Project Documentation: Building a Stewart-Platform

Hendrik Burgdörfer
Fabian Rühle

Interdisciplinary Center for Scientific Computing
Ruprecht-Karls Universität Heidelberg

01.11.2006 to 01.04.2007

Contents

1 Motivation

A Stewart Platform or Hexapod is a special parallel kinematic machine. The first name is in honor of D. Stewart who was the first to introduce such a concept in 1965. The second name originates from the greek words *hexa: six* and *pod: foot*.
As the name tells, it consists of six actuators whose length can be varied. The actuators are fixed at one end and are connected to a platform at the other. By changing the length of the different actuators one can access six degrees of freedom - three translational and three rotational degrees.
Hexapods are widely used in flight simulators. Here they move the cabin with the passengers according to the movie that is being displayed or the user action that is performed. Hexapods can also be used to mount large telescopes. They can then be repositioned to point to any desired position in the sky. Further applications in industries include the flexible positioning of workpieces under fixed tools and the transport of sensitive or hazardous material that has to be kept in a specific postion at all times during the transport.
The advantages of a hexapod are:

- High dynamics
- Good payload to net weight ratio
- Great postioning precision due to parallel kinematic
- High flexibility with many accessible degrees of freedom

2 Construction of the hexapod

This section describes the different steps that have to be performed to assemble the Stewart-Platform. Basically all actuators that can be bought work either hydraulically or pneumatically. They have a very high payload and accuracy and are designed for industrial use. However, we needed a simple, inexpensive solution that provides the basic features of a hexapod without causing too much costs and overhead. So we decided to assemble the actuators ourselves.

2.1 Actuators

The six actuators basically consist of a thread rod and a nut. One end of the tread rod is connected via a coupler to a cardan joint which in turn connects to a bipolar stepper motor. The nut is attached to one end of a brazen pipe. At the other end of the pipe one finds a ball joint and a self-constructed connector.

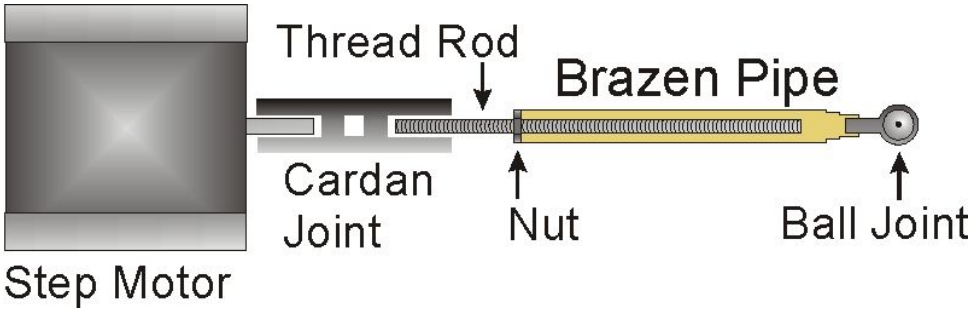


Figure 1: Schematic drawing of one linear actuator

The length of the tread rod is approximately 15 cm. The maximum bent for the cardan joint is 45 degree. The maximum range for the ball joint at the other end is 35 degree. The step motor supports 1.8 degree steps thus there are 200 steps per rotation (for more information see 6.1).

2.2 Platform

The platform was constructed according to *Figure 2*. The actuators are connected to the platform via a skrew that leads through the ball joint of the actuator and an angle bracket at the platform (compare *Figure 3*). To ensure maximum flexibility of the ball joint while still keeping its position fixed we inserted two thin brazen pipes between the nut and the ball joint at either side.

