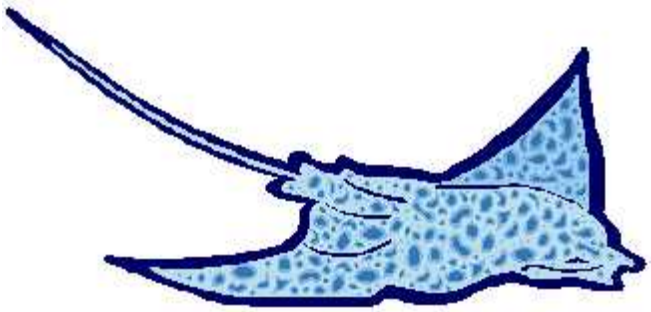
ftComputing

FishDevices40 für C#

Ulrich Müller



Inhaltsverzeichnis

Übersichten

Konzept

Allgemeines

Klasse **FishDevices**:**FishBase** mit Basis-Funktionen und der zentralen Verwaltung der aktiven Device-Objekte in einer DeviceList. Start der zugehörenden Threads.

Device-Klassen (Basis **DeviceBase**) für jeden Input-, Analog-Eingangstyp sowie die Outputtypen und die komplexen Klassen mit dem Zusammenwirken von mehreren Devices. Meist gibt es allgemeine Basistypen von denen dann rechteinfache spezielle abgeleitet werden (z.B. BinaryInput -> PhotoTransistor).

Wenn eine Device-Klasse über **Ereignisse** verfügt, stellt sie diese in einem eigenen EventLoop fest und löst sie auch aus. Die Ereignisverarbeitung ist pro Objekt abwählbar.

Eine Device-Klasse kann mehrere Ereignisse(Routinen) auslösen. Sie laufen im **Thread** der Ereignis-Erkennung(EventLoop). Langlaufende Ereignis-Routinen können also die Ereigniserkennung eines einzelnen Objektes beeinflussen. Die zugehörenden Ereignisroutinen haben als ersten Parameter stets object sender mit dem aktuellen Parameter this. Bei der Instanzierung der Device-Klassen kann festgelegt werden (WithEvents) ob Events ausgeführt werden sollen.

Unterbrechbarkeit

Die Unterbrechbarkeit einer Anwendung (z.B. um Anzeigen auf der Form machen zu können) ist durch Ablauf der Eventroutinen in einem separaten Thread und durch Einbau von Application.DoEvents in die Mehrzahl der FishDevices / FishBase- Methoden gewährleiltet.

Abbrechbarkeit

Die eingesetzten Threads sind Background-Threads, die mit Beenden des Hauptthreads automatisch beendet werden. Sie können durch Suspend vorrübergehend angehalten und dann mit Resume wieder aufgenommen werden.

In einigen Methoden (Finish und Wait ...) werden Schleifen eingesetzt, die über die Eigenschaft NotHalt = true und die ESC-Taste beendet werden können dadurch werden gleichzeitig alle Threads beendet. Ein so unterbrochenes Programm kann erst wieder aufgenommen werden, wenn die Objekte, die Threads nutzen, neu instanziert wurden.

Klassenstruktur

Namespace : FishDevices40

- **FishDevices** :FishBase : Zentrale Steuerung, Triggern der Ereignisse, Hilfsmethoden.
- **BinaryInput** : DeviceBase : Verarbeiten der I-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.
- **IRInput** : DeviceBase : Verarbeiten der IR-Eingänge.
- **AnalogInput** : DeviceBase : Verarbeiten der Analog-Eingänge im RawMode.
  - **PhotoResistor** : Klasse für einen Photowiderstand
  - **NTC** : Klasse für einen Temperatursensor
  - **PotentioMeter** : Klasse für ein Potentiometer
- **VoltageInput** : DeviceBase : Verarbeiten der Spannungs-Eingänge im EchtMode (double Volt).
- **DualOutput** : DeviceBase : 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** : DeviceBase : Ansteuerung eines einpoligen (ein O-Pin, Masse) O-Ausganges.
  - **Lamp** : Klasse für Lampen
  - **Magnet** : Klasse für Magneten
  - **PneuValve** : Klasse für Pneumatik-Ventile
- *CombinedOutput*
  - **LightBarrier** : DeviceBase : Lichtschranke aus Lampe(Lamp/DLamp) und Phototransitor (PhotoTransistor)
  - Lights
  - **LimitedMotor** : DeviceBase : Motor (Motor), dessen Bewegungsraum durch zwei I-Eingänge (BinaryInput) begrenzt wird.
  - **RobMotor** : DualOutput : Kombination von Motor, Impuls- und Endtaster.
  - **RobMotors** : DeviceBase : Verbund von mehreren RobMotor-Objekten zu einem Roboter.
  - **ImpulseSensor** : DeviceBase : Zählen von Impulsen an einem I-Eingang.
  - StepMotor / StepMotors

