Project work for the course Computer and Microcontroller Technology WS21/22

Faculty of Biology and Preclinical Medicine
5DOF robotic arm
Markus Reichold
26.01.2022

Table of contents

1. Introduction

1.1 Background

More and more industrial processes are being carried out automatically by robots. The geometry of these robots varies greatly with their tasks, which is why the kinetics are difficult to standardize. One possibility for standardization was developed in 1955 by Jacques Denavit and Richard S. Hartenberg and is now considered the standard method for forward kinematics. With only four parameters per joint (so-called DH parameters), the entire robot can be described in terms of forward kinematics. In simplified terms, this means that in addition to the arm lengths, the angles of the rotational joints must also be known in order to calculate the position of the effector. The effector is the last link in the kinematic chain. This is often a tool, such as a gripper arm, a painting nozzle or a welding device. The center of the tool or a fixed point on it is called the effector.
Called the "Tool Center Point" (TCP). All movements of the robot are normally referred to this point.
While forward kinematics is mathematically easy to handle and provides clear results, inverse kinematics is much more complicated. Conversely, inverse kinematics means that the effector position (and the arm lengths) are given and the required joint angles of the motors are to be calculated from this. This type of kinematics is much more interesting and important for robotics, but unfortunately there are no standardized mathematical methods for it. One of the problems is that several joint angle configurations are often possible in order to achieve the desired effector position. For simple robots, geometric approaches are often used to calculate the inverse kinematics. More advanced methods are the creation of a pseudo inverse from the transformation matrix of the forward kinematics and the creation of a Jacobian matrix with gradient descent. These methods have the disadvantage that they are mathematically very complex and therefore require a fast processor if the movements are to be calculated in real time.
The aim of this work was to build a simple 5DOF robot arm (DOF = "degrees of freedom") and to implement its forward kinematics using the DH parameters. In addition, a rudimentary form of inverse kinematics (geometric approach for joints 1-3) was to be integrated. In addition to this automated form of positioning, it should also be possible to control the joint motors manually using joysticks.

1.2 Brief description

Five servo motors were used to move the links, 3x MG996R and 2x MG90S. The arm links including the gripper arm and the robot base were made from Lego technology. An ATmega328p microcontroller was used for the control (Arduino Uno).
The robot can be positioned using two different modes, manual and automatic mode. In manual mode, the servomotors are controlled via three (thumb) joysticks ("PS2 Joystick Game Controller XY Dual-Axis Joystick"). The direction of the joystick deflection determines the direction of movement and the width of the deflection determines the motor speed.
In automatic mode, however, the microcontroller must be connected to a computer. Coordinates (x, y and z) can be specified via the serial monitor, which the effector then controls. In order to position the effector more or less precisely, the servomotors must be calibrated. The inverse kinematics were solved with the help of analytical geometry for the first three motors. There are always two possible joint positions to achieve the target. The software can be used to set which of the

both options should be used. If a point cannot be approached, a corresponding error message is displayed on the PC's serial monitor. In addition, the calculated effector position from the inverse kinematics is recalculated for control via forward kinematics. A model of the robot according to Denavit-Hartenberg (DH parameters) was created for this purpose and a transformation matrix was produced.
Finally, info LEDs were integrated into the project. One green, one yellow and one red LED per motor provide information about the joint positions and possible problems at runtime. The 15 LEDs are controlled by two shift registers (74HC595) connected in series, which means that only three pins on the Arduino are occupied. The green LED lights up if there are no problems. The yellow LED lights up when the servo has reached a specified minimum or maximum position ("stop brake"). The red LEDs are particularly important for automatic mode. Among other things, they indicate whether the desired effector position can (mathematically) be reached at all.

2. Main part

2.1 Explanation of the structure

2.1.1 Scheme

Figure 1 shows the schematic of the robot arm with joints (G1-G5), relevant length specifications (L1-L3), joint angles (Φ1-Φ5, positive direction of rotation +) and the (clockwise) coordinate systems for each joint. The robot arm is in the neutral position, in which the joint angles Φ1-Φ4 are zero by definition. The servomotor for the gripper arm (G5) was not included in the DH parameters. The end of the closed gripper arm was selected as the "tool center point". The DH parameters in Table 1 were determined with the help of Fig. 1.

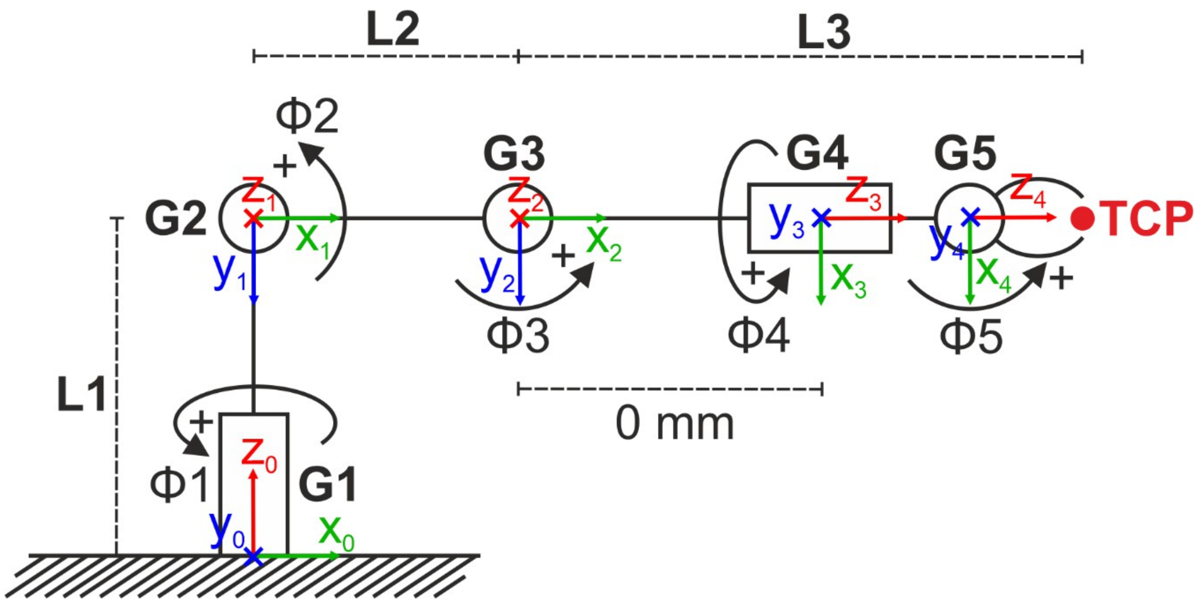

Figure 1: Schematic of the robot arm in neutral position. G1-5: Joints in the form of servo motors, Φ1-Φ5: Joint angles in the joints G1-G5 ("+" means positive direction of rotation), L1-3: Lengths of the relevant arm links, TCP: Position of the "Tool Center Point". The coordinate systems for each joint are also shown.