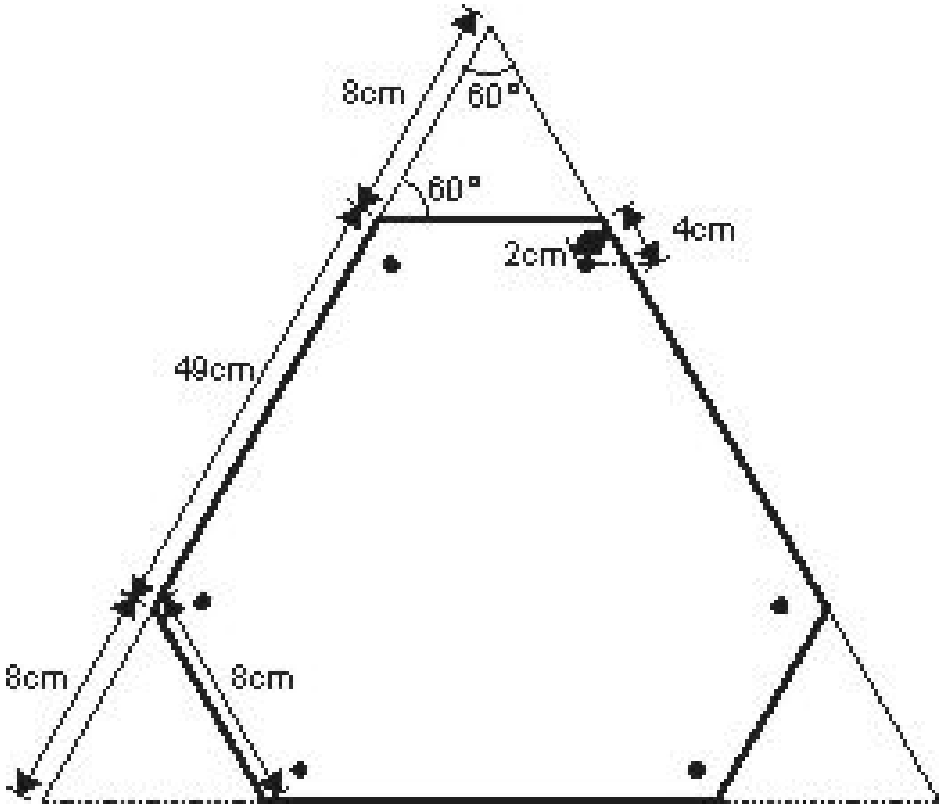


Figure 2: Schematic drawing of the platform

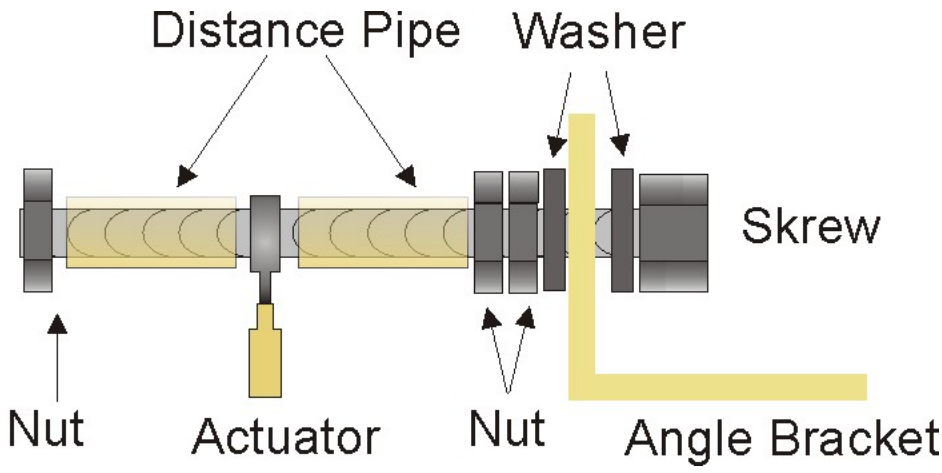


Figure 3: Schematic drawing of the fixation device for the actuators

3 Platform control

The platform is controlled via a graphical user interface at the PC. Data is sent over the serial port (RS232) to a connector [4] (compare *Figure 9*). This connector basically translates from COM to IIC. The IIC bus then goes to a microcontroller [5] (*Figure 10*) which does some consistency checks and translates the signal coming from the UI into a signal readable by the six stepper modules [6] (*Figure 11*) which control the motion of the motors. The signal is distributed to the stepper modules via a switch[6] (*Figure 12*). The GUI was programmed completely in *C#* and the microcontroller in *CCBASIC*. The setup of the different parts can be seen in the connection diagram in *Figure 4*.

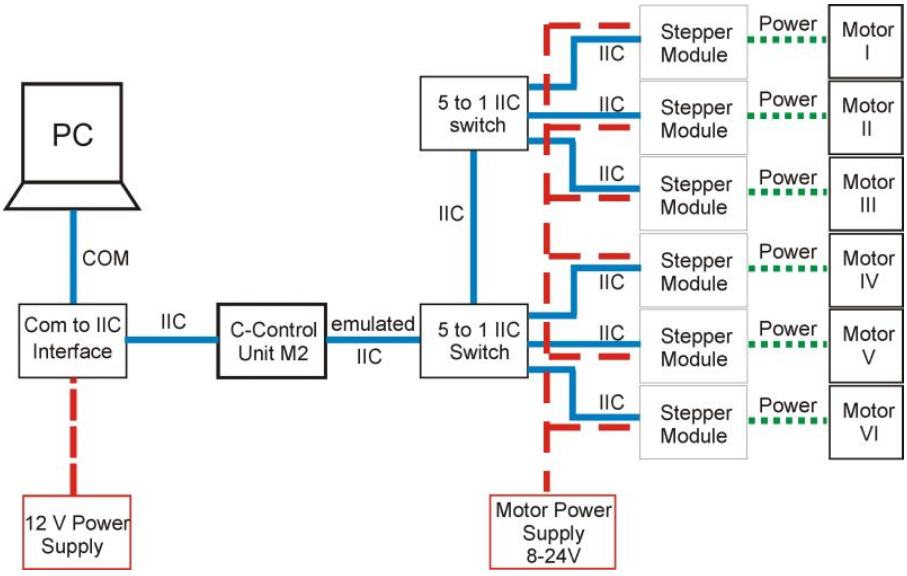


Figure 4: Connection diagram for the Stewart platform components

3.1 Communication protocol

To send the data from the UI to the microcontroller we implemented an easy handshake protocol. By using a microcontroller as arbiter rather than writing direcly from COM on the IIC bus we can ensure that the data sent is correct by counterchecking it on the other side. This filters falsified data that might come from an inconsistent state of the computer (e.g. during a system crash) or from uncontrollable bit flips during communication. These data could lead to repositioning commands that are mechanically impossible and could thus result in damage of the platform or the motor or even both. Or it could lead to an irreversible reprogramming of the OTP memory of a stepper module which renders the chip unuseful.

