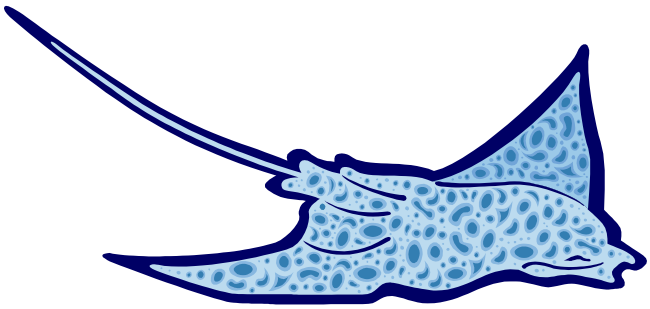
Notizen und Übersichten zu

RoboFace

Ulrich Müller



Inhaltsverzeichnis

Graphik : Einfügen | Grafik | AusDatei | Office | Rochen.WMF

Übersichten, Konzepte

Konzept

Klasse RoboFace verwaltet die Resource Interface. In einem eigenen Thread werden mögliche Ereignisse festgestellt. Die xxxInput / yyyOutput Objekte beschreiben einen einzelnen Port des Interfaces. Sie bereiten die Events auf, die RoboFace anbietet und lösen ggf. eine zugeordnete Ereignisroutine aus. Zusätzlich besitzen sie noch Methoden, die meist mit den Ereignissen korrepondieren. Bei ComplexOutput-Objekten werden meist mehrere Einzelobjekte zu einer Funktionseinheit zusammengefaßt (z.B. Lichtschranke).

Da RoboFace einen eigenen Thread nutzt, greift es auf die Resourcen des Interfaces zu ohne dabei unterbrechbar zu sein (kein DoEvents). Der Thread kann aber über NotHalt abgebrochen werden. Die Komponenten-Objects greifen meist unter Nutzung von DoEvents (kann beim ConnectRobo unterbunden werden) auf die Interface-Resourcen zu, so ist bei engen Schleifen die Unterbrechbarkeit gewährleistet.

Die Anregung zu dieser Klassenbibliothek verdanke ich der Diplomarbeit von Stephan Jätzold : "Objekte, Threads und Events für Roboter – Ein Toolkit zur hardware-unabhängigen Robotersteuerung in Java" (www.jaetzold.de/art/). Zu der es dann aber auch ganz deutliche Unterschiede gibt :

- RoboFace ist zur ausschließlichen Unterstützung von fischertechnik Interfaces (z.Zt. das Intelligent Interface) ausgelegt. Die Ansteuerung des Interfaces erfolgt nicht über einen Bytestream der COM-Schnittstelle, sondern über eine in VC++ geschriebene systemkonforme DLL, die das Polling der Interface-Eingänge übernimmt.
- Die Programmiersprache ist C#
- Die Ereignisorientierung der Objekte ist immer von korrespondierenden Methoden begleitet. Das erlaubt auch eine problemlose "lineare" Programmierung, die bei einer Vielzahl von Aufgaben deutlich praktischer ist.
- Es wird zusätzlich zum Thread der Anwendung nur ein weiterer Thread zur Handhabung der Ereignisse eingesetzt (die genutzte umFish30.DLL ihrerseits setzt noch einen – durch den MultiMediaTimer gesteuerten – Thread für das Polling des Interfaces ein). Dadurch liegen die Ereignisroutinen selber im Ereignis-Thread. Sie beeinflussen dadurch die Laufzeit des Ereignis-Threads relevant und unterliegen deswegen entsprechenden Größenbeschränkungen und auch Beschränkungen beim Einsatz von Methoden der Input/Output-Objekte.

RoboFace zeigt bei ereignisgesteuerten Programme ein sehr gutes Zeitverhalten, wenn man die o.g. Restriktionen beachtet. Gemischte Programme (lineare Programme mit ereignisgesteuerter Anzeige von Werten) sind völlig unproblematisch. Bei rein linearen Programmen kann der Ereignis-Thread abgeschaltet werden. Die Zeitaussagen beziehen sich auf einen 1,7 GHz-Rechner unter Windows 2000 SP4 und .NET Final SP2.

Klassenstruktur

- **RoboFace**:RoboBase : Zentrale Steuerung, Triggern der Ereignisse, Hilfsmethoden.
- **BinaryInput** : Verarbeiten der E-Eingänge im RawMode.
  - **Sensor** : Klasse für Taster
  - **PhotoTransistor** : Klasse für eine Phototransitor
  - **ReedContact** : Klasse für einen Reedkontakt
  - **ImpulseSensor** : Zählen von Impulsen an einem E-Eingang.
- **AnalogInput** : Verarbeiten der Analog-Eingänge im RawMode.
  - **PhotoResistor** : Klasse für einen Photowiderstand
  - **NTC** : Klasse für einen Temperatursensor
  - **PotentioMeter** : Klasse für ein Potentiometer
- **DualOutput** : Ansteuerung eines zweipoligen (Plus/Minus schaltbar) M-Ausganges.
  - **Motor** : Klasse für Motoren
  - **RobMotor** : Verbund von Motor (Motor), Endeingang und Impulseingang. Der Motor kann auf eine vorgebbare Position (Anzahl Impulse ab Home) verfahren werden.
  - **DLamp** : Klasse für Lampen
  - **DMagnet** : Klasse für Magneten.
  - **DPneuValve** : Klasse für Pneumatik-Ventile
  - **Buzzer** : Klasse für einen Summer
- **MonoOutput** : Ansteuerung eine einpoligen (ein M-Pin, Masse) M-Ausganges.
  - **Lamp** : Klasse für Lampen
  - **Magnet** : Klasse für Magneten
  - **PneuValve** : Klasse für Pneumatik-Ventile
- CombinedOutput
  - **LightBarrier** : Lichtschranke aus Lampe(Lamp/DLamp) und Phototransitor (PhotoTransistor)
  - Lights
  - **LimitedMotor** : Motor (Motor), dessen Bewegungsraum durch zwei e-Eingänge (BinaryInput) begrenzt werden.
  - **RobMotors** : Verbund von mehreren RobMotor-Objekten zu einem Roboter.
  - StepMotor
  - StepMotors

Zentrale Klassen

RoboFace

Zentrale Steuerung. Abgeleitet von RoboBase (abgemagerte Version von FishFace)

Auslösen der Ereignisse : BinaryEvent und AnalogEvent (Eigener Thread : RoboThread(RoboExecute)), die dann in den entsprechenden Input-Objekten zu eigenen Ereignissen verarbeitet werden.

Allgemein : Das Exception Handling hängt noch.
Fehlt : Alive, feststellen, signalisieren, ob RoboExecute noch läuft.

Member

ConnectRobo, **DisconnectRobo** (M, p)

StartRobo, **StopRobo** (M, p)

internal AddEvent (M, p) und die internal Klassen BinaryEvent und AnalogEvent

Exceptions

RoboFaceException

RoboBa30

Funktionen, die direkt (über cs-Funktionen von umFish30.DLL) auf das Interface zugreifen. Zusätzlich die Enums und Exception-Klassen. Methoden sind teilweise internal um einen Zugriff von Klassen des RoboFa30, aber nicht der Anwendung zu erlauben.

LPT-Eigenschaften wurden entfernt, nicht aber LPT.

