Institut für Robotik und Prozessinformatik

Technische Universität Braunschweig

Prof. Dr. J. Steil

ROBOTIKPRAKTIKUM

Versuch 2

Roboterprogrammierung

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieser Versuch beschäftigt sich mit der Programmierung eines realen Roboters. Im Versuch 1 wurde mit Hilfe eines Roboter-Simulationssystems der Stäubli RX60-Roboter simuliert. Nachdem dort bereits eine Palettierung gezeigt wurde, soll nun für den realen RX60 ein solches Palettierungsprogramm in *Zero++* geschrieben werden. Dazu gehört u.a. auch das Bestimmen eines lokalen Palettenframes aufgrund drei ge*teach*ter Positionen.

Kapitel 2

Der Stäubli RX-60-Roboter

2.1 Inbetriebnahme

Der Hauptschalter des Roboters befindet sich am Steuerschranks (großer grauer Drehknopf). Nach dem Einschalten des Roboters wird zunächst die Hardware initialisiert und überprüft. Dieser Vorgang dauert ein wenig und kann auf dem Terminal des Roboters verfolgt werden.

Der Roboter ist mit einem Knickschutz ausgestattet, der bei einer Kollision der Hand mit einem Hindernis den Roboter sofort anhält. Dieser Knickschutz wird mit Druckluft betrieben. Zum Aktivieren des Knickschutzes befinden sich auf der hinteren Seite des Arbeitstisches drei Absperrventile (siehe Abbildung 2.1):

- links unten befindet sich das Hauptventil, welches die gesamte Druckluftzufuhr regelt
- das mittlere Ventil steuert die Luftzufuhr des Knickschutzes
- mit dem rechten Ventil wird der Druck für die Greiferbacken geregelt.

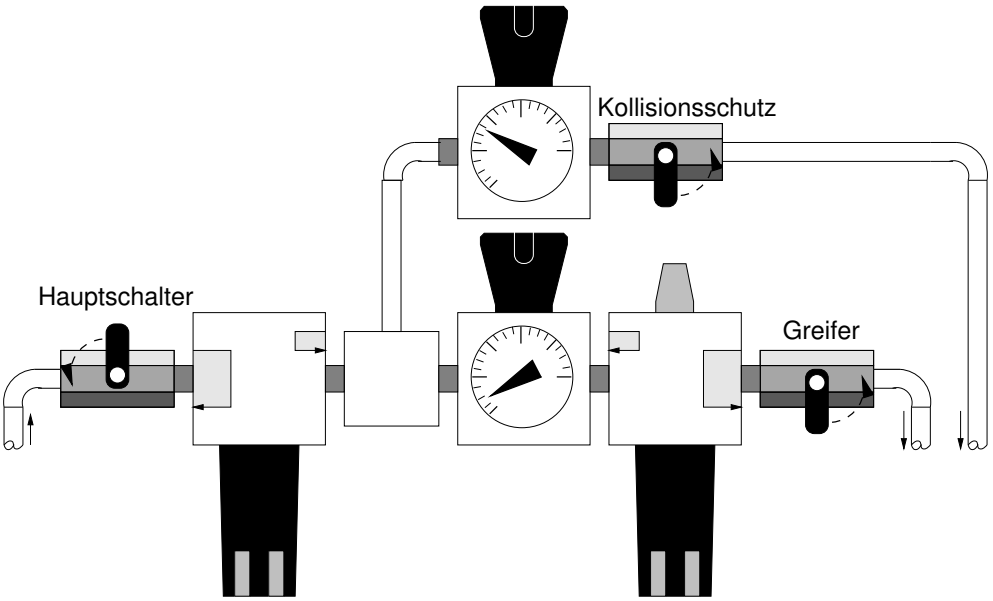

Abbildung 2.1: Druckluft-Absperrventile

Beim Einschalten des Roboters ist der Knickschutz meist nicht arretiert. Nachdem das Druckluftabsperrventil des Knickschutzes geöffnet wurde, muss deshalb der Greifer des Roboters vom Flansch weggezogen werden. Dabei muß er so gedreht werden, daß keine Druckluft mehr entweicht und die Hand fest im Knickschutz sitzt.
Anschließend sollte der Knickschutz mit dem separaten Knopf (blaue Taste mit roter LED) aktiviert werden. Die LED sollte nun nicht mehr leuchten. Dieser Vorgang ist nur erforderlich, falls der Roboter bereits eingeschaltet ist, bevor der Knickschutz aktiviert wird.

Hinweis:
Sollte der Roboter mit einem Hindernis kollidieren und dabei der Knickschutz aktiviert werden, so muß der Roboter in der Regel manuell in eine freie Position bewegt werden, bevor die Hand wieder in den Knickschutz arretiert werden kann.

Dieser Vorgang sollte vom betreuenden HiWi vorgenommen werden !

Um das zischende Geräusch der entweichenden Druckluft zu beenden, kann zuvor das Druckluftventil des Knickschutzes geschlossen werden.

Nach dem Einschalten des Roboters wird der Roboterarm noch nicht mit Strom versorgt, Deswegen muß der Arm aktiviert werden, indem zunächst auf dem Teach-Panel die Taste COMP/PWR gedrückt wird. Daraufhin sollte am Steuerschrank eine grüne Taste (ARM POWER ON) blinken, welche dann zu drücken ist. Sollte dies nicht der Fall sein, ist zu überprüfen, ob einer der drei Notausschalter bzw. die Entriegelungstaste des Knickschutzes noch arretiert ist. In diesem Fall, ist der Vorgang nach Lösen des Notausschalters zu wiederholen.

2.2 Bedienung des Teach-Panels

Das Teach-Panel dient zur interaktiven Steuerung des Roboters. Das Panel besitzt eine Reihe von Tasten (vgl. Bild 2.2), von die wichtigsten im folgenden beschrieben werden.

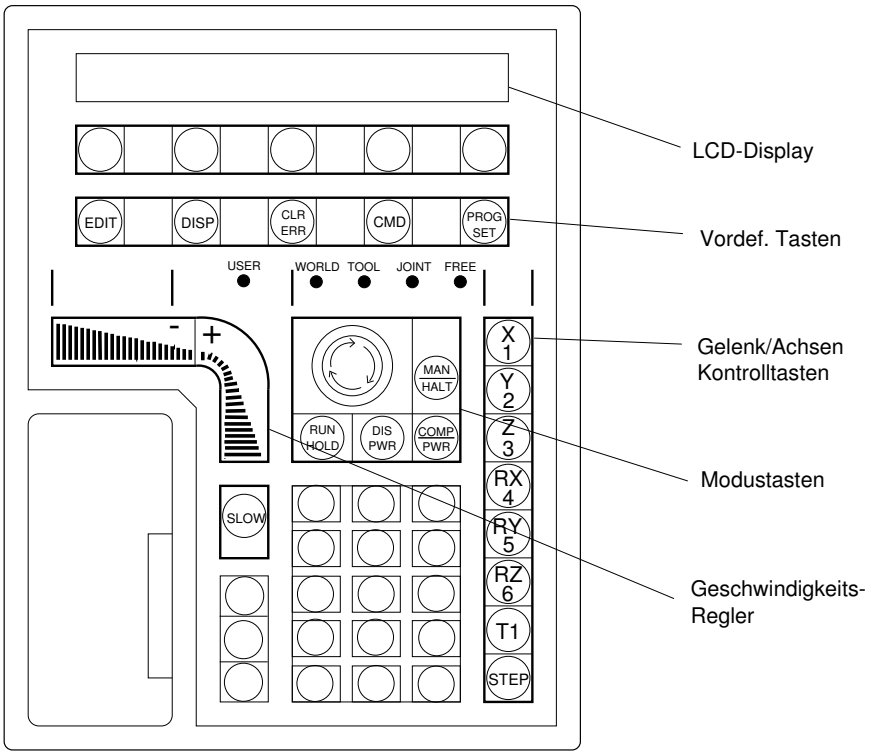

Abbildung 2.2: MCP

2.2.1 Moduswechsel

Der Roboter unterscheidet drei Modi:

- **deaktiviert:** Der Roboterarm hat keinen Strom (s.o.)
- **Comp-Modus:** Mit der COMP/PWR-Taste kann der Roboter in den Comp-Modus geschaltet werden, in dem er entweder über Befehle gesteuert werden kann, die über das Terminal eingegeben oder in Form eines Programms ausgeführt werden.
- **Interaktiver-Modus:** Der Roboter kann mit dem Teach-Panel bewegt werden. In diesen Modus gelangt man mit der MAN/HALT-Taste.