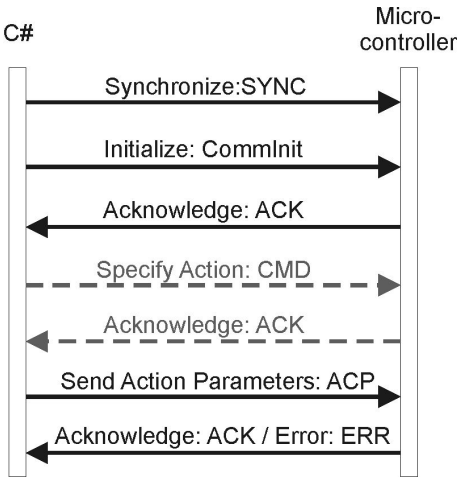


Figure 5: Communication protocol

First C# sends a synchronization byte in order to set the microcontroller into a well-defined state. Then a byte that initializes the communication is sent and confirmed upon derivation. After that C# sends for special actions (i.e. everything but a move motor command) a command byte indicating what action shall be

performed. After receiving the acknowledgement from the microcontroller, C# sends action-specific parameters. At the end the microcontroller acknowledges again and the communication cycle is completed. *Figure 5* illustrates the process.
"0x" indicates hexadecimal representation. *Table 1* provides a lsit with the allowed values for the bytes introduced in *Figure 5*. The action parameter itself is one byte containing information

Byte	Value (HEX)	Meaning
Sync	0x10	Brings microprocessor into a well-defined state
CommInit	0xFF	Indicates communication request
ACK	0x00	Acknowledgement of received data
CMD	- 0xC0 0xE0 0xF0	If left out, action move motor is performed Sends status request to wake up motors Causes the microcontroller to shut down Indicates a status report request
ERR	0x11 0x22	Wrong number of '1' in action parameter Wrong parity of action parameter

Table 1: List of communication bytes

about the motor that shall be addressed and the direction of motion. With bit zero being the least significant bit, it reads as follows:

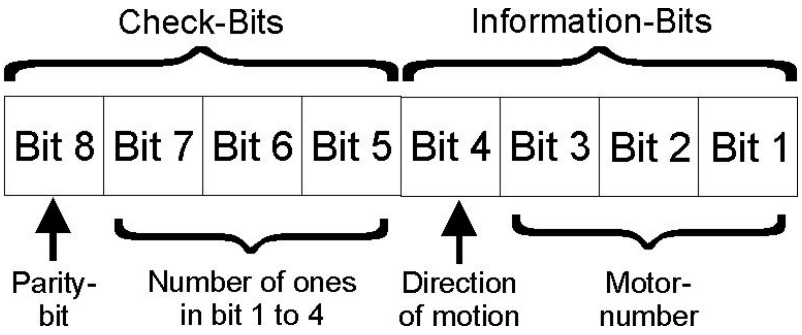


Figure 6: Communication byte composition

Bit	Meaning	Value (Binary)	Semantic
(2:0)	Motor Number	000 & 111 001 010 011 100 101 110	N/A Motor1 Motor2 Motor3 Motor4 Motor5 Motor6
3	Direction of motion	0 1	counterclockwise (up) clockwise(down)
(4:6)	Number of '1'		Binary encoding of number of ones in bits (3:0)
7	Parity		Ensures even number of ones in bit (0:7)

Table 2: List of communication bytes

3.2 Platform Commisioning

Firstly, the motor power supply has to be connected to an appropriate 12V DC power supply. After that the COM → IIC connector has to be plugged in, also supplied with 12V. When SHARP is closed or after a long idle time, the C-Control or the connector shuts down. In this case, the power supply must be cut and reinstalled in order to have the C-Control reboot. If not all motors are spinning correctly, probably the power supply does not provide enough current respectively power.

4 The ICP

On the microprocessor of the C-Control Unit the ICP (Intercommunicational Program) is stored. The task of ICP is to transform information received by the terminal program (SHARP) into information readable for the IIC stepper modules connected to the C-Control and vice versa.
If the C-Control Unit is jumpered correctly (see [3] for more information), ICP runs automatically as soon as the pins GND and V5+ are connected to a charge pump.
First of all ICP sends a 'Get Full Status1' command to all IIC stepper modules and receives the information sent back by them. This is necessary to 'wake up' the modules i.e. to get them out of the shutdown mode which they are set in automatically after connecting the modules to the mains supply (for more information see [3]). The information they send back is unimportant at this time, the only purpose is to wake up the IIC stepper modules.
After that ICP sends the 'SetMotorParam' command to set the motor parameters. 9 parameters are set within this procedure (please see [3] for detailed information):
'Irun' and 'Ihold' are both set to 0xD.
'Vmax' and 'Vmin' are both set to 0xF.
'SecPos' is set to 0 and so is 'Shaft'. 'Acc' is set to 0xF.
'AccShape' is set to 0, and 'StepMode' is set to 1/16.
The parameters are the same for each stepper.
After the motor parameters are set, ICP jumps into an endless loop where it reads the incoming data from the terminal program (SHARP) continuously.
If a data byte 0xFF (0x means hexadecimal) is received, ICP leaves the loop and sends 0x00 to the terminal program. Then it reads the byte that comes in immediatley next; normally that would be a positioning byte from SHARP . Such a byte consists of four information bits - namely the four least significant bits - and four check bits.
The three least significant bits encode the number of the IIC stepper module the information should be sent to, it reaches from binary 001 to 110.
The 4^{th} bit encodes the direction the stepper should turn. (0 for clockwise and 1 for counterclockwise rotation).
The 5^{th}, 6^{th} and 7^{th} bit encode the number of information bits that are 1 in binary representation, for example .010.... if two of the information bits are 1.
The most significant bit is a parity bit such that the number of bits that are 1 in the whole byte is even.
After receiving the positioning byte, ICP generates the adress of the IIC stepper module out of the three LSB and checks whether the four check bits fit the information bits. It then resets the position counter of the corresponding module, i.e. the actual position is set to 0. The next command sets the target position to 1 respectively -1 (binary complement of 1), depending on the direction of rotation that was received with the 4^{th} information bit from the terminal program.
After finishing the communication with the IIC stepper module, ICP sends 0x00 to the termi-