Enums

Dir, Nr, Port, Speed, Wait

Klassen

FishFaceException, RoboBase

RoboBase-Member (Property/Method/Event,Delegate, public/internal)

AnalogScan (P, p)

AnalogEX, AnalogEY (P, i)

ClearCounter, ClearCounters, ClearMotors(M, i)

CloseInterface (M, i)

ESC (P, p)

Finish (M, p)

GetAnalog, GetAnalogDirect (M, i)
GetAnalogPur (M, i)

GetCounter (M, i)

GetInput, GetInputs (M, i)

Inputs (P, i)

NotHalt (P, p)

OpenInterface (M, i)

Outputs (P, i)

Pause (M, p)

PollInterval (P, p)

PortName (P, p)

Setcounter (M, i)

SetLamp (M, i)

SetMotor /SetMotors (M, i)

Slave (P, p)

Version (P, p, s)

WaitForChange (M, i)

WaitForInput (M, i)

WaitForMotor (M, i)

Einfache Komponenten-Klassen

BinaryInput

Verarbeiten der E-Eingänge. BinaryInput ist der RawMode.
Weitere Klassen **Sensor** (Taster), **PhotoTransistor**, **ReedContact** z.Zt. ohne weitere Eigenschaften (base : BinaryInput).

Konstruktor

BinaryInput(RoboFace ft, int InputNumber)
BinaryInput(RoboFace ft, Nr InputNumber)

Ereignisse

ChangedToTrue : Der E-Eingang hat sich von false auf true verändert

ChangedToFalse : Der E-Eingang hat sich von true auf false verändert

Changed : Der Delegate dazu (object sender)

Methoden

WaitForHigh : Warten auf Wechsel nach true

WaitForInput : Warten auf Erreichen des vorgegebenen Standes (true/false)

WaitForLow : Warten auf Wechsel nach false

SignalEvent : interne Methode zum Signalisieren der Ereignisse

Eigenschaften

IsTrue : Stand des E-Einganges (true/false)

InputNumber : Nummer des zugehörenden E-Eingangs

ImpulseSensor

Zählen der Impulse an einem einzelnen Eingang. Öffnen/Schließen werden als einzelne Impulse gezählt. Die Zählung kann durch Erreichen des wahlweise angebbaren Endtasters vorzeitig abgebrochen werden.
base : BinaryInput. **Zusätzlich** gelten natürlich die Member von BinaryInput

Konstruktor

ImpulseSensor(RoboFace ft, int InputNumber)
ImpulseSensor(RoboFace ft, int InputNumber, BinaryInput TermInput)
ImpulseSensor(RoboFace ft, Nr InputNumber)
ImpulseSensor(RoboFace ft, Nr InputNumber, BinaryInput TermInput)

Ereignisse

ChangedCount : Der Zählerstand hat sich geändert

FinalCount : Der vorgegebene Endstand wurde erreicht, alternativ der Endtaster

ImpulseCount : Der Delegate dazu (object sender, int ActualCount, bool Ready)

Methoden

CountChanges : Zähle eine vorgegebenen Anzahl von Impulsen, beginnend ab Null

CountPosition : Zähle bis zum vorgegebenen Zählerstand, beginnend mit dem Stand des **ActualCounter**. Ist der ActualCounter > als der vorgegebene Zählerstand, wird rückwärts gezählt (abgezogen).

WaitForDone : Warten auf das Ende der Zählung.

Eigenschaften

ActualCounter : Aktueller Counterstand nach abgeschlossener Zählung

AnalogInput

Verarbeiten der Analog-Eingänge. AnalogInput ist der RawMode.
Weitere Klassen (base : AnalogInput) : **PhotoResistor**, **NTC**, **PotentioMeter** z.Zt. ohne weitere Eigenschaften.

Konstruktor

AnalogInput(RoboFace ft, int AnalogNumber, int LowValue, int HighValue)
AnalogInput(RoboFace ft, Nr AnalogNumber, int LowValue, int HighValue)

Ereignisse

LimitLow : Der Analog-Eingang hat einen Wert unter LowValue angenommen.

LimitHigh : Der Analog-Eingang hat einen Wert über HighValue angenommen.

LimitNormal : Der Analog-Eingang hat einen Wert zwischen LowValue und HighValue angenommen (jeweils einschließlich)

Limit : Der Delegate dazu (object sender, int Value)

Methoden

SignalEvent : internal Methode zum Signalisieren der Ereignisse

Eigenschaften

AnalogNumber : Nummer des Analog-Einganges (EX = 0, EY = 1)

ActValue : Aktueller Wert des Analog-Einganges

ValueHigh : Ende des Normalbereichs

ValueLow : Beginn des Normalbereichs

DualOutput

Ansteuerung eines zweipoligen (Plus/Minus-schaltbar) M-Ausganges

Konstruktor

DualOutput(RoboFace ft, int OutputNumber)
DualOutput(RoboFace ft, Nr OutputNumber)

Methoden

Left : Einschalten des des M-Ausganges, nominell links (1) mit höchster Spannung

Right : Einschalten des M-Ausganges, nominell rechts (2) mit höchster Spannung

Off : Ausschalten des M-Ausganges (0)

Go : Einschalten des M-Ausganges mit Richtungsvorgabe, Spannung wählbar

Eigenschaften

State : Status des M-Ausganges (Links : 1, Rechts : 2, Aus : 0)

OutputNumber : Nummer des M-Ausganges

Abgeleitet

Mit zusätzlichen Bezeichnungen (Methoden) für die Schaltzustände.

Motor : Forward, Backward mit wählbarer Geschwindigkeit 1 - 15

DLamp : On

DMagnet : On

DPneuValve : On

Buzzer : On

RobMotor

Betrieb eines Motors für eine bestimmte Anzahl von Impulsen durch einen Verbund von Motor, Impulstaster, Endtaster. Der Impulstaster wird durch ein auf einer vom Motor getriebenen Welle sitzendes "Impulsrad" betätigt. Die Taster sind fest zugeordnet (M1 , E2, E1 ....) . Aus der Anzahl der gezählten Impulse läßt sich die Position des vom Motor getriebenen Bauteils ableiten. base : DualOutput.

Konstruktor

RobMotor(RoboFace ft, int MotNr, int MaxPosition)
RobMotor(RoboFace ft, Nr MotNr, int MaxPosition)

Ereignisse

ChangedPosition : Die aktuelle Position hat sich geändert.

FinalPosition : Die Endposition wurde erreicht (laut Fahrauftrag, Endtaster, MaxPosition)

Position : Der Delegate dazu (object sender, int Pos, bool Ready)

Methoden

DriveHome : Anfahren der Home-Position. Erreichen des Endtasters, linksdrehend.

DriveTo : Anfahren einer vorgegebenen Position in Impulsen ab Endtaster (0)

DriveDelta : Fahren von der aktuellen Position um eine vorgegebene Anzahl von Impulsen. Negative Werte nach links zum Endtaster.

WaitForDone : Warten auf Erreichen der vorgegebenen Position bzw. Endtaster / MaxPosition.

Eigenschaften

MotorNumber : M-Ausgangsnummer des Motors

EndSwitch : E-Eingangsnummer des Endtasters

ImpulseSwitch : E-Eingangsnummer des Impulstasters

ActualPosition : Erreichte Position in Impulsen ab Endtaster (im Stillstand, laufende Position : Ereignis ChangedPosition und internal property ActPosition).

State : Aktueller Status (1 : dreht links, 2 : rechts, 0 : steht).

MaxPosition : Maximal mögliche Position in Impulsen ab Endtaster

MonoOutput

Ansteuerung eines einpoligen (ein M-Pin, Masse) M-Ausganges

Konstruktor

MonoOutput(RoboFace ft, int OutputNumber)
MonoOutput(RoboFace ft, Nr OutputNumber)

Methoden

On : Einschalten des Ausganges

Off : Ausschalten des Ausganges

Eigenschaften

OutputNumber : Nummer des M-Pins, Zählung 1 für M1 vorn, 2 M1 hinten, bis 8

State : Status des Ausganges (0 : aus, 1 : an).