2.2.2 Roboterbewegung

Um den Roboter zu bewegen, muss er mit der MAN/HALT-Taste des Modusbereiches (siehe Abb. 2.2) in das gewünschte Referenzsystem gewechselt werden. Das aktuell eingestellte System wird durch die Leuchtdioden WORLD, TOOL, JOINT oder FREE

angezeigt. Durch wiederholtes Drücken der MAN/HALT-Taste kann zwischen den verschiedenen Bezugssystemen gewechselt werden.

Anschließend kann mit den Gelenk-/Achsen-Kontrolltasten das zu bewegende Gelenk bzw. die gewünschte Bewegungsrichtung selektiert werden. Die Bewegung erfolgt mit den Geschwindigkeitsreglern.
Das Öffnen und Schließen des Greifers erfolgt analog zur Bewegung mit der Taste T1 sowie den Geschwindigkeitsreglern.

2.2.3 Koordinatenanzeige

Zum Anzeigen der aktuellen Roboterkoordinaten bzw. -gelenkstellungen dient die **DISP**-Taste. In der Anzeige erscheint ein Menü, aus dem mit den darunterliegenden Tasten entweder die Darstellung der Roboterposition mit Gelenkstellungen (*JOINT VALUES*) oder kartesischen Koordinaten (*WORLD LOCATION*) ausgewählt werden kann.

Hinweis:
Bei der Koordinatenanzeige handelt es sich bei der im Display angezeigten Orientierung **nicht** um RPY (auch wenn dort y=..., p=... und r=... steht), sondern um Euler-Winkel! Das heißt, y=φ_y, p=φ_p und r=φ_r bedeutet in Euler-Notation:

$$\begin{aligned}\mathbf{Euler}(\varphi,\theta,\psi) &= \mathbf{Rot}(z,\varphi)\cdot\mathbf{Rot}(y,\theta)\cdot\mathbf{Rot}(z,\psi)\\ &= \mathbf{Rot}(z,\varphi_y)\cdot\mathbf{Rot}(y,\varphi_p)\cdot\mathbf{Rot}(z,\varphi_r)\ .\end{aligned}$$

2.3 Abschalten des Roboters

Bevor der Roboter abgeschaltet wird, muss er in die „ready“-Position bewegt werden. Dazu muss auf dem Terminal der Befehl

DO READY

eingegeben und mit <RETURN> bestätigt werden. Anschließend kann der Strom mit dem roten Hauptschalter an der Rückseite der Robotersteuerung abgeschaltet werden.
Zusätzlich muß das Hauptventil (links) der Druckluftzufuhr abgesperrt werden.

Kapitel 3

Windows und Visual-C++

3.1 Starten von Windows

Bevor es los gehen kann, müssen die Hinweise im Abschnitt 2 beachtet werden.

3.2 Das Praktikumsprojekt

Zum Start von Visual C++ befindet sich auf dem Desktop das Icon „MS-Dev“. Nach einem Doppelklick auf dieses Symbol erscheint das Visual-C++ Fenster. Mit dem Menüpunkt *Datei/Arbeitsbereich öffnen ...* wird das Praktikumsprojekt geladen. Das vorbereitete Programmgerüst `Palette.dsw` befindet sich in folgendem Verzeichnis:
`M:\Prak\Gruppe`*n*[1]
Im Programmfenster wird daraufhin die (erste) Stelle angezeigt, an der Sie das Programm ergänzen müssen.
Das Projekt kann mit der Funktion *Datei/Arbeitsbereich speichern* auf der Festplatte gesichert werden.

3.3 Ausführen des Programms

Das Compilieren des Programm kann entweder mit dem Menüpunkt *Erstellen/erstellen...* oder durch die Taste F7 erreicht werden. Wurde das Programm fehlerfrei übersetzt, kann es nun ausgeführt werden. Dazu wird der Menüpunkte *Erstellen/ausführen von ...* aufgerufen bzw. der Tastenkombination Ctrl-F5 gedrückt.

Hinweis:

Denken Sie daran, den Server auf dem Terminal des Stäubli-Roboters zu starten, da sonst keine Befehle zum Roboter übertragen werden können ! (Kapitel 5)

[1] *n* steht für die Nummer ihrer Praktikumsgruppe

3.4 Fehlersuche mit dem Debugger

Sollte sich Ihr Programm nicht wie erwartet verhalten, können Sie das Programm auch im Debugger starten. Dazu ist das Programm nicht wie oben beschrieben mit mit Ctrl-F5 sondern mit dem Befehl *Erstellen/Debug starten/ausführen* (F5) zu starten. Beim Ablauf des Programms erscheint nun ein weiteres Fenster in welchem sich Knöpfe zur Bedienung des Debuggers befinden. Die wichtigsten Funktionen des Debuggers sind:

- Haltepunkt einfügen/löschen (F9):
  Setzt bzw. löscht einen Haltepunkt in der Zeile, in der sich der Cursor befindet. Bei gesetztem Haltepunkt wird die Ausführung des Programms gestoppt sobald diese Zeile erreicht wird.
- In Aufruf springen (F11):
  befindet sich der aktuelle Programmzeiger bei einer Funktion wird in diese hineingesprungen
- Aufruf als ein Schritt (F10):
  Führt die aktuelle Funktion im Programm in einem Schritt aus.
- Ausführen bis Rücksprung Ctrl-F11:
  Die aktuelle Funktion wird bis zum Rücksprung zum aufrufenden Programteil ausgeführt.
- Schnellüberwachung (Shift-F9):
  Zeigt den Wert eines Datenelementes in einer Liste an. Beim Anklicken des Variablennamen mit der linken Maustaste erscheint der aktuelle Wert in einem kleinen Popup-Fenster.
- Debugger beenden (Shift-F5):
  Beendet den Debugger

Kapitel 4

Klassen für Vektoren, Matrizen und Frames

Dieses Kapitel behandelt die Klassen, die *ZERO++* für das Arbeiten mit Vektoren, Matrizen und Frames zur Verfügung stellt.

4.1 Klasse: HVECTOR

Die Klasse HVECTOR definiert die Konstruktoren und Destruktoren, sowie die Operatoren für das Arbeiten mit Vektoren. Der Aufruf des Destruktors wird von "C++" automatisch ausgeführt. Nur bei dynamisch erzeugten Instanzen (*new HVECTOR(...)*) muß der Destruktor mittels *delete* aufgerufen werden.

Beispiel:

```
[...]
    HVECTOR *v = new HVECTOR (2.0, 1.0, 2.5);
[...]
    delete v ;
```

4.1.1 Konstruktoren

Folgende Konstruktoren sind in dieser Klasse definiert:

1. *HVECTOR (HVECTOR& source);*
   Legt eine Kopie des bereits bestehenden Vektors *source* an.