Klassen

NameSpace

FishDevices40

Klasse : FishDevices: FishBase

Verwaltung der Resourcen eines Interfaces. Liste aller angemeldeten DeviceObjekte (DeviceList).

FishDevices()

Datei : FishDev40.CS

Methoden : internal **AddDevice**(DeviceBase DeviceObject)
Geführt in ArrayList DeviceList(DeviceBase)

Connect, **Disconnect** Verbindung zum Interface

Start der EventLoops der Device-Objekte

Resume der EventLoops der Device-Objekte – Wiederaufnehmen der Threads nach einem vorhergenenden Suspend

Suspend der EventLoops der Device-Objekte -
vorübergehendes Aussetzen der Threads

Ein Stop der DeviceThreads ist nicht erforderlich, da sie mit IsBackground = true laufen und bei Programmende automatisch beendet werden. Üblicherweise werden sie aber über DisConnect durch setzen von NotHalt = true durch Auslaufenlassen der Execute-Routine beendet. Dort werden vor dem Beenden der Verbindung zum Interface alle M-Ausgänge abgeschaltet.

Klasse : FishBase

Zugriff auf das Interface über cs-Schnittstellen-Funktionen auf Basis von umFish30.DLL

Datei : FishBa40.CS.

Enums : **IFTypen**, **Dir**, **Inp**, **IRCode**, **IRKeys**, **Out**, **Port**, **Speed**, **Wait**

Properties : **ActDevice**, **ESC**, **NotHalt**, **Outputs**, **Version**

Methoden public : **Finish**, **Pause**
protected : Sleep, GetTickCount, GetAsyncKeyState
internal : ClearCounter, ClearCounters, ClearMotors, CloseInterface, GetAnalog, GetAnalogPur, GetCounter, GetInput, GetInputs, GetIRKey, GetVoltage, OpenInterface, Outputs, SetCounter, SetLamp, SetMotor, SetMotors, WaitForChange, WaitForInput, WaitForMotor.

Abgestrippte Version von FishFace40.CS. Basiert auf umFish40.DLL mit FtLib von fischertechnik.

Basis-Klasse DeviceBase

Basis für alle weiteren Device-Klassen, wenn nicht besonders angegeben.

DeviceBase(fd, WithEvents, DeviceNumber)

Eigenschaften : **DeviceNumber**
ThreadInterval
WithEvent

internal : deviceThread

Device-Klassen

Allgemeines

EventLoop : Mit eigenem Thread in dem die für das Device relevanten Ereignisse in der Execute-Routine EventLoop ausgelöst werden. Der Thread ist IsBackground = true, beendet sich also auch bei Ende des Hauptthreads. Normalerweise wird er durch FishDevices.DisConnect durch Setzen von NotHalt = true beendet (ESC tuts auch).
WithEvents : Unterbrechbar.
Delegate : Name des Delegates wird angegeben.
Event : Aufzählung der angebotenen Ereignisse.

Klasse : BinaryInput

Verarbeiten der I-Eingänge (I1 – I32, bzw. E1 – E16) im Raw-Mode.