Abgeleitet

Lamp, **Magnet**, **PneuValve**

z.Zt. ohne weitere Eigenschaften

Komplexe Komponenten-Klassen

Abgeleitet von object.

LightBarrier

Lichtschranke aus einer Lampe (wahlweise ein- (Lamp) oder zweipolig(DLamp)) und einem PhotoTransistor. Basis : object.

Konstruktor

LightBarrier(RoboFace ft, DLamp Lampe, PhotoTransitor Photo)
LightBarrier(RoboFace ft, Lamp Lampe, PhotoTransistor Photo)

Ereignisse

ChangedToBroken : Lichtschranke ist unterbrochen

ChangedToFree : Lichtschranke ist OK

Changed : Delegate dazu (object sender)

Methoden

On : Einschalten der Lichtschranke

Off : Ausschalten der Lichschranke

WaitForBroken : Warten auf Unterbrechung der Lichtschranke

WaitForFree : Warten auf eine "heile" Lichtschranke

WaitForPassed : Warten auf einen Durchgang durch die Lichtschranke

Die Wait-Methoden korrepondieren mit den Ereignissen und können alternativ eingesetzt werden.

Eigenschaften

IsBroken : Ist die Lichtschranke unterbrochen

IsFree : Ist die Lichtschranke "heil".

LimitedMotor

Motor, dessen Bewegungsraum durch zwei Enddtaster begrenzt wird, Bewegungen immer von Endtaster nach Endtaster (z.B. Motor für Parkhausschranke).

Konstruktor

LimitedMotor(RoboFace ft, Motor Mot, BinaryInput ELeft, BinaryInput ERight)

Ereignisse

LeftToTrue : Der linke Endanschlag wurde erreicht

RightToTrue : Der rechte Endanschlag wurde erreicht
Changed : Der Delegate dazu (object sender)

Methoden

GoLeft : Drehen bis zum Linksanschlag, der Motor wird dann abgeschaltet.

GoRight : Drehen bis zum Rechtsanschlag, dto.

WaitForDone : Warten auf Erreichen eines Endanschlages. Der Motor wird dann abgeschaltet.
Korrespondiert mit den Ereignissen und kann alternativ eingesetzt werden.

Eigenschaften

MotorNumber : M-Ausgangsnummer des Motors

LeftSwitch : E-Eingangsnummer des linken Endanschlages.

RightSwitch : dto. rechts.

State : Status des Motors (0 : steht, 1 : links-, 2 : rechtsdrehend)

IsRight : Ist der rechte Endanschlag erreicht.

IsLeft : dto. links.

RobMotors

Betrieb einer Liste von RobMotor-Objekten in einer (Robot-)Einheit.

Konstruktor

RobMotors(RoboFace ft, params RobMotor[] RMotList)

Ereignisse

ChangedPosition : Änderung der aktuellen Position min. eines der beteiligten RobMotor-Objekte.

FinalPosition : Die vorgegebene Position aller beteiligten RobMotor-Objekte wurde erreicht.

PositionS : Der Delegate dazu (object sender, int[] Pos, bool Ready)

Methoden

MoveHome : Anfahren der Home-Position aller beteiligten RobMotor-Objekte : Erreichen des jeweiligen Endtasters.

MoveTo : Anfahren der vorgegebenen Positionen in Impulsen ab Endtaster (0). Die Positionen werden in der Reihenfolge angegeben in der die RobMotor-Objekte bei der Instanzierung angegeben wurden. Unveränderte Positionen am Ende der Liste können entfallen.

MoveDelta : Fahren von der aktuellen Position um eine vorgegebene Anzahl von Impulsen mit allen beteiligten RobMotor-Objekten. Reihenfolge wie bei der Instanzierung, unveränderte Position am Ende der Liste kann entfallen. Negative Werte : hin zum jeweiligen Endtaster.

WaitForDone : Warten auf Erreichen der vorgegebenen Positionen bzw. der zugehörenden Endtaster / MaxPosition.

Eigenschaften

State : Status des RobMotoren-Verbundes 0 : alle Motoren stehen, 1 : min. einer läuft.

Beispielprogramme

Aufbau einer Anwendung

Windows-Anwendung mit einer Hauptform.

Klassendeklarationen

Zusammenfassende Deklarationen der Klasse Roboface und der Klassen-Deklarationen aller genutzten Komponenten des Interfaces. Sollte zu Beginn der Hauptform der (Windows) Anwendung stehen.

Instanzierung, Ereigniszuweisung

Instanzierung aller Klassen im Konstruktor der Hauptform (Eine Instanzierung im Body der Form-Klasse ist nicht möglich). Ebenso eine Zuweisung der Ereignisroutinen zu den Komponenten-Objekten. Ein späteres Löschen ist über den Operator -= möglich.

Betriebsroutinen der Anwendung

Methoden der Klasse frmMain mit Teilaufgaben der Anwendung.

RoboFace Ereignis-Routinen

Stehen zu Beginn des Programmes in einem geschlossenen Block.

WinForm Ereignis-Routinen

Folgen danach. Typisch sind Click-Routinen für Command-Buttons in denen RoboFace gestartet und beendet werden kann. Bei dafür geeigneten Anwendungen sind keine weiteren Anwendungs-Routinen mehr erforderlich, die Verarbeitung erfolgt in den RoboFace Ereignis-Routinen. In den anderen Fällen, das dürfte die Mehrzahl sein, Start einer Anwendung über einen entsprechenden Button und dann weiter, wie bei Windows gewohnt.

RoboFaceTest

Testrahmen für einzelne Objekte, wird nach Bedarf modifiziert. Hier der aktuelle Stand.

Mobile Robot (Trusty)

Bumper Robot mit zwei einzeln angetriebenen Rädern und einem dritten Stützrad. Vorn ist ein Stoßfänger mit je einem Taster (Schließer) links und rechts, die die Anstöße registrieren.

Das Beispiel ist eigentlich recht ungeeignet für diese Klassenbibliothek, da hier ständig eine Kabelverbindung zum Robot bestehen muß, die ihn (den Robot) stark bei der Arbeit beeinträchtigt. Jedoch ist sie als Vergleichsobjekt zur (Diplom)arbeit von Jätzold von Interesse.

MR2.ftC

VBA-Programm mit vbaFish30 als Entwicklungsumgebung. Dient als knappe Übersicht der Aufgabenstellung und als Vergleich mit einem "EinObjektProgramm".

```
Const LeftMotor      = 1
Const RightMotor     = 2
Const ImpulseSensor  = 1
Const LeftSensor     = 3
Const RightSensor    = 4
Const Forward        = ftiLeft
Const Backward       = ftiRight
Const BackCount      = 16
Const TurnCount      = 24

Sub Main
  Do
    RunForward
    If GetInput(LeftSensor) Then
      RunBack BackCount
      LeftTurn TurnCount
    ElseIf GetInput(RightSensor) Then
      RunBack BackCount
      RightTurn TurnCount
    End If
  Loop Until Finish
End Sub

Sub RunBack(Inc)
  SetMotor LeftMotor, Backward
  SetMotor RightMotor, Backward
  WaitForChange ImpulseSensor, Inc
End Sub

Sub LeftTurn(Inc)
  SetMotor LeftMotor, Forward
  SetMotor RightMotor, Backward
  WaitForChange ImpulseSensor, Inc
End Sub

Sub RightTurn(Inc)
  SetMotor LeftMotor, Backward
  SetMotor RightMotor, Forward
  WaitForChange ImpulseSensor, Inc
End Sub

Sub RunForward()
  SetMotor LeftMotor, Forward
  SetMotor RightMotor, Forward
End Sub
```