2. *HVECTOR (double *array);*
   Ein Vektorobjekt wird erzeugt und mit den Werten aus dem Feld *array* initialisiert. Dabei gilt folgende Zuordnung:

   $$x = array[0], y = array[1], z = array[2], w = array[3]$$

3. *HVECTOR (double x = 0.0, double y = 0.0, double z = 0.0, double w = 1.0);*
   Es wird ein Vektorobjekt erzeugt und mit den übergebenen Werten initialisiert. Die im Prototypen angegebenen Werte werden dann verwendet, wenn der

entsprechende Parameter beim Aufruf nicht angegeben worden ist. Um beispielsweise ein Vektorobjekt zu erzeugen, dessen x und w Komponenten auf 1.0 gesetzt sind, reicht es, *HVECTOR neu (1.0);* zu schreiben.

4.1.2 Operatoren

Die nachfolgenden Operatoren können auf Objekte des Types *HVECTOR* ausgeführt werden:

void operator= (const HVECTOR&)

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ w_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \\ w_2 \end{pmatrix}$$

Beispiel:

```
HVECTOR v1, v2 ;

v1 = v2 ;
```

int operator== (const HVECTOR&) const

Vergleicht zwei Vektoren miteinander.

Beispiel:

```
HVECTOR v1, v2 ;

if( v1 == v2 )
{ [...] }
else
{ [...] }
```

HVECTOR operator+(const HVECTOR&) const

$$\vec{v} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ w_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \\ z_2 \\ w_2 \end{pmatrix}$$

Beispiel:

```
HVECTOR v, a, b ;

v = a + b ;
```

HVECTOR operator-(const HVECTOR&) const

$$\vec{v} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ w_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \\ z_2 \\ w_2 \end{pmatrix}$$

Beispiel:

```
HVECTOR v, a, b ;

v = a - b ;
```

double operator*(const HVECTOR&) const

$$scalar = \frac{x_1 x_2 + y_1 y_2 + z_1 z_2}{\|\vec{v}_1\| \, \|\vec{v}_2\|}$$

Beispiel:

```
HVECTOR a, b ;
double scalar ;

scalar = a * b ;
```

HVECTOR operator* (double) const

Multipliziert die einzelnen Komponenten eines Vektors mit *scalar*.

Beispiel:

```
HVECTOR v ;
double scalar ;

v = v * scalar ;
```

HVECTOR operator^(const HVECTOR&) const

$$\vec{v} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ w_1 \end{pmatrix} \times \begin{pmatrix} x_2 \\ y_2 \\ z_2 \\ w_2 \end{pmatrix}$$

Beispiel:

```
HVECTOR v, a, b ;

v = a ^ b ;
```

double operator!(void) const

Liefert die Länge des Vektors (Euklidische Norm im $\mathcal{R}^3$) als *double* zurück.

$$norm = \sqrt{x^2 + y^2 + z^2}$$

Beispiel:

```
HVECTOR v ;
double length ;

length = !v ;
```

HVECTOR operator∼(void) const

Normiert den Vektor v.

$$\vec{v} = \frac{\vec{v}}{\|\vec{v}\|}$$

Könnte auch mittels `v = v * ( 1 / !v )` berechnet werden.

Beispiel:

```
HVECTOR v ;

v = ∼v ;
```

double& operator[](int)

Erlaubt sowohl den lesenden als auch den schreibenden indizierten Zugriff auf einzelne Komponenten eines Vektorobjekts.

Beispiel:

```
HVECTOR v ;
double z ;

v[ 1 ] = 5.0 ;
z = v[ 2 ] ;
```

4.1.3 Weitere Beispiele

1. Berechnung eines orthonormierten Koordinatensystems aus den Vektoren $\vec{o}, \vec{a}$ mit $\vec{a}$ als "Orthonormierungsbasis":

   ```
   HVECTOR n,o,a ;

   n = (~o) ^ (~a) ;
   o = (~a) ^ (~n) ;
   a = ~a ;
   ```

2. Berechnung des Winkels zwischen zwei Vektoren $\vec{a}$ und $\vec{b}$:

   ```
   HVECTOR a,b ;
   double winkel ;

   winkel = acos( a * b ) ;
   ```

4.2 Klasse: MATRIX

In der Klasse MATRIX sind die notwendigen Konstruktoren, Operatoren und Methoden enthalten, die das bequeme Arbeiten mit homogenen Transformationsmatrizen erlauben. Hier sind auch diejenigen Operatoren definiert, die die Interaktion zwischen den Klassen HVECTOR und MATRIX ermöglichen. Die Klasse MATRIX ist die interne Basisklasse der Klasse FRAME, die weiter unten beschrieben wird. Deshalb sollte auf die MATRIX- Klasse nicht zugegriffen werden. Statt desssen können die Methoden in der Klasse FRAME benutzt werden.

4.2.1 Konstruktoren

Die nachfolgenden Konstruktoren für Objekte der Klasse MATRIX sind verfügbar:

1. *MATRIX (MatrixType mt = Identity)*
   Wird dem Konstruktor einer MATRIX-Instanz kein Argument mitgegeben, so wird ein Matrixobjekt erzeugt, das mit der Einheitsmatrix initialisiert wird. Wird jedoch der Typ `MATRIX::ZeroMatrix` als Argument angegeben, so wird das Objekt mit der Nullmatrix initialisiert.

   Beispiel:
   ```
   MATRIX I ;                          // I ist eine Einheitsmatrix
   MATRIX N (MATRIX::ZeroMatrix); // N ist eine Nullmatrix
   ```

2. *MATRIX (HVECTOR&, HVECTOR&, HVECTOR&, HVECTOR&)*
   Es wird ein MATRIX-Objekt erzeugt, welches mit den übergebenen vier **Spaltenvektoren** initialisiert wird.

4.2.2 Operatoren

Nachfolgend sind diejenigen Operatoren aufgeführt, die auf MATRIX-Objekte als auch auf Mischterme aus MATRIX- und HVECTOR-Objekten ausgeführt werden können.

void operator= (const MATRIX&)

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

Beispiel:

```
MATRIX A,B ;

A = B ;
```

int operator== (const MATRIX&) const

Vergleicht die einzelnen Elemente zweier Matrizen miteinander. Das Ergebnis ist nur dann **true**, wenn alle Elemente gleich sind.

Beispiel:

```
MATRIX A,B ;

if (A == B)
{ ... }
else
{ ... }
```

MATRIX operator+ (const MATRIX&) const

Addiert elementweise zwei Matrizen.

$$[C] = [A] + [B]$$

Beispiel:

```
MATRIX A,B,C ;

A = B + C ;
```

MATRIX operator- (const MATRIX&) const

Subtrahiert elementweise die Elemente der zweiten Matrix von der Ersten.

$$[C] = [A] - [B]$$

Beispiel:

```
MATRIX A,B,C ;

A = B - C ;
```

MATRIX operator* (const MATRIX&) const

Multipliziert zwei Matrizen miteinander (**auf Reihenfolge achten !**).

$$[C] = [A] \cdot [B]$$

Beispiel:

```
MATRIX A,B,C ;

A = B * C ;
```

HVECTOR operator* (HVECTOR&) const

Mittels dieses Operators können Vektoren unter Verwendung einer gegebenen Matrix transformiert werden.

$$\vec{c} = [T] \cdot \vec{x}$$

Beispiel:

```
HVECTOR c,x ;
MATRIX T ;

c = T * x ;
```

MATRIX operator~ (void) const

Berechnet die inverse Matrix. Dabei wird davon ausgegangen, dass es sich um eine homogene Transformationsmatrix handelt. (Es wird also der aus der VL bekannte vereinfachte Algorithmus zur Berechnung der Inversen verwendet.)