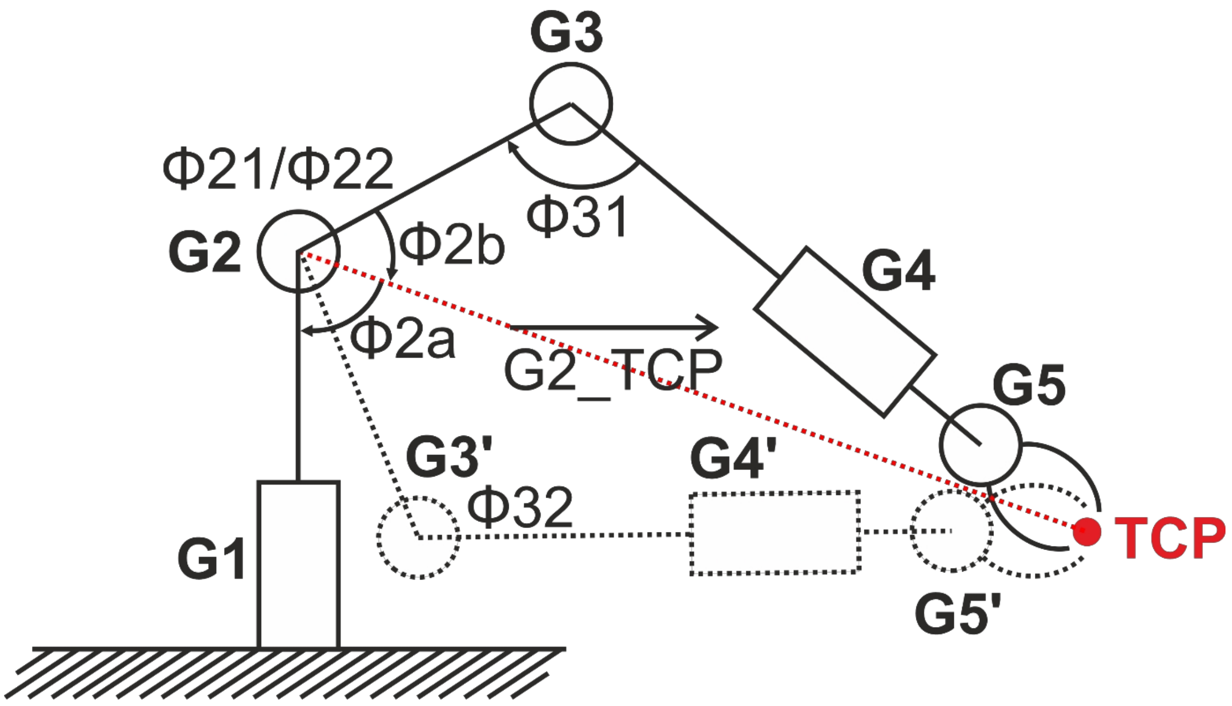

Figure 2: Auxiliary angles, vectors and alternative joint position. The alternative joint position, which can be assumed for (almost) every TCP (G3' - G5'), is shown as a dashed line. For this position, Φ2 must be calculated differently and the sign of Φ3 must be reversed. Φ2 is calculated in two steps. Φ2a is the angle between the vectors G2_0 (0 = origin of the world coordinate system) and G2_TCP. Φ2b, on the other hand, can be calculated using the cosine theorem, just like Φ3. The upper joint configuration (Φ21) is reached when Φ21 = Φ2a + Φ2b, the lower one (Φ22) when Φ22 = Φ2a
- Φ2b. The angles obtained must still be related to the neutral position of the robot: Φ21/Φ22 = (Φ21/Φ22 - 90°) * -1 and: Φ31 = 180° - Φ 3 1 ; Φ 3 2 = Φ31 * -1.

Table 1: DH parameters of joints 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

Length specifications:
L1 = 160 mm
L2 = 90 mm
L3 = 190 mm

The position of G4 between G3 and G5 is not relevant. Although G4 is closer to G5 than to G3 in the real robot, the axis of rotation of G4 was placed on the axis of rotation of G3 to determine the DH parameters, i.e. there is a distance of 0 mm between G3 and G4.

2.1.2 Circuit diagram

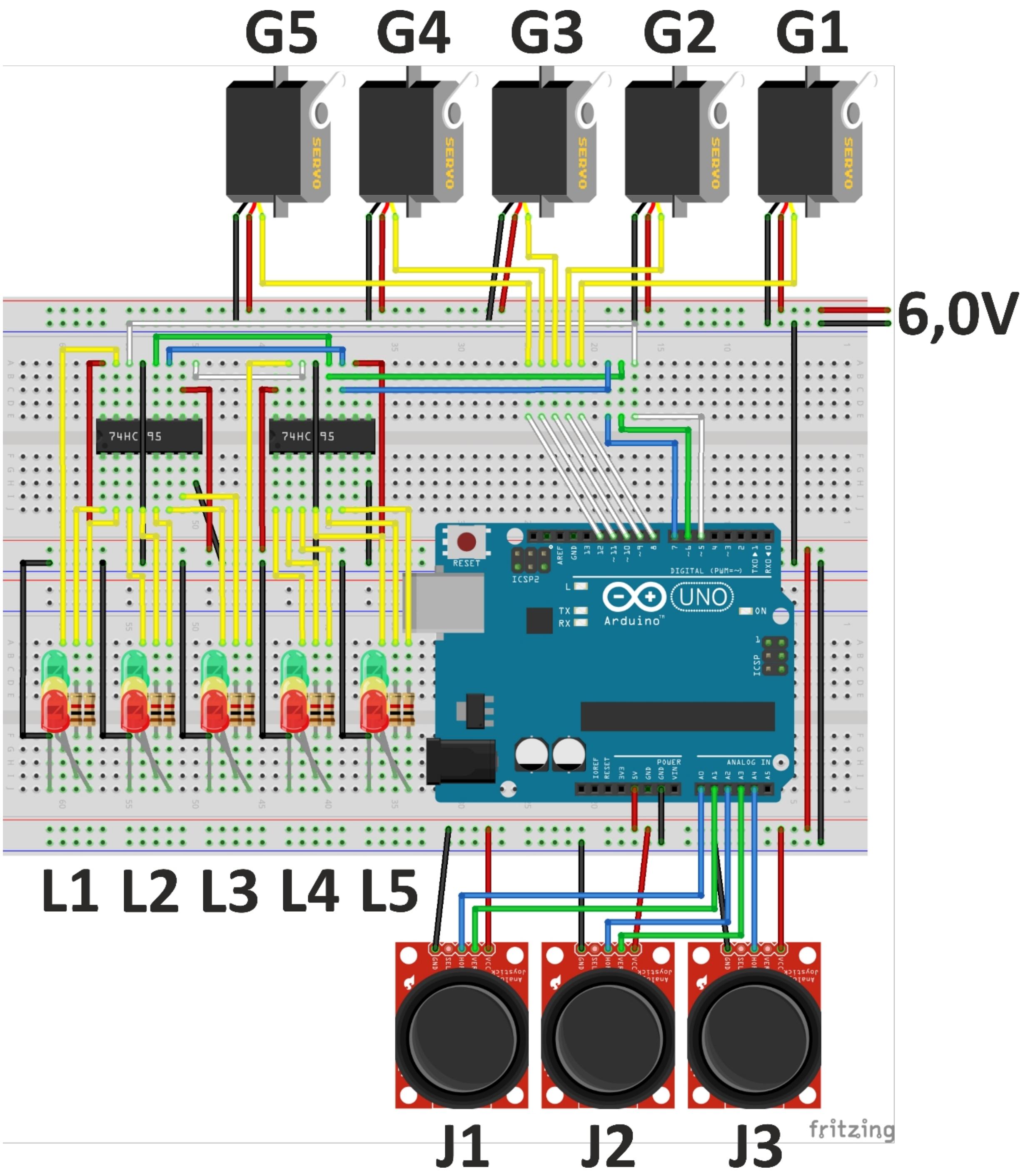

Figure 3: Circuit diagram. G1-G5: Servomotors of the five joints. Unlike in the illustration, two different servomotors were used, namely 3x MG996R (G1-3) and 2x MG90S (G4-5). L1-5: Three info LEDs are assigned to each servo (green, yellow, red). The numbers refer to the joints, i.e. the info LEDs L1 indicate the status of the servo motor of joint 1. A 1kΩ resistor is connected upstream of each LED. Two shift registers (74HC595), which are responsible for controlling the 15 LEDs, are located on the left of the upper breadboard. The registers are connected in series (16 bits). J1-3: Joysticks for manual control of the servo motors. Joysticks 1 and 2 use the x and y axes, whereas joystick 3 only uses the x axis. Joysticks, shift registers and LEDs are supplied with 5.0 V via the Arduino's USB port. The servo motors, on the other hand, have their own 6.0 V power supply.