nal program to signal readiness to receive a new command and jumps back into the endless loop.

In case that the binary number of information bits that are 1, encoded in the check bits, doesn't fit the number of information bits that are in fact 1, ICP doesn't communicate with any IIC stepper module but sends back 0x11 to the terminal program. After that, it jumps back into the endless loop.
In case that the number of bits that are 1 in the whole byte is odd, ICP doesn't start a communication with any module either but sends back 0x22 to the terminal program and jumps back into the endless loop.

Beside the positioning bytes, there are three special bytes that can be sent by the terminal program immediately after the 0xFF byte that makes ICP break the endless loop. These special bytes are 0x10, 0xE0 and 0xF0.
The 0x10 byte makes the ICP jump back into the endless loop. ICP is constructed in a way that whenever 0x10 is received it jumps back into the loop, so after sending 0x10 one can be sure which state ICP is in. That is important especially in cases one assumes that the terminal program and the ICP aren't synchronized anymore for whatever reason. The first byte sent by the terminal program to ICP should always be 0x10.
0xE0 is the kill signal which terminates ICP immediately.
0xF0 is the information command. It allows the terminal program to get several information about temperature, electrical defects etc. from an IIC stepper module. When receiving 0xF0, ICP sends back 0x00 and reads the byte that comes in immediately next. ICP uses that byte only to filter out the module number from the three LSB, so the terminal program can send a usual positioning byte for the stepper it wants to get the information from. After that, ICP starts the communication with the corresponding IIC module and sends the two commands 'Get Full Status1' and 'Get Full Status2'. The information received from the module is stored in 12 bytes and sent to the terminal program after sending a 0x00 byte first. Following information is encoded in the 12 bytes (see [3] for detailed information.): commands.

1st byte:	adress of the module
2nd byte:	Irun(3:0) Ihold(3:0)
3rd byte:	Vmax(3:0) Vmin(3:0)
4th byte:	AccShape StepMode(1:0) Shaft ACC(3:0)
5th byte:	VdReset StepLoss ElDef UV2 TSD TW Tinfo(1:0)
6th byte:	Motion(2:0) ESW OVC1 OVC2 1 CPFail
7th byte:	ActPos(15:8)
8th byte:	ActPos(7:0)
9th byte:	TagPos(15:8)
10th byte:	TagPos(7:0)
11th byte:	SecPos(7:0)
12th byte:	1 1 1 1 1 1 SecPos(10:8)

After the transmition is finished, the ICP jumps back to the endless loop to wait for further commands

5 Graphical User Interface

5.1 General Remarks

The graphical user interface was programmed with C#. Its name is SHARP in honor of the language that we used for programming. At the same time it is an acronym for ***S**upported **H**exapod **A**ctuator **R**epositioning **P**rogram*. The user can preselect a position for the Stewart Platform which is illustrated graphically. Once the postion is confirmed, C# checks whether the position is reachable from the current one (i.e. whether the actuators can be lowered or raised enough). If this is the case the new actuator length is computed from the desired position. This length is then translated in a number of rotations which is sent via the RS232 interface to the microcontroller. Otherwise the user is asked to specify a new valid position.

5.2 The Menu Bar

5.2.1 File

The following subchapter will shorty introduce SHARP and its functionalities

- **Open Position**: This allows the user to open a previously saved position file. Note that the position will only be loaded into the graphical preview (for more information see *Section 5.3*). In order to have the Stewart-Platform go to this position the "Move platform" button must still be pressed.
- **Save Position As...**: This allows for saving the current position of the platform. Note that here again only the postion that is currently selected in the preview is saved, not the actual position the platform is in (for more information see *Section 5.3*).
- **Settings...**: Here one can specify the COM-port which connects to the actuator control and the default path for opening and saving files
- **Bullriding Mode**: The user enters a timespan in seconds during which SHARP computes random positions for the platform.
- **Exit**: Closes the program.

5.2.2 Debug

- **Recovery Mode**: As there are no sensors at the platform it is impossible to find its current position once the program files storing this information are in an inconsistent state, which might come from a system crash or improper treatment of the program files (for more information see *Section 5.3*). In this case the user has to reset the platform manually to a predefined position (all actuators lowered as much as possible). This must be done by specifying manually the amount and directions of rotations for each actuator. Upon leaving the recovery mode the program files are updated and SHARP can be used normally again.
- **Motor status**: This provides a list of data characterizing the motors of the actuators. Among these information are temperature, current and voltage supply and information about problems and errors within the step motor.

