wurden zwei verschiedene Servomotoren verwendet, nämlich 3x MG996R (G1-3) und 2x MG90S (G4-5). L1-5: Jedem Servo sind drei Info-LEDs zugeordnet (grün, gelb, rot). Die Zahlen beziehen sich auf die Gelenke, d.h. die Info-LEDs L1 geben den Status des Servomotors von Gelenk 1 aus. Jeder LED wurde ein 1kΩ Widerstand vorgeschaltet. Links auf dem oberen Steckbrett sind zwei Schieberegister lokalisiert (74HC595), welche für die Ansteuerung der 15 LEDs verantwortlich sind. Die Register sind hintereinandergeschaltet (16 Bits). J1-3: Joysticks für die manuelle Steuerung der Servo-Motoren. Bei Joystick 1 und 2 werden die x- und y-Achsen verwendet, bei Joystick 3 hingegen nur die x-Achse. Joysticks, Schieberegister und LEDs werden über den USB Anschluss des Arduinos mit 5,0 V Spannung versorgt. Die Servomotoren hingegen besitzen eine eigene Spannungsversorgung von 6,0 V.

2.1.3 Bilder

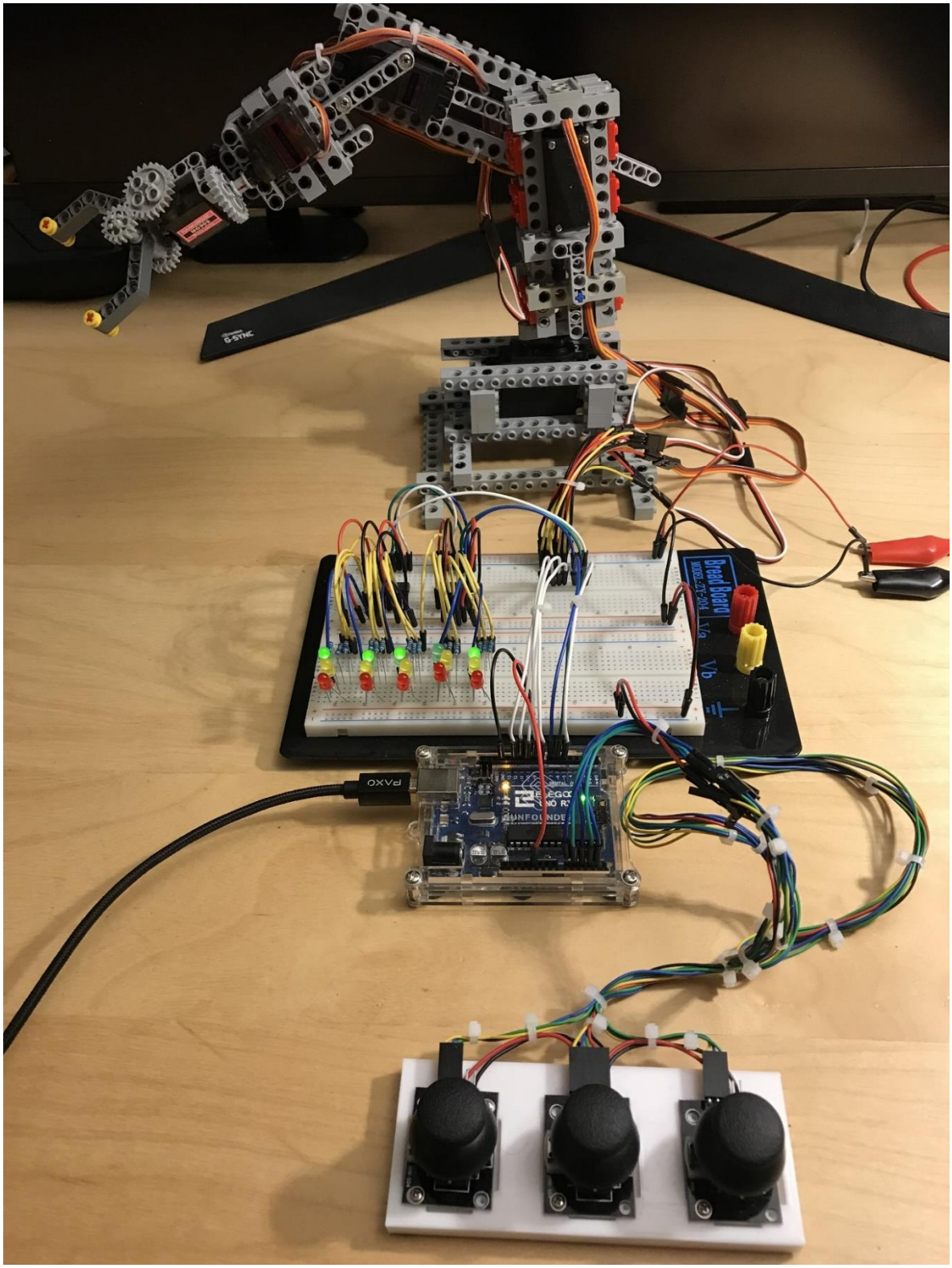

Abbildung 4: Foto des Aufbaus. Anders als in Abb. 3 ist der Arduino Uno hier nicht auf dem Steckbrett positioniert. Die drei Joysticks (unten) sind auf eine 3D-gedruckte Platte geschraubt. Man sieht die 5,0 V Spannungsversorgung über USB am Arduino PCB und die 6,0 V Spannungsversorgung für die Servo-Motoren (Klemmen rechts oben), die von einem Labornetzteil stammen. Beide Stromkreise haben eine gemeinsame Masse (GND). Die drei großen MG996R-Servos (G1-3) sind lediglich in die Lego Bausteine geklemmt, während die zwei kleinen 2x MG90S-Servos (G4-5) mit doppelseitigen Klebeband befestigt sind.

2.2 Erklärung Programm/Quelltext

2.2.1 Generierung des PWM-Signals:

Die Ansteuerung mehrerer Servomotoren gleichzeitig basiert auf folgender Anleitung: https://www.youtube.com/watch?v=dmjvHvw3Rc8 und wurde für das Projekt modifiziert. Die Generierung des PWM-Signals erfolgt über Timer 1 und kann an jedem beliebigen Port des Microcontrollers ausgegeben werden. Timer 1 wird dabei im Mode 14 (Fast PWM, Top: ICR1, WGM11, 12 und 13 = 1) mit einem Prescaler von 8 betrieben (CS11 = 1). Bei einer CPU-Frequenz von 16 MHz ergibt sich die nötige Periodendauer von 20 ms, wenn ICR1 den Wert 39999 erreicht hat. Dann wird ein Interrupt ausgelöst, der alle Datenpins der Servo-Motoren (in diesem Fall an Port B) gleichzeitig auf HIGH setzt:

```
//Timer 1 Handler für PWM-Signal
ISR(TIMER1_COMPA_vect)
{
	//Alle Pins an Port B auf HIGH setzen
	//Servos werden synchron angesteuert
	PORTB = 0xFF;
}
```

Das Zurücksetzen auf LOW erfolgt, wenn ein vorgegebener Wert im Register TCNT1 erreicht ist. Diese Werte werden für alle fünf Motoren im Struct „Pul“ gespeichert (Pul.pul_1 - Pul.pul_5). 1 ms bzw. 1000 µs entsprechen einem Wert von 2000 in TCNT1. Um also ein PWM-Signal mit x µs HIGH-Level zu generieren, muss der Wert für TCNT1 mit 2 multipliziert werden. Das zurücksetzen der Servo-Pins auf LOW soll hier am Beispiel von zwei Servos gezeigt werden:

```
//Servo-Pins auf LOW setzen
if(TCNT1 >= 1000 && TCNT1 <= 5200)
{
	//Pin B0: Servo 1 (Phi_1):
	if(TCNT1 >= Pul.pul_1 && bit_is_set(PORTB, SERVO_const[0][0]))
	{
		PORTB &= ~(1<<SERVO_const[0][0]);
	}
	//Pin B1: Servo 2 (Phi_2):
	if(TCNT1 >= Pul.pul_2 && bit_is_set(PORTB, SERVO_const[1][0]))
	{
		PORTB &= ~(1<<SERVO_const[1][0]);
	}
	//usw.....
}
```

Das Zurücksetzen der Servo-Pins erfolgt nur, wenn:
1. der Wert in TCTN1 erreicht wird (TCNT1 >= Pul.pul_1, bestimmt die Pulslänge)
2. der Pin auf HIGH ist (bit_is_set(), SERVO_const[x][0] = globale Konstanten für Servo Pins) und
3. wenn der Counter in dem Bereich liegt, wo das Umschalten erfolgen soll (if(TCNT1 >= 1000 && TCNT1 <= 5200)).

Alle anderen Aktionen des Programms sollen nur ausgeführt werden, wenn der Counter außerhalb dieses Bereichs liegt:

```
if(TCNT1 < 1000 || TCNT1 > 5200)
{
	//Anderer Code als PWM-Generierung…..
}
```

Dies soll für eine deutlich bessere Genauigkeit des PWM-Signals sorgen, da der Programmablauf nicht mit anderem Code in Konflikt kommt. Die Grenzen 1000-5200 wurden so gewählt, weil die verwendeten MG996R-Servos einen großen Pulsweitenbereich von 500 µs bis 2600 µs haben (muss für TCTN1 x2 genommen werden!). Der Code sorgt zusammenfassend also dafür, dass alle Servos gleichzeitig auf HIGH gesetzt werden, jedoch zu unterschiedlichen Zeiten wieder zurück auf LOW gesetzt werden. Ferner wird eine Periodendauer von 20 ms generiert.

2.2.2 Manueller Modus:

Im manuellen Modus werden die Servo-Motoren mit Hilfe von drei Daumen-Joysticks gesteuert. Die Richtung der Auslenkung entspricht der Servo-Richtung und die Weite der Auslenkung der Servo-Geschwindigkeit (je weiter die Auslenkung desto größer die Geschwindigkeit). Von den sechs vorhandenen Achsen wurden nur fünf verwendet, da nur fünf Motoren verbaut sind. Die Druckschalter an jedem Joystick wurden nicht mit Funktionen belegt. Folgende Zuordnung der analogen Achsen zu den Gelenken/Servos wurde festgelegt:

- Joystick 1, x-Achse: Horizontale Rotation der Roboterbasis in Gelenk 1
- Joystick 1, y-Achse: Vertikale Rotation in Gelenk 2
- Joystick 2, y-Achse: Vertikale Rotation in Gelenk 3
- Joystick 2, x-Achse: Rotation des Greifarms in Gelenk 4
- Joystick 3, x-Achse: Öffnen und Schließen des Greifarms in Gelenk 5

Der Analog-Digital-Konverter des Arduinos hat (maximal) 10 Bit Auflösung, d.h. dass das digitalisierte Joystick-Signal Werte von 0-1023 erreicht. Betrachtet man als Beispiel die x-Achsen, dann wäre der ganz linke Ausschlag bei 0, der ganz rechte Ausschlag bei 1023 und die Mitte würde sich bei 511 befinden. Bei der y-Achse wäre entsprechend die ganz untere Position 0 und die ganz obere 1023.
Jeder Achse wurde eine „Joystick Deadzone“ (Bezeichnung im Gaming Bereich) von etwa 5% der maximalen Auslenkung in jede Richtung zugewiesen (globale Konstante „JOY_dz“ = 25, siehe Funktion joy_to_pulse()). Dies verhindert, dass die Motoren sich ohne Bedienung des Joysticks von selbst bewegen, weil die Mitte nicht exakt stimmt. Auf die Implementierung einer Joystick-Kalibrierung wurde verzichtet.
Um die analogen Eigenschaften der Joysticks voll auszunutzen, wurde die Servo-Geschwindigkeit von deren Auslenkung abhängig gemacht. Generell wird die Position der Servos durch die Pulsweite des PWM-Signals bestimmt (500-2500 µs Puls bei 20 ms Periodendauer). Eine Pulsweitenänderung bedeutet demnach eine Positionsänderung. Ist die Pulsweitenänderung hingegen nicht linear, dann beschleunigt der Servo oder er bremst ab.