MR2Linear.CS

C#-Programm mit einem linearen Programm auf Basis der Klassenbibliothek RoboFa30.DLL

Klassendeklarationen

```
private RoboFace      ft;
private Motor         leftMotor;
private Motor         rightMotor;
private ImpulseSensor impulseSensor;
private Sensor        leftSensor;
private Sensor        rightSensor;
```

Instanzierung, Ereignisse

```
ft            = new RoboFace();
leftMotor     = new Motor(ft, 1);
rightMotor    = new Motor(ft, 2);
impulseSensor = new ImpulseSensor(ft, 1);
leftSensor    = new Sensor(ft, 3);
rightSensor   = new Sensor(ft, 4);
```

MR2 : Betriebsroutinen

```
private void RunBack(int Inc) {
      leftMotor.Backward();
      rightMotor.Forward();
      impulseSensor.CountChanges(Inc);
      impulseSensor.WaitForDone();}
private void LeftTurn(int Inc) {
      leftMotor.Forward();
      rightMotor.Backward();
      impulseSensor.CountChanges(Inc);
      impulseSensor.WaitForDone();}
private void RightTurn(int Inc) {
      leftMotor.Backward();
      rightMotor.Forward();
      impulseSensor.CountChanges(Inc);
      impulseSensor.WaitForDone();}
private void RunForward() {
      leftMotor.Forward();
      rightMotor.Forward();}
```

MR2 : Hauptprogramm

Untergebracht in der Click-Ereignisroutine des Buttons cmdAction. Zusätzlich vorhanden ist ein Label-Control für Status-Anzeigen.

```
try {
      ft.ConnectRobo(2);
      ft.StartRobo();
      cmdAction.Enabled = false;
      lblStatus.Text = "--- MR2 gestartet ---";
      do {
            RunForward();
            if(leftSensor.IsTrue) {
                  RunBack(BackCount);
                  LeftTurn(TurnCount);
            }
            else if(rightSensor.IsTrue) {
                  RunBack(BackCount);
                  RightTurn(TurnCount);
            }
      } while(!ft.Finish());
      lblStatus.Text = "--- MR2 wird beendet ---";
      ft.Pause(1234);
      ft.StopRobo();
      ft.DisconnectRobo();
      this.Close();
}
catch (RoboFaceException eft) {
lblStatus.Text = eft.Message;
}
```

Das Programm wird in einer Endlosschleife betrieben, die durch ESC abgebrochen werden kann.

MR2Event.CS

C#-Programm mit einem ereignisgesteuerten Programm, ebenfalls auf Basis von RoboFa30.DLL

Klassendeklarationen

Wie gehabt.

Instanzierung, Ereignisse

Instanzierung wie gehabt, hinzukommt die Anmeldung der genutzten Ereignisroutinen

```
leftSensor.ChangedToTrue  += new BinaryInput.Changed(TurnLeft);
rightSensor.ChangedToTrue += new BinaryInput.Changed(TurnRight);
```

MR2 : Betriebsroutinen

Wie gehabt

RoboFace Ereignis-Routinen

```
private void TurnLeft(object sender) {
      RunBack(BackCount);
      LeftTurn(TurnCount);
      RunForward();}
private void TurnRight(object sender) {
      RunBack(BackCount);
      RightTurn(TurnCount);
      RunForward();}
```

Aufgerufen bei left/rightSensor == true. Hinzugekommen ist jeweils ein RunForward, das bisher im Endlos do zentral stand.

MR2 : Hauptprogramm

```
try {
      if(cmdAction.Text == "&START") {
            ft.ConnectRobo(2);
            ft.StartRobo();
            cmdAction.Text = "&HALT";
            lblStatus.Text = "--- MR2 gestartet ---";
            RunForward();
      }
      else {
            lblStatus.Text = "--- MR2 wird beendet ---";
            ft.Pause(1234);
            ft.StopRobo();
            ft.DisconnectRobo();
            this.Close();
            }
      }
catch (RoboFaceException eft) {
      lblStatus.Text = eft.Message;
}
```

Die Rahmenteile wie gehabt. Hinzugekommen ist ein RunForward um "das Ding in Gang zzubringen". Weggefallen ist die gesamte, endlose, do Schleife. Das eigentliche Programm findet zwischen den Ereignissen statt : Instanzierung, Ereignisse, RoboFace Ereignis-routinen. s.o.

Parkhausschranke

 Eine durch einem Motor angetriebene Schranke, deren Endlagen durch Taster begrenzt wird. Ein Durchfahrtwunsch wird durch einen Taster signalisiert, die Schranke öffnet und schließt dann wieder, wenn eine Lichtschranke durchfahren wurde. Der Betriebszustand wird durch eine rote und eine grüne lampe signalisiert.

Parkhaus1.ftC

VBA-Programm mit vbaFish30 als IDE. Dient als knappe Übersicht der Aufgabenstellung und als Vergleich mit einem "EinObjektProgramm".

```
Const mSchranke = 1, mRot = 2, mGruen = 3,   mLicht = 4
Const eZu = 1,       eAuf = 2, eOeffnen = 3, ePhoto = 4
Const sZu = 1,       sAuf = 2

Sub Main
  SetMotor mLicht,    ftiEin
  SetMotor mRot,      ftiEin
  Do
  SetMotor     mSchranke, sZu
  WaitForInput            eZu
  SetMotor     mSchranke, ftiAus
  WaitForHigh  eOeffnen
  SetMotor     mSchranke, sAuf
  WaitForInput eAuf
  SetMotor     mSchranke, ftiAus
  SetMotor     mRot,      ftiAus
  SetMotor     mGruen,    ftiEin
  WaitForLow   ePhoto
  WaitForHigh  ePhoto
  WaitForTime  250*EA
  SetMotor     mGruen,    ftiAus
  SetMotor     mRot,      ftiEin
  Loop Until Finish
End Sub
```

ParkhausLinear.CS

C#-Programm mit einem linearen Programm auf Basis von RoboFa30.DLL. Bei diesem Beispiel werden für die Lichtschranke (Lampe und Phototransitor) und den Schrankenmotor (Motor und zwei Endtaster) ComplexOutputs genutzt.

Klassendeklarationen

```
private RoboFace        ft;
private LimitedMotor    Schranke;
private LightBarrier    Durchfahrt;
private Sensor          Anforderung;
private DLamp           Rot;
private DLamp           Gruen;
```

Instanzierung, Ereignisse

```
ft          = new RoboFace();
Schranke    = new LimitedMotor(ft, new Motor(ft, 3),
                    new Sensor(ft, 5), new Sensor(ft, 7));
Durchfahrt  = new LightBarrier(ft, new DLamp(ft, 4),
                     new PhotoTransistor(ft, 2));
Anforderung = new Sensor(ft, 1);
Rot         = new DLamp(ft, 2);
Gruen       = new DLamp(ft, 1);
```

Betriebsroutinen

Keine, es konnte alles im Hauptprogramm untergebracht werden.