5.2.3 Help

- **Sharp Help**: Explains the functionality of SHARP with screenshots and comments.
- **About**: Displays version and program information of SHARP .

5.3 Program files

Here the principal way how SHARP stores the platform information is explained.

5.3.1 *.pos files

These are user-generated position files. They store in plain ASCII the position vector of the head of each actuator followed by the amount that has been rotated or translated with respect to the default position:

x position of actuator 1
y position of actuator 1
z position of actuator 1
x position of actuator 2
⋮
z position of actuator 6
Amount of translation about x axis
Amount of translation about y axis
Amount of translation about z axis
Amount of rotation about x axis
Amount of rotation about y axis
Amount of rotation about z axis

5.3.2 config.cfg file

This file is generated by SHARP and here it stores *all* program information. This is also the file used for the initial load at the startup of the program. Besides the information of the preview, it contains information about the user-specified program settings and the position of the real actuators. This file should never be manipulated, replaced, edited or deleted, as this results in unreconstructable loss of information of the platform position! The information is also stored as ASCII characters as follows:

COM Port connecting to the microprocessor
Default path for file dialogs
Information about the position of the platform stored in the same way as in Section 5.3.1
Information about the position of the preview stored in the same way as in Section 5.3.1

5.4 Program Structure

The program consists mainly of five form classes for the several GUI's. Then there is an abstract class which holds the basic parameters and is inherited by the graphical platform class and the real platform class. Additionally an actuator class which has actuator specific members and methods is implemented. Furthermore there is a motor class which implements methods for

communicaton with and movement of the motors. A vector class with overridden mathematical operators simplifies storing and manipulating the position information. The UML diagram is shown below in *Figure 7*.

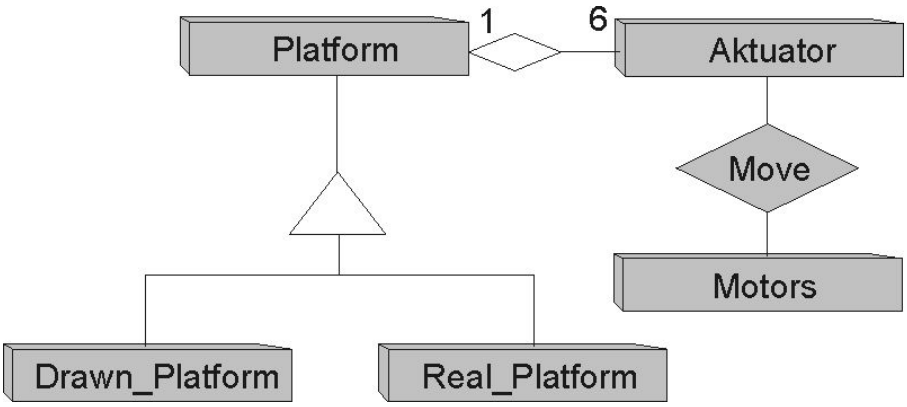


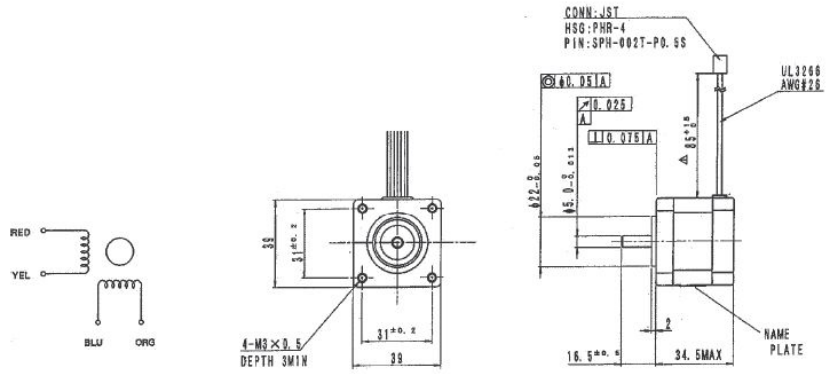
Figure 7: Simplified UML diagram for SHARP

5.5 Algorithmic Realization

Each actuator is represented by a threedimensional vector. The problem is that from the knowledge of the total amount of translation and rotation about each axes, the position of the platform cannot be reconstructed as rotations about different axes *do not* commute. Thus each motion command must be processed immediately.
The new position is computed for translations by simply adding a constant offset to the respective coordinate. For rotations all vectors representing the actuators are moved to the origin, multiplied with the standard rotation matrices in $\mathbb{R}^3$ and then brought back to the real position.
For the preview, the same action is performed. Subsequently the vectors are projected into a twodimensional plane with the third axis pointing at 45°with a contraction factor of $\frac{\sqrt{2}}{2}$ to suggest a threedimensional effect.
The so computed new position is then translated into a rotation amount and direction and written on the RS232 bus in plain ASCII as explained in *Section 3.1*