Die Pulsweitenänderung wurde über folgende Formel implementiert (am Beispiel einer x-Achse):
Ausschlag nach rechts: $\Delta Pulse = ((j\_val - JOY\_mid) / JOY\_div)^2$
Ausschlag nach links: $\Delta Pulse = ((JOY\_mid - j\_val) / JOY\_div)^2$

j_val: digitalisierter Joystick-Wert (0-1023)
JOY_mid: globale Konstante für Joystick-Mittelstellung (511)
JOY_div: globale Konstante für einen Teiler, der die Geschwindigkeitsänderung festlegt. Je höher der Wert, desto langsamer die Geschwindigkeitsänderung

Erfahrungsgemäß sollte die Pulsänderung in den Extrempositionen maximal 40 betragen, da die Servos sonst zu schnell werden. Zu beachten ist, dass die Pulsweitenänderung in der Formel nicht linear von statten geht, sondern quadratisch. Dies sorgt dafür, dass die Motor-Geschwindigkeit bei geringer Joystick-Auslenkung zunächst nur langsam zunimmt und bei hoher Auslenkung sehr schnell. Dies wird im Gaming Bereich als „Joystick Sensitivity Curve“ bezeichnet und bewirkt, dass die Motoren mit den Joysticks präzise positioniert werden können. Bei einer linearen Pulsänderung hat sich gezeigt, dass sich die Geschwindigkeit bei geringer Auslenkung zu schnell ändert. Experimente mit Exponenten 3 oder 4 zeigten eine noch bessere Sensitivität bei kleiner Auslenkung. Allerdings wurde befürchtet, dass die Rechenlast dadurch unnötig groß wird.
Zu erwähnen ist zudem, dass das Joystick-Signal alle 12 ms ausgelesen wird und die Servo-Position alle 24 ms aktualisiert wird. Die Aktualisierungsrate der Servo-Position hat einen großen Einfluss auf die Servo-Geschwindigkeit. Es wurde für dieses Projekt allerdings entschieden, diese Rate konstant zu lassen und die Geschwindigkeitsanpassung über den oben beschriebenen Teiler umzusetzen.
Für die Abtastrate der Joysticks und die Aktualisierungsrate der Servo-Position wurde Timer 0 verwendet, der jede Millisekunde einen Interrupt erzeugt. In der Interrupt-Routine werden Variablen inkrementiert, bis der entsprechende Wert von 12 bzw. 24 erreicht wurde:

```
//Timer 0 Handler für die Abtastrate der Joysticks und die Servo-Geschwindigkeit
(Interrupt jede ms)
ISR(TIMER0_COMPA_vect)
{
	Joy_ms_count++;
	Serv_ms_count++;
}
```

Dann wird die Aktion ausgelöst und die Laufvariablen zurückgesetzt (hier am Beispiel der Joystick-Aktualisierung):

```
//Alle x ms die Joystick-Eingabe speichern
if(Joy_ms_count >= JOY_ms_refresh)
{
	//Hier Aktualisierung der Joystick-Eingaben, d.h. die Werte werden im Struct
	„adcJ“ gespeichert

	//Reset Counter
	Joy_ms_count = 0;
}
```

2.2.3 Automatischer Modus

Im automatischen Modus kann die Position des Roboterarms über Koordinaten (x-, y- und z-Koordinaten in mm) vorgegeben werden. Dafür muss der Arduino Uno mit einem PC verbunden sein. Wird das Programm gestartet, dann erhält man über den seriellen Monitor am PC eine Aufforderung zur Eingabe der Koordinaten (int oder double, kommasepariert). Erfolgt eine Eingabe, löst dies einen Interrupt aus (ISR(USART_RX_vect)), welcher die eingegebenen Zeichen **als String (!)** in die Variable „usart_rx_buffer[]“ schreibt. **Es ist wichtig, dass die Eingabe am PC mit einem abschließenden \r\n erfolgt** (entsprechenden Haken beim seriellen Monitor im AtmelStudio setzen). Zur Kontrolle werden die eingegebenen Koordinaten danach am seriellen Monitor des PCs ausgegeben.
Ist der String fertig eingelesen, wird die Variable „usart_string_compl“ auf 1 (= fertig) gesetzt. Erst dann erfolgt die weitere Verarbeitung, nämlich das Parsen des Strings (Funktion pars_rx_strg()). Das bedeutet, dass aus dem String wieder drei Double-Variablen gemacht werden sollen. Die Funktion, die dafür verwendet wurde (strtok()), kann Strings mit Hilfe eines definierten Delimiters in Substrings unterteilen. In diesem Fall wurde als Delimiter das Komma gewählt, mit dem die Koordinaten bei der Eingabe getrennt wurden. Mit Hilfe der Funktion atof() werden die drei Substrings in Double-Variablen

umgewandelt und im Struct „TCP“ gespeichert. Ist das Umwandeln nicht möglich, gibt die Funktion 0 zurück. Um die erneute Eingabe von Koordinaten zu ermöglichen, wird die Variable „usart_string_compl“ wieder auf 0 zurückgesetzt.
Als nächstes erfolgt die Berechnung der Gelenkwinkel mit Hilfe der inversen Kinematik (Funktion inv_kinematic()). Auf die Details der Berechnung soll hier aus Platzgründen nicht eingegangen werden, es basiert aber auf einfachen trigonometrischen Überlegungen. Die berechneten Winkel werden im Struct „Ang“ gespeichert. Dabei werden immer beide möglichen Gelenkstellungen berechnet (Phi_21 und Phi_22 bzw. Phi_31 und Phi_32). Ist eine Berechnung nicht möglich oder führt die Berechnung zu sinnlosen Daten (nan, inf), dann gibt die Funktion 0 zurück (ansonsten 1). Ist die Berechnung hingegen erfolgreich, werden die berechneten Gelenkwinkel mit Hilfe der Vorwärtskinematik noch einmal kontrolliert. Bei dieser Funktion (fwd_kinematic()) werden die Gelenkwinkel übergeben und Punkt 0, 0, 0 (Ursprung des Welt-Koordinatensystems) transformiert. Das Ergebnis der Berechnung wird im Struct „TCP_calc“ gespeichert und mit den initial eingegebenen Koordinaten verglichen. Da vorgegebene und berechnete Koordinaten (sehr) geringfügig voneinander abweichen können, werden diese mit einer gewissen Toleranz verglichen (0,01 mm). Ist der Vergleich nicht erfolgreich, wird eine Warnung am seriellen Monitor ausgegeben und der Roboterarm verbleibt in der aktuellen Position.
Für die Berechnung der Vorwärtskinematik wurden zunächst die DH-Parameter des Roboters bestimmt (siehe Tabelle 1) und dann für jedes Gelenk in die unten aufgeführte Matrix eingesetzt:

a)

$$ {}^{n-1}T_n = \mathrm{Rot}(z_{n-1},\theta_n)\cdot\mathrm{Trans}(z_{n-1},d_n)\cdot\mathrm{Trans}(x_n,a_n)\cdot\mathrm{Rot}(x_n,\alpha_n) $$

$$ = \begin{pmatrix} \cos\theta_n & -\sin\theta_n\cos\alpha_n & \sin\theta_n\sin\alpha_n & a_n\cos\theta_n \\ \sin\theta_n & \cos\theta_n\cos\alpha_n & -\cos\theta_n\sin\alpha_n & a_n\sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix}. $$

b)

```
//Transformationsmatrizen
matr M01 = {{cos(Phi_1),    0,      -sin(Phi_1),    0},
            {sin(Phi_1),    0,       cos(Phi_1),    0},
            {0,            -1.0,     0,             L_1},
            {0,             0,       0,             1.0}};

matr M12 = {{cos(Phi_2),    -sin(Phi_2),    0,      L_2*cos(Phi_2)},
            {sin(Phi_2),     cos(Phi_2),    0,      L_2*sin(Phi_2)},
            {0,              0,             1.0,    0},
            {0,              0,             0,      1.0}};

matr M23 = {{cos(Phi_3+PI/2),    0,      sin(Phi_3+PI/2),      0},
            {sin(Phi_3+PI/2),    0,     -cos(Phi_3+PI/2),      0},
            {0,                             1.0,    0,         0},
            {0,                             0,      0,         1.0}};

matr M34 = {{cos(Phi_4),    -sin(Phi_4),    0,      0},
            {sin(Phi_4),     cos(Phi_4),    0,      0},
            {0,              0,             1.0,    L_3},
            {0,              0,             0,      1.0}};
```

Abbildung 5: Vorwärtskinematik a) Transformationsmatrix (Bildquelle: Wikipedia). b) Die Transformationsmatrizen für jedes Einzelgelenk erhält man, wenn die DH-Parameter aus Tabelle 1 in Matrix eingesetzt werden. Die gezeigten Matrizen wurden vereinfacht, indem Koeffizienten, die 0 oder 1 ergaben, entsprechend ersetzt wurden. Für Gelenk 1 wird die zugehörige Matrix als M01 bezeichnet, für Gelenk 2 M12, für Gelenk 3 M23 und für Gelenk 4 M34. Durch die Multiplikation dieser Matrizen erhält man eine Gesamt-Matrix M04, welche Transformationen vom Welt-Koordinatensystem in das Koordinatensystem von Gelenk 4 beschreibt.

Durch die Multiplikation der Transformationsmatrix eines jeden Gelenks erhält man die Gesamtmatrix, welche Transformationen vom Koordinatensystem 0 (Welt-Koordinatensystem) zum Koordinatensystem 4 beschreibt. Multipliziert man diese Transformationsmatrix mit dem Ursprung des Welt-Koordinatensystems (0, 0, 0), erhält man die Position (und Orientierung) des TCPs. Entsprechende Funktionen für Vektor- und Matrizenrechnungen wurden als Funktionen implementiert.