2.1.3 Pictures

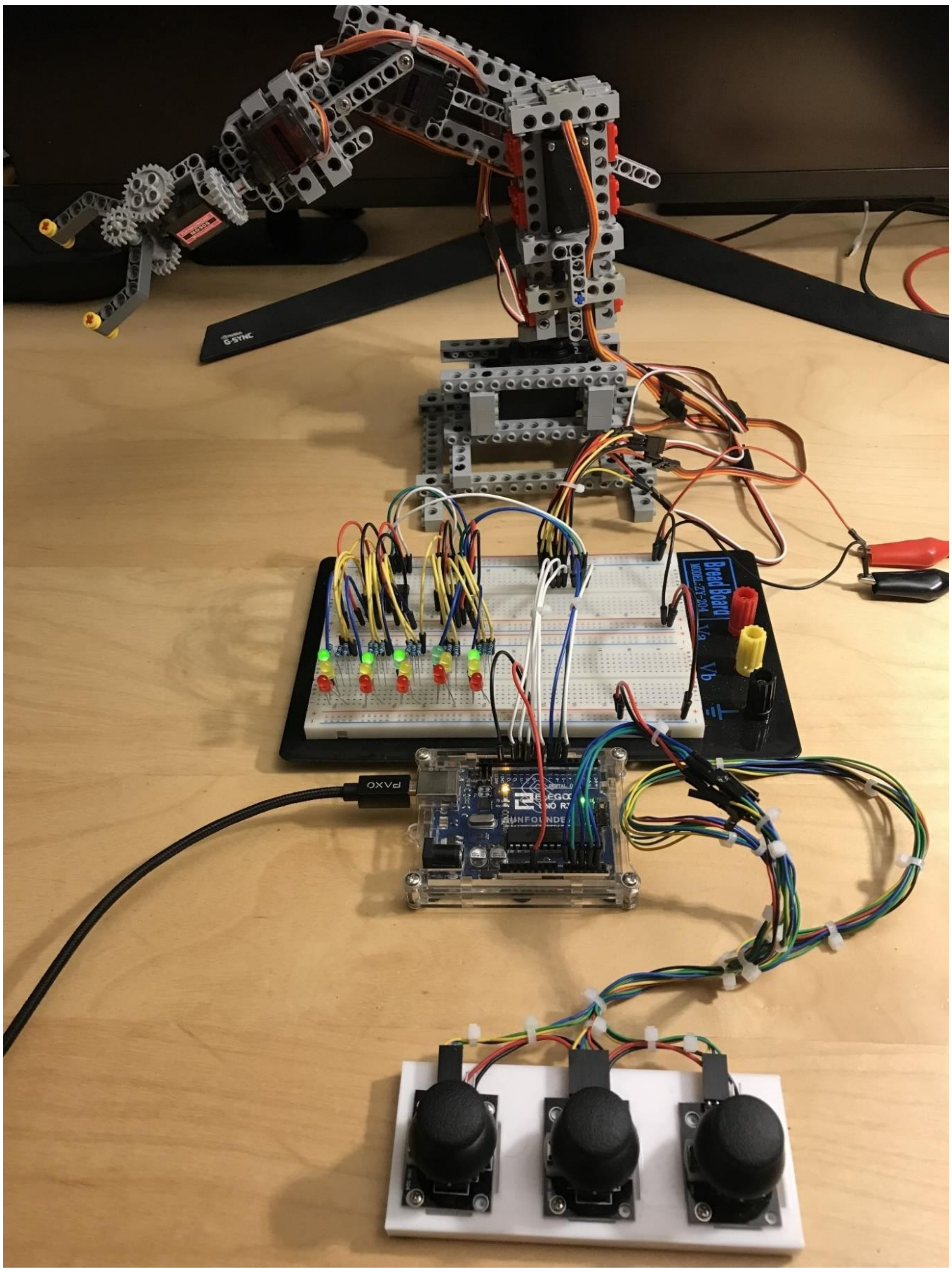

Figure 4: Photo of the setup. Unlike in Fig. 3, the Arduino Uno is not positioned on the breadboard here. The three joysticks (below) are screwed onto a 3D-printed plate. You can see the 5.0 V power supply via USB on the Arduino PCB and the 6.0 V power supply for the servo motors (terminals top right), which come from a laboratory power supply. Both circuits have a common ground (GND). The three large MG996R servos (G1-3) are simply clamped into the Lego building blocks, while the two small 2x MG90S servos (G4-5) are attached with double-sided adhesive tape.

2.2 Explanation program/source text

2.2.1 Generation of the PWM signal:

The control of several servomotors simultaneously is based on the following instructions: https://www.youtube.com/watch?v=dmjvHvw3Rc8 and has been modified for the project. The PWM signal is generated via timer 1 and can be output at any port of the microcontroller. Timer 1 is operated in mode 14 (Fast PWM, Top: ICR1, WGM11, 12 and 13 = 1) with a prescaler of 8 (CS11 = 1). With a CPU frequency of 16 MHz, the required period duration is 20 ms when ICR1 has reached the value 39999. An interrupt is then triggered, which sets all data pins of the servo motors (in this case on port B) to HIGH at the same time:

```
//Timer 1 handler for PWM signal
ISR(TIMER1_COMPA_vect)
{
        //Set all pins on port B to HIGH
        //Servos are controlled synchronously PORTB
        = 0xFF;
}
```

The reset to LOW takes place when a specified value in register TCNT1 is reached. These values are saved for all five motors in the "Pul" structure (Pul.pul_1 - Pul.pul_5). 1 ms or 1000 µs corresponds to a value of 2000 in TCNT1. To generate a PWM signal with x µs HIGH level, the value for TCNT1 must be multiplied by 2. Resetting the servo pins to LOW is shown here using the example of two servos:

```
//Set servo pins to LOW 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.....
}
```

The servo pins are only reset if:
1. the value in TCTN1 is reached (TCNT1 >= Pul.pul_1, determines the pulse length)
2. the pin is HIGH (bit_is_set(), SERVO_const[x][0] = global constants for servo pins) and
3. if the counter is in the range where the switchover is to take place (if(TCNT1 >= 1000 && TCNT1 <= 5200)).

All other actions of the program should only be executed if the counter is outside this range:

```
if(TCNT1 < 1000 || TCNT1 > 5200)
{
        //Additional code as PWM generation.....
}
```

This should ensure significantly better accuracy of the PWM signal, as the program sequence does not conflict with other code. The limits 1000-5200 were chosen because the MG996R servos used have a large pulse width range of 500 µs to 2600 µs (must be used for TCTN1 x2!). In summary, the code ensures that all servos are set to HIGH at the same time, but are set back to LOW at different times. Furthermore, a period duration of 20 ms is generated.

2.2.2 Manual mode:

In manual mode, the servo motors are controlled using three thumb joysticks. The direction of the deflection corresponds to the servo direction and the width of the deflection corresponds to the servo speed (the wider the deflection, the greater the speed). Only five of the six available axes were used, as only five motors are installed. No functions were assigned to the pushbuttons on each joystick. The following assignment of the analog axes to the joints/servos was defined:

- Joystick 1, x-axis: Horizontal rotation of the robot base in joint 1
- Joystick 1, y-axis: Vertical rotation in joint 2
- Joystick 2, y-axis: Vertical rotation in joint 3
- Joystick 2, x-axis: Rotation of the gripper arm in joint 4
- Joystick 3, x-axis: Opening and closing the gripper arm in joint 5

The Arduino's analogue-digital converter has a (maximum) resolution of 10 bits, i.e. the digitized joystick signal reaches values of 0-1023. Looking at the x-axis as an example, the leftmost deflection would be 0, the rightmost deflection would be 1023 and the middle would be 511. For the y-axis, the lowest position would be 0 and the highest 1023.
Each axis has been assigned a "joystick deadzone" (term used in gaming) of around 5% of the maximum deflection in each direction (global constant "JOY_dz" = 25, see joy_to_pulse() function). This prevents the motors from moving by themselves without operating the joystick because the center is not exactly right. The implementation of a joystick calibration has been omitted.
To make full use of the analog properties of the joysticks, the servo speed was made dependent on their deflection. In general, the position of the servos is determined by the pulse width of the PWM signal (500-2500 µs pulse with a period of 20 ms). A change in pulse width therefore means a change in position. However, if the pulse width change is not linear, the servo accelerates or brakes.

The change in pulse width was implemented using the following formula (using an x-axis as an example): deflection to the right: $\Delta Pulse = ((j\_val - JOY\_mid) / JOY\_div)^2$
Left deflection: $\Delta Pulse = ((JOY\_mid - j\_val) / JOY\_div)^2$

j_val: digitized joystick value (0-1023)
JOY_mid: global constant for joystick center position (511)
JOY_div: global constant for a divider that determines the speed change. The higher the value, the slower the change in speed

Experience has shown that the pulse change in the extreme positions should be a maximum of 40, otherwise the servos will become too fast. It should be noted that the pulse width change in the formula is not linear, but quadratic. This ensures that the motor speed initially increases only slowly at low joystick deflection and very quickly at high deflection. In gaming, this is known as the "joystick sensitivity curve" and means that the motors can be positioned precisely with the joysticks. With a linear pulse change, it has been shown that the speed changes too quickly at low deflection. Experiments with exponents 3 or 4 showed even better sensitivity at low displacement. However, it was feared that this would increase the computing load unnecessarily.
It should also be mentioned that the joystick signal is read out every 12 ms and the servo position is updated every 24 ms. The update rate of the servo position has a major influence on the servo speed. For this project, however, it was decided to leave this rate constant and to implement the speed adjustment via the divider described above.
Timer 0, which generates an interrupt every millisecond, was used for the sampling rate of the joysticks and the update rate of the servo position. In the interrupt routine, variables are incremented until the corresponding value of 12 or 24 is reached:

```
//Timer 0 Handler for the sampling rate of the joysticks and the servo speed (interrupt
every ms)
ISR(TIMER0_COMPA_vect)
{
        Joy_ms_count++;
        Serv_ms_count++;
}
```