Hauptprogramm

```
try {
      ft.ConnectRobo(2);
      ft.StartRobo();
      cmdAction.Enabled = false;
      Durchfahrt.On();
      Rot.On();
      lblStatus.Text = "--- Schranke betriebsbereit ---";
      do {
            Schranke.GoLeft();
            Schranke.WaitForDone();
            Anforderung.WaitForHigh();
            Schranke.GoRight();
            Schranke.WaitForDone();
            Rot.Off();
            Gruen.On();
            Durchfahrt.WaitForPassed();
            ft.Pause(555);
            Rot.On();
            Gruen.Off();
      } while(!ft.Finish());
      lblStatus.Text = "--- Schrankenbetrieb wird beendet ---";
      ft.Pause(1234);
      ft.StopRobo();
      ft.DisconnectRobo();
      this.Close();
}
catch (RoboFaceException eft) {
      lblStatus.Text = eft.Message;
}
```

Das Programm wird auch hier in einer Endlosschleife betrieben, die durch ESC abgebrochen werden kann. Die Schranke ist hier – zusammen mit den beiden begrenzenden Endtastern – ein geschlossenes Objekt. Das gleiche gilt für die Lichtschranke mit Lampe und Phototransistor. Die Nutzung ist hier geschlossener. Bei der Implementierung der LimitedMotor-Methoden wurden der asychronen Form (analog Motor) der Vorzug gegeben. Das erfordert hier ein entsprechendes Wait. Die Wait-Routinen laufen intern nicht in einem separaten Thread. Sie verwenden Schleifen, die durch Abgabe der Kontrolle an die zentrale Windowsschleife (DoEvents) unterbrechbar sind. Zur Resourcenschonung zusätzlich noch ein Thread.Sleep genutzt.

ParkhausEvent.CS

C#-Programm mit einem ereignisgesteuerten Programm, ebenfalls auf Basis von RoboF30.DLL.

Klassendeklarationen

Wie gehabt.

Instanzierung, Ereignisse

Instanzierung wie gehabt, hinzukommt die Anmeldung der genutzten Ereignisroutinen

```
Anforderung.ChangedToTrue +=
new BinaryInput.Changed(SchrankeOeffnen);
Schranke.RightToTrue      +=
new LimitedMotor.Changed(SchrankeFreigeben);
Durchfahrt.ChangedToFree  +=
new LightBarrier.Changed(SchrankeSchliessen);
Schranke.LeftToTrue       +=
new LimitedMotor.Changed(SchrankeWarten);

SchrankeBusy = false;
```

Zusätzlich wird hier noch ein globales Feld zur Sperrung von Ereignisses eingeführt. Alternativ wäre ein dyn. An- und Abmelden der betroffenen Ereignisse (anstelle des Setzens von SchrankeBusy) möglich (Anmelden +=, Abmelden -=).

Ereignisroutinen

```
private void SchrankeOeffnen(object sender){
      if(SchrankeBusy) return;
      SchrankeBusy = true;
      lblStatus.Text = "--- SchrankeBusy ---";
      Schranke.GoRight();}
private void SchrankeFreigeben(object sender){
      Rot.Off();
      Gruen.On(); }
private void SchrankeSchliessen(object sender) {
      ft.Pause(555);
      Rot.On();
      Gruen.Off();
      Schranke.GoLeft();}
private void SchrankeWarten(object sender) {
      SchrankeBusy = false;
      lblStatus.Text = "--- Schranke betriebsbereit ---";}
```

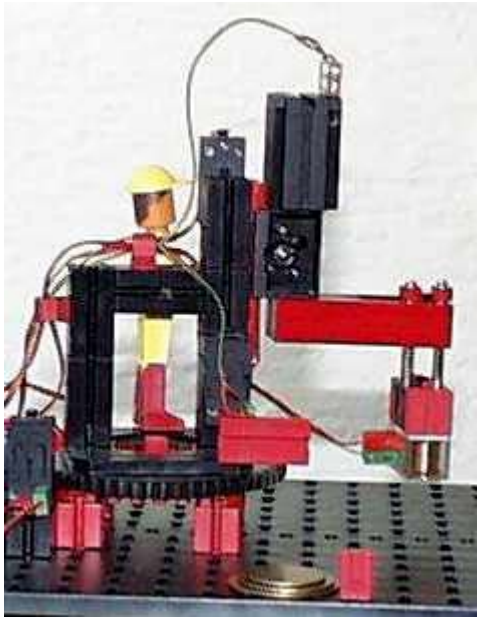
Die Ereignisroutinen sind in der Reihenfolge aufgeführt, wie sie in einem Betriebszyklus genutzt werden. SchrankeWarten tritt zusätzlich auch bei Start des Programms in Aktion.

Hauptprogramm

```
try {
      if(cmdAction.Text == "&START") {
            ft.ConnectRobo(2);
            ft.StartRobo();
            cmdAction.Text = "&HALT";
            Durchfahrt.On();
            Rot.On();
            Schranke.GoLeft();
      }
else {
      lblStatus.Text = "--- Schrankenbetrieb wird beendet ---";
            ft.Pause(1234);
            ft.StopRobo();
            ft.DisconnectRobo();
            this.Close();
      }
}
catch (RoboFaceException eft) {
      lblStatus.Text = eft.Message;
}
```

Hier wird zu Anlauf des Programmes noch etwas zur Herstellung einer definierten Ausgangslage getan : Schranke zu (die Methode ist asynchron, auf "zu" reagiert wird in der Ereignisroutine SchrankeWarten. Der eigentliche Betrieb findet zwischen der Ereignissen statt. Wenn man wissen will, was da läuft, sieht man am besten in der linearen Variante nach – oder gleich bei VBA.

HanoiRobot


Ein Stapel von Eisenscheiben wird von einem gefedert gelagerten Magneten von der Ausgangsposition unter Nutzung einer Zwischenposition auf die Endposition umgestapelt. Bedingung : nur kleinere Scheiben dürfen auf die größeren gestapelt werden. Siehe auch www.ftcomputing.de/hanoivba.htm

HanoiRobot.CLS

Hier wieder als Referenz eine VBA-Lösung (IDE : vbaFish30) vorgestellt. Das Programm besteht aus einem Hauptprogramm (RobotMain.ftC) und der hier vorgestellten Klasse HanoiRobot, in der die Funktionen zur Handhabung des Robots (nicht die Problemlösung) zusammengefaßt sind :

```
' --- Eigenschaften ------------
Public mSaule&
Public mArm&
Public mGreifer&
Public eArmOben&
Public eArmUnten&
Public PosA&
Public PosB&
Public PosC&

' --- Interne Variable ----------
Private RobotPos&
Private eSauleImpuls&
Private eSauleEnde&

' --- public Methoden --------
Public Sub Grundstellung()
' --- Fahren auf Grundstellung ---
  SetMotor mGreifer, ftiAus
  ArmHeben
  RobotPos = 999
  SauleNach 0
  SauleNach PosA
End Sub
```

```
Public Sub SauleNach(ZielPos&)
' --- Fahren auf ZielPosition --------
  If RobotPos < ZielPos Then
    SetMotor mSaule, ftiRechts, ftiFull, ZielPos-RobotPos
    WaitForMotors 0, mSaule
  Else
    SetMotor mSaule, ftiLinks, ftiFull, RobotPos-ZielPos
    WaitForMotors 0, mSaule
  End If
  RobotPos = ZielPos + GetCounter(eSauleImpuls)
End Sub

Public Sub ArmSenken()
' --- Senken des Greiferarms -----
  SetMotor mArm, ftiLinks
  WaitForInput eArmUnten
  SetMotor mArm, ftiAus
End Sub
Public Sub ArmHeben()
' --- Heben des Greiferarms ------
  SetMotor mArm, ftiRechts
  WaitForInput eArmOben
  SetMotor mArm, ftiAus
End Sub
Public Sub ScheibeLegen()
' --- Ablegen der Scheibe ----
  SetMotor mGreifer, ftiAus
End Sub
Public Sub ScheibeGreifen()
' --- Aufnehmen der Scheibe ---
  SetMotor mGreifer, ftiEin
End Sub

' --- Konstruktor ----------------------
Private Sub Class_Initialize()
' --- Setzen der Standardwerte ---
  mSaule       = 1
  mArm         = 2
  mGreifer     = 3
  eArmOben     = 3
  eArmUnten    = 4
  eSauleImpuls = (mSaule-1)*2+2
  eSauleEnde   = (mSaule-1)*2+1
  PosA         = 30
  PosB         = 75
  PosC         = 120
End Sub
```

HanoiRobot.CS (Linear)

C#Klasse mit linerarer Programmierung, die ebenfalls die Robot-Funktionen abbildet.