Ist die Berechnung der Vorwärtskinematik erfolgreich, werden die berechneten Gelenkwinkel in Pulsweiten umgewandelt (Funktion ang_to_puls()). Dafür müssen die Motoren kalibriert werden, d.h. es muss die Pulsweite von mindestens zwei Winkeln bekannt sein. Für die Zweipunkt-Kalibrierung wurde für die Servos 1-5 jeweils der Winkel 0° (Neutralstellung) und der Winkel -90° verwendet. Mit Hilfe einer einfachen Geradengleichung kann dann aus den Winkeln die entsprechenden Pulsweiten berechnet werden. Man muss nur beachten, dass die Werte für die Joystick-Auslenkung nach links bzw. unten anders berechnet werden müssen als bei einer Auslenkung nach rechts bzw. oben. Das wäre nicht notwendig, wenn die Joysticks in Mittelstellung bei 0 wären. Ferner müssen die berechneten Pulslängen in µs für Timer 1 in noch mit 2 multipliziert werden.
Die Funktion set_pulse() schreibt die berechneten Pulslängen für jeden Servo in das Struct „Pul“. Bei dieser Funktion kann gewählt werden, ob Gelenkstellung 1 oder 2 verwendet werden soll. Zudem wird überprüft, ob die Pulsweite einen definierten Maximal- bzw. Minimalwert erreicht (“Anschlagsbremse“). Pin, Pulsweiten bei 0° und -90° sowie Minimal- und Maximalauslenkung eines jeden Servos sind als globale Konstanten im Array „SERVO_const“ gespeichert. Werden Minimal- bzw. Maximalauslenkung über- bzw. unterschritten, werden sie auf die Minimal- bzw. Maximalauslenkung zurückgesetzt. Der Motor bleibt also an dieser Position stehen.

2.2.4 Info-LEDs und Schieberegister

In die Schaltung wurden LEDs integriert, um Informationen über den Status der Motoren während der Laufzeit anzuzeigen. Jedem der 5 Servo-Motoren sind drei LEDs zugeordnet: eine grüne, eine gelbe und eine blaue LED. Damit überhaupt 15 LEDs über den Arduino Uno angesteuert werden können, werden Schieberegister benötigt, da ansonsten die Anzahl an Pins nicht ausreicht. Da die hier verwendeten Schieberegister 74HC595 nur acht Bit haben, wurden zwei davon hintereinander geschalten. Damit erhält man 16 Bit, was für 15 LEDs ausreicht. Es leuchtet immer nur eine LED gleichzeitig pro Servo, weswegen sich der Strombedarf, der über den Spannungswandler des Arduino Uno PCBs gedeckt wird, in Grenzen hält.
Der Zustand der LEDs wird im Struct „Led“ gespeichert (HIGH=1, LOW=0). Für die Ausgabe werden die Daten des Structs in eine 16 Bit lange, binäre Variabel umgewandelt (uint16_t data). Das erfolgt über die Funktion shiftr_data(), wo zudem die Zuordnung LEDs zu den zugehörigen Bits erfolgt. Über die Funktion shiftr_init() werden die Schieberegister initialisiert, mit shiftr_shift() das nächste Bit in das Register geladen und mit shiftr_latch() erfolgt die Ausgabe der 16 Bits.

Die Farben der LEDs wurde folgenden Ereignissen zugeordnet:

- **Grün:** Alles in Ordnung. Die geplante Bewegung kann ordnungsgemäß ausgeführt werden.
- **Gelb:** Der entsprechende Servo ist an der vorgegebenen Maximal- bzw. Minimalposition angekommen und kann nicht weiterbewegt werden. Dies gilt sowohl für den manuellen als auch für den automatischen Modus. Im automatischen Modus bedeutet das allerdings, dass der gewünschte Punkt nicht angefahren werden kann, weil ein oder auch mehrere Servo-Motoren nicht in die notwendige Position gebracht werden können. Bei einem idealen Roboter ohne Einschränkung der Bewegungsfreiheit (vor allem bezüglich Kollisionen und einer Rotation von 360° pro Gelenk anstatt 180°) würde der Punkt mathematisch aber schon erreicht werden können.

- **Rot:** Nur für den automatischen Modus wichtig. Diese LEDs leuchten alle gemeinsam auf, weil die Fehlermeldungen nicht durch einen einzelnen Servo verursacht werden. Prinzipiell hätte hier auch nur eine einzige LED gereicht. Bei folgenden Ereignissen leuchten die roten LEDs auf:
  - Die Funktion der inversen Kinematik (inv_kinematic()) gibt 0 zurück. Das kann bedeuten, dass 1. als x- und y-Koordinate 0 gewählt wurde, 2. das Ziel nicht erreicht werden kann, weil es zu nah am Roboter ist und 3. die Berechnung eines der Winkel den Wert Unendlich oder „not a number“ (nan) ergeben hat.
  - Die Berechnung der Vorwärtskinematik (fwd_kinematic()) gibt 0 zurück. Das passiert, wenn der berechnete Wert Unendlich oder „not a number“ (nan) ergibt.
  - Der berechnete TCP aus der Vorwärtskinematik stimmt innerhalb einer gewissen Toleranz (0,01 mm) nicht mit dem vorgegebenen TCP überein.

Momentan leuchten die roten LEDs nur für 50 ms auf, weil die Gelenkstellung danach vom manuellen Modus als korrekt erkannt wird. Um dieses Problem zu umgehen, könnte man die Druckschalter der Joysticks dazu benutzen, zwischen manuellem und automatischem Modus zu wechseln. Dann würden die roten LEDs so lange aufleuchten, bis Koordinaten eingegeben werden, die vom Roboterarm angefahren werden können. Aktuell laufen beide Modi gleichzeitig.

3. Fazit

3.1 Schwierigkeiten/Probleme

Tatsächlich haben sich im Laufe des Projekts viele Probleme offenbart, die über den vermittelten Stoff des Kurses hinausgingen. Eine große Herausforderung war die Ansteuerung von 5 Servomotoren gleichzeitig, ohne die Bibliothek servo.h zu benutzen. Eine mögliche Lösung wurde im Internet gefunden und in dieses Projekt mit Erfolg integriert. Mit dieser Methode konnte nicht nur ein PWM-Signal für mehrere Servo-Motoren gleichzeitig generiert werden, die Pins waren zudem unabhängig von den sechs nativen PWM-Pins des ATmega328p und konnten beliebig gewählt werden.
Eine weitere Schwierigkeit war das Einlesen der Koordinaten im automatischen Modus über den seriellen Monitor. Problematisch war vor allem, dass Double-Variablen gesendet werden sollten, welche beim ATmega328p 32 Bits (= 4 Bytes) belegen. Dies erfordert, dass die Daten per USART in Form mehrerer Pakete (mit 8 jeweils Bit) gesendet werden müssen. Die hier umgesetzte Lösung funktioniert, ist aber vermutlich nicht besonders sparsam mit den Ressourcen des Microcontrollers, da mit Strings (C Standard-Bibliothek string.h) gearbeitet wird.
Ferner hat es auch etwas Zeit gekostet, die Steuerung der Servo-Geschwindigkeit mit Hilfe der Joysticks zu integrieren. Das war primär auch nicht geplant. Prinzipiell wäre es ausreichend gewesen, die Joystick-Richtung auszulesen und die Motoren entsprechend mit konstanter Geschwindigkeit zu bewegen. Das Hauptproblem war jedoch, dass sich die Motoren ohne Geschwindigkeitskontrolle zu schnell bewegt haben, was die exakte Platzierung des Effektors schwierig bis unmöglich machte.
Bezüglich der Joysticks musste zudem eine „Joystick Deadzone“ von etwa 5% in jede Richtung eingebaut werden, da es sich bei den verwendeten Joysticks um billige und damit unpräzise Ware gehandelt hat. In diesem Fall bedeutet das, dass die Joysticks in Mittelstellung faktisch nie einen Wert von 511 erreicht haben. Eine „Deadzone“ von 5% konnte dieses Problem beheben.

3.2 Verbesserungsmöglichkeiten

Zunächst sollten die Armglieder des Roboters mechanisch stabiler konstruiert werden und die Servo-Motoren fest mit den Bauteilen verschraubt/verklebt sein. Da die drei großen Servos nur in das Lego geklemmt sind, bleibt zu viel Bewegungsspielraum. Beim automatischen Modus ist die Positionierung deshalb nicht sehr genau. Noch besser wäre es, wenn die Glieder 3D-gedruckt werden würden. Lego war bei diesem Projekt eigentlich nur Mittel zum Zweck, um den Roboterarm schnell und unkompliziert bauen zu können. Es wäre auch von Vorteil gewesen, die Servos nicht direkt als Rotationsachsen zu verwenden, sondern in einfache Getriebe zu integrieren. Auf diese Weise können mit Hilfe von Lego-Zahnrädern sehr leicht die normalen 180° Rotationsweite der Motoren auf 360° ausgedehnt werden.
Bezüglich der Programmierung wäre es schön, eine Kollisionserkennung für den manuellen (aber auch den automatischen) Modus hinzuzufügen. Die einzelnen Bausteine dafür sind prinzipiell schon vorhanden. Was noch fehlt, wäre eine inverse Funktion puls_to_ang() zur Funktion ang_to_puls(). In diesem Fall würden die Pulslängen übergeben und daraus die Gelenkwinkel berechnet werden. Jedes Mal, wenn die Pulsweite über die Joystickeingabe erhöht oder erniedrigt werden würde (24 ms Takt), könnten damit die daraus resultierenden Gelenkwinkelkonfigurationen bestimmt werden und über die Funktion zur Vorwärtskinematik in die Effektorposition umgerechnet werden. Damit wäre zu jeder Zeit klar, wo sich der Effektor gerade befindet. Auf diese Weise kann zum Beispiel vermieden werden, dass der Effektor in die negative z-Richtung (also in den Boden hinein) bewegt wird. So eine Bewegung kann mit einer Minimal-/Maximalauslenkung, wie es in diesem Projekt für jeden Servo umgesetzt wurde, nicht verhindert werden, da zwei Motoren gleichzeitig dafür verantwortlich sein können (z.B. G2 und G3). Auch die Geometrie der Roboterbasis kann grob mathematisch nachgebildet werden und auf Kollisionen hin untersucht werden. Zudem könnte berechnet werden, ob es bei einer der beiden Gelenkwinkelkonfigurationen nicht zur Kollision kommt.
Als letztes könnten die drei Info-LEDs für jeden Servo gegen eine RGB-LED ausgetauscht werden. Das ist deshalb möglich, weil sowieso immer nur eine der drei LEDs leuchtet. Damit können auch mehr Farben gemacht werden, um mehr Informationen darzustellen. Außerdem würde es Platz sparen.

3.3 Fazit

Meine Projektarbeit hat viele Inhalte aufgegriffen, die im Laufe des Kurses vermittelt wurden. Dies beinhaltet die Generierung eines PWM-Signals, die Ansteuerung von Servo-Motoren, die Verwendung eines Schieberegisters, eines Timers und eines Analog-Digital-Konverters. Dieses Wissen wurde für diese Projektarbeit zum Teil noch erweitert und vertieft. Beispielsweise wurden fünf Servo-Motoren gleichzeitig angesteuert, was deutlich komplexer ist als die Kursübungen mit nur einem Servo. Ferner wurden zwei Schieberegister hintereinandergeschaltet, was auch über den gelernten Kursinhalt hinausgeht. Nicht im Kurs behandelt wurde hingegen die Verwendung von (Daumen)Joysticks. Dieses Wissen wurde selbstständig erarbeitet.
Insgesamt war das ein sehr spannender und lehrreicher Kurs. Besonders die Assembler-Programmierung war neu für mich und hat meinen Horizont deutlich erweitert. Darüber hinaus hat mir der Kurs eine neue Sicht auf die Programmiersprache C gegeben. Hier habe ich gelernt, dass C nicht gleich C ist (AVR C vs. Arduino Skript), wie viel Arbeit eigentlich vom Compiler abgenommen wird und dass der Compiler nicht immer alles optimal macht.