Beispiel:

```
MATRIX A ;

A = ~A ;
```

4.2.3 Methoden

Im folgenden sind noch die auf Objekte des Typs MATRIX anwendbaren Methoden beschrieben. In C++ ist das Anwenden von Methoden analog der Referenzierung von Strukturelementen unter C. Um beispielsweise die Methode *XYZ()* auf ein Objekt des Typs Matrix anzuwenden, gibt es folgende Möglichkeiten:

```
MATRIX M ;
MATRIX *P = new MATRIX() ;

M.XYZ() ;      // da 'M' eine Instanz darstellt
P->XYZ() ;     // da 'P' ein Pointer auf eine Instanz darstellt
```

HVECTOR GetEuler_I (void)

Berechnet die Eulerwinkel (RotZ,RotX,RotZ) aus dem rotatorischen Unteranteil der Matrix

Beispiel:

```
HVECTOR euler ;
MATRIX M ;

euler = M.GetEuler_I() ;
```

HVECTOR GetEuler_II (void)

Berechnet die Eulerwinkel (RotZ,RotY,RotZ) aus dem rotatorischen Unteranteil der Matrix

Beispiel:

```
HVECTOR euler ;
MATRIX M ;

euler = M.GetEuler_II() ;
```

HVECTOR GetRPY (void)

Berechnet die Roll-, Pitch- und Yaw-Winkel aus dem rotatorischen Unteranteil der Matrix. In dem zurückgegebenen Vektor sind die Winkel in der Reihenfolge YAW, PITCH, ROLL enthalten.

Beispiel:

```
HVECTOR rpy ;
MATRIX M ;

rpy = M.GetRPY() ;
```

void SetColumn (HVECTOR&, ColumnVector)

Ersetzt den angegebenen Spaltenvektor durch den übergebenen Vektor

Beispiel:

```
HVECTOR p_neu ;
MATRIX M ;

M.SetColumn(p_neu, MATRIX::P);
```

void SetRow (HVECTOR&, int)

Ersetzt den spezifizierten Zeilenvektor durch den übergebenen Vektor

Beispiel:

```
HVECTOR neu ;
MATRIX M ;

M.SetRow (neu, 0);
```

void SetRow (double, double, double, double, int)

Ersetzt den spezifizierten Zeilenvektor durch die übergebenen Werte

Beispiel:

```
MATRIX M ;

M.SetRow (1.0, 2.5, 3.0, 1.0, 0);
```

HVECTOR GetRow(int)

Liefert den spezifizierten Zeilenvektor als Vektorobjekt zurück

Beispiel:

```
HVECTOR zeile ;
MATRIX M ;

zeile = M.GetRow (0);
```

void Orthonorm (void)

Orthonormiert die Matrix. Als Bezugsvektor wird dabei der Approach-Vektor verwendet. Mit Hilfe des normierten Approach- und Openvektors wird mittels des Kreuzproduktes der Normalenvektor berechnet. Daraufhin wird, ebenfalls mit Hilfe des Vektorproduktes, der orthonormierte Openvektor berechnet. Die alten Werte der rotatorischen Untermatrix gehen dabei verloren.

Beispiel:

```
MATRIX M ;

M.Orthonorm() ;
```

4.3 Klasse: FRAME

Die Klasse FRAME ist von der internen Klasse MATRIX abgeleitet und enthält die für die eindeutige Berechnung der inversen Kinematik notwendigen Roboter-Konfigurationsflags. Ansonsten können die Operatoren und Methoden, die bereits aus der Klasse MATRIX bekannt sind, weiterhin verwendet werden.

4.3.1 Konstruktoren

Die nachfolgenden Konstruktoren für Objekte der Klasse FRAME sind verfügbar:

1. *FRAME (HVECTOR&, HVECTOR&, HVECTOR&, HVECTOR&, int rc = 0)*
   Erzeugt aus den vier übergebenen Vektoren und der optional spezifizierten Konfiguration ein FRAME-Objekt.
2. *FRAME (void)*
   Erzeugt ein FRAME, welches die Einheitsmatrix mit der Null-Konfiguration darstellt.

4.3.2 zusätzliche Methoden

Die Klasse FRAME verfügt über zwei zusätzliche Methoden (zu denen der Klasse MATRIX), die für das Lesen und Schreiben der Konfiguration eines FRAME-Objektes notwendig sind.

void SetConfig (int rc)

Setzt die Konfiguration des FRAME-Objektes auf den übergebenen Wert.

Beispiel:

```
FRAME F ;

F.SetConfig (0x02) ;
```

int GetConfig (void)

Gibt die Konfiguration des FRAME-Objektes zurück.

Beispiel:

```
FRAME F ;
int config ;

config = F.GetConfig() ;
```

4.4 Besonderheiten

Durch die Überlagerung des Index-Operators (*operator[]*) in den beschriebenen Klassen, muß bei der Verwendung von Arrays oder Pointern aufgepaßt werden. Diese sind zuerst mit dem '*'-Operator zu referenzieren. Erst dann kann der Index-Operator in der jeweils beschriebenen Weise angewendet werden.

Beispiel:

Verwendung des Index-Operators bei einem normalen Pointer:

```
FRAME  *f ;
HVECTOR v ;

f = new FRAME () ;          // 'Einheits'-Frame
v = (*f)[ FRAME::P ] ;    // Zugriff auf p-Vektor
```

Verwendung des Index-Operators bei Arrays von z.B. Matrizen:

```
FRAME *m ;
HVECTOR v ;

v = (m[ ArrayIndex ])[ FRAME::N ] ;    // Zugriff auf n-Vektor
```

4.5 Globale Funktionen

Mit globale Funktionen sind mathematische Funktionen gemeint, die Elemente der Klasse FRAME als Ergebnis zurückliefern. Dazu gehören u.a. Euler- und RPY-Winkel, oder auch reine Translationsmatrizen.

FRAME FTransXYZ (double x, double y, double z)

Berechnet die Translationsmatrix für eine Verschiebung um die übergebenen Werte.

Beispiel:

```
FRAME M;

M = FTrans (10.0, 60.0, 1.5);
```

FRAME FRotX (double winkel)

Berechnet die Rotationsmatrix für eine Drehung um die X-Achse des Basissystems. Winkelangaben im Bogenmaß!

Beispiel:

```
FRAME M;

M = FRotX (0.5);
```

FRAME FRotY (double winkel)

Berechnet die Rotationsmatrix für eine Drehung um die Y-Achse des Basissystems. Winkelangaben im Bogenmaß!

Beispiel:

```
FRAME M;

M = FRotY (0.5);
```

FRAME FRotZ (double winkel)

Berechnet die Rotationsmatrix für eine Drehung um die Z-Achse des Basissystems. Winkelangaben im Bogenmaß!

Beispiel:

```
FRAME M;

M = FRotZ (0.5);
```

FRAME FRotRPY (double Roll, double Pitch, double Yaw)

Berechnet die Rotationsmatrix für eine Drehung um die RPY-Winkel. Winkelangaben im Bogenmaß!

Beispiel:

```
FRAME M;

M = FRotRPY (0.0, 0.0, 0.5);
```

FRAME FRotEuler_I (double z1, double x, double z2)

Berechnet die Rotationsmatrix für eine Drehung um die Euler-Winkel (RotZ,RotX,RotZ). Winkelangaben im Bogenmaß!

Beispiel:

```
FRAME M;

M = FRotEuler_I (0.0, 0.0, 0.5);
```

FRAME FRotEuler_II (double z1, double y, double z2)

Berechnet die Rotationsmatrix für eine Drehung um die Euler-Winkel (RotZ,RotY,RotZ). Winkelangaben im Bogenmaß!

Beispiel:

```
FRAME M;

M = FRotEuler_II (0.0, 0.0, 0.5);
```

FRAME FRotVectr (HVECTOR& v, double winkel)

Berechnet die Rotationsmatrix für eine Drehung um einen beliebigen Vektor v um den Winkel *winkel*. Die Funktion normiert den Vektor v selbständig. Winkelangaben im Bogenmaß!