Klassendeklarationen

Im klassenglobalen Teil :

```
private RoboFace     ft;
private RobMotor     mSaule;
private LimitedMotor mArm;
private DMagnet      mGreifer;
```

Instanzierung

Im Konstruktor die Instanzierung und die Initialisierung der Hanoi-Positionen (die zugehörenden Eigenschaften im weiteren Teil der Klasse fehlen hier) :

```
this.PortNr = PortNr;
ft = new RoboFace();
mSaule   = new RobMotor(ft, Nr.M1, 180);
mArm     = new LimitedMotor(ft, new Motor(ft, Nr.M2),
                            new Sensor(ft, Nr.E4),
                             new Sensor(ft, Nr.E3));
mGreifer = new DMagnet(ft, Nr.M3);

PosA = 14;
PosB = 52;
PosC = 95;
```

Die Methoden

```
public void Start() {
      ft.ConnectRobo(PortNr);
      this.Grundstellung();}
public void Stop() {
      ft.Pause(555);
      ft.DisconnectRobo();}
public void Grundstellung(){
      mGreifer.Off();
      mArm.GoRight();
      mArm.WaitForDone();
      mSaule.DriveHome();
      mSaule.WaitForDone();
      mSaule.DriveTo(PosA);
      mSaule.WaitForDone();}
public void SauleNach(int ZielPos) {
      mSaule.DriveTo(ZielPos);
      mSaule.WaitForDone();}
public void ArmSenken() {
      mArm.GoLeft();
      mArm.WaitForDone();}
public void ArmHeben() {
      mArm.GoRight();
      mArm.WaitForDone();}
public void ScheibeLegen() {
      mGreifer.Off();}
public void ScheibeGreifen() {
      mGreifer.On();}
```

sind eigentlich überraschend einfach. Das liegt natürlich auch an der Verwendung der komplexen Objekte RobMotor und LimitedMotor, die die zugehörenden Tasterabfragen einschließen. RobMotor macht auch noch eine Buchhaltung über die aktuelle Position.

Bei Start und Stop wurde auf die Angabe von ft.StartRobo / ft.StopRobo verzichtet, da hier keine Ereignisse eingesetzt werden. Man könnte aber, wenn man die aktuelle Position von mSaule anzeigen will.

In Grundstellung wird nacheinander der Robot in seine Ausgangsposition gebracht :

- Greifermagnet abschalten
- Arm in obere Position fahren
- Säule gegen Endtaster fahren und dann auf PosA

Da die letzten beiden Methoden asynchron sind, könnte man auch mit

```
mArm.GoRight();
mSaule.DriveHome();
mArm.WaitForDone();
mSaule.WaitForDone();
```

die Positionen simultan anfahren. Es ist aber nicht zu empfehlen, da ein tiefstehender Greifer verhaken könnte.

Die weiteren Methoden kapseln ganz einfach Methoden des jeweiligen Robot-Objektes.

Hauptprogramm

"verhandelt" dann nur noch mit Methoden der Klasse HanoiRobot :

```
  private HanoiRobot hr;
  hr = new HanoiRobot(2);

  private void Ziehe(int Von, int Nach) {
        Hole(Von);
        Bringe(Nach);}
  private void Hole(int Pos) {
        hr.SauleNach(Pos);
        hr.ArmSenken();
        hr.ScheibeGreifen();
        hr.ArmHeben();}
  private void Bringe(int Pos) {
        hr.SauleNach(Pos);
        hr.ArmSenken();
        hr.ScheibeLegen();
        hr.ArmHeben();}
```

in der Click-Routine cmdAction_Click wird dann die Hanoilogik angestoßen :

```
        hr.Start();
        Hanoi(3, hr.PosLinks, hr.PosMitte, hr.PosRechts);
        hr.Stop();
```

Die Hanoi-Routine sieht dann so aus :

```
  private void Hanoi(int n, int TAnf, int TMit, int TEnd) {
    if(n == 1) {
        Ziehe(TAnf, TEnd);}
    else {
        Hanoi(n-1, TAnf, TEnd, TMit);
        Ziehe(TAnf, TEnd);
        Hanoi(n-1, TMit, TAnf, TEnd); }
    }
```

Industry Robot

Industry Robot von 1998, Aufbau nach Bauanleitung. Geeignet sind der Knickarm-Robot und der Säulen-Robot, der zweite ist im Test wesentlich nervenschonender.

RoboStep.CS

Mit der nebenstehenden Bedienoberfläche zum Betrieb des Robots in Form von einzelnen Aktionen (Teil des Projektes RobStep.SLN). Die Aktionen sind in den zum jeweiligen Button passenden Click-Routinen untergebracht. Die oberen fünf Buttons sind auf einem Panel plaziert über das sie gemeinsam Enabled/Disabled werden können um eine Mehrfachbediendung zu verhhindern. Hinter dem ENDE-Button steckt noch eine Parken-Routine, mit der der Robot wieder so zusammengefaltet werden kann, daß er ins Regal paßt.

Klassendeklaration

```
private RoboFace  ft;
private RobMotor  mSaule;
private RobMotor  mArmV;
private RobMotor  mArmH;
private RobMotor  mGreifer;
```

Instanzierung

```
ft       = new RoboFace();
mSaule   = new RobMotor(ft, Nr.M1, 222);
mArmV    = new RobMotor(ft, Nr.M3, 111);
mArmH    = new RobMotor(ft, Nr.M2, 88);
mGreifer = new RobMotor(ft, Nr.M4, 26);
```

Zu beachten ist, daß beim RobMotor zu zugehörenden E-Eingänge festzugeordnet sind : M1 mit Endtaster E1 und Impulstaster E2 ...

Start / Ende

```
private void cmdAction_Click(object sender, System.EventArgs e) {
try{
      ft.ConnectRobo(cboPortName.SelectedIndex);
      cmdAction.Enabled = false;
      lblStatus.Text    = "läuft";}
catch(RoboFaceException eft) {
      lblStatus.Text = eft.Message; }}
private void cmdEnde_Click(object sender, System.EventArgs e) {
      Parken();
      ft.DisconnectRobo();
      this.Close();
}
```

Verbindung zum Interface über ft.ConnectRobo, Interface-Anschluß aus ComboBox, keine Event-Routinen. Vor dem Beenden wird noch Parken() aufgerufen s.o.