4. Literaturverzeichnis

4.1 Verwendete Datenblätter

- Datenblatt ATmega328p
- Datenblatt 74HC595

4.2 Webseiten

USART:

- https://www.xanthium.in/how-to-avr-atmega328p-microcontroller-usart-uart-embedded-programming-avrgcc
- https://embedds.com/programming-avr-usart-with-avr-gcc-part-1/
- https://www.mikrocontroller.net/articles/AVR-GCC-Tutorial/Der_UART

PWM und Servo-Steuerung:

- https://www.mikrocontroller.net/articles/AVR-Tutorial:_PWM
- https://www.newbiehack.com/MicrocontrollerControlAHobbyServo.aspx
- https://docs.arduino.cc/tutorials/generic/secrets-of-arduino-pwm
- https://www.youtube.com/watch?v=9WeewNNGs5E
- https://www.youtube.com/watch?v=dmjvHvw3Rc8

Analog-Digital-Konverter:

- https://www.mikrocontroller.net/articles/AVR-Tutorial:_ADC
- https://sites.google.com/site/avrasmintro/home/usisng-the-adc
- https://scienceprog.com/programming-avr-adc-module-with-avr-gcc/
- https://maxembedded.com/2011/06/the-adc-of-the-avr/

Schieberegister:

- https://embedds.com/interfacing-shift-register-with-avr/
- https://extremeelectronics.co.in/avr-tutorials/using-shift-registers-with-avr-micro-avr-tutorial/
- http://adam-meyer.com/arduino/74HC595

C-Funktionen:

- https://www.educative.io/edpresso/splitting-a-string-using-strtok-in-c

Vektor- und Matrizenrechnung, 3D-Transformationen:

- https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/

DH-Parameter:

- https://de.wikipedia.org/wiki/Denavit-Hartenberg-Transformation
- https://www.youtube.com/watch?v=JBNmPq8eg8w&t=322s