6 Appendix

6.1 Step Motor Data Sheet



ITEM	SPECIFICATION
RATED VOLTAGE	4.2 V
RATED CURRENT	0.6 A / Phase
MAX. INPUT	5.0 W
STEP ANGLE	1.8 °
MAX. HOLDING TORQUE	0.196 Nm
WINDING RESISTANCE	7.0 Ohm ± 10 %
WINDING INDUCTANCE	10.6 mH
TEMPERATUR RISE	80 °C
WEIGHT	220 g

eine Vollumdrehung = 200 Schritte

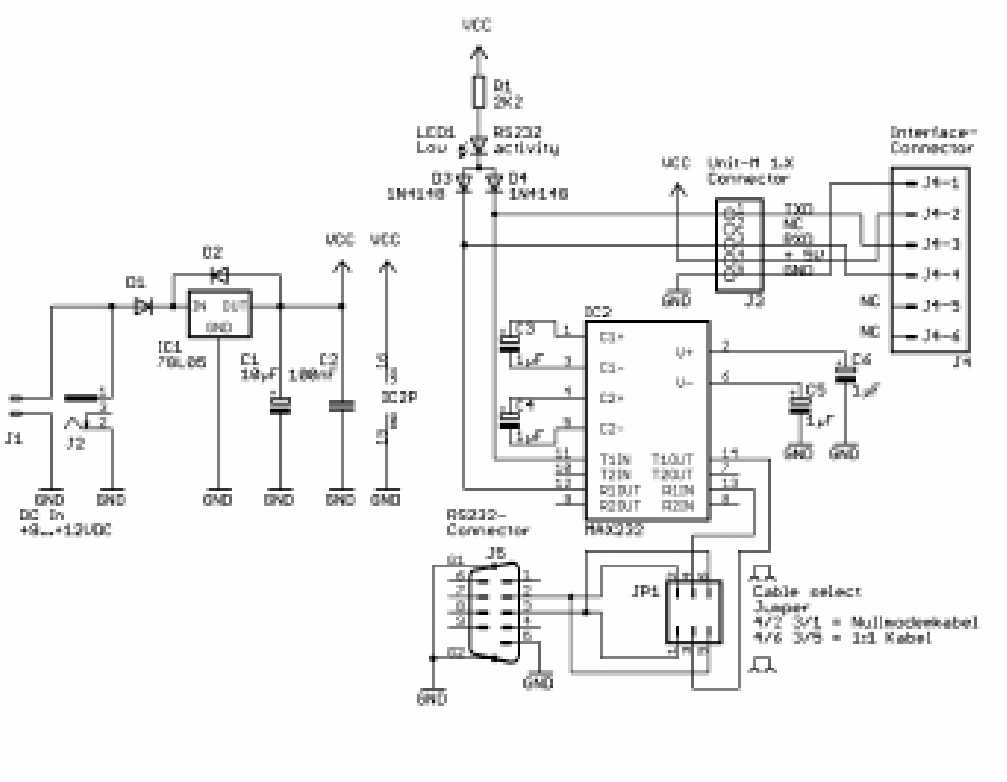
Figure 8: Data sheet of the step motor [1]

6.2 Parts

This is a list of several parts that were used for the construction of the Stewart-Platform.

article	**quantity**	**article number at Conrad®**
Step Motor SM 42051	6	198398-62
I^2C Stepper Module	6	198266-62
C-Control Unit M 2.0	1	198822-62
Adapter RS232 → I^2C	1	198834-62
I^2C-Bus Wire	8	198876-62
Cardan Joint	6	226467-62
Inset Piece ⊘ 5 mm	6	226505-29
Inset Piece M4	6	226513-29
Thread Rod	1	237108-62
Brazen Pipe	1	221797-62
Ball Joint	6	216488-62

6.3 RS232 to IIC Connection Module Data Sheet



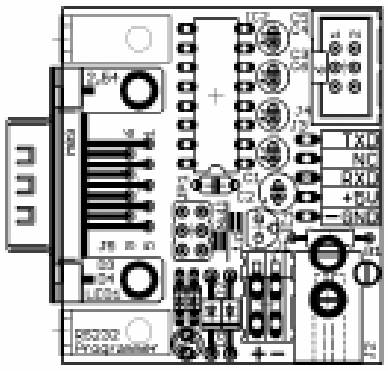


Figure 9: Data sheet of Connection Module [4]