The action is then triggered and the run variables are reset (here using the example of the joystick update):

```
//Save the joystick input every x ms
if(Joy_ms_count >= JOY_ms_refresh)
{
        //Here the joystick inputs are updated, i.e. the values are saved in the Struct
        "adcJ" saved

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

2.2.3 Automatic mode

In automatic mode, the position of the robot arm can be specified using coordinates (x, y and z coordinates in mm). To do this, the Arduino Uno must be connected to a PC. When the program is started, you will be prompted to enter the coordinates (int or double, comma-separated) via the serial monitor on the PC. If an input is made, this triggers an interrupt (ISR(USART_RX_vect)), which stores the characters entered **as a string (!)** in the variable
writes "usart_rx_buffer[]". It **is important that the input on the PC is made with a final \r\n** (check the corresponding box for the serial monitor in AtmelStudio). The entered coordinates are then output on the serial monitor of the PC for checking purposes.
Once the string has been read in, the variable "usart_string_compl" is set to 1 (= finished). Only then does further processing take place, namely parsing the string (function pars_rx_strg()). This means that three double variables are to be created from the string again. The function used for this (strtok()) can divide strings into substrings with the help of a defined delimiter. In this case, the comma used to separate the coordinates during input was selected as the delimiter. With the help of the atof() function, the three substrings are converted into double variables

and saved in the "TCP" struct. If conversion is not possible, the function returns 0. To e n a b l e the re-entry of coordinates, the variable
"usart_string_compl" is reset to 0.
Next, the joint angles are calculated using inverse kinematics (inv_kinematic() function). For reasons of space, we will not go into the details of the calculation here, but it is based on simple trigonometric considerations. The calculated angles are saved in the "Ang" struct. Both possible joint positions are always calculated (Phi_21 and Phi_22 or Phi_31 and Phi_32). If a calculation is not possible or if the calculation results in meaningless data (nan, inf), the function returns 0 (otherwise 1). However, if the calculation is successful, the calculated joint angles are checked a g a i n using forward kinematics. In this function (fwd_kinematic()), the joint angles are transferred and point 0, 0, 0 (origin of the world coordinate system) is transformed. The result of the calculation is saved in the "TCP_calc" struct and compared with the initially entered coordinates. As the specified and calculated coordinates may differ (very) slightly, they are compared with a certain tolerance (0.01 mm). If the comparison is not successful, a warning is displayed on the serial monitor and the robot arm remains in the current position. To calculate the forward kinematics, the DH parameters of the robot were first determined (see Table 1) and then inserted into the matrix below for each joint:

a)

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

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

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}};
```

Figure 5: Forward kinematics a) Transformation matrix (image source: Wikipedia). b) The transformation matrices for each individual joint are obtained by inserting the DH parameters from Table 1 into the matrix. The matrices shown were simplified by replacing coefficients that resulted in 0 or 1 accordingly. For joint 1 the corresponding matrix is called M01, for joint 2 M12, for joint 3 M23 and for joint 4 M34. By multiplying these matrices, a total matrix M04 is obtained, which describes transformations from the world coordinate system to the coordinate system of joint 4.

By multiplying the transformation matrix of each joint, the total matrix is obtained, which describes transformations from coordinate system 0 (world coordinate system) to coordinate system 4. If this transformation matrix is multiplied by the origin of the world coordinate system (0, 0, 0), the position (and orientation) of the TCP is obtained. Corresponding functions for vector and matrix calculations have been implemented as functions.

If the calculation of the forward kinematics is successful, the calculated joint angles are converted into pulse widths (ang_to_puls() function). To do this, the motors must be calibrated, i.e. the pulse width of at least two angles must be known. For the two-point calibration, the angle 0° (neutral position) and the angle -90° were used for servos 1-5. The corresponding pulse widths can then be calculated from the angles using a simple linear equation. It is important to note that the values for the joystick deflection to the left or down must be calculated differently than for a deflection to the right or up. This would not be necessary if the joysticks were in the center position at 0. Furthermore, the calculated pulse lengths in µs for timer 1 must be multiplied by 2.
The set_pulse() function writes the calculated pulse lengths for each servo to the "Pul" structure. This function can be used to select whether joint position 1 or 2 should be used. It also checks whether the pulse width reaches a defined maximum or minimum value ("stop brake"). Pin, pulse widths at 0° and -90° as well as minimum and maximum deflection of each servo are saved as global constants in the "SERVO_const" array. If the minimum or maximum deflection is exceeded or undershot, they are reset to the minimum or maximum deflection. The motor therefore remains at this position.

2.2.4 Info LEDs and shift register

LEDs have been integrated into the circuit to display information about the status of the motors during operation. Three LEDs are assigned to each of the 5 servo motors: one green, one yellow and one blue LED. Shift registers are required so that 15 LEDs can be controlled via the Arduino Uno, as otherwise the number of pins would not be sufficient. As the 74HC595 shift registers used here only have eight bits, two of them were connected in series. This results in 16 bits, which is sufficient for 15 LEDs. Only one LED lights up at a time per servo, which is why the current requirement, which is covered by the voltage converter of the Arduino Uno PCB, is kept within limits.
The status of the LEDs is saved in the "Led" struct (HIGH=1, LOW=0). For the output, the data of the struct is converted into a 16-bit binary variable (uint16_t data). This is done using the shiftr_data() function, where the LEDs are also assigned to the corresponding bits. The shift registers are initialized via the shiftr_init() function, the next bit is loaded into the register with shiftr_shift() and the 16 bits are output with shiftr_latch().

The colors of the LEDs were assigned to the following events:

- **Green:** Everything is OK. The planned movement can be carried out correctly.
- **Yellow:** The corresponding servo has reached the specified maximum or minimum position and cannot be moved any further. This applies to both manual and automatic mode. In automatic mode, however, this means that the desired point cannot be approached because one or more servo motors cannot be moved to the required position. However, in an ideal robot without restrictions on freedom of movement (especially with regard to collisions and a rotation of 360° per joint instead of 180°), the point could be reached mathematically.

- **Red:** Only important for automatic mode. These LEDs all light up together because the error messages are not caused by a single servo. In principle, only one LED would have been sufficient here. The red LEDs light up for the following events:
  - The inverse kinematics function (inv_kinematic()) returns 0. This may mean that 1. 0 was selected as the x and y coordinates, 2. the target cannot be reached because it is too close to the robot and 3. the calculation of one of the angles is infinity or "not a number" (nan) has resulted.
  - The calculation of the forward kinematics (fwd_kinematic()) returns 0. This happens if the calculated value is infinity or "not a number" (nan).
  - The calculated TCP from the forward kinematics does not match the specified TCP within a certain tolerance (0.01 mm).

At the moment, the red LEDs only light up for 50 ms because the joint position is then recognized as correct by the manual mode. To avoid this problem, the joystick pushbuttons could be used to switch between manual and automatic mode. The red LEDs would then light up until coordinates are entered that can be approached by the robot arm. Currently, both modes run simultaneously.

3. Conclusion

3.1 Difficulties/problems

In fact, many problems arose during the course of the project that went beyond the material covered in the course. One major challenge was controlling 5 servomotors simultaneously without using the servo.h library. A possible solution was found on the Internet and successfully integrated into this project. This method not only made it possible to generate a PWM signal for several servo motors at the same time, the pins were also independent of the six native PWM pins of the ATmega328p and could be selected as required.
Another difficulty was reading in the coordinates in automatic mode via the serial monitor. The main problem was that double variables were to be sent, which occupy 32 bits (= 4 bytes) on the ATmega328p. This requires the data to be sent via USART in the form of several packets (with 8 bits each). The solution implemented here works, but is probably not particularly economical with the resources of the microcontroller, as it works with strings (C standard library string.h).
It also took some time to integrate the servo speed control using the joysticks. This was not the primary plan. In principle, it would have been sufficient to read out the joystick direction and move the motors accordingly at a constant speed. However, the main problem was that the motors moved too quickly without speed control, which made it difficult or even impossible to position the effector precisely.
With regard to the joysticks, a "joystick deadzone" of around 5% in each direction also had to be built in, as the joysticks used were cheap and therefore imprecise. In this case, this means that the joysticks never actually reached a value of 511 in the center position. A "deadzone" of 5% could solve this problem.

3.2 Opportunities for improvement

First of all, the robot's arms should be designed to be mechanically more stable and the servo motors should be firmly screwed/glued to the components. As the three large servos are only clamped into the Lego, there is too much room for movement. In automatic mode, the positioning is therefore not very accurate. It would be even better if the links were 3D-printed. In this project, Lego was really only a means to an end, so that the robot arm could be built quickly and easily. It would also have been advantageous not to use the servos directly as rotation axes, but to integrate them into simple gears. In this way, the normal 180° rotation range of the motors can easily be extended to 360° with the help of Lego gears. In terms of programming, it would be nice to add collision detection for manual (but also automatic) mode. The individual modules for this are already available in principle. What is still missing would be an inverse function puls_to_ang() to the function ang_to_puls(). In this case, the pulse lengths would be transferred and the joint angles would be calculated from them. Each time the pulse width is increased or decreased via the joystick input (24 ms cycle), the resulting joint angle configurations could be determined and converted into the effector position via the forward kinematics function. This would make it clear at all times where the effector is currently located. In this way, for example, it is possible to avoid moving the effector in the negative z-direction (i.e. into the ground). Such a movement cannot be prevented with a minimum/maximum deflection, as implemented for each servo in this project, as two motors can be responsible for this at the same time (e.g. G2 and G3). The geometry of the robot base can also be roughly mathematically simulated and examined for collisions. It could also be calculated whether a collision will not occur with one of the two joint angle configurations.
Finally, the three info LEDs for each servo could be replaced with an RGB LED. This is possible because only one of the three LEDs lights up at any one time. This would also allow more colors to be used to display more information. It would also save space.