5. Anhang

5.1 Quelltext

```c
//µC-Kurs, WS 21/22
//Projektarbeit
//Markus Reichold
//5DOF-Roboterarm

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include "typedef.h"
#include "functions.h"
#include "var.h"

//Defines
#define USART_RX_MAX 100

//////////
// Main //
//////////

int main(void)
{
    //PWM-Signal (Timer 1) initialisieren
    pwm_init();
    //USART initialisieren
    usart_init();
    //ADC für die Joysticks initialisieren
    adc_init();
    //Timer 0 für die Abtastfrequenz der Joysticks und die Servogeschwindigkeit
      initialisieren
    ms_init();
    //Schieberegister (2xHC595) initialisieren
    shiftr_init();
    //Interrupts aktivieren
    sei();

    //Gelenkwinkel und TCP auf Neutralstellung setzen
    reset_ang(&Ang);
    reset_tcp(&TCP);
    //Gelenkwinkel in Pulsweite (*2) umrechnen
    //und im Strut Pul speichern
    //Letzter Parameter gibt an, welche der zwei möglichen Gelenkkonfigurationen
      verwendet werden soll
    set_pulse(&Ang, &Pul, 1, &Led);

    //Eingabeaufforderung
    //Bei folgender Art der String-Initialisierung wird das \0 vom Compiler
      automatisch angehängt
    char text1[] = "Koordinaten [mm] im Format: x,y,z eingeben:\n\r";
    usart_send_strg(text1);

    //////////
    // Loop //
    //////////

    while(1)
    {
```

```
        ////////////////////////////
        // PWM Signalgenerierung //
        ////////////////////////////

        //Servo-Pins auf LOW setzen, wenn
        //1) der gewünschte Wert in TCNT1 erreicht ist (bestimmt Pulslänge)
        //2) der Pin auf HIGH gesetzt ist
        //3) die Pulslänge in dem Bereich liegt, wo das Umschalten nötig ist
          (500-2600 x2 ms)
        if(TCNT1 >= 1000 && TCNT1 <= 5200)
        {
            //Pin B0: Servo 1 (Phi_1):
            if(TCNT1 >= Pul.pul_1 && bit_is_set(PORTB, SERVO_const[0][0]))
            {
                PORTB &= ~(1<<SERVO_const[0][0]);
            }
            //Pin B1: Servo 2 (Phi_2):
            if(TCNT1 >= Pul.pul_2 && bit_is_set(PORTB, SERVO_const[1][0]))
            {
                PORTB &= ~(1<<SERVO_const[1][0]);
            }
            //Pin B2: Servo 3 (Phi_3):
            if(TCNT1 >= Pul.pul_3 && bit_is_set(PORTB, SERVO_const[2][0]))
            {
                PORTB &= ~(1<<SERVO_const[2][0]);
            }
            //Pin B3: Servo 4 (Phi_4):
            if(TCNT1 >= Pul.pul_4 && bit_is_set(PORTB, SERVO_const[3][0]))
            {
                PORTB &= ~(1<<SERVO_const[3][0]);
            }
            //Pin B4: Servo 5 (Greifarm, Phi_5):
            if(TCNT1 >= Pul.pul_5 && bit_is_set(PORTB, SERVO_const[4][0]))
            {
                PORTB &= ~(1<<SERVO_const[4][0]);
            }
        }

        //Anderer Code soll nur ausgeführt werden, wenn die Pulslänge außerhalb von
           500-2600 (x2) ms liegt
        //und die Servoposition nicht gesetzt werden muss -> sorgt für bessere
          Geauigkeit beim PWM-Signal
        if(TCNT1 < 1000 || TCNT1 > 5200)
        {
            ////////////////////////////////
            // Manueller Modus (Joysticks) //
            ////////////////////////////////

            //Alle x ms die Pulsweite erhöhen/erniedrigen
            //Das Delta der Erhöhung/Erniedrigung wird über die Joystickauslenkung
              bestimmt (Servogeschwindigkeit)
            if(Serv_ms_count >= SERVO_ms_refresh)
            {
                Pul.pul_1 = joy_to_pulse(adcJ.joy_1_x, 1, Pul.pul_1, 0, &Led);
                Pul.pul_2 = joy_to_pulse(adcJ.joy_1_y, 2, Pul.pul_2, 1, &Led);
                Pul.pul_3 = joy_to_pulse(adcJ.joy_2_y, 3, Pul.pul_3, 0, &Led);
```

```
            Pul.pul_4 = joy_to_pulse(adcJ.joy_2_x, 4, Pul.pul_4, 0, &Led);
            Pul.pul_5 = joy_to_pulse(adcJ.joy_3_x, 5, Pul.pul_5, 0, &Led);

            //Info-LEDs anzeigen
            shiftr_out(shiftr_data(&Led));

            //Reset Counter
            Serv_ms_count = 0;
        }

        //Alle x ms die Joystick-Eingabe speichern
        if(Joy_ms_count >= JOY_ms_refresh)
        {
            //ADC-Signal vom Joystick von 0-1023
            //Joysick-Mittelstellung bei 511
            //Joystick x-Achsen: links 0-511, rechts 512-1023
            //Joystick y-Achsen: oben 512-1023, unten 0-511

            //Joystick 1:
            adcJ.joy_1_x = adc_read(0);
            adcJ.joy_1_y = adc_read(1);
            //Joystick 2:
            adcJ.joy_2_x = adc_read(2);
            adcJ.joy_2_y = adc_read(3);
            //Joystick 3:
            adcJ.joy_3_x = adc_read(4);

            //Reset Counter
            Joy_ms_count = 0;
        }

        ////////////////////////////////////////
        // Automatischer Modus (Koordinaten) //
        ////////////////////////////////////////

        //Wenn Koordinaten über USART erfolgreich eingelesen wurden
        if(usart_strg_compl == 1)
        {
            //Parsen des Strings und setzen der TCP-Koordinaten
            pars_rx_strg(usart_rx_buffer, &TCP);
            //Ausgabe der Koordinaten
            char text2[60];
            sprintf(text2, "Eingegebene Koordinaten: x=%.2f, y=%.2f, z=%.2f\n
              \r", TCP.x ,TCP.y ,TCP.z);
            usart_send_strg(text2);

            //Gelenkwinkel über inverse Kinematik berechnen
            if(!inv_kinematic(&TCP, &Ang))
            {
                //Wenn die Berechnung der Gelenkwinkel nicht möglich ist:

                //Gelenkwinkel und TCP auf Neutralstellung setzen
                //reset_ang(&Ang);
                //reset_tcp(&TCP);

                //Info-LEDs: Alle Rot
```

```
                    for(uint8_t i=0; i<5; i++)
                    {
                        Led.led_g[i] = 0;
                        Led.led_y[i] = 0;
                        Led.led_r[i] = 1;
                    }
                    //Warnmeldung senden
                    char text3[] = "Berechnung der Gelenkwinkel nicht moeglich!\n
                     \r";
                    usart_send_strg(text3);
                }
                else
                {
                    //TCP über Vorwärtskinematik berechnen und mit ursprünglichen
                     TCP vergleichen
                    if(!fwd_kinematic(&Ang, &P_Zero, &TCP_calc, 1))
                    {
                        //Wenn der TCP nicht über die Vorwärtskinematik
                     zurückgerechnet werden konnte:

                        //Gelenkwinkel und TCP auf Neutralstellung setzen
                        //reset_ang(&Ang);
                        //reset_tcp(&TCP);

                        //Info-LEDs: Alle Rot
                        for(uint8_t i=0; i<5; i++)
                        {
                            Led.led_g[i] = 0;
                            Led.led_y[i] = 0;
                            Led.led_r[i] = 1;
                        }
                        //Warnmeldung senden
                        char text4[] = "Berechnung der Vorwaertskinematik nicht
                     moeglich!\n\r";
                        usart_send_strg(text4);
                    }
                    else
                    {
                        //Vergleich mit Toleranz (0,01 mm)
                        double tol = 0.01;
                        if(TCP_calc.x < (TCP.x-tol) || TCP_calc.x > (TCP.x+tol) ||
                           TCP_calc.y < (TCP.y-tol) || TCP_calc.y > (TCP.y+tol) ||
                           TCP_calc.z < (TCP.z-tol) || TCP_calc.z > (TCP.z+tol))
                        {
                            //Wenn der vorgegebene TCP vom berechneten TCP
                     abweicht:

                            //Gelenkwinkel und TCP auf Neutralstellung setzen
                            //reset_ang(&Ang);
                            //reset_tcp(&TCP);

                            //Info-LEDs: Alle Rot
                            for(uint8_t i=0; i<5; i++)
                            {
                                Led.led_g[i] = 0;
                                Led.led_y[i] = 0;
```

```
                        Led.led_r[i] = 1;
                    }
                    //Warnmeldung senden
                    char text5[] = "Fehler bei der Berechnung der Vorwaertskinematik!\n\r";
                    usart_send_strg(text5);
                }
                else
                {
                    //Info-LEDs: Alle grün
                    for(uint8_t i=0; i<5; i++)
                    {
                        Led.led_g[i] = 1;
                        Led.led_y[i] = 0;
                        Led.led_r[i] = 0;
                    }

                    //Gelenkwinkel erfolgreich gesetzt
                    //grüne LED leuchten lassen
                    //Erfolgsmeldung ausgeben
                    char text6[120];
                    sprintf(text6, "Berechnete Gelenkwinkel: Phi_1=%.2f, Phi_21=%.2f, Phi_22=%.2f, Phi_31=%.2f, Phi_32=%.2f\n\r",
                    rad_in_grad(Ang.Phi_1),
                    rad_in_grad(Ang.Phi_21),
                    rad_in_grad(Ang.Phi_22),
                    rad_in_grad(Ang.Phi_31),
                    rad_in_grad(Ang.Phi_32));
                    usart_send_strg(text6);

                    //Ausgabe des berechneten TCPs über die Vorwärtskinematik
                    char text7[60];
                    sprintf(text7, "Berechneter TCP: x=%.2f, y=%.2f, z=%.2f\n\r",
                    TCP_calc.x,
                    TCP_calc.y,
                    TCP_calc.z);
                    usart_send_strg(text7);

                    //Gelenkwinkel in Pulsweite (*2) umrechnen
                    //und im Strut Pul speichern
                    set_pulse(&Ang, &Pul, 1, &Led);
                    //Erfolgsmeldung ausgeben
                    char text8[120];
                    sprintf(text8, "Berechnete Pulslängen: Pul_1=%i, Pul_2=%i, Pul_3=%i, Pul_4=%i, Pul_5=%i\n\r",
                    Pul.pul_1/2,
                    Pul.pul_2/2,
                    Pul.pul_3/2,
                    Pul.pul_4/2,
                    Pul.pul_5/2);
                    usart_send_strg(text8);
                }
            }
        }
```

```
                //Puffer für das Einlesen neuer Koordinaten vorbereiten
                usart_strg_compl = 0;
                //Info-LEDs anzeigen
                shiftr_out(shiftr_data(&Led));
                //Eingabeaufforderung für die nächste Eingabe
                usart_send_strg(text1);
            }
        }
    }
}

//////////////////////
// Interrupt Handler //
//////////////////////

//Timer 1 Handler für PWM-Signal
ISR(TIMER1_COMPA_vect)
{
    //Alle Pins an Port B auf HIGH setzen
    //Servos werden synchron angesteuert
    PORTB = 0xFF;
}

//Timer 0 Handler für die Abtastrate der Joysticks und die Servogeschwindigkeit
  (Interrupt jede ms)
ISR(TIMER0_COMPA_vect)
{
    Joy_ms_count++;
    Serv_ms_count++;
}

//USART Handler: USART Rx Complete (0x0024)
ISR(USART_RX_vect)
{
    //Übergabe der x-, y- und z-Koordinaten per Eingabe am seriellen Monitor (x, y,
       z)
    //Zeichen werden im C-String "usart_rx_buffer" abgelegt
    //wenn alle Zeichen des Strings in den Puffer geladen wurden,
    //wird die Variable usart_strg_complete auf 1 gesetzt
    uint8_t nextChar;
    //Zeichen für Zeichen einlesen
    nextChar = UDR0;
    if(usart_strg_compl == 0)
    {
        if(nextChar != '\n' && nextChar != '\r' && usart_strg_count < USART_RX_MAX)

        {
            usart_rx_buffer[usart_strg_count] = nextChar;
            usart_strg_count++;
        }
        else
        {
            usart_rx_buffer[usart_strg_count] = '\0';
            usart_strg_count = 0;
            usart_strg_compl = 1;
```

```
        }
    }
}
```

```
#ifndef TYPEDEF_H
#define TYPEDEF_H

//Typedef für Vektoren im 3D-Raum
typedef struct vector
{
    double x;
    double y;
    double z;
} vec;

//Typedef für Punkte im 3D-Raum
typedef struct point
{
    double x;
    double y;
    double z;
} dot;

//Typedef für 4x4 Matrizen
typedef double matr[4][4];

//Typedef für Struct, welches die Gelenkwinkel enthält
//Angabe in Rad
//_21/_22 bzw. _31/_32 beziehen sich auf zwei Möglichkeiten
//der Gelenkstellungen, um den TCP zu erreichen
typedef struct angles
{
    double Phi_1;
    double Phi_21;
    double Phi_22;
    double Phi_31;
    double Phi_32;
    double Phi_4;
    double Phi_5;
} ang;

//Typedef für Struct, dass die Pulsweiten für die Servos enthält
//Pulsweiten in µs müssen als Eintrag in TCNT1 x 2 genommen werden
typedef struct pulsewidth
{
    uint16_t pul_1;
    uint16_t pul_2;
    uint16_t pul_3;
    uint16_t pul_4;
    uint16_t pul_5;
} pul;

//Typedef für Struct, dass die analogen Werte für die drei Joysticks enthält
typedef struct ADC_joystick
{
    uint16_t joy_1_x;
    uint16_t joy_1_y;
    uint16_t joy_2_x;
    uint16_t joy_2_y;
    uint16_t joy_3_x;
```

```
    uint16_t joy_3_y;
} adcj;

//Typedef für Struct, dass den Zustand der Info-LEDs enthält (0 oder 1)
typedef struct LED_info
{
    uint8_t led_g[5];
    uint8_t led_y[5];
    uint8_t led_r[5];
} led;

#endif
```

```
#ifndef CONST_H
#define CONST_H

#include <avr/io.h>

//Deklaration von globalen Konstanten

//Pi
extern const double PI;

//Längen der Glieder [mm]
extern const double L_1;
extern const double L_2;
extern const double L_3;

//Servospezifische Konstanten
//0: Servo-Pin am Arduino (PORTB)
//1: P_0 = Pulslänge bei Winkel 0 (Neutralstellung)
//2: P_-90 = Pulslänge bei -90 Grad
//3: P_Min = Minimale Pulslänge für den Servo
//4: P_Max = Maximale Pulslänge für den Servo
extern const uint16_t SERVO_const[5][5];
//Aktualisierungsrate der Servoposition in ms
extern const uint8_t SERVO_ms_refresh;

//Joystickspezifische Konstanten
//Joystick Deadzone (für alle Joysticks/Achsen gleich)
//Gilt in +- x- und y-Richtung, ca. 5% der Auslenkung in jede Richtung
  (512x0,05=25,6)
//Verhindert automatisches Laufen der Servos, wenn die Mittelstellung der Joysticks
   nicht genau bei 512 ist
extern const uint8_t JOY_dz;
//Joystick Mittelstellung
//10 bit = (1024/2) - 1 = 511
extern const uint16_t JOY_mid;
//Testkonstante für die Servogeschwindigkeit
extern const uint8_t JOY_div;
//Abtastrate der Joysticks in ms
extern const uint8_t JOY_ms_refresh;

#endif
```

```c
#include "const.h"

//Definition von globalen Konstanten

//Pi
const double PI = 3.14159265;

//Längen der Glieder [mm]
const double L_1 = 160.0;
const double L_2 = 90.0;
const double L_3 = 190.0;

//Servospezifische Konstanten
//0: Servo-Pin am Arduino (PORTB)
//1: P_0 = Pulslänge bei Winkel 0 (Neutralstellung)
//2: P_-90 = Pulslänge bei -90° (leichter zu mesen als +90°)
//3: P_Min = Minimale Pulslänge für den Servo
//4: P_Max = Maximale Pulslänge für den Servo
//Pulslängen in µs, für Timer 1 x 2 nehmen
const uint16_t SERVO_const[5][5] =
{
    //Servo G1
    {PINB0, 1573, 590, 590, 2496},
    //Servo G2
    {PINB1, 1554, 2446, 1030, 2446},
    //Servo G3
    {PINB2, 1546, 580, 580, 2496},
    //Servo G4
    {PINB3, 2300, 1146, 600, 2340},
    //Servo G5
    {PINB4, 2026, 976, 1615, 2066}
};
//Aktualisierungsrate der Servoposition in ms (24 ms)
const uint8_t SERVO_ms_refresh = 23;

//Joystickspezifische Konstanten
//Joystick Deadzone (für alle Joysticks/Achsen gleich)
//Gilt in +- x- und y-Richtung, ca. 5% der Auslenkung in jede Richtung (512 x 0,05 ↵
  = 25,6)
//Verhindert automatisches Laufen der Servos, wenn die Mittelstellung der Joysticks↵
   nicht genau bei 512 ist
const uint8_t JOY_dz = 25;
//Joystick Mittelstellung
//10 bit = (1024/2) - 1 = 511
const uint16_t JOY_mid = 511;
//Testkonstante für die Servogeschwindigkeit
//Je höher, desto langsamer die Servos
const uint8_t JOY_div = 80;
//Abtastrate der Joysticks in ms (12 ms)
const uint8_t JOY_ms_refresh = 11;
```

```
#ifndef VAR_H
#define VAR_H

//Defines
#define USART_RX_MAX 100

//Includes
#include "typedef.h"

//Deklaration von globalen Variablen

//Tool Center Point = gewünschte Effektorposition
dot TCP;
//Gelenkwinkel des Roboters
ang Ang;
//(modifizierte) Pulsweite für die Servos
pul Pul;
//ADC-Werte der drei Joysticks (x- und y-Achsen)
volatile adcj adcJ;
//berechneter TCP über Vorwärtskinematik
dot TCP_calc;
//Nullpunkt des Weltkoordinatensystems
dot P_Zero;
//USART Empfangspuffer für die Koordinaten
char usart_rx_buffer[USART_RX_MAX + 1];
//Variable gibt an, ob der String mit den Koordinaten über USART komplett empfangen
   wurde
//1 = komplett, 0 = noch nicht komplett
volatile uint8_t usart_strg_compl;
//Zähler für die Anzahl an übergebenen Zeichen über USART
volatile uint8_t usart_strg_count;
//Zähler für ms für die Abtastrate der Joysticks (Timer 0)
volatile uint8_t Joy_ms_count;
//Zähler für ms für die Aktualisierungsrate der Servoposition (Timer 0)
volatile uint8_t Serv_ms_count;
//Struct für den Zustand der Info-LEDs
led Led;

#endif
```

```c
#include "var.h"

//Definition von globalen Variablen

//Tool Center Point = gewünschte Effektorposition
dot TCP = {0, 0, 0};
//Gelenkwinkel des Roboters
ang Ang = {0, 0, 0, 0, 0, 0, 0};
//(modifizierte) Pulsweite für die Servos
pul Pul = {0, 0, 0, 0, 0};
//ADC-Werte der drei Joysticks (x- und y-Achsen)
adcj adcJ = {0, 0, 0, 0, 0, 0};
//berechneter TCP über Vorwärtskinematik
dot TCP_calc = {0, 0, 0};
//Nullpunkt des Weltkoordinatensystems
dot P_Zero = {0, 0, 0};
//USART Empfangspuffer für die Koordinaten
char usart_rx_buffer[USART_RX_MAX + 1] = "";
//Variable gibt an, ob der String mit den Koordinaten über USART komplett empfangen
   wurde
//1 = komplett, 0 = noch nicht komplett
uint8_t usart_strg_compl = 0;
//Zähler für die Anzahl an übergebenen Zeichen über USART
uint8_t usart_strg_count = 0;
//Zähler für ms für die Abtastrate der Joysticks (Timer 0)
uint8_t Joy_ms_count = 0;
//Zähler für ms für die Aktualisierungsrate der Servoposition (Timer 0)
uint8_t Serv_ms_count = 0;
//Struct für den Zustand der Info-LEDs
led Led = {{1, 1, 1, 1, 1}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}};
```