6.4 C Control Unit 2 Data Sheet

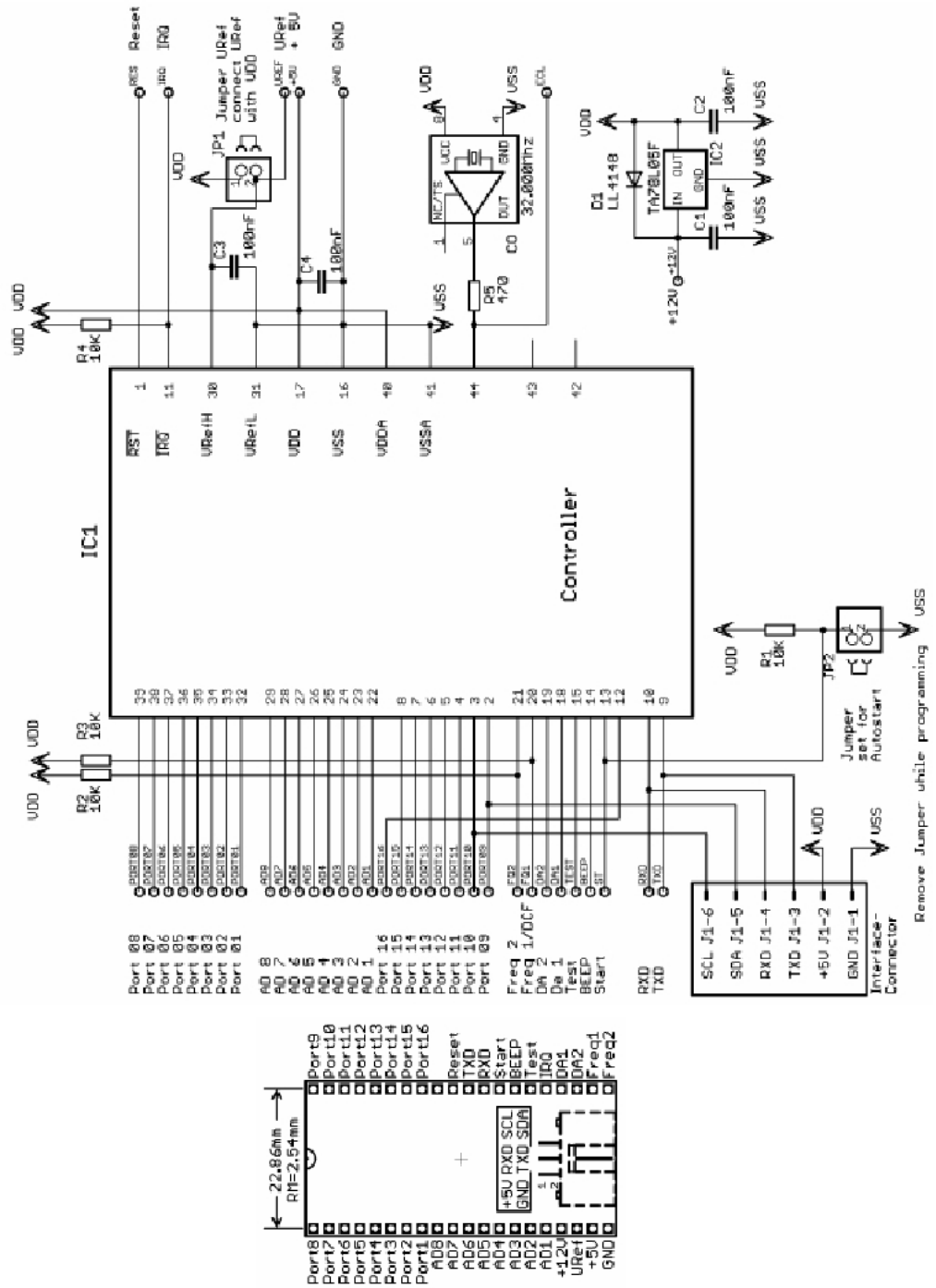
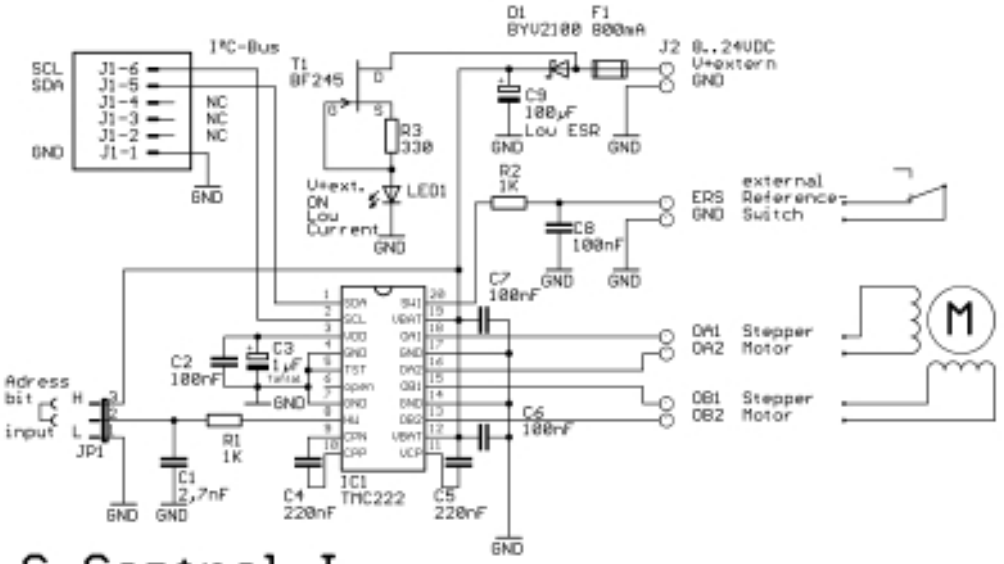


Figure 10: Data sheet of C Control [5]

6.5 Interface Module Data Sheet

Technische Daten:

Betriebsspannung:	8...24VDC
Spulenstrom programmierbar:	max. Strom/Phase 800mA
Temperatur Bereich:	–50....+150 C°
Schrittfrequenz:	bis 1kHz
Schrittweite:	bis 4 Bit Microstepping (1/16)
Positionszähler:	16 Bit
Anschlussklemmen:	1 mm²
Abm. (L x B x H):	47 x 42 x 12 mm