Beispiel:

```
FRAME M;
HVECTOR v (0.0, 0.0, 0.1);

M = FRotVectr (v, 0.5);
```

Kapitel 5

Zero++

Dieses Kapitel beschreibt diejenigen Funktionen von *Zero++*, die für die erfolgreiche Absolvierung des Robotik-Praktikums notwendig sind.

5.1 Kommunikation mit dem Roboter

Die Steuerbefehle werden über eine Socket-Verbindung an den Roboter geschickt, dort verarbeitet und eine Antwort zurückgeschickt. Zu diesem Zweck muß auf dem Roboter ein Serverprogramm gestartet werden, welches die Befehle entgegennimmt und in die entsprechenenden Steuersequenzen umsetzt. Hierzu ist der Roboter zunächst in den *Comp-Modus* zu schalten und sind folgenden Befehle über die Konsole einzugeben. Das Laden des Programms erfolgt durch den Befehl

```
LOAD SERVER5.V2
```

Anschließend kann das Programm mit dem Befehl

```
EXE SERVER5
```

gestartet werden.

5.2 Die Klasse RX60

In dem gegebenen Programmgerüst wurde die Header-Datei „robot.h“eingebunden. In dieser Datei wird die Klasse RX60 definiert, welche die Client-Seite der Kommunikation beinhaltet. Beim Instanziieren dieser Klasse wird automatisch eine Verbindung zum Roboter aufgebaut, welche beim Entfernen der Instanz wieder geschlossen wird.

Ein minimales *Zero++*-Programm sieht daher wie folgt aus:

```
#include "robot.h"

int main()
{
  RX60 Robot;

  [...]
}
```

5.3 Bewegen des Roboters

Mit Hilfe der nachfolgenden Funktionen kann der Praktikumsroboter (*Stäubli RX60*) bewegt werden. Dabei wird zwischen zwei Funktionen unterschieden, die jedoch denselben Namen besitzen: **MoveTo**(). Sie werden vom Compiler anhand der übergebenen Parameter unterschieden. In *Zero++* sind zusätzlich einige Konstanten definiert, die sich auf das Verhalten der **MoveTo**()-Routine auswirken. Auf diese Konstanten wird im folgenden kurz eingegangen.

5.3.1 Konstanten für die MoveTo()-Routine

Die Bewegung des Roboters kann auf zwei Arten interpoliert werden:

- JointInterp:
  Bei dieser Form der Interpolation wird für jedes Gelenk linear zwischen den Winkelstellungen interpoliert. Wird der Roboter nur mit einem Gelenk bewegt, fährt er als einen Kreisbogen.
- FrameInterp:
  Diese zweite Form der Interpolation bezieht sich auf den kartesischen Raum. In diesem Fall wird eine Gerade zwischen den angegebenen Positionen berechnet und die Orientierung ebenfalls linear interpoliert.

Diese Konstanten können beispielsweise durch den Ausdruck *RX60::JointInterp* eingesetzt werden.

5.3.2 Konstanten für die Handsensoren

In der Roboterhand des *Stäubli RX60* befinden sich eine Lichtschranke und in jeder Greiferbacke jeweils ein Näherungssensor. Für die Sensoren wurden Konstanten definiert mit denen sie z.B. bei der Funktion *SetMonitor* spezifiziert werden können:

- NO_SENSOR:
  Schaltet alle Sensoren für die entsprechende Funktion ab.
- HAND_LEFT:
  Selektiert den Nährungssensor in der linken Greiferbacke. Der Sensor ist so eingestellt, dass er reagiert, sobald eine **weiße** Fläche in seine Nähe kommt.

- HAND_RIGHT:
  Entspricht *HAND_LEFT* jedoch für die rechte Greiferbacke.
- LIGHT_BARRIER:
  Bestimmt die Lichtschranke, welche sich zwischen den beiden Greiferbacken der Hand befindet.

Die Angabe eines Sensors erfolgt durch den Ausdruck *RX60::*SENSOR (für die Lichtschranke also RX60::LIGHT_BARRIER).

5.3.3 Konstruktoren

Mit der Methode *RX60 (void)* kann ein neues Objekt der Klasse *RX60* erzeugt werden.

Hinweis:
Beim Instanzieren dieser Klasse wird automatisch eine Socket-Verbindung zum Roboter hergestellt. Damit diese erfolgreich aufgebaut werden kann, muss im Roboter das Serverprogramm „SERVER5“ laufen, welches über die Konsole gestartet werden kann (s.o.) !

5.3.4 Methoden

int MoveTo (FRAME& frame, MotionInterp MInterp, SensorType Monitor)

In *frame* wird in Form eines *FRAME*s das gewünschte Ziel der Bewegung des Roboters angegeben. Für den zweiten und dritten Parameter gibt es sogenannte „default“-Werte. D.h. dass diese Parameter nicht notwendigerweise immer mit angegeben werden müssen.
Beispiel:

```
[...]

Robot.MoveTo (Ziel);
// Bewegung erfolgt gelenkinterpoliert ohne Monitor

Robot.MoveTo (Home, RX60::FrameInterp);
// Linear interpoliert, keine Überwachung der Bewegung

Robot.MoveTo (Rutsche, RX60::JointInterp, RX60::LIGHT_BARRIER);
// Gelenkinterpoliert mit Lichtschrankenüberwachung
```

Mit dem zweiten Parameter *MInterp* kann die Interpolationsart angegeben werden. Zur Auswahl stehen die Typen *FrameInterp* (linear interpoliert) und *JointInterp* (gelenkinterpoliert). Wird kein Wert angegeben, wird die Gelenkinterpolation benutzt.
Der dritte Parameter ermöglicht die sensorüberwachte Bewegung des Roboters. Die einstellbaren Sensoren sind:

- *NO_SENSOR*: keine Sensorüberwachung
- *HAND_LEFT*: Näherungssensor in der linken Greiferbacke
- *HAND_RIGHT*: Näherungssensor in der rechten Greiferbacke

- *LIGHT_BARRIER*: Lichtschranke zwischen den Greiferbacken

Wenn der Parameter weggelassen wird, erfolgt die Bewegung ohne Überwachung. Bei aktivierter Überwachung wird im Fall der Sensorauslösung (Lichtschranke unterbrochen, bzw. Objekt berührt Greiferbacke) die Bewegung unterbrochen und eine entsprechender Wert zurückgegeben. Die Sensorüberwachung ist nur während des **MoveTo**()-Befehls aktiv.

Die **MoveTo**()-Routine liefert einen **int**-Wert mit folgender Bedeutung zurück:

-1 Die Bewegung kann nicht ausgeführt werden (z.B. liegt die Zielposition außerhalb des Arbeitsbereiches.