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

//Defines für USART
#define F_CPU 16000000UL
#define BUAD 9600
#define UBRR ((F_CPU/16/BUAD) - 1)
//Defines für Schieberegister (Port und Pins)
//Zwei Schieberegister hintereinander geschaltet (16 Bit)
#define HC595_PORT      PORTD
#define HC595_DDR       DDRD
#define HC595_DS_POS    PD5     //Data pin (DS oder SER)
#define HC595_SH_CP_POS PD7     //Shift Clock Pin (SH_CP oder SRCLK)
#define HC595_ST_CP_POS PD6     //Store Clock Pin (ST_CP oder RCLK)
#define SHIFTR_BITS     16      //Anzahl der Bits in beiden Schieberegistern

//Includes
#include <avr/io.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "typedef.h"
#include "const.h"

//Funktionsdeklarationen

//Umrechnung von Rad in Grad
double rad_in_grad(double rad);
double grad_in_rad(double grad);
//Arduino map() Funktion für double-Variablen
double mapd(double val, double in_min, double in_max, double out_min, double
  out_max);

//USART
//USART initialisieren
void usart_init();
//Einzelnes Zeichen über USART empfangen (wird nicht benötigt)
uint8_t usart_rec_char(void);
//String über USART empfangen und in Puffer ablegen (wird nicht benötigt)
void usart_rec_strg(char *Buffer, uint8_t MaxLen);
//Einzelnes Zeichen über USART an PC senden
void usart_send_char(char ch);
//String über USART an PC senden
void usart_send_strg(char *strg);
//Parsen des gesendeten Strings vom PC (Delimiter: ,)
//Koordinaten müssen in der Form: x,y,z gesendet werden
//Umwandeln der Substrings in Double-Variablen und speichern im TCP-Struct
void pars_rx_strg(char *strg, dot *TCP);

//Timer 1 für PWM-Signal
void pwm_init();

//ADC initialisieren (Joysticks)
void adc_init();
//ADC auslesen
```

```
uint16_t adc_read(uint8_t ch);

//Timer 0 für die Abtastfrequenz der Joysticks und die Servogeschwindigkeit
void ms_init();

//Vektorrechnungen
//Betrag eines Vektors berechnen
double vec_betrag(vec *v);
//Skalarprodukt von zwei Vektoren berechnen
double vec_skalarprod(vec *v1, vec *v2);
//Zwei Vektoren subtrahieren -> Vektor
vec vec_subtract(vec *v1, vec *v2);
//Ortsvektoren zweier Punkte subtrahieren -> Vektor
vec dot_subtract(dot *p1, dot *p2);

//Matrizenrechnungen
//4x4 Matrizen multiplizieren
//Übergeben werden Matrix 1 und 2
//Das Ergebnis der Multiplikation wird in Matrix 3 geschrieben
void matr_x_matr(matr m_src1, matr m_src2, matr m_dst);
//4x4 Matrix mit Punkt multiplizieren
//p_src = Punkt für die Multiplikation
//p_dst = Ergebnis der Multiplikation
//Punkte werden der Einfachheit halber nicht als homogene Koordinaten dargestellt
  (=4. Variable h)
//h ist in aller Regel = 1, ausser die Matrix ist eine Projektionsmatrix (für
  perspektivische Darstellung)
//Dann müssen die x,y,z-Koordinaten durch h geteilt werden
void matr_x_dot(matr m, dot *p_src, dot *p_dst);

//Kinematik
//Funktion für die inverse Kinematik
//Übergeben wird der gewünschte TCP
//Die berechneten Gelenkwinkel werden im globalen Struct Ang gespeichert
//Können die Winkel nicht berechnet werden, gibt die Funktion 0 zurück, ansonsten 1
uint8_t inv_kinematic(dot *TCP, ang *Ang);
//Funktion für die Vorwärtskinematik
//Übergeben wird das Struct Ang mit den Gelenkwinkeln,
//die Transformationsmatrix und der Punkt, der transformiert werden soll
//Ausserdem der Punkt, in dem die Transformation gespeichert wird
//Der letzte Parameter x gibt an, ob Gelenkstellung 1 oder 2 verwendet werden soll
//Kann der Punkt nicht berechnet werden, gibt die Funktion 0 zurück, ansonsten 1
uint8_t fwd_kinematic(ang *Ang, dot *p_src, dot *p_dst, uint8_t x);

//Berechnung von Pulslängen
uint16_t ang_to_puls(uint8_t s_nr, double ang, led *Led);
//Pulslängen in Struct Pul speichern
//Der letzte Parameter x gibt an, ob Gelenkstellung 1 oder 2 verwendet werden soll
void set_pulse(ang *Ang, pul *Pul, uint8_t x, led *Led);
//Wandelt die Joystickeingaben in eine Pulsweite um
//j_val = analoger Wert der Joystickachse (1-1024)
//s_nr = Servonummer (1-5)
//s_puls_alt = aktueller Servopuls (µs*2)
//s_dir = gewünschte Servorichtung (1,0)
//led *Led = Referenz zum Struct für die Info-LEDs
uint16_t joy_to_pulse(uint16_t j_val, uint8_t s_nr, uint16_t s_puls_alt, uint8_t
```

```
  s_dir, led *Led);

//Gelenkwinkel auf die Neutralstellung setzen
void reset_ang(ang *Ang);
//TCP auf die Neutralstellung setzen
void reset_tcp(dot *TCP);

//Funktionen für die Schieberegister (2xHC595)
//HC595 initialisieren
void shiftr_init();
//Puls auf Shift Clock Pin (SH_CP oder SRCLK)
void shiftr_shift();
//Puls auf Store Clock Pin (ST_CP oder RCLK)
void shiftr_latch();
//Zustände der Info-LEDs aus dem Struct "Led" Bit für Bit in uint16_t Variable
  schreiben
//Auffüllen von Rechts
//Bit 16 (links): ungenutzt, immer 0
uint16_t shiftr_data(led *Led);
//Funktion gibt den Zustand der Info-LEDs über zwei Schieberegister (= 16 Byte) aus
//Da nur 15 LEDs vorhanden sind, ist das letzte Byte immer 0
void shiftr_out(uint16_t data);

#endif
```

```c
#include "functions.h"

//Funktionsdefinitionen

//Umrechnung von Rad in Grad
double rad_in_grad(double rad)
{
    return rad * (180.0 / PI);
}
double grad_in_rad(double grad)
{
    return grad * (PI / 180.0);
}
//Arduino map() Funktion für double-Variablen
double mapd(double val, double in_min, double in_max, double out_min, double
  out_max)
{
    return (val - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

//USART
//USART initialisieren
void usart_init()
{
    //UBRR0:
    //Baudrate von 9600 bps im Register UBRR0H und UBRR0L festlegen (16 bit)
    //Register UBRR0H muss vor dem Register UBRR0L beschrieben werden!
    UBRR0H = (UBRR>>8);
    UBRR0L = (UBRR);
    //UCSR0B: Transmitter (TXEN0)/Receiver (RXEN0) enable
    //RXCIE0: Interrupts aktivieren, wenn empfangenes Byte bereit steht
    UCSR0B = (1<<RXCIE0)|(1<<RXEN0)|(1<<TXEN0);
    //UCSR0C:
    //USARD Mode Select: UMSEL01 und UMSEL00 = 0 -> Asynchronous Mode
    //Parity Mode: UPM01 und UPM00 = 0 -> Disabled (kein Parity Bit)
    //Stop Bit: USBS0 = 0 -> Ein Stop Bit
    //Character Size: UCSZ02 = 0, UCSZ01 = 1, UCSZ00 = 1 -> 8-bit
    UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
}
//Einzelnes Zeichen über USART empfangen
uint8_t usart_rec_char(void)
{
    //Warten bis das USART data register bereit/leer ist
    while(!(UCSR0A & (1<<UDRE0))) {};
    //Empfangenes Zeichen als 8 bit Char zurückgeben
    return UDR0;
}
//String über USART empfangen und in Puffer ablegen
void usart_rec_strg(char *Buffer, uint8_t MaxLen)
{
    uint8_t NextChar;
    uint8_t StringLen = 0;
    //Auf nächstes Zeichen warten und empfangen
    NextChar = usart_rec_char();
    //Solange Zeichen in den Puffer schreiben, bis
    //entweder der String aus ist (\n)
```

```
    //oder der Puffer voll ist
    while(NextChar != '\n' && NextChar != '\r' && StringLen < MaxLen)
    {
        *Buffer++ = NextChar;
        StringLen++;
        NextChar = usart_rec_char();
    }
    //'\0' für C-String anhängen
    *Buffer = '\0';
}
//Einzelnes Zeichen über USART senden
void usart_send_char(char ch)
{
    //Warten bis das USART data register bereit/leer ist
    while(!(UCSR0A & (1<<UDRE0))) {}
    //Zeichen senden
    UDR0 = ch;
}
//String über USART an PC senden
void usart_send_strg(char *strg)
{
    while(*strg)
    {
        usart_send_char(*strg);
        strg++;
    }

    /*
    uint8_t i = 0;
    while(strg[i] != 0)
    {
        usart_send_char(strg[i]);
    }
    */
}
//Parsen des gesendeten Strings vom PC (Delimiter: ,)
//Koordinaten müssen in der Form: x,y,z gesendet werden
//Umwandeln der Substrings in Double-Variablen und  im TCP-Struct speichern
void pars_rx_strg(char *strg, dot *TCP)
{
    //String nach Delimiter auftrennen
    char *substring[3];
    char delimit[] = ",";
    uint8_t i=0;
    substring[i] = strtok(strg, delimit);
    while(substring[i] != NULL)
    {
        i++;
        substring[i] = strtok(NULL, delimit);
    }
    //Substrings in Double umwandeln
    //und x, y und z Werte im TCP-Struct speichern
    //Rückgabe von 0.0, wenn keine Umwandlung möglich ist
    TCP->x = atof(substring[0]);
    TCP->y = atof(substring[1]);
    TCP->z = atof(substring[2]);
```

```
}

//Timer 1 für PWM-Signal
void pwm_init()
{
    //Alle Pins an Port B für die Servos auf Output stellen
    DDRB = 0xFF;
    //Register setzen
    //WGM11, 12 und 13: Mode 14
    //CS11: Prescaler 8 -> 40.000 clks für 20 ms
    //Output Compare A Match Interrupt Enable
    TCCR1A = (1<<WGM11);
    TCCR1B = (1<<WGM12) | (1<<WGM13) | (1<<CS11);
    TIMSK1 = (1<<OCIE1A);
    //ICR1: Wert entspricht 20 ms = eine Periode
    ICR1 = 39999;
    //40000 = 20 ms
    //2000 = 1 ms = 1000 µs
    //gewünschte Pulslänge x 2 = Wert in TCNT1
    //Pulslänge von 500 µs = TCNT1 1000
    //Pulslänge von 1000 µs = TCNT1 2000
    //Pulslänge von 1500 µs (Mittelstellung) = TCNT1 3000
    //Pulslänge von 2000 µs = TCNT1 4000
    //Pulslänge von 2500 µs = TCNT1 5000
}