Ereignisse :	Eigener EventLoop, WithEvents, Changed **ChangedToTrue** **ChangedToFalse**
Eigenschaften :	**InputNumber** **IsTrue**
Methoden :	**WaitForInput**([OnOff]) **WaitForHigh**() **WaitForLow**()

Abgeleitete Klassen : Sensor, PhotoTransistor, ReedContact

z.Zt. nur Konstruktor

Klasse : IRInput

Verarbeiten der IR-Eingänge

Ereignisse :	Eigener EventLoop, WithEvents, Changed **ChangedToTrue** **ChangedToFalse**
Eigenschaften :	**IRCodeNr** **IRKeyNr** **IsTrue**
Methoden :	**WaitForKey**([OnOff])

Klasse : AnalogInput

Verarbeiten der Widerstands-Eingänge (AX, AY, AXS1-3 bzw. EX, EY) im Raw-Mode.

Ereignisse : Eigener EventLoop, WithEvents, Limit
ChangedToLow
ChangedToHigh
ChangedToNormal
ShowActualValue

Eigenschaften : **AnalogNumber**
ActualValue
LimitHigh
LimitLow

Abgeleitete Klassen : PhotoResistor, NTC, PotentioMeter

z.Zt. nur Konstruktor

Klasse : VoltageInput

Verarbeiten der Spannungs-Eingänge (A1, A2, AV) im Real-Mode

Ereignisse : Eigener EventLoop, WithEvents, Limit
ChangedToLow
ChangedToHigh
ChangedToNormal
ShowActualValue

Eigenschaften : **VoltNumber**
ActualValue (in double Volt)
LimitHigh (in double Volt)
LimitLow (in double Volt)

Klasse : DualOutput

Ansteuerung eines zweipoligen (Plus/Minus schaltbar) M-Ausganges (M1 – M16).

Ereignisse : keine, kein WithEvents

Eigenschaften : **State**
OutputNumber

Methoden : **On**([Speed])
Off()
Left([Speed])
Right([Speed])
Go(Dir [,Speed])

Abgeleitete Klassen

Motor

Spezielle Klasse für Motoren

Methoden : **Forward**([Speed])
Backward([Speed])
Up([Speed])
Down([Speed])

DLamp

Spezielle Klasse für zweipolig geschaltete Lampen

Methoden : **BlinkingOn**([FreqOn, FreqOff]) // --- Läuft im eigenen Thread
BlinkingOff()

DMagnet, DPneuValve

Spezielle Klassen für zweipolig geschaltete Magnete und Pneumatic-Schalter.
Z.Zt.nur Konstruktor

Buzzer

Spezielle Klasse für zweipolig geschalteten Summer.

Z.Zt. nur Konstruktor

Klasse : MonoOutput

Ansteuerung eines einpoligen (ein O-Pin, Masse) geschalteten O-Ausganges

Ereignisse : keine, kein WithEvents

Eigenschaften : **State**
OutputNumber

Methoden : **On**()
Off()

Abgeleitete Klassen

Lamp

Ansteuerung einer einpolig (ein O-Pin, Masse) geschalteten Lampe

Methoden : **BlinkingOn**([FreqOn, FreqOff]) // --- Läuft im eigenen Thread
BlinkingOff()

Magnet, PneuValve

nur Konstruktor

Komplexe Klassen

LightBarrier

Lichtschranke aus Lampe (Lamp / DLamp) und Phototransistor (PhotoTransistor).

Ereignis :	nutzt Events von BinaryInput(PhotoTransistor), WithEvents, Changed **ChangedToFree** **ChangedToBroken**
Eigenschaften :	**PhotoNumber** **LampNumber** **IsFree** **IsBroken**
Methoden :	**On**() **Off**() **WaitForBroken**() **WaitForFree**() **WaitForPassed**()

LimitedMotor

Motor (Motor), dessen Bewegungsraum durch zwei I-Eingänge (BinaryInput) begrenzt wird.

Ereignis :	nutzt Events von BinaryInput(Left-,RightSwitch), WithEvents, Changed **LeftToTrue** **RightToTrue**
Eigenschaften :	**MotorNumber** **MotorState** **LeftSwitch** **RightSwitch** **IsLeft** **IsRight**
Methoden :	**GoLeft**([Speed]) **GoRight**([Speed]) **WaitForDone**()

RobMotor

Kombination von Motor (Motor), Impuls- und Endtaster
(festzugeordnet M1 : I2, I1, M4 : I8, I7).

base :	DualOutput
Ereignis :	Eigener EventLoop, WithEvents, Position **ChangedPosition** **FinalPosition**
Eigenschaften :	**MotorNumber** **EndSwitch** **ImpulseSwitch** **MaxPosition** **CurrentPosition** **ActualPosition**
Methoden :	**DriveHome**() **DriveDelta**(Inc) **DriveTo**(Pos) **WaitForDone**()

RobMotors

Verbund von mehreren RobMotor-Objekten zu einem Roboter.

Ereignis :	Eigener EventTrigger – nutzt Events von RobMotor, WithEvents, PositionS **ChangedPosition** **FinalPosition**
Eigenschaften :	**State**
Methoden :	**MoveHome**() **MoveDelta**(DestDelta[]) **MoveTo**(DestPos[]) **WaitForDone**()

ImpulseSensor

Zählen von Impulsen an einem I-Eingang.

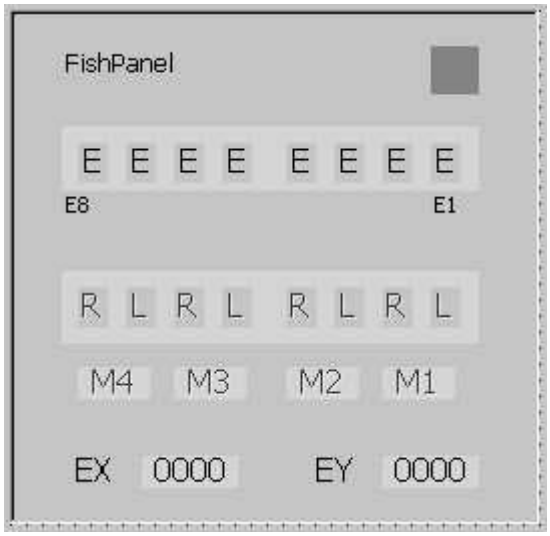
base :	BinaryInput
Ereignisse :	Eigener EventLoop, WithEvents, ImpulseCount **ChangedCount** **FinalCount**
Eigenschaften :	**CurrentCounter** (InputNumber)
Methoden :	**CountChanges** **CountPosition** **WaitForDone**

Leere Klassen

StepMotor, StepMotors, Lights

Controls

FishPanel



Anzeige der Interfacewerte und Bedienung der M-Ausgänge mit der Maus.

Eigenschaften :

Analog : mit/ohne Anzeige der Analogwerte

Slave : mit/ohne Anzeige der Slavewerte

Interface : Zuordnung eines Interfaceobjektes vom Typ FishDevices

Methoden :

Start : Start der Anzeige

Stop : Stop der Anzeige

Installieren

Menü Extras | Toolbox anpassen ...
.NET Frameworkkomponenten | Durchsuchen | FishDevices40.DLL auswählen.
Erscheint dann in der Toolbox (v1.0 ganz unten, v1.1 in extra Gruppe)

Beispielprogramme

Aufbau einer Anwendung

Windows-Anwendung mit einer Hauptform.

Klassendeklarationen

Zusammenfassende Deklarationen der Klasse FishDevices 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.

FishDevicesTest

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.

Der Robot sollte über das ROBO RF Datalink betrieben werden, da hier ein Kabel sehr störend ist.

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 FishDevices40.DLL

Klassendeklarationen, Konstanten

```
private FishDevices   fd;
private Motor         leftMotor;
private Motor         rightMotor;
private ImpulseSensor impulseSensor;
private Sensor        leftSensor;
private Sensor        rightSensor;

private const int BackCount = 8;
private const int TurnCount = 8;
```

Instanzierung, Ereignisse

```
fd            = new FishDevices();
leftMotor     = new Motor(fd, 1);
rightMotor    = new Motor(fd, 2);
impulseSensor = new ImpulseSensor(fd, false, 1);
leftSensor    = new Sensor(fd, false, 4);
rightSensor   = new Sensor(fd, false, 3);
```

MR2 : Betriebsroutinen

```
private void RunBack(int Inc) {
private void RunBack(int Inc) {
      leftMotor.Backward();