3.3 Conclusion

My project work has taken up a lot of the content that was taught during the course. This includes the generation of a PWM signal, the control of servo motors, the use of a shift register, a timer and an analog-digital converter. Some of this knowledge was expanded and deepened for this project work. For example, five servo motors were controlled simultaneously, which is significantly more complex than the course exercises with just one servo. Furthermore, two shift registers were connected in series, which also goes beyond the content of the course. However, the use of (thumb) joysticks was not covered in the course. This knowledge was acquired independently.
Overall, it was a very exciting and instructive course. The assembler programming in particular was new to me and broadened my horizons considerably. The course also gave me a new perspective on the C programming language. I learned that C is not just C (AVR C vs. Arduino script), how much work is actually done by the compiler and that the compiler doesn't always do everything perfectly.

4. Bibliography

4.1 Data sheets used

- Data sheet ATmega328p
- Data sheet 74HC595

4.2 Websites

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 and servo control:

- 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 converter:

- 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/

Shift register:

- 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 functions:

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

Vector and matrix calculation, 3D transformations:

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

DH parameters:

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

5. Appendix

5.1 Source text

```c
//µC course, WS 21/22
//Project work
//Markus Reichold
//5DOF robot arm

#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) initialize pwm_init();
    //USART initialize
    usart_init();
    //Initialize the ADC for the joysticks
    adc_init();
    //Initialize timer 0 for the sampling frequency of the joysticks and the servo speed
    ms_init();
    //Initialize shift register (2xHC595) shiftr_init();
    //Enable interrupts
    is();

    //Set joint angle and TCP to neutral position
    reset_ang(&Ang);
    reset_tcp(&TCP);
    //Convert joint angle to pulse width (*2)
    //and save in Strut Pul
    //Last parameter specifies which of the two possible joint configurations is to be used
    set_pulse(&Ang, &Pul, 1, &Led);

    //Prompt
    //With the following type of string initialization, the \0 is automatically appended
      by the compiler
    char text1[] = "Enter coordinates [mm] in the format: x,y,z:\n\r";
    usart_send_strg(text1);

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

    while(1)
    {
        //////////////////////////
        // PWM signal generation //
        //////////////////////////

        //Set servo pins to LOW if
        //1) the desired value in TCNT1 is reached (determines pulse length)
        //2) the pin is set to HIGH
        //3) the pulse length is in the range where switching is necessary
          (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 (gripper arm, Phi_5):
            if(TCNT1 >= Pul.pul_5 && bit_is_set(PORTB, SERVO_const[4][0]))
            {
                PORTB &= ~ (1<<SERVO_const[4][0]);
            }
        }

        //Additional code should only be executed if the pulse length is outside 500-
           2600 (x2) ms
        //and the servo position does not have to be set -> ensures better
          accuracy of the PWM signal
        if(TCNT1 < 1000 || TCNT1 > 5200)
        {
            //////////////////////////////////
            // Manual mode (joysticks) //
            //////////////////////////////////

            //Increase/decrease the pulse width every x ms
            //The delta of the increase/decrease is determined by the joystick deflection
              (servo speed)
            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,  O, &Led);
                Pul.pul_5 = joy_to_pulse(adcJ.joy_3_x, 5, Pul.pul_5,  O, &Led);

                //Display info LEDs shiftr_out(shiftr_data(&Led));


                //Reset Counter
                Serv_ms_count = 0;
            }

            //Save the joystick input every x ms
            if(Joy_ms_count >= JOY_ms_refresh)
            {
                //ADC signal from the joystick from 0-1023
                //Joysick center position at 511
                //Joystick x-axes: left 0-511, right 512-1023
                //Joystick y-axes: top 512-1023, bottom 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;
            }

            //////////////////////////////////////
            // Automatic mode (coordinates) //
            //////////////////////////////////////

            //When coordinates have been successfully read in via USART
            if(usart_strg_compl == 1)
            {
                //Parse the string and set the TCP coordinates
                pars_rx_strg(usart_rx_buffer, &TCP);
                //Output of the coordinates
                char text2[60];
                sprintf(text2, "Entered coordinates: x=%.2f, y=%.2f, z=%.2f\n
                  \r", TCP.x ,TCP.y ,TCP.z);
                usart_send_strg(text2);

                //Calculate joint angle using inverse kinematics
                if(!inv_kinematic(&TCP, &Ang))
                {
                    //When it is not possible to calculate the joint angles:

                    //Set joint angle and TCP to neutral position
                    //reset_ang(&Ang);
                    //reset_tcp(&TCP);

                    //Info LEDs: All red
                    for(uint8_t i=0; i<5; i++)
                    {
                        Led.led_g[i] = 0;
                        Led.led_y[i] = 0;
                        Led.led_r[i] = 1;
                    }
                    //Send warning message
                    char text3[] = "Calculation of the joint angles not possible!
                     \r";
                    usart_send_strg(text3);
                }
                else
                {
                    Calculate //TCP via forward kinematics and compare with original
                     TCP
                    if(!fwd_kinematic(&Ang, &P_Zero, &TCP_calc, 1))
                    {
                        //When the TCP could not be recalculated via the
                     forward kinematics:

                        //Set joint angle and TCP to neutral position
                        //reset_ang(&Ang);
                        //reset_tcp(&TCP);

                        //Info LEDs: All red
                        for(uint8_t i=0; i<5; i++)
                        {
                            Led.led_g[i] = 0;
                            Led.led_y[i] = 0;
                            Led.led_r[i] = 1;
                        }
                        //Send warning message
                        char text4[] = "Calculation of the precursor kinematics not
                     possible!\n\r";
                        usart_send_strg(text4);
                    }
                    else
                    {
                        //Comparison with tolerance (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))
                        {
                            //When the specified TCP differs from the calculated TCP
                     deviates:

                            //Set joint angle and TCP to neutral position
                            //reset_ang(&Ang);
                            //reset_tcp(&TCP);

                            //Info LEDs: All red
                            for(uint8_t i=0; i<5; i++)
                            {
                                Led.led_g[i] = 0;
                                Led.led_y[i] = 0;
                                Led.led_r[i] = 1;
                            }
                            //Send warning message
                            char text5[] = "Error in the calculation of the 
            prewartskinematics!\n\r";
                            usart_send_strg(text5);
                        }
                        else
                        {
                            //Info LEDs: All green
                            for(uint8_t i=0; i<5; i++)
                            {
                                Led.led_g[i] = 1;
                                Led.led_y[i] = 0;
                                Led.led_r[i] = 0;
                            }

                            //Joint angle successfully set
                            //light up the green LED
                            //Output success message
                            char text6[120];
                            sprintf(text6, "Calculated joint angles: 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);

                            //Output of the calculated TCP via the forward 
            kinematics
                            char text7[60];
                            sprintf(text7, "Calculated TCP: x=%.2f, y=%.2f, z=%.2f 
            \n\r",
                            TCP_calc.x,
                            TCP_calc.y,
                            TCP_calc.z);
                            usart_send_strg(text7);

                            //Convert joint angle to pulse width (*2)
                            //and save in strut Pul
                            set_pulse(&Ang, &Pul, 1, &Led);
                            //Output success message
                            char text8[120];
                            sprintf(text8, "Calculated pulse lengths: 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);
                        }
                    }
                }
```

```
                //Prepare buffer for reading in new coordinates usart_strg_compl = 0;
                //Display info LEDs
                shiftr_out(shiftr_data(&Led));
                //Prompt for the next input usart_send_strg(text1);
            }
        }
    }
}

///////////////////////
// Interrupt handler //
///////////////////////

//Timer 1 handler for PWM signal
ISR(TIMER1_COMPA_vect)
{
    //Set all pins on port B to HIGH
    //Servos are controlled synchronously PORTB
    = 0xFF;
}

//Timer 0 Handler for the sampling rate of the joysticks and the servo speed (interrupt
  every ms)
ISR(TIMER0_COMPA_vect)
{
    Joy_ms_count++;
    Serv_ms_count++;
}

//USART Handler: USART Rx Complete (0x0024)
ISR(USART_RX_vect)
{
    //Transfer of the x, y and z coordinates via input on the serial monitor (x, y, z)
    //Characters are stored in the C-string "usart_rx_buffer"
    //when all characters of the string have been loaded into the buffer,
    //the variable usart_strg_complete is set to 1
    uint8_t nextChar;
    //Read in character by character
    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 for vectors in 3D space
typedef struct vector
{
	double x;
	double y;
	double z;
} vec;

//Typedef for points in 3D space
typedef struct point
{
	double x;
	double y;
	double z;
} dot;

//Typedef for 4x4 matrices
typedef double matr[4][4];

//Typedef for Struct, which contains the joint angles
//Specification in wheel
//_21/_22 or _31/_32 refer to two possibilities
//of the joint positions to reach the TCP 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 for Struct that contains the pulse widths for the servos