C-Control I
I²C-Bus Stepper Driver

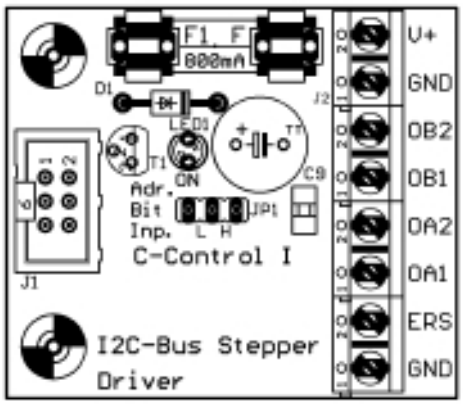


Figure 11: Data sheet of Interface Module [6]

6.6 IIC Bus Switch Data Sheet

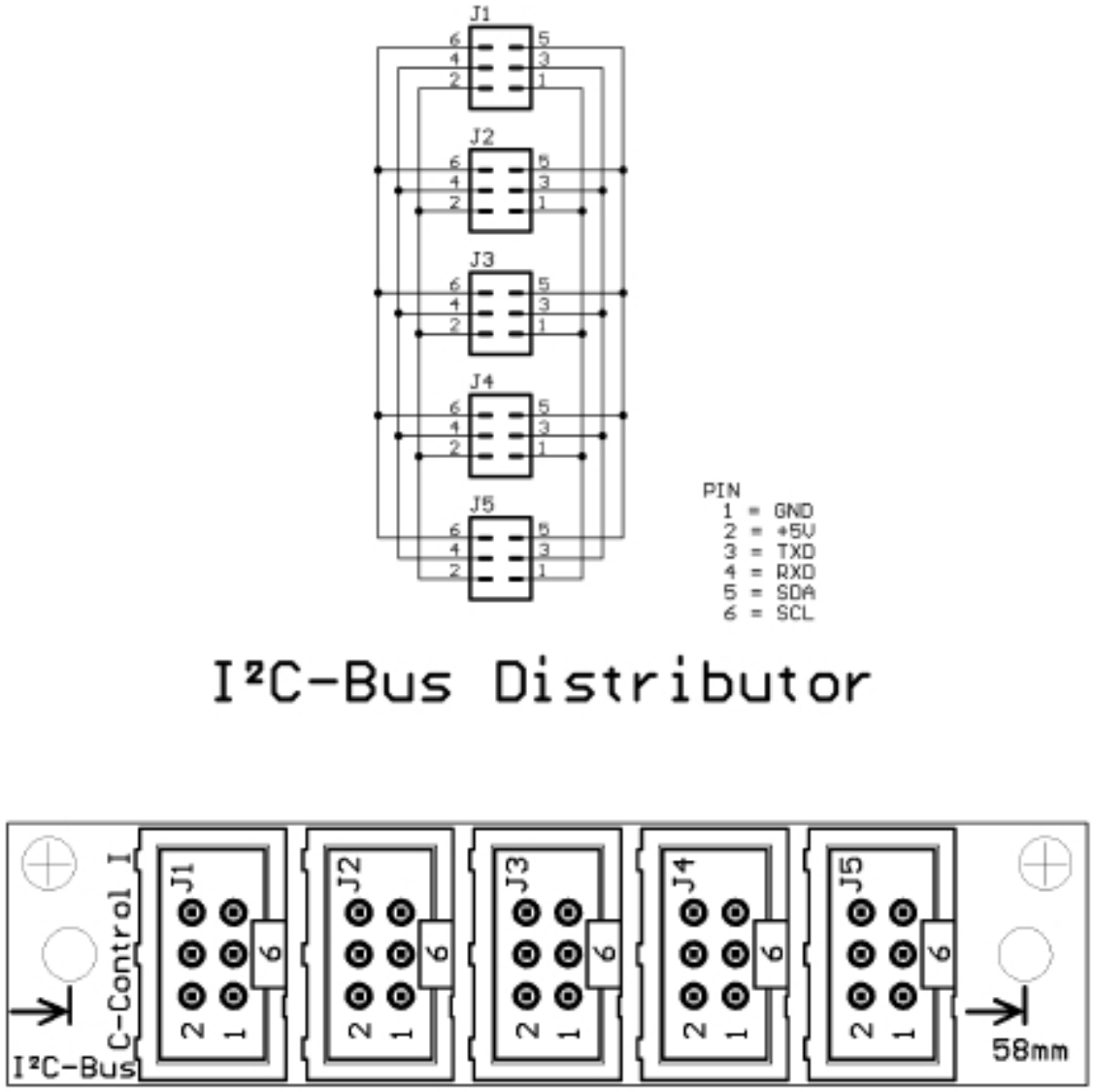


Figure 12: Data sheet of the Switch [6]

References

[1] Step motor data sheet. Accessed 03/07. http://www2.produktinfo.conrad.com/datenblaetter/175000-199999/198398-da-01-en-Schrittmotor_SM_42051.pdf

[2] TMC222 data sheet. Accessed 03/07. http://www.c-control-support.net/downloads/tmc222.pdf

[3] C-Control CCBASIC Manual. Accessed 03/07. http://www.c-control-support.net/downloads/MANUAL_M2.0_M1.2.pdf

[4] C-Control Connection Interface Module. Accessed 03/07. http://www.c-control-support.net/downloads/G_MANUAL_BA002_PROGRAMMER_A5.pdf

[5] C-Control Unit 2 Hardware Manual. Accessed 03/07. http://www.c-control-support.net/downloads/G_MANUAL_BA001_UNITs_A4.pdf

[6] IIC Bus Interface Module. Accessed 03/07. http://www.c-control-support.net/downloads/BA005.pdf