Aktions-Click-Routinen

```
private void cmdNachA_Click(object sender, System.EventArgs e) {
      pnlSteps.Enabled = false;
      lblStatus.Text = "Nach A";
      mSaule.DriveTo(110);
      mSaule.WaitForDone();
      mArmV.DriveTo(45);
      mArmH.DriveTo(80);
      mArmV.WaitForDone();
      mArmH.WaitForDone();
      pnlSteps.Enabled = true;
}
```

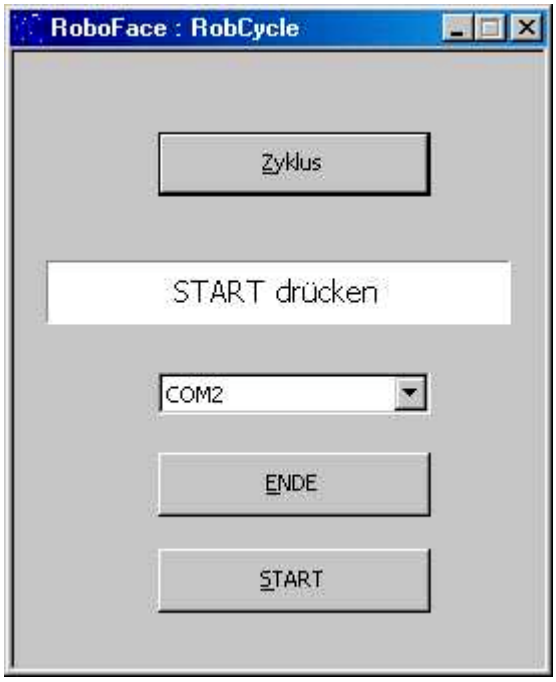
Zuerst wird das Panel, auf dem der zugehörende Button plaziert ist, Disabled um weitere Clicks zu unterbinden. Am Schluß der Routine wird das Panel wieder Enabled.

Die eigentliche Routine ist simpel : Säule auf Position fahren, Arm vertikal und horizontal starten und auf Ende warten. Das wars. Man hätte hier auch noch die Säule simultan betreiben können, doch in Praxis kann sich der Arm dann leicht irgendwo verhaken.

Die weiteren Click-Routinen sind nach dem gleichen Schema aufgebaut.

Die vollständige Source ist im ZIP-Päckchen enthalten.

RobCycle.CS in Projekt RobCycle.SLN



Ablauf ähnlich RobStep. Hier sind aber die Einzelschritte zu einem Zyklus zusammengefaßt, der beliebig oft aufgerufen werden kann.

Der wesentlichste Unterschied gegenüber RobStep ist der Einsatz der komplexen Klasse RobMotors anstelle von viermal RobMotor. RobMotors faßt die vier Motoren des Robots zu einer Einheit zusammen. Anstelle der Methode DriveTo für ein RobMotor Objekt, tritt hier die Methode MoveTo des Objektes RobMotors mit der alle vier Motoren gleichzeitig angesteuert werden können.

Außerdem wird im Statusfeld während des Zyklusbetriebes laufend die aktuelle Position des Robots angezeigt. Das geschieht über eine Ereignisroutine.

Klassendeklaration

```
private RoboFace    ft;
private RobMotors   iRob;
```

Instanzierung

```
ft    = new RoboFace();
iRob  = new RobMotors(ft,
      new RobMotor(ft, Nr.M1, 222),     // --- Säule
      new RobMotor(ft, Nr.M3, 111),     // --- Arm vertikal
      new RobMotor(ft, Nr.M2, 88),      // --- Arm horizontal
      new RobMotor(ft, Nr.M4, 26));     // --- Greifer
```

Als Parameter für iRob werden die bekannten RobMotor Objekte verwendet. Ihre Reihenfolge ist für die Methoden MoveTo und MoveDelta verbindlich.

Startroutine

Ist die Buttonroutine cmdAction_Click

```
private void cmdAction_Click(object sender, System.EventArgs e) {
try{
      ft.ConnectRobo(cboPortName.SelectedIndex);
      ft.StartRobo();
      cmdAction.Enabled = false;
      lblStatus.Text = "Nach Hause";
      iRob.MoveHome();
      iRob.WaitForDone();
      iRob.ChangedPosition +=
            new RobMotors.PositionS(RobPosition);
      }
      catch(RoboFaceException eft) {
            lblStatus.Text = eft.Message; }
}
```

Hier erfolgt über StartRobo jetzt auch ein Start der zentralen Ereignisroutine und mit iRob.ChangedPosition += ... die Zuordnung der Ereignisroutine für die Positionsanzeige (RobPosition). Das erfolgt erst hier nach dem iRob.MoveHome, da eine Anzeige während des MoveHome sinnlos ist, da die aktuelle Position hier nicht bekannt ist. In der Routine cmdEnde_Click wird sie dann vor Anfahren der Parkposition aus dem gleichen Grunde wieder abgeschaltet.

Ereignisroutine

```
private void RobPosition(object sender, int[] Pos, bool Ready) {
lblStatus.Text = Pos[0].ToString() + " - " + Pos[1].ToString()
      + " - " + Pos[2].ToString();}
```

Zyklus

```
private void cmdZyklus_Click(object sender, System.EventArgs e) {
      cmdZyklus.Enabled = false;
      iRob.MoveTo(145);                // --- Säule auf Pos 145
      iRob.WaitForDone();
      iRob.MoveTo(145, 45, 45);        // --- Arm auf Pos 45v / 45h
      iRob.WaitForDone();
      iRob.MoveTo(145, 45, 45, 24);    // --- Greifer schließen
      iRob.WaitForDone();
      iRob.MoveTo(145, 0);             // --- Arm hoch auf 0
      iRob.WaitForDone();
      iRob.MoveTo( 45);                // --- Säule auf Pos 45
      iRob.WaitForDone();
      iRob.MoveTo( 45, 75);            // --- Arm vertikal 75
      iRob.WaitForDone();
      iRob.MoveTo( 45, 75, 45, 0);     // --- Greifer öffnen
      iRob.WaitForDone();
      cmdZyklus.Enabled = true;
}
```

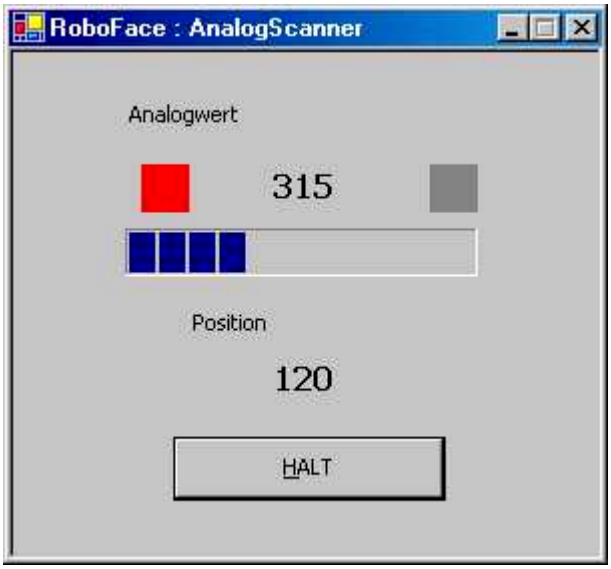
Die Clickroutine cmdZyklus_Click enthält den vollständigen Code zur Steuerung des angegebenen Zyklus. Verwendet wird hier die Methode MoveTo und die Routine WaitForDone um die Ausführung des Fahrbefehl abzuwarten. Bei MoveTo werden anzufahrenden Positionen in Form von Positionen für die einzelnen Motoren in der Reihenfolge "ihres Erscheinens" bei der Instanzierung angegeben. Unveränderte Positionen am Ende der Liste müssen nicht angegeben werden. Die Positionsangaben erfolgen in Impulsen ab 0 (Position am Endtaster).
Der Button wird während des Betriebes abgeschaltet.

Die vollständige Source ist auch hier im ZIP-Päckchen enthalten.