//Pulse widths in µs must be taken as an entry in TCNT1 x 2 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 for Struct that contains the analog values for the three joysticks 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 for Struct that contains the state of the info LEDs (0 or 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>

//Declaration of global constants

//Pi
external const double PI;

//Lengths of the links [mm]
extern const double L_1;
extern const double L_2;
extern const double L_3;

//Servo-specific constants
//0: Servo pin on the Arduino (PORTB)
//1: P_0 = pulse length at angle 0 (neutral position)
//2: P_-90 = pulse length at -90 degrees
//3: P_Min = Minimum pulse length for the servo
//4: P_Max = Maximum pulse length for the servo
external const uint16_t SERVO_const[5][5];
//Refresh rate of the servo position in ms external
const uint8_t SERVO_ms_refresh;

//Joystick-specific constants
//Joystick deadzone (the same for all joysticks/axes)
//Applies in +- x- and y-direction, approx. 5% of the deflection in each
   direction (512x0.05=25.6)
//Prevents the servos from running automatically if the center position of the joysticks is
    not exactly at 512
external const uint8_t JOY_dz;
//Joystick center position
//10 bit = (1024/2) - 1 = 511
extern const uint16_t JOY_mid;
//Test constant for the servo speed external
const uint8_t JOY_div;
//Sampling rate of the joysticks in
ms extern const uint8_t
JOY_ms_refresh;

#endif
```

```
#include "const.h"

//Definition of global constants

//Pi
const double PI = 3.14159265;

//Lengths of the links [mm]
const double L_1 = 160.0;
const double L_2 = 90.0;
const double L_3 = 190.0;

//Servo-specific constants
//0: Servo pin on the Arduino (PORTB)
//1: P_0 = pulse length at angle 0 (neutral position)
//2: P_-90 = pulse length at -90° (easier to measure than +90°)
//3: P_Min = Minimum pulse length for the servo
//4: P_Max = Maximum pulse length for the servo
//Pulse lengths in µs, for timer 1 x 2 take
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}
};
//Refresh rate of the servo position in ms (24 ms)
const uint8_t SERVO_ms_refresh = 23;

//Joystick-specific constants
//Joystick deadzone (the same for all joysticks/axes)
//Applies in +- x- and y-direction, approx. 5% of the deflection in each direction (512 x 0.05 
   = 25,6)
//Prevents the servos from running automatically if the center position of the joysticks is 
   not exactly at 512
const uint8_t JOY_dz = 25;
//Joystick center position
//10 bit = (1024/2) - 1 = 511
const uint16_t JOY_mid = 511;
//Test constant for the servo speed
//The higher, the slower the servos const
uint8_t JOY_div = 80;
//Sampling rate of the 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"

//Declaration of global variables

//Tool Center Point = desired effector position dot
TCP;
//Joint angle of the robot
ang Ang;
//(modified) pulse width for the servos
pul Pul;
//ADC values of the three joysticks (x and y axes)
volatile adcj adcJ;
//calculated TCP via forward kinematics dot
TCP_calc;
//Zero point of the world coordinate
system dot P_Zero;
//USART receive buffer for the coordinates char
usart_rx_buffer[USART_RX_MAX + 1];
//Variable indicates whether the string with the coordinates was received in full via USART
//1 = complete, 0 = not yet complete volatile
uint8_t usart_strg_compl;
//Counter for the number of characters transferred via USART
volatile uint8_t usart_strg_count;
//Counter for ms for the sampling rate of the joysticks
(timer 0) volatile uint8_t Joy_ms_count;
//Counter for ms for the update rate of the servo position (timer 0) volatile
uint8_t Serv_ms_count;
//Struct for the status of the Info-LEDs
led Led;

#endif
```

```
#include "var.h"

//Definition of global variables

//Tool Center Point = desired effector position dot
TCP = {0, 0, 0};
//Joint angle of the robot
ang Ang = {0, 0, 0, 0, 0, 0, 0};
//(modified) pulse width for the servos pul
Pul = {0, 0, 0, 0, 0};
//ADC values of the three joysticks (x and y axes)
adcj adcJ = {0, 0, 0, 0, 0, 0};
//calculated TCP via forward kinematics dot
TCP_calc = {0, 0, 0};
//Null point of the world coordinate
system dot P_Zero = {0, 0, 0};
//USART receive buffer for the coordinates
char usart_rx_buffer[USART_RX_MAX + 1] = "";
//Variable indicates whether the string with the coordinates was received in full via USART
//1 = complete, 0 = not yet complete
uint8_t usart_strg_compl = 0;
//Counter for the number of characters transferred via USART
uint8_t usart_strg_count = 0;
//Counter for ms for the sampling rate of the joysticks (timer 0)
uint8_t Joy_ms_count = 0;
//Counter for ms for the update rate of the servo position (timer 0)
uint8_t Serv_ms_count = 0;
//Struct for the status of the 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 for USART
#define F_CPU 16000000UL
#define BUAD 9600
#define UBRR ((F_CPU/16/BUAD) - 1)
//Defines for shift registers (port and pins)
//Two shift registers connected in series (16 bit) #define
HC595_PORT              PORTD
#define HC595_DDR       DDRD
#define HC595_DS_POS    PD5     //Data pin (DS or SER)
#define HC595_SH_CP_POS PD7     //Shift Clock Pin (SH_CP or SRCLK)
#define HC595_ST_CP_POS PD6     //Store Clock Pin ( ST_CP or RCLK)
#define SHIFTR_BITS     16      //Number of bits in both shift registers

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

//Functional declarations

//Conversion from wheel to
degrees double
rad_in_grad(double rad); double
grad_in_rad(double grad);
//Arduino map() function for double variables
double mapd(double val, double in_min, double in_max, double out_min, double
  out_max);

//USART
Initialize //USART void
usart_init();
//Single character received via USART (not required)
uint8_t usart_rec_char(void);
//String received via USART and stored in buffer (not required) void
usart_rec_strg(char *Buffer, uint8_t MaxLen);
//Send single character to PC via USART void
usart_send_char(char ch);
//Send string to PC via USART void
usart_send_strg(char *strg);
//Parse the string sent from the PC (Delimiter: ,)
//Coordinates must be sent in the form: x,y,z
//Convert the substrings into double variables and save in the TCP struct void
pars_rx_strg(char *strg, dot *TCP);

//Timer 1 for PWM signal
void pwm_init();

//Initialize ADC (joysticks)
void adc_init();
//Read out the ADC
```

```
uint16_t adc_read(uint8_t ch);

//Timer 0 for the sampling frequency of the joysticks and the servo speed
void ms_init();

//Vector calculations
//Calculate the amount of a vector
double vec_amount(vec *v);
//Calculate scalar product of two vectors
double vec_scalarprod(vec *v1, vec *v2);
//Subtract two vectors -> vector vec
vec_subtract(vec *v1, vec *v2);
//Subtract location vectors of two points -> vector vec
dot_subtract(dot *p1, dot *p2);

//Matrix calculations
//4x4 Multiply matrices
//Matrix 1 and 2 are transferred
//The result of the multiplication is written to matrix 3 void
matr_x_matr(matr m_src1, matr m_src2, matr m_dst);
//4x4 Multiply matrix with point
//p_src = point for the multiplication
//p_dst = result of the multiplication
//Points are not shown as homogeneous coordinates for the sake of simplicity (=4th
   variable h)
//h is usually = 1, unless the matrix is a projection matrix (for
   perspective representation)
//Then the x,y,z coordinates must be divided by h void
matr_x_dot(matr m, dot *p_src, dot *p_dst);

//Kinematics
//Function for inverse kinematics
//The desired TCP is transferred
//The calculated joint angles are saved in the global Struct Ang
//If the angles cannot be calculated, the function returns 0, otherwise 1
uint8_t inv_kinematic(dot *TCP, ang *Ang);
//Function for forward kinematics
//The Struct Ang is transferred with the joint angles,
//the transformation matrix and the point to be transformed
//Additionally, the point in which the transformation is saved
//The last parameter x specifies whether joint position 1 or 2 should be used
//If the point cannot be calculated, the function returns 0, otherwise 1
uint8_t fwd_kinematic(ang *Ang, dot *p_src, dot *p_dst, uint8_t x);

//Calculation of pulse lengths
uint16_t ang_to_puls(uint8_t s_nr, double ang, led *Led);
//Save pulse lengths in Struct Pul
//The last parameter x specifies whether joint position 1 or 2 is to be used void
set_pulse(ang *Ang, pul *Pul, uint8_t x, led *Led);
//Converts the joystick inputs into a pulse width
//j_val = analog value of the joystick axis (1-1024)
//s_nr = servo number (1-5)
//s_puls_alt = current servo pulse (µs*2)
//s_dir = desired servo direction (1,0)
//led *Led = Reference to the struct for the 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);

//Set joint angle to neutral position void
reset_ang(ang *Ang);
//TCP set to neutral position void
reset_tcp(dot *TCP);

//Functions for the shift registers (2xHC595)
//HC595 initialize
void shiftr_init();
//Pulse on shift clock pin (SH_CP or SRCLK)
void shiftr_shift();
//Pulse on store clock pin (ST_CP or RCLK)
void shiftr_latch();
//States of the info LEDs from the "Led" struct bit by bit in uint16_t variable
  write
//Filling from the right
//Bit 16 (left): unused, always 0
uint16_t shiftr_data(led *Led);
//Function outputs the status of the info LEDs via two shift registers (= 16 bytes)
//Since there are only 15 LEDs, the last byte is always 0 void
shiftr_out(uint16_t data);

#endif
```

```
#include "functions.h"

//Function definitions

//Conversion from wheel to
degrees double rad_in_grad(double
rad)
{
	return rad * (180.0 / PI);
}
double grad_in_rad(double grad)
{
	return grad * (PI / 180.0);
}
//Arduino map() function for double variables
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
Initialize //USART void
usart_init()
{
	//UBRR0:
	//Define baud rate of 9600 bps in register UBRR0H and UBRR0L (16 bit)
	//Register UBRR0H must be written before register UBRR0L! UBRR0H =
	(UBRR>>8);
	UBRR0L = (UBRR);
	//UCSR0B: Transmitter (TXEN0)/Receiver (RXEN0) enable
	//RXCIE0: Activate interrupts when received byte is ready UCSR0B =
	(1<<RXCIE0)|(1<<RXEN0)|(1<<TXEN0);
	//UCSR0C:
	//USARD Mode Select: UMSEL01 and UMSEL00 = 0 -> Asynchronous Mode
	//Parity mode: UPM01 and UPM00 = 0 -> Disabled (no parity bit)
	//Stop bit: USBS0 = 0 -> On stop bit
	//Character Size: UCSZ02 = 0, UCSZ01 = 1, UCSZ00 = 1 -> 8-bit
	UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
}
//Single character received via USART
uint8_t usart_rec_char(void)
{
	//Wait until the USART data register is ready/empty
	while(!(UCSR0A & (1<<UDRE0))) {};
	//Return received character as 8 bit char return
	UDR0;
}
//Receive string via USART and store in buffer void
usart_rec_strg(char *Buffer, uint8_t MaxLen)
{
	uint8_t NextChar;
	uint8_t StringLen = 0;
	//Wait for next character and receive NextChar =
	usart_rec_char();
	//Write characters to the buffer until
	//either the string is off (\n)
```

```
    //or the buffer is full
    while(NextChar != '\n' && NextChar != '\r' && StringLen < MaxLen)
    {
        *Buffer++ = NextChar;
        StringLen++;
        NextChar = usart_rec_char();
    }
    //Append '\0' for C-string
    *Buffer = '\0';
}
//Send single character via USART void
usart_send_char(char ch)
{
    //Wait until the USART data register is ready/empty
    while(!(UCSR0A & (1<<UDRE0))) {}
    //Send character
    UDR0 = ch;
}
//Send string to PC via USART 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]);
    }
    */
}
//Parse the string sent from the PC (Delimiter: ,)
//Coordinates must be sent in the form: x,y,z