//ADC für die Joysticks
void adc_init()
{
    //ADCSRA: ADC Control and Status Register A
    //ADEN=1: ADC aktivieren
    //ADPS0-2: Prescalar (ADPS2=1, ADPS1=1, ADPS0=1 -> 128 Teiler)
    //Benötigte Frequenz: 50-200kHz; 16.000.000/128 = 125.000 (einzig möglicher
      Teiler, ansonsten > 200kHz)
    ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
    //ADMUX: ADC Multiplexer Selection Register
    //REFS1=0 und REFS0=1: AVcc (with external capacitor at AREF pin)
    ADMUX = (1<<REFS0);
}
//ADC auslesen
//Parameter ist der Kanal, der ausgelesen weden soll (PC0-PC5)
uint16_t adc_read(uint8_t ch)
{
    //7 mögliche Kanäle
    ch &= 0b00000111;
    ADMUX = (ADMUX & 0xF8)|ch;
    //Single convertion starten (1 in ADSC)
    ADCSRA |= (1<<ADSC);
    //Warten, dass die Konvertierung abgeschlossen ist (ADSC wird wieder 0)
    while(ADCSRA & (1<<ADSC));

    return (ADC);
}

//Timer 0 für die Abtastfrequenz der Joysticks und die Servogeschwindigkeit
void ms_init()
```

```
{
    //Abtastrate soll 1000 Hz btragen (alle 1 ms)
    //CPU Frequenz = 16.000.000 Hz / Teiler 64 = 250.000 [s]
    //1 ms entspricht also 250

    //Timer in CTC Mode -> vergleicht mit OCR0A
    //CTC-Mode -> In TCCR0A: WGM00=0, WGM01=1, In TCCR0B: WGM02=0
    TCCR0A = (1<<WGM01);
    //Teiler: clk/64 -> In TCCR0B: CS02=0, CS01=1, CS00=1
    TCCR0B = (1<<CS00)|(1<<CS01);
    //Vergleichsregister OCR0A auf msTimer = 249 = 1 ms setzen
    OCR0A = 0xF9;
    //Interrupt wenn OCR0A erreicht ist -> In TIMSK0: OCIE0A=1
    TIMSK0 = (1<<OCIE0A);

    // initialize counter
    //TCNT0 = 0 ;
}

//Vektorrechnungen
//Betrag eines Vektors berechnen
double vec_betrag(vec *v)
{
    return sqrt((v->x * v->x) + (v->y * v->y) + (v->z * v->z));
}
//Skalarprodukt von zwei Vektoren berechnen
double vec_skalarprod(vec *v1, vec *v2)
{
    return (v1->x * v2->x) + (v1->y * v2->y) + (v1->z * v2->z);;
}
//Zwei Vektoren subtrahieren -> Vektor
vec vec_subtract(vec *v1, vec *v2)
{
    vec v3 = {v1->x - v2->x, v1->y - v2->y, v1->z - v2->z};
    return v3;
}
//Ortsvektoren zweier Punkte subtrahieren -> Vektor
vec dot_subtract(dot *p1, dot *p2)
{
    vec v = {p1->x - p2->x, p1->y - p2->y, p1->z - p2->z};
    return v;
}

//Matrizenrechnungen
//4x4 Matrizen multiplizieren
//Übergeben werden Matrix 1 und 2
//Das Ergebnis der Multiplikation wird in Matrix 3 geschrieben
void matr_x_matr(matr m_src1, matr m_src2, matr m_dst)
{
    uint8_t i, j;
    for(i = 0; i < 4; ++i)
    {
        for(j = 0; j < 4; ++j)
        {
            m_dst[i][j] =
            m_src1[i][0] * m_src2[0][j] +
```

```
            m_src1[i][1] * m_src2[1][j] +
            m_src1[i][2] * m_src2[2][j] +
            m_src1[i][3] * m_src2[3][j];
        }
    }
}
//4x4 Matrix mit Punkt multiplizieren
//p_src = Punkt für die Multiplikation
//p_dst = Ergebnis der Multiplikation
//Punkte werden der Einfachheit halber nicht als homogene Koordinaten dargestellt (= 4. Variable h)
//h ist in aller Regel = 1, ausser die Matrix ist eine Projektionsmatrix (für perspektivische Darstellung)
//Dann müssen die x,y,z-Koordinaten durch h geteilt werden
void matr_x_dot(matr m, dot *p_src, dot *p_dst)
{
    //COLUMN-MAJOR Variante
    p_dst->x = m[0][0] * p_src->x + m[0][1] * p_src->y + m[0][2] * p_src->z + m[0][3];
    p_dst->y = m[1][0] * p_src->x + m[1][1] * p_src->y + m[1][2] * p_src->z + m[1][3];
    p_dst->z = m[2][0] * p_src->x + m[2][1] * p_src->y + m[2][2] * p_src->z + m[2][3];
}

//Kinematik
//Funktion für die inverse Kinematik
//Übergeben wird der gewünschte TCP
//Die berechneten Gelenkwinkel werden im globalen Struct Ang gespeichert
//Können die Winkel nicht berechnet werden, gibt die Funktion 0 zurück, ansonsten 1
uint8_t inv_kinematic(dot *TCP, ang *Ang)
{
    //x und y des TCP dürfen nicht beide 0 sein
    if(TCP->x == 0 && TCP->y == 0)
    {
        return 0;
    }
    else
    {
        ////////////////////////
        // Berechnung von Phi_1 //
        ////////////////////////

        //Vektor mit den xy-Koordinaten des TCP erzeugen (vom Ursprung aus)
        vec V_TCPxy = {TCP->x, TCP->y, 0};
        //Betrag des Vektors berechnen (Hypotenuse)
        double L_TCPxy = vec_betrag(&V_TCPxy);
        //Phi1 berechnen (Kosinus)
        Ang->Phi_1 = acos(TCP->x/L_TCPxy);
        //Vorzeichen des Winkels bestimmen
        if(TCP->y < 0)
        {
            Ang->Phi_1 *= -1.0;
        }

        ////////////////////////
```

```
        // Berechnung von Phi_2 //
        //////////////////////////

        //Koordinaten des Punktes G2
        //Unbeweglich, abhängig von L1
        dot P_G2 = {0, 0, L_1};
        //Vektor G2_TCP berechnen
        vec V_G2_TCP = dot_subtract(TCP, &P_G2);
        //Betrag von V_G2_TCP bestimmen
        double L_G2_TCP = vec_betrag(&V_G2_TCP);
        //Wenn L_G2_TCP < L_2+L_3 ist, dann kann das Ziel nicht erreicht werden
        if(L_G2_TCP > (L_2 + L_3))
        {
            return 0;
        }
        else
        {
            //Vektor parallel zur z-Achse
            vec V_z = {0, 0, -1.0};
            //Hilfswinkel Phi_2a berechnen = Winkel zwischen V_G2_TCP und V_z
            //Immer nur der spitze Winkel wird berechnet
            double Phi_2a = acos(vec_skalarprod(&V_G2_TCP, &V_z)/L_G2_TCP);

            //Hilfswinkel Phi_2b über den Kosinussatz berechnen
            double Phi_2b = acos((L_2*L_2 + L_G2_TCP*L_G2_TCP - L_3*L_3)/(2.0 * L_2
               * L_G2_TCP));
            //Phi_21 und Phi_22 berechnen (2 Möglichkeiten für die Gelenkstellung)
            Ang->Phi_21 = Phi_2a + Phi_2b;
            Ang->Phi_22 = Phi_2a - Phi_2b;
            //Phi_2a und Phi_2a auf die Neutralstellung des Roboters beziehen
            Ang->Phi_21 = (Ang->Phi_21 - PI/2.0) * -1.0;
            Ang->Phi_22 = (Ang->Phi_22 - PI/2.0) * -1.0;

            //////////////////////////
            // Berechnung von Phi_3 //
            //////////////////////////

            //Phi_31 berechnen (Kosinussatz)
            Ang->Phi_31 = acos((L_3*L_3 + L_2*L_2 - L_G2_TCP*L_G2_TCP)/(2.0 * L_3 *
               L_2));
            //Phi_31 und Phi_32 auf die Neutralstellung des Roboters beziehen
            Ang->Phi_31 = PI - Ang->Phi_31;
            Ang->Phi_32 = Ang->Phi_31 * -1.0;

            //Winkel auf unendliche Zahlen oder NAN prüfen
            if(isnan(Ang->Phi_1) || isnan(Ang->Phi_21) || isnan(Ang->Phi_22) ||
              isnan(Ang->Phi_31) ||
               isinf(Ang->Phi_1) || isinf(Ang->Phi_21) || isinf(Ang->Phi_22) ||
                 isinf(Ang->Phi_31))
            {
                return 0;
            }
        }
    }
    return 1;
}
```

```
//Funktion für die Vorwärtskinematik
//Übergeben wird das Struct Ang mit den Gelenkwinkeln,
//die Transformationsmatrix und der Punkt, der transformiert werden soll
//Ausserdem der Punkt, in dem die Transformation gespeichert wird
//Der letzte Parameter x gibt an, ob Gelenkstellung 1 oder 2 verwendet werden soll
//Kann der Punkt nicht berechnet werden, gibt die Funktion 0 zurück, ansonsten 1
uint8_t fwd_kinematic(ang *Ang, dot *p_src, dot *p_dst, uint8_t x)
{
    //Winkel setzen
    double Phi_1, Phi_2, Phi_3, Phi_4;
    Phi_1 = Ang->Phi_1;
    if(x == 1)
    {
        Phi_2 = Ang->Phi_21;
        Phi_3 = Ang->Phi_31;
    }
    else
    {
        Phi_2 = Ang->Phi_22;
        Phi_3 = Ang->Phi_32;
    }
    Phi_4 = Ang->Phi_4;

    //Transformationsmatrizen
    matr M01 = {{cos(Phi_1),    0,      -sin(Phi_1),    0},
                {sin(Phi_1),    0,       cos(Phi_1),    0},
                {0,            -1.0,     0,             L_1},
                {0,             0,       0,             1.0}};

    matr M12 = {{cos(Phi_2),    -sin(Phi_2),    0,      L_2*cos(Phi_2)},
                {sin(Phi_2),     cos(Phi_2),    0,      L_2*sin(Phi_2)},
                {0,              0,             1.0,    0},
                {0,              0,             0,      1.0}};

    matr M23 = {{cos(Phi_3+PI/2),    0,      sin(Phi_3+PI/2),      0},
                {sin(Phi_3+PI/2),    0,     -cos(Phi_3+PI/2),      0},
                {0,                             1.0,    0,         0},
                {0,                             0,      0,         1.0}};

    matr M34 = {{cos(Phi_4),    -sin(Phi_4),    0,      0},
                {sin(Phi_4),     cos(Phi_4),    0,      0},
                {0,              0,             1.0,    L_3},
                {0,              0,             0,      1.0}};