0 Der Sensor wurde aktiviert

1 die Bewegung konnte korrekt ausgeführt werden

int MoveTo (JOINT& joint, MotionInterp MInterp, SensorType Monitor)

Diese Routine entspricht der vorhergehenden, jedoch wird hier die Zielposition durch die Gelenkstellungen angegeben.

void Approach (double& dist, double &speed, SensorType Monitor)

Mit der *Approach*-Funktion kann der Roboter in positive Richtung des Approach-Vektors der aktuellen Handorientierung bewegt werden. Mit dem Parameter *dist* wird die gewünschte Entfernung (in Millimetern) angegeben. Der *speed*-Parameter legt die Bewegungsgeschwindigkeit fest. Wird als Geschwindigkeit 0 angegeben oder der Parameter weggelassen, so wird die zuletzt eingestellte Geschwindigkeit beibehalten.
Als letzter Paramter (*Monitor*) kann wiederum bei der Bewegung ein Sensor angegeben werden, welcher während der Bewegung überprüft wird.
Am Ende der Routine wird die Geschwindigkeit wieder auf den zuvor eingestellten Wert gesetzt.
Beispiel:

```
[...]

Robot.Approach (40, 10);
// Bewegt die Hand 4 cm in Richtung des Approach-Vektors. Die
// Geschwindigkeit beträgt 10 \% der Maximalgeschwindigkeit.
```

void Depart (double& dist, double &speed, SensorType Monitor)

Diese Routine bewegt den Roboter analog zur *Approach*-Funktion in die entgegengesetzte Richtung (negativer Approach-Vektor).

void SetMonitor (SensorType Monitor, int Priority)

Diese Routine bestimmt einen Sensor, welcher bei den folgenden Bewegungen überprüft werden soll. Der Monitor bleibt solange aktiv bis

- Der Sensor aktiviert wurde
- Der Sensor gestoppt wird (siehe *StopMonitor*)
- Ein anderer Sensor als Monitor aktiviert wird.

Wird der Sensor während einer Bewegung aktiviert, so wird diese Bewegung abgebrochen.
Der *Priority*-Parameter ist optional und erlaubt es, die Priorität der Sensorabfrage zu verändern.

void StopMonitor (SensorType Monitor)

Hiermit kann eine zuvor gestartete Monitorroutine gestoppt werden.

JOINTS GetJoints ()

Die *GetJoints*-Routine ermittelt die aktuellen Roboter-Gelenkstellungen und liefert sie im Rückgabeparameter zurück.

Der Rückgabewert enthält die sechs Gelenkstellungen im Bogenmaß

FRAME GetPosition ()

GetPosition liefert die aktuelle Roboter-Position als Frame. Das zurückgegebene Frame enthält die Position und Orientierung des Roboters in Form eines Frames.

void SetSpeed (double& p_speed)

Die *SetSpeed*-Funktion setzt die Geschwindigkeit des Roboters in % der Maximalgeschwindigkeit. Aus Sicherheitsgründen sollte die eingestellte Geschwindigkeit nicht größer als 40% sein !

double GetSpeed ()

Hiermit kann die aktuell eingestellte Bewegungsgeschwindigkeit des Roboters ermittelt werden. Der Wert gibt den Prozentsatz von der maximal möglichen Geschwindigkeit an.
Der zurückgegebene Geschwindigkeitswert entspricht dem Prozentsatz der maximalen Geschwindigkeit.

5.3.5 Roboter-spezifische Funktionen

FRAME FwdKin (JOINT & Joint, bool & check)

Diese Routine wandelt die gegebenen Gelenkstellungen in ein Frame um. Mit dem *check*-Parameter kann bestimmt werden, ob die Stellung auf Gültigkeit überprüft werden soll.
Es wird das zu den Gelenkwinkeln entsprechende Frame zurückgegeben.

void OpenHand ()

Mit dieser Routine wird der Greifer geöffnet. Ist der Greifer beim Aufruf dieser Routine bereits offen, so wird keine Änderung vorgenommen.

void CloseHand ()

Die *CloseGripper*-Routine schließt die Greiferbacken des Roboters, falls diese geöffnet sind.

bool InRange (FRAME & p_goalFrame)

Für den übergebenen Frame wird geprüft, ob die dadurch definierte Position und Orientierung vom Endeffektor des Roboters erreichbar ist.
Liegt die angegebene Position und Orientierung innerhalb des Arbeitsraumes, liefert die Funktion den Wert **True**. Anderenfalls ist der Rückgabewert **False**.

int GetUSSensor ()

Mit dieser Routine wird der Werte des Ultraschallsensors ermittelt.
Der Wert entspricht der Entfernung des Sensors von einem Gegenstand (Tisch, Klotz, ...). Der Sensor ist so eingestellt, dass er bei einer Entfernung von 20 cm einen Wert von 200 liefert. Für die Justierung des Sensors ist der betreuende HiWi zuständig.
Der Sensor hat nur einen eingeschränkten Meßbereich (ca. 5 - 25 cm). Außerhalb dieses Bereiches wird der Wert 0 zurückgegeben.

Kapitel 6

Aufgabenstellungen

Wie bereits im Versuch 1 am Simulationssystem gezeigt, soll nun ein realer Roboter eine Palette mit Muttern belegen. Wie Sie sehen, wurde die simulierte Umgebung der realen Umgebung nachgebildet. Auch hier sind die Muttern in einer Zuführeinrichtung aufgereiht. Da es sich jetzt um einen realen Roboter handelt, müssen die folgenden Punkte unbedingt beachtet werden:

- Um eine Beschädigung des Roboters zu vermeiden, **muss**, während sich der Roboter bewegt, ein Mitglied der Gruppe die Hand am Notaus–Schalter haben !
- Die Greiferbacken müssen beim Zugreifen parallel zu den Mutterseitenflächen ausgerichtet sein, damit die Greifkraft optimal wirken kann. Darauf ist beim *Teachen* der Greifposition besonders zu achten !
- Beim Entnehmen der Mutter muss unter **Beibehaltung der Handorientierung** eine Depart–Bewegung erfolgen, damit ein Verkanten der Mutter verhindert wird.
- Analog zur Entnahme der Mutter aus der Zuführeinrichtung, soll bei der Ablage der Mutter eine Approach-Bewegung programmiert werden. Die Approach-Bewegung selbst soll wiederum unter **Beibehaltung der Handorientierung** erfolgen. Die Orientierung der abgelegten Muttern ist so zu modifizieren, dass die Seitenflächen angrenzender Muttern parallel zueinander sind. Auf diese Weise wird der Abstand zwischen den Muttern optimal ausgenutzt.
- Es soll vor und nach dem Absetzen einer Mutter auf der Palette eine Sicherheitsposition angefahren werden, um Kollisionen mit der Palette oder der Zuführeinrichtung zu verhindern. Da zwischen der Entnahme- und Ablageposition gelenkinterpoliert verfahren werden soll, kann dieser Fall leicht eintreten, da die kartesische Bahn des Endeffektors nicht vorhersagbar ist. Die Lage der Sicherheitsposition sollte ca. 30cm über der Arbeitsplattenoberfläche zwischen der Palette und der Zuführeinrichtung liegen.

6.1 AUFGABE 1

Machen Sie sich mit der Bedienung des Roboters und der Sicherheitseinrichtung vertraut. Starten Sie den Roboter und aktivieren Sie den Knickschutz. Fahren Sie mit

dem Roboter anschließend mit Hilfe des Teach-Panels einige Positionen mit geöffnetem und geschlossenem Greifer an. Zur Palettierung soll nun ein Programm in der Sprache V+ geschrieben werden. Als Grundlage kann der Anhang dieses Skripts dienen. Da es sich bei dem Beispiel jedoch um ein *VAL II*-Programm handelt, welches die die Vorgängerversion der am Institut eingesetzten Roboter-Programmiersprache *V+* ist, sollte es nicht nur abgetippt, sondern auch korrigiert werden. Wichtig ist vor allem das Verständnis, wie das Programm funktioniert.