//Convert the substrings into double variables and     in the TCP struct void
pars_rx_strg(char *strg, dot *TCP)
{
    //Split string after delimiter char
    *substring[3];
    char delimit[] = ",";
    uint8_t i=0;
    substring[i] = strtok(strg, delimit);
    while(substring[i] != NULL)
    {
        i++;
        substring[i] = strtok(NULL, delimit);
    }
    //Convert substrings to double
    //and save x, y and z values in the TCP struct
    //Return of 0.0 if no conversion is possible TCP->x =
    atof(substring[0]);
    TCP->y = atof(substring[1]);
    TCP->z = atof(substring[2]);
```

```
}

//Timer 1 for PWM signal
void pwm_init()
{
	//Set all pins on port B for the servos to output DDRB =
	0xFF;
	//Set register
	//WGM11, 12 and 13: Mode 14
	//CS11: Prescaler 8 -> 40,000 clks for 20 ms
	//Output Compare A Match Interrupt Enable
	TCCR1A = (1<<WGM11);
	TCCR1B = (1<<WGM12) | (1<<WGM13) | (1<<CS11);
	TIMSK1 = (1<<OCIE1A);
	//ICR1: Value corresponds to 20 ms = one
	period ICR1 = 39999;
	//40000 = 20 ms
	//2000 = 1 ms = 1000 µs
	//Desired pulse length x 2 = Value in TCNT1
	//pulse length of 500 µs = TCNT1 1000
	//pulse length of 1000 µs = TCNT1 2000
	//pulse length of 1500 µs (center position) = TCNT1 3000
	//pulse length of 2000 µs = TCNT1 4000
	//pulse length of 2500 µs = TCNT1 5000
}

//ADC for the joysticks
void adc_init()
{
	//ADCSRA: ADC Control and Status Register A
	//ADC=1: Activate ADC
	//ADPS0-2: Prescalar (ADPS2=1, ADPS1=1, ADPS0=1 -> 128 dividers)
	//Required frequency: 50-200kHz; 16,000,000/128 = 125,000 (only possible
	  divider, otherwise > 200kHz)
	ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
	//ADMUX: ADC Multiplexer Selection Register
	//REFS1=0 and REFS0=1: AVcc (with external capacitor at AREF pin) ADMUX
	= (1<<REFS0);
}
//Read out ADC
//Parameter is the channel to be read out (PC0-PC5)
uint16_t adc_read(uint8_t ch)
{
	//7 possible channels
	ch &= 0b00000111;
	ADMUX = (ADMUX & 0xF8)|ch;
	//Start single conversion (1 in ADSC)
	ADCSRA |= (1<<ADSC);
	//Wait for the conversion to be completed (ADSC becomes 0 again) while(ADCSRA &
	(1<<ADSC));

	return (ADC);
}

//Timer 0 for the sampling frequency of the joysticks and the servo speed
void ms_init()
```

```
{
	//Sampling rate should be 1000 Hz (every 1 ms)
	//CPU frequency = 16,000,000 Hz / divider 64 = 250,000 [s]
	//1 ms therefore corresponds to 250

	//Timer in CTC mode -> compares with OCR0A
	//CTC mode -> In TCCR0A: WGM00=0, WGM01=1, In TCCR0B: WGM02=0
	TCCR0A = (1<<WGM01);
	//Divider: clk/64 -> In TCCR0B: CS02=0, CS01=1, CS00=1
	TCCR0B = (1<<CS00)|(1<<CS01);
	//Set comparison register OCR0A to msTimer = 249 = 1 ms
	OCR0A = 0xF9;
	//Interrupt when OCR0A is reached -> In TIMSK0: OCIE0A=1 TIMSK0
	= (1<<OCIE0A);

	// initialize counter
	//TCNT0 = 0 ;
}

//Vector calculations
//Calculate the amount of a vector
double vec_amount(vec *v)
{
	return sqrt((v->x * v->x) + (v->y * v->y) + (v->z * v->z));
}
//Calculate scalar product of two vectors
double vec_scalarprod(vec *v1, vec *v2)
{
	return (v1->x * v2->x) + (v1->y * v2->y) + (v1->z * v2->z);;
}
//Subtract two vectors -> vector vec
vec_subtract(vec *v1, vec *v2)
{
	vec v3 = {v1->x - v2->x, v1->y - v2->y, v1->z - v2->z};
	return v3;
}
//Subtract location vectors of two points -> vector vec
dot_subtract(dot *p1, dot *p2)
{
	vec v = {p1->x - p2->x, p1->y - p2->y, p1->z - p2->z};
	return v;
}

//Matrix calculations
//4x4 Multiply matrices
//Matrix 1 and 2 are transferred
//The result of the multiplication is written to matrix 3 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 Multiply matrix with point
//p_src = point for the multiplication
//p_dst = result of the multiplication
//Points are not shown as homogeneous coordinates for the sake of simplicity (= 4th ↩
  variable h)
//h is usually = 1, unless the matrix is a projection matrix (for perspective display)
//Then the x,y,z coordinates must be divided by h void
matr_x_dot(matr m, dot *p_src, dot *p_dst)
{
    //COLUMN-MAJOR variant
    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];
}

//Kinematics
//Function for inverse kinematics
//The desired TCP is transferred
//The calculated joint angles are saved in the global Struct Ang
//If the angles cannot be calculated, the function returns 0, otherwise 1
uint8_t inv_kinematic(dot *TCP, ang *Ang)
{
    //x and y of the TCP must not both be 0 if(TCP-
    >x == 0 && TCP->y == 0)
    {
        return 0;
    }
    else
    {
        /////////////////////////
        // Calculation of Phi_1 //
        /////////////////////////

        //Vector with the xy-coordinates of the TCP (from the origin) vec
        V_TCPxy = {TCP->x, TCP->y, 0};
        //Calculate the amount of the vector
        (hypotenuse) double L_TCPxy =
        vec_amount(&V_TCPxy);
        //Phi1 calculate (cosine)
        Ang->Phi_1 = acos(TCP->x/L_TCPxy);
        //Determine the sign of the angle
        if(TCP->y < 0)
        {
            Ang->Phi_1 *= -1.0;
        }

        /////////////////////////
        // Calculation of Phi_2 //
        /////////////////////////

        //Coordinates of point G2
        //Immovable, dependent on L1
        dot P_G2 = {0, 0, L_1};
        //Calculate vector G2_TCP
        vec V_G2_TCP = dot_subtract(TCP, &P_G2);
        //Determine amount of V_G2_TCP
        double L_G2_TCP = vec_betrag(&V_G2_TCP);
        //If L_G2_TCP < L_2+L_3, then the target cannot be reached if(L_G2_TCP > (L_2 +
        L_3))
        {
            return 0;
        }
        else
        {
            //Vector parallel to the z-axis
            vec V_z = {0, 0, -1.0};
            //Calculate auxiliary angle Phi_2a = angle between V_G2_TCP and V_z
            //Only the acute angle is calculated
            double Phi_2a = acos(vec_skalarprod(&V_G2_TCP, &V_z)/L_G2_TCP);

            //Calculate the auxiliary angle Phi_2b using the cosine theorem
            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 and Phi_22 (2 possibilities for the joint position) Ang->Phi_21 =
            Phi_2a + Phi_2b;
            Ang->Phi_22 = Phi_2a - Phi_2b;
            //Phi_2a and Phi_2a refer to the neutral position of the robot Ang-
            >Phi_21 = (Ang->Phi_21 - PI/2.0) * -1.0;
            Ang->Phi_22 = (Ang->Phi_22 - PI/2.0) * -1.0;

            /////////////////////////
            // Calculation of Phi_3 //
            /////////////////////////

            //Phi_31 calculate (cosine theorem)
            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 and Phi_32 refer to the neutral position of the robot Ang-
            >Phi_31 = PI - Ang->Phi_31;
            Ang->Phi_32 = Ang->Phi_31 * -1.0;

            //Check angle for infinite numbers or NAN
            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;
}
```

```c
//Function for forward kinematics
//The Struct Ang is transferred with the joint angles,
//the transformation matrix and the point to be transformed
//Additionally, the point in which the transformation is saved
//The last parameter x specifies whether joint position 1 or 2 should be used
//If the point cannot be calculated, the function returns 0, otherwise 1
uint8_t fwd_kinematic(ang *Ang, dot *p_src, dot *p_dst, uint8_t x)
{
    //Set angle
    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;

    //Transformation matrices
    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}};

    //Multiply rotation matrices
    //Overloading of operators in C not possible :-(
    matr M02, M03, M04;
    matr_x_matr(M01, M12, M02);
    matr_x_matr(M02, M23, M03);
    matr_x_matr(M03, M34, M04);

    //Multiply the total matrix by the given dot
    matr_x_dot(M04, p_src, p_dst);

    //Test the coordinates of the transformed point for inf and nan
    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;
    }
}

//Conversion of the joint angles [rad] into pulse lengths [µs*2]
//Transfer of the servo number (1-5) and the angle [wheel]
uint16_t ang_to_puls(uint8_t s_nr, double ang, led *Led)
{
    // Linear equation with the servo constants:
    //pulse y = ((pulse(0°)-pulse(-90°))/(Pi/2)) * angle x + pulse(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];
    //Restrict freedom of movement as a stop brake: MIN_MAX_PULSE
    if(pulswidth >= SERVO_const[(s_nr-1)][4])
    {
        pulswidth = (double)SERVO_const[(s_nr-1)][4];
        //Info LEDs: Yellow
        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: Yellow
        Led->led_g[(s_nr-1)] = 0;
        Led->led_y[(s_nr-1)] = 1;
        Led->led_r[(s_nr-1)] = 0;
    }
    else
    {
        //Info LEDs: Green
        Led->led_g[(s_nr-1)] = 1;
        Led->led_y[(s_nr-1)] = 0;
        Led->led_r[(s_nr-1)] = 0;
    }
    //Calculated pulse widths in µs must be multiplied by 2 for the entry in
      TCNT1
    pulswidth *= 2.0;
    //Pulse width rounding (conversion to int)
    return (uint16_t)(pulswidth + 0.5);
}

//Save pulse lengths in Struct Pul
//The last parameter x specifies whether joint position 1 or 2 is to be used 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);
}

//Converts the joystick inputs into a pulse width
//j_val = analog value of the joystick axis (1-1024)
//s_nr = servo number (1-5)
//s_puls_alt = current servo pulse (µs*2)
//s_dir = desired servo direction (1,0)
//led *Led = Reference to the struct for the 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)
{
    //Return value
    uint16_t s_puls_neu;
    //Calculate the pulse change via the joystick input
    uint16_t dpulse;
    if(j_val > (JOY_mid + JOY_dz))
    {
        //Quadratic function, as a linear function leads too quickly to a high servo speed
        //The highest pulse change should be approx. 40
        dpulse = pow(((j_val - JOY_mid) / JOY_div), 2);
        //Add or subtract the pulse change to the current pulse width
          (servo direction)
        if(s_dir == 1)
        {s_pulse_new = s_pulse_old +
        dpulse;} else
        {s_pulse_new = s_pulse_old - dpulse;}
    }
    else if(j_val < (JOY_mid - JOY_dz))
    {
        dpulse = pow(((JOY_mid - j_val) / JOY_div), 2);
        //Add or subtract the pulse change to the current pulse width (servo direction)
        if(s_dir == 1)
        {s_pulse_new = s_pulse_old -
        dpulse;} else
        {s_pulse_new = s_pulse_old + dpulse;}
    }
    else
    {
        s_puls_new = s_puls_old;
    }
    //Restrict the freedom of movement of the servos
    //and set info LEDs (servo stop) if(s_puls_neu >=
    (SERVO_const[(s_nr-1)][4]*2))
    {
        s_puls_neu = SERVO_const[(s_nr-1)][4]*2;
        //Info LEDs: Yellow
        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: Yellow
        Led->led_g[(s_nr-1)] = 0;
        Led->led_y[(s_nr-1)] = 1;
        Led->led_r[(s_nr-1)] = 0;
    }
    else
    {
        //Info LEDs: Green
        Led->led_g[(s_nr-1)] = 1;
        Led->led_y[(s_nr-1)] = 0;
        Led->led_r[(s_nr-1)] = 0;
    }
    return s_puls_new;
}

//Set joint angle to neutral position 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 set to neutral position void
reset_tcp(dot *TCP)
{
    TCP->x = L_1+L_2;
    TCP->y = 0;
    TCP->z = L_3;
}

//Functions for the shift registers (2xHC595)
//HC595 initialize
void shiftr_init()
{
    //Set DS, SH_CP and ST_CP pins to output
    HC595_DDR |= (1<<HC595_SH_CP_POS)|(1<<HC595_ST_CP_POS)|(1<<HC595_DS_POS);
}

//Pulse on shift clock pin (SH_CP or SRCLK)
void shiftr_shift()
{
    HC595_PORT |= (1<<HC595_SH_CP_POS);        //HIGH
```