AnalogScanner

Scannen der Analoganzeigen eines Photowiderstandes, der auf einem Robot-Turm oszillierend gedreht wird. Aufbau : hier Aufsatz eines Photowiderstandes, der mit einem Winkelstein am Motor des HanoiRobots (siehe Kapitel HanoiRobot) befestigt ist.

AnalogScanner.CS



Der Analogwert wird als digitaler Wert und als Balken angezeigt. Wird der LowValue unterschritten, geht die linke "Lampe" an, wird der HighValue überschritten geht die rechte an, im Normalbereich sind beide Lampen aus.

Klassendeklaration

```
private RoboFace      ft;
private RobMotor      Turm;
private PhotoResistor Photo;
```

Instanzierung

```
ft    = new RoboFace(true, false, 0);
Turm  = new RobMotor(ft, Nr.M1, 175);
Photo = new PhotoResistor(ft, Nr.EX, 333, 555);
```

Bei RoboFace hier der Parameter AnalogScan = true. Der Turm dreht einen Winkel von 175 Impulsen (Vorsicht Motor des Robots vorherhochfahren). Der Photo(Resistor) löst bei < 333 LowValue und bei > 555 HighValue aus.

Start / Ende

Wie gewohnt in der cmdAction_Click Routine :

```
try {
      if(cmdAction.Text == "&START") {
            ft.ConnectRobo(2);
            ft.StartRobo();
            cmdAction.Text = "&HALT";

            Turm.DriveHome();
            Turm.WaitForDone();
```

```
            Turm.ChangedPosition +=
               new RobMotor.Position(ScanAnalog);
            Photo.LimitHigh += new AnalogInput.Limit(Hoch);
            Photo.LimitLow  += new AnalogInput.Limit(Niedrig);
            Photo.LimitNormal += new AnalogInput.Limit(Normal);

            Normal(this,0);
            Turm.DriveTo(Turm.MaxPosition);
        }
        else {
```

Beim Start wird wie gewohnt die Verbindung zum Interface hergestellt und der RoboThread gestartet, anschließend fährt der Turm auf HomePosition. Dann werden die Ereignisroutinen für ScanAnalog, Hoch, Niedrig und Normal aktiviert. Das Normal-Ereignis wird angestoßen, der Turm fährt zur MaxPosition. Weitere Action sieht man nicht, die liegt in den Eventroutinen.

Der Rest ist bekannt.

Ereignisroutinen

ScanAnalog : Anzeige des aktuellen Analogwertes und der Position des Turmes. Außerdem wird die Drehrichtung des Turmes bei Bedarf (Ready == true) umgekehrt. Scananalog ist dem Turm zugeordnet.

```
  private void ScanAnalog(object sender, int Pos, bool Ready) {
        lblPhoto.Text = Photo.ActValue.ToString();
        pgbPhoto.Value = Photo.ActValue;
        lblPos.Text   = Pos.ToString();
        if(Ready && Pos == 0) Turm.DriveTo(Turm.MaxPosition);
        else if(Ready && Pos == Turm.MaxPosition) Turm.DriveTo(0);
  }
```

Bereichsüberschreitungen

Die Eventroutinen Hoch, Niedrig, Normal sind dem Objekt Photo zugeordnet und lösen entsprechend aus :

```
  private void Hoch(object sender, int Wert) {
        lblLow.BackColor  = Color.Gray;
        lblHigh.BackColor = Color.Red;
  }
  private void Niedrig(object sender, int Wert) {
        lblLow.BackColor  = Color.Red;
        lblHigh.BackColor = Color.Gray;
  }
  private void Normal(object sender, int Wert) {
        lblLow.BackColor  = Color.Gray;
        lblHigh.BackColor = Color.Gray;
  }
```

AnalogScannerLinear.CS

Lösung wie AnalogScanner.CS. Der Unteschied besteht in der Reduktion der Eventroutine ScanAnalog. Hier fehlt das if für die Drehrichtungsumkehr dafür ist in cmdAction eine entsprechende do-Schleife enthalten :

```
  Normal(this,0);
  do {
        Turm.DriveTo(Turm.MaxPosition);
        Turm.WaitForDone();
        Turm.DriveTo(0);
        Turm.WaitForDone();
  } while(!ft.Finish());
}
else {
```

Technische Anmerkungen

Ereignis-Loop RoboExecute in RoboFace

RoboExecute ist eine zentrale Einrichtung von RoboFace, die zur Erkennung von möglichen Ereignissen. Der Loop ist nur erforderlich, wenn in der Anwendung Ereignisse genutzt werden sollen. Er wird durch RoboStart gestartet und durch RoboStop beendet (geht z.Zt. nur einmal im Leben einer Anwendung). RoboExecute ist die CallBack-Routine des Threads RoboThread von RoboFace.

Jedes Robo-Objekt mit Bedarf an Ereignissen trägt bei seiner Instanzierung über RoboFace.AddEvent ein EventObject (BinaryEvent oder AnalogEvent) in die interne EventList von RoboFace ein. RoboExecute stellt in einem Loop die möglichen Ereignisse : Veränderung eines E-Einganges und Veränderung des Wertebereichs eines Analog-Einganges fest und ruft dann die SignalEvent-Methoden der betroffenen Robo-Objekte auf ohne zu prüfen, ob tatsächlich ein Anwendungsevent vorhanden ist. Die jeweilige SignalEvent-Methode prüft das dann und ruft ihrerseits die zugeordnete Eventroutine auf.

Bei komplexen Robo-Ojekten – d.h. Objekten, die ihrerseits RoboObjekte nutzen – können diese Ereignisse in eigenen Routinen abgefangen und weiterverarbeitet werden. Das führt dann zum Aufruf einer diesem (komplexen) Objekt zugeordneten Ereignisroutine.

Ereignisroutinen liegen logisch im Thread des Ereignisloops. Ihre Abarbeitung stoppt den Loop solange. Sie müssen deswegen sehr kurz sein oder in einen eigenen Thread verlegt werden. Ebenso wird die Taktzeit des Ereignisloops natürlich auch durch die Anzahl der Einträge in der Ereignisliste beeinflußt. Allerdings bei weitem nicht so erheblich wie es durch große Ereignisroutinen geschehen kann.

C# Details

Ereigniss-Routinen werden durch einen delegate beschrieben, z.B. :

```
public delegate void Position(object sender, int Pos, bool Ready);
```

zur Beschreibung eines RobMotor Events. sender enthält immer das auslösende Object (hier RobMotor) und kann in der Ereignisroutine der Anwendung zur Bestimmung des Kontextes herangezogen werden. Der Parameter Pos enthält die aktuelle Position des RobMotors und Ready gibt an, ob die Bewegung abgeschlossen wurde.

Mit :

```
public event Position ChangedPosition;
public event Position FinalPosition;
```

werden die zugehörenden Events deklariert.

und mit

```
rMotor.ChangedPosition += RobMotor.Position(rMotPos);
```

der Ereignisroutine rMotoPos in der Anwendung zugeordnet :

```
private void rMotPos(object sender, int Pos, bool Ready) {
    lblStatus.Text = Pos.ToString();}
```

In der Ereignisroutine hier wird lediglich die aktuelle Position angezeigt.

Mit :

```
rMotor.ChangedPosition -= RobMotor.Position(rMotPos);
```

kann die Ereignisroutine wieder entfernt werden.

In diesem Beispiel erfolgt die Anmeldung des "Ereignisbedarfs" bei RoboFace indirekt über das von RobMotor angelegte Objekt Sensor ImpulseCounter durch ein :

```
ft.AddEvent(new BinaryEvent(this, ENr, true);
```