Anmerkungen:

- Zum Einloggen auf den PC benötigen Sie kein Passwort. Sie werden beim ersten Mal aufgefordert ein entsprechendes Passwort einzugeben.
- Erstellen Sie Ihr V+ Programm mit dem PC. Sie können das Programm mit dem Editor erstellen, welcher im *Start*-Menü unter *Programme/Zubehör/Editor* erreichbar ist.
  Da ein direkter Zugriff des Roboters auf die PC-Laufwerke nicht möglich ist, muss das erstellte Programm in Ihr Unix-Verzeichnis gespeichert werden. Der Zugriff auf dieses Verzeichnis erfolgte beim Einloggen und wurde bereits in Kapitel 3 beschrieben. Schreiben Sie Ihr *V+* Programm in das Verzeichnis
  `F:\`
  Von dort aus kann es über die Konsole des Roboters mit dem Befehl:
  `LOAD NFS>STAEUBLI:\Programmname`
  geladen werden. Bevor sie ein Programm erneut laden, muß ein geladenes Programm durch Eingabe von
  `ZERO`
  und Bestätigung mit 'Y' gelöscht werden.
  Die Ausführung des Programms erfolgt durch den Befehl:
  `EXEC Programmname`

Wichtig:

Da es aus organisatorischen Gründen nicht möglich ist, dass das Programm in Ihrem lokalen Verzeichnis gespeichert wird (bei dem Verzeichnis *F:* handelt es sich um einen Link), sollte das Programm nach Fertigstellung in ein **eigenes** Verzeichnis kopiert werden. Anschließend sollte das Programm im *F:*-Verzeichnis gelöscht werden.

6.2 AUFGABE 2

Die Palettierung soll nun vollständig in *Zero++*durchgeführt werden. Zur Vereinfachung wurden die benoetigten Koordinaten bereits in das Programmgerüst eingetragen. Aus den drei Positionen der Palette muß zunächst das Palettenframe erstellt werden, d.h. die Position und Orientierung der Palette bezüglich der Welt (Basis des Roboters), berechnen. Da die Orientierung der vorgegebenen Palettenframes nicht identisch mit der Ausrichtung der Palette ist (siehe auch Aufgabe 3), muß diese Orientierung genau berechnet werden.
Die Abmessungen der Objekte, sowie die Anordnung und Abstände der Stifte auf der Palette können Sie aus der Zeichnung 6.1 entnehmen.

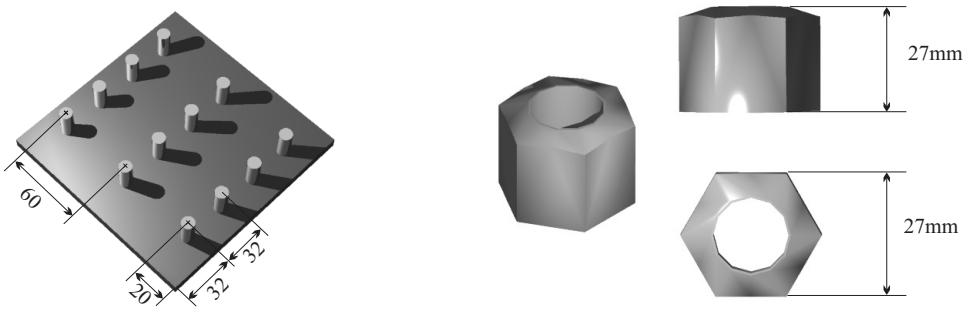

Abbildung 6.1: Abmessungen der Objekte *Mutter* und *Palette*

Im Anschluss an die Berechnung des Frames soll die Palettierung durchgeführt werden.

Für die erfolgreiche Ausführung dieser Aufgabe erfüllen Sie bitte u.a. folgende Punkte:

1. Benutzen Sie die Klassen VECTOR und FRAME, sowie die von *Zero++* angebotenen Operatoren auf diesen Klassen.
2. Arbeiten Sie mit einem lokalen Palettenkoordinatensystem, mit dessen Hilfe Sie die einzelnen Palettenpositionen leicht berechnen können.

Anmerkungen:

- Das zu ergänzende Visual C++-Projekt befindet sich im Verzeichnis M:\Prak\Gruppe*n*\Versuch2\vc++
- Nach dem Laden des Projektes wird die Datei *PaletteView.cpp* angezeigt, welche von Ihnen zu ergänzen ist.
- Verwenden Sie an geeigneten Stellen des Programms die Funktion *MessageBox(...)*, um das Anfahren der nächsten Position durch einen Mausklick bestätigen zu können.

6.3 AUFGABE 3

Um zu überprüfen, ob das lokale Koordinatensystem der Palette auch korrekt berechnet worden ist, soll nun die Palettierung mit einer neuen Palettenposition wiederholt werden. Die neue Position soll gegenüber des Koordinatensystems der Basis des Roboters derart verdreht werden, daß die Achsen der jeweiligen lokalen Systeme **nicht** mehr orthogonal zueinander sind.

Anhang A

Appendix

A.1 V+, Nachfolger von VAL II

Monitor-Befehle:

CALIBRATE	Synchronisation von Mechanik und Elektronik
SPEED *factor*	Verringert die Monitor-Geschwindigkeit um *factor* der Standardgeschwindigkeit 100%
EXECUTE *program*	Führt das Roboterprogramm *program* aus
ABORT *tasknum*	Abbruch des Roboterprogramms mit der Task-Nummer *tasknum*
ENABLE *parameter*	Systemparameter *parameter* wird aktiviert
DISABLE *parameter*	*parameter* wird deaktivert
SWITCH *parameter*	Zustand des Systemparameters *parameter* wird geändert
POINT *var = val*	Setzen der Umgebungsvarible *var*, speziell *val* für Precision-Point-Variablen
HERE *var*	Abspeichern der aktuellen Gelenkwinkel in den Variablen
BASE *x,y,z,θ*	Setzen der Roboterbasis : Trans(*x,y,z*) * Rot(Z, *θ*)
TOOL *transform*	Setzen der Tooltransformation
BREAK	Stoppt die Programmausführung so lange, wie die Roboterbewegung dauert. Diese Anweisung ist speziell im CP-Mode sinnvoll, wenn z.B. Objekte gegriffen werden sollen

Roboter-Kontrolle:

MOVE *loc*	gelenkinterpolierte Bewegung zur Position *loc*
MOVES *loc*	linearinterpolierte Bewegung zur Position *loc*
MOVET *loc, val*	wie MOVE, während der Bewegung wird die Hand auf *val* geöffnet
MOVEST *loc, val*	wie MOVET, aber linearinterpoliert
APPRO *loc, dist*	wie MOVE, es wird jedoch eine Position angefahren, die *dist* mm von *loc* in Richtung der Tool-Z-Achse entfernt ist
APPROS *loc, dist*	
DEPART *dist*	bewegt den Effektor *dist* mm in Richtung der negativen Tool-Z-Achse
DEPARTS *dist*	
MOVEF	Initiiert eine Pick and Place Bewegung (gelenkvariablen interpolierte Transferbewegung)
MOVEF	Initiiert eine Pick and Place Bewegung (kartesisch interpolierte Transferbewegung)
DRIVE *j, ch, sp*	bewegt das Gelenk *j* um *ch* Grad mit *sp*% der Monitorgeschwindigkeit
READY	Roboter fährt in die Homoposition
DELAY *time*	Stoppen der Roboterbewegung für *time* Sekunden
HERE *var*	setzt *var* gleich der aktuellen Position

Bewegungs-Parameter:

SPEED *factor* [ALWAYS]	setzt die Geschwindigkeit auf *factor* Monitorgeschwindigkeit; fehlt ALWAYS, so gilt *factor* nur für die nächste Bewegung
SPEED *val, unit* [ALWAYS]	absolute Geschwindigkeit in *unit* : MMPS = mm/s oder IPS = inch/s
DURATION time [ALWAYS]	setzt die minimale Zeit, die die folgenden Bewegungen dauern sollten
COARSE [ALWAYS]	erlauben von größeren Positionsfehlern pro Gelenk (±1° bzw. ±1mm)
FINE [ALWAYS]	höchste Genauigkeit
CP	Systemparameter, der angibt, ob einzelne Bewegungssegmente zu einer glatten Bahn verschmolzen werden (continous path)

Funktionen:

NULL	Nulltransformation (Einheitsmatrix)
BASE	liefert aktuelle BASE-Transformation
DEST	aktuelles Ziel
HERE	aktuelle Gelenkwinkel
TOOL	aktuelle TOOL-Transformation
INVERSE(*loc*)	inverse Transformation von *loc*
DECOMPOSE *array[]=pos*	schreibt die Variablen in *pos* in das *array*
NORMAL(*loc*)	die Orientierungsvektoren von *loc* werden orthonormiert
RX(θ), RY(θ), RZ(θ)	Rotationstransformationen um X-, Y- bzw. Z-Achse
TRANS(*x,y,z,o,a,t*)	liefert Trans(*x,y,z*) * Rot(Z, *o*) * Rot(Y, *a*) * Rot(X, *t*)
DISTANCE(*loc1*, *loc2*)	Abstand zwischen *loc1* und *loc2*
DX(*loc*), DY(*loc*), DZ(*loc*)	liefert den X-, Y- bzw. Z-Anteil der Transformation *loc*
IDENTICAL(*loc1*, *loc2*)	Test auf Identität zweier Transformationen
INRANGE(*loc*)	liefert den Wert 0, falls *loc* erreicht werden kann
FRAME(l_1, ..., l_4)	liefert ein Frame mit Ursprung gegeben durch l_4 , x-Achse parallel zu der durch l_1, l_2, l_3 definierten Ebene; Richtung der Y-Achse muss von der X-Achse zu l_3 zeigen
SCALE(*loc* BY *s*)	liefert die Transformation *loc* um den Faktor *s* skaliert
SHIFT(*loc* BY *x,y,z*)	liefert die Transformation *loc* um den Vektor $[x,y,z]^T$ verschoben
#PPOINT(j_1, ..., j_6)	Precision-Point

Konfigurations- und Handkontrolle

RIGHTY, LEFTY	Gelenk 1 und 2 werden wie ein rechter bzw. linker Arm gesteuert
ABOVE, BELOW	Ellenbogengelenk oben bzw. unten
FLIP, NOFLIP	Gelenk 5 arbeitet im negativen bzw. positiven Winkelbereich
OPEN	öffnen der Hand während der nächsten Bewegung
OPENI	sofortiges öffnen der Hand
CLOSE, CLOSEI	(dito) schließen der Hand
HAND	liefert 1, falls die Hand offen ist, sonst 0

Programmablauf:

GOTO *label*	Sprung zur Marke *label*
CALL *pro(arg_list)*	ruft das Programm *pro* auf, (Rekursionen sind nicht möglich)
RETURN	Ende des Unterprogramms, Rücksprung zur Aufrufstelle
HALT	Abbruch des Programmlaufs (irreversibel)
STOP	beendet derzeitigen Programmzyklus
PAUSE	Unterbrechen des Programms; kann mit PROCEED wieder gestartet werden
WAIT *condition*	Programm wartet bis *condition* den Wert *true* liefert
REACTE *err_prg*	Aufsetzen einer eigenen Fehlerbehandlung

Einige Kontrollstrukturen:

```
FOR LoopVar = val1 TO val2 STEP val3 ... END
WHILE condition DO ... END
DO ... UNTIL condition
IF condition GOTO label
IF condition THEN ... ELSE ... END
```

A.2 Beispielprogramm in V+

```
.PROGRAM teach_in( teach_frame, &msg )
;
; Mit Hilfe dieses Unterprogramms wird während des Programmlaufs über das
; Manual-Control-Pendant (MCP) interaktiv die Location „teach-frame" definiert.

      LOCAL $clear_display                    ; String-Variable

      $clear_display = $CHR(12) + $CHR(7)     ; Löschsequenz für MCP-Display
      ATTACH(1)                               ; MCP aktivieren
      DETACH(0)                               ; Roboterkontrolle abgeben

      WRITE(1) $clear_display$, $msg          ; Meldung ins MCP-Display schreibe
      WRITE(1) /X17, "RECORD", $CHR(5), /S
      WRITE(1) $CHR(30), $CHR(3), /S
      WAIT PENDANT(3)                         ; auf MCP-Tastendruck warten
      HERE teach_frame                        ; aktuelle Position festhalten

      ATTACH(0)                               ; Kontrolle übernehmen
      DETACH(1)                               ; MCP deaktivieren

      RETURN
.END


.PROGRAM def_pallet_frame( pallet)
;
; Bestimmen der Position der Palette mit Hilfe der Prozedur „teach_in":
; Dazu muss der Benutzer den Koordinaten-Nullpunkt, einen Punkt auf der X-Achse
; und einen Punkt auf der Y-Achse der Palette interaktiv angeben.
;
      LOCAL pallet_ref_point, pallet_x, pallet_y

      CALL teach_in( pallet_ref_point, "Referenzpunkt der Palette" )
      CALL teach_in( pallet_x, "Punkt auf der X-Achse der Palette" )
      CALL teach_in( pallet_y, "Punkt auf der Y-Achse der Palette" )

      SET pallet = FRAME( pallet_ref_point, pallet_x, pallet_ref_point)

      RETURN
.END


.PROGRAM def_pickup_frame( pickup )
;
; Festlegen der Aufnahmeposition:
;
      CALL teach_in( pickup, "Aufnahmestelle" )
      RETURN
.END
```

```
.PROGRAM do_pallet( pallet, pick_up, x_max, y_max, x_shift, y_shift )
;
; Die Palettierung erfolgt in zwei ineinander geschachtelten Schleifen. Die
; äußere Schleife läuft über den Y-Index, die innere Schleife über den X-Index.
; Die Location "pallet" und "pick_up" geben den Paletten-Nullpunkt
; bzw. den Aufnahmepunkt an. Die Anzahl der Palettenplätze in X- bzw. in
; Y-Richtung werden durch "x_max" und "y_max" festgelegt. "x_shift" und
; "y_shift" gegen den Abstand zweier Palettenplätze in X- und Y-Richtung an.
;
      LOCAL palref, palref_y                ; lokale Positionen bzgl. "pallet"
      LOCAL x, y                            ; Schleifenindizes
      LOCAL place                           ; aktuelle Position auf der Palette

      OPENI
      SET palref_y = NULL                   ; Start der Palettierung bei "pallet"
      FOR y = 1 TO y_max                    ; Schleife über Y-Richtung
            SET palref = palref_y           ; aktuelle Transformation setzen
            FOR x = 1 TO x_max              ; Schleife über X-Richtung
                  APPRO pick_up, 100.0      ; Anfahren der Aufnahmeposition
                  MOVES pick_up
                  CLOSEI                    ; Greifer schließen
                  DELAY 1.0                 ; 1 Sekunde warten
                  DEPARTS 100.0
                  APPRO place, 100.0        ; Anfahren der akt. Ablageposition
                  MOVES place
                  OPENI                     ; Greifer öffnen
                  DELAY 1.0                 ; 1 Sekunde warten
                  DEPARTS 100.0
                  SET palref = SHIFT( palref BY x_shift, 0.0, 0.0 )
                              ; eine Position auf Palette in X weiter gehen
            END
            SET palref_y = SHIFT( palref_y by 0.0, y_shift, 0.0 )
                              ; eine Position auf Palette in Y weiter gehen
      END
      RETURN
.END


.PROGRAM pallet_main()
;
; Palettierungsprogramm: Zunächst werden durch die Unterprogramme
; "def_pallet_frame" und "def_pickup_frame" der Referenzpunkt der Palette und
; eine Aufnahmeposition bestimmt. Anschließend wird mit Hilfe des Unterprogramms
; "do_pallet" die Palettierung durchgeführt.
;
      CALL def_pallet_frame( pallet )
      CALL def_pickup_frame( pickup )
      CALL do_pallet( pallet, pick_up, 6, 6, 100, 200 )
      STOP
.END
```