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

//Pulse on store clock pin (ST_CP or 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);
}

//Write the states of the info LEDs from the "Led" struct bit by bit to the uint16_t 
  variable
//Filling from the right
//Bit 16 (left): unused, always 0
uint16_t shiftr_data(led *Led)
{
	uint16_t data = 0b0000000000000000;

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

	return data;
}

//Function outputs the status of the info LEDs via two shift registers (= 16 bytes)
//Since there are only 15 LEDs, the last byte is always 0
```

```
void shiftr_out(uint16_t data)
{
    //Low level macros to set Data Pin (DS) to HIGH or LOW
    //#define HC595DataHigh() (HC595_PORT |= (1<<HC595_DS_POS))
    //#define HC595DataLow() (HC595_PORT &= (~(1<<HC595_DS_POS)))

    //Send each of the 15 bits serially (MSB first)
    for(uint16_t i=0; i<SHIFTR_BITS; i++)
    {
        //Output via DS depending on the value of the MSB (HIGH
        or LOW) if(data & 0b1000000000000000)
        {
            //MSB is1 1: output HIGH
            HC595_PORT |= (1<<HC595_DS_POS);            //HC595DataHigh();
        }
        else
        {
            //MSB is 0: output LOW
            HC595_PORT &= (~(1<<HC595_DS_POS));         //HC595DataLow();
        }
        //Pulse on shift clock (SH_CP)
        shiftr_shift();
        //Bring the next bit to the MSB position data =
        (data<<1);
    }
    //All 15 bits loaded into the shift register
    //Pulse to Store Clock (ST_CP) -> Output
    shiftr_latch();
}
```