    //Rotationsmatrizen multiplizieren
    //Überladung von Operatoren in C nicht möglich :-(
    matr M02, M03, M04;
    matr_x_matr(M01, M12, M02);
    matr_x_matr(M02, M23, M03);
    matr_x_matr(M03, M34, M04);

    //Gesamtmatrix mit übergebenem Punk multiplizieren
    matr_x_dot(M04, p_src, p_dst);

    //Koordinaten des transformierten Punktes auf inf und nan testen
    if(isnan(p_dst->x) || isnan(p_dst->y) || isnan(p_dst->z) ||
```

```
       isinf(p_dst->x) || isinf(p_dst->y) || isinf(p_dst->z))
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

//Umrechnung der Gelenkwinkel [rad] in Pulslängen [µs*2]
//Übergabe der Servonummer (1-5) und des Winkels [Rad]
uint16_t ang_to_puls(uint8_t s_nr, double ang, led *Led)
{
    //Geradengleichung mit den Servo-Konstanten:
    //Puls y = ((Puls(0°)-Puls(-90°))/(Pi/2)) * Winkel x + Puls(0°)
    double pulswidth = ((((double)SERVO_const[(s_nr-1)][1] - SERVO_const[(s_nr-1)]
      [2])/(PI/2)) * ang) + SERVO_const[(s_nr-1)][1];
    //Bewegungsfreiheit als Anschlagsbremse einschränken: MIN_MAX_PULSE
    if(pulswidth >= SERVO_const[(s_nr-1)][4])
    {
        pulswidth = (double)SERVO_const[(s_nr-1)][4];
        //Info-LEDs: Gelb
        Led->led_g[(s_nr-1)] = 0;
        Led->led_y[(s_nr-1)] = 1;
        Led->led_r[(s_nr-1)] = 0;
    }
    else if(pulswidth <= SERVO_const[(s_nr-1)][3])
    {
        pulswidth = (double)SERVO_const[(s_nr-1)][3];
        //Info-LEDs: Gelb
        Led->led_g[(s_nr-1)] = 0;
        Led->led_y[(s_nr-1)] = 1;
        Led->led_r[(s_nr-1)] = 0;
    }
    else
    {
        //Info-LEDs: Grün
        Led->led_g[(s_nr-1)] = 1;
        Led->led_y[(s_nr-1)] = 0;
        Led->led_r[(s_nr-1)] = 0;
    }
    //Errechnete Pulsweiten in µs müssen für den Eintrag in TCNT1 mit 2
      multipliziert werden
    pulswidth *= 2.0;
    //Pulsweite runden (Umwandlung in int)
    return (uint16_t)(pulswidth + 0.5);
}

//Pulslängen in Struct Pul speichern
//Der letzte Parameter x gibt an, ob Gelenkstellung 1 oder 2 verwendet werden soll
void set_pulse(ang *Ang, pul *Pul, uint8_t x, led *Led)
{
    Pul->pul_1 = ang_to_puls(1, Ang->Phi_1, Led);
    if(x == 1)
    {
```

```
        Pul->pul_2 = ang_to_puls(2, Ang->Phi_21, Led);
        Pul->pul_3 = ang_to_puls(3, Ang->Phi_31, Led);
    }
    else
    {
        Pul->pul_2 = ang_to_puls(2, Ang->Phi_22, Led);
        Pul->pul_3 = ang_to_puls(3, Ang->Phi_32, Led);
    }
    Pul->pul_4 = ang_to_puls(4, Ang->Phi_4, Led);
    Pul->pul_5 = ang_to_puls(5, Ang->Phi_5, Led);
}

//Wandelt die Joystickeingaben in eine Pulsweite um
//j_val = analoger Wert der Joystickachse (1-1024)
//s_nr = Servonummer (1-5)
//s_puls_alt = aktueller Servopuls (µs*2)
//s_dir = gewünschte Servorichtung (1,0)
//led *Led = Referenz zum Struct für die Info-LEDs
uint16_t joy_to_pulse(uint16_t j_val, uint8_t s_nr, uint16_t s_puls_alt, uint8_t
  s_dir, led *Led)
{
    //Rückgabewert
    uint16_t s_puls_neu;
    //Die Pulsänderung über die Joystickeingabe berechnen
    uint16_t dpulse;
    if(j_val > (JOY_mid + JOY_dz))
    {
        //Quadratische Funktion, da eine lineare Funktion zu schnell zu einer hohen
           Servogeschwindigkeit führt
        //Die höchste Pulsänderung sollte bei ca 40 liegen
        dpulse = pow(((j_val - JOY_mid) / JOY_div), 2);
        //Die Pulsänderung zur aktuellen Pulsweite addieren bzw subtrahieren
          (Servorichtung)
        if(s_dir == 1)
        {s_puls_neu = s_puls_alt + dpulse;}
        else
        {s_puls_neu = s_puls_alt - dpulse;}
    }
    else if(j_val < (JOY_mid - JOY_dz))
    {
        dpulse = pow(((JOY_mid - j_val) / JOY_div), 2);
        //Die Pulsänderung zur aktuellen Pulsweite addieren bzw subtrahieren
          (Servorichtung)
        if(s_dir == 1)
        {s_puls_neu = s_puls_alt - dpulse;}
        else
        {s_puls_neu = s_puls_alt + dpulse;}
    }
    else
    {
        s_puls_neu = s_puls_alt;
    }
    //Die Bewegungsfreiheit der Servos einschränken
    //und Info-LEDs setzen (Servoanschlag)
    if(s_puls_neu >= (SERVO_const[(s_nr-1)][4]*2))
    {
```

```
        s_puls_neu = SERVO_const[(s_nr-1)][4]*2;
        //Info-LEDs: Gelb
        Led->led_g[(s_nr-1)] = 0;
        Led->led_y[(s_nr-1)] = 1;
        Led->led_r[(s_nr-1)] = 0;
    }
    else if(s_puls_neu <= (SERVO_const[(s_nr-1)][3]*2))
    {
        s_puls_neu = SERVO_const[(s_nr-1)][3]*2;
        //Info-LEDs: Gelb
        Led->led_g[(s_nr-1)] = 0;
        Led->led_y[(s_nr-1)] = 1;
        Led->led_r[(s_nr-1)] = 0;
    }
    else
    {
        //Info-LEDs: Grün
        Led->led_g[(s_nr-1)] = 1;
        Led->led_y[(s_nr-1)] = 0;
        Led->led_r[(s_nr-1)] = 0;
    }
    return s_puls_neu;
}

//Gelenkwinkel auf die Neutralstellung setzen
void reset_ang(ang *Ang)
{
    Ang->Phi_1 = 0;
    Ang->Phi_21 = 0;
    Ang->Phi_22 = 0;
    Ang->Phi_31 = 0;
    Ang->Phi_32 = 0;
    Ang->Phi_4 = 0;
    Ang->Phi_5 = 0;
}

//TCP auf die Neutralstellung setzen
void reset_tcp(dot *TCP)
{
    TCP->x = L_1+L_2;
    TCP->y = 0;
    TCP->z = L_3;
}

//Funktionen für die Schieberegister (2xHC595)
//HC595 initialisieren
void shiftr_init()
{
    //DS, SH_CP und ST_CP Pins auf Output stellen
    HC595_DDR |= (1<<HC595_SH_CP_POS)|(1<<HC595_ST_CP_POS)|(1<<HC595_DS_POS);
}

//Puls auf Shift Clock Pin (SH_CP oder SRCLK)
void shiftr_shift()
{
    HC595_PORT |= (1<<HC595_SH_CP_POS);     //HIGH
```

```
    HC595_PORT &= (~(1<<HC595_SH_CP_POS));  //LOW
}

//Puls auf Store Clock Pin (ST_CP oder RCLK)
void shiftr_latch()
{
    HC595_PORT |= (1<<HC595_ST_CP_POS);     //HIGH
    //_delay_loop_1(1);
    HC595_PORT &= (~(1<<HC595_ST_CP_POS));  //LOW
    //_delay_loop_1(1);
}

//Zustände der Info-LEDs aus dem Struct "Led" Bit für Bit in uint16_t Variable
  schreiben
//Auffüllen von Rechts
//Bit 16 (links): ungenutzt, immer 0
uint16_t shiftr_data(led *Led)
{
    uint16_t data = 0b0000000000000000;

    //Servo 1: grün
    if(Led->led_g[0] == 1) {data |= (1<<0);}
    //Servo 1: orange
    if(Led->led_y[0] == 1) {data |= (1<<1);}
    //Servo 1: rot
    if(Led->led_r[0] == 1) {data |= (1<<2);}
    //Servo 2: grün
    if(Led->led_g[1] == 1) {data |= (1<<3);}
    //Servo 2: orange
    if(Led->led_y[1] == 1) {data |= (1<<4);}
    //Servo 2: rot
    if(Led->led_r[1] == 1) {data |= (1<<5);}
    //Servo 3: grün
    if(Led->led_g[2] == 1) {data |= (1<<6);}
    //Servo 3: orange
    if(Led->led_y[2] == 1) {data |= (1<<7);}
    //Servo 3: rot
    if(Led->led_r[2] == 1) {data |= (1<<8);}
    //Servo 4: grün
    if(Led->led_g[3] == 1) {data |= (1<<9);}
    //Servo 4: orange
    if(Led->led_y[3] == 1) {data |= (1<<10);}
    //Servo 4: rot
    if(Led->led_r[3] == 1) {data |= (1<<11);}
    //Servo 5: grün
    if(Led->led_g[4] == 1) {data |= (1<<12);}
    //Servo 5: orange
    if(Led->led_y[4] == 1) {data |= (1<<13);}
    //Servo 5: rot
    if(Led->led_r[4] == 1) {data |= (1<<14);}

    return data;
}

//Funktion gibt den Zustand der Info-LEDs über zwei Schieberegister (= 16 Byte) aus
//Da nur 15 LEDs vorhanden sind, ist das letzte Byte immer 0
```

```c
void shiftr_out(uint16_t data)
{
    //Low level macros um Data Pin (DS) auf HIGH oder LOW zu setzen
    //#define HC595DataHigh() (HC595_PORT |= (1<<HC595_DS_POS))
    //#define HC595DataLow() (HC595_PORT &= (~(1<<HC595_DS_POS)))

    //Jeden der 15 Bits seriell senden (MSB zuerst)
    for(uint16_t i=0; i<SHIFTR_BITS; i++)
    {
        //Output über DS je nach Wert des MSB (HIGH oder LOW)
        if(data & 0b1000000000000000)
        {
            //MSB is1 1: output HIGH
            HC595_PORT |= (1<<HC595_DS_POS);        //HC595DataHigh();
        }
        else
        {
            //MSB ist 0: output LOW
            HC595_PORT &= (~(1<<HC595_DS_POS));     //HC595DataLow();
        }
        //Puls auf Shift-Clock (SH_CP)
        shiftr_shift();
        //Nächstes Bit auf die MSB Position bringen
        data = (data<<1);
    }
    //Alle 15 Bits ins Schieberegister geladen
    //Puls auf Store Clock (ST_CP) -> Output
    shiftr_latch();
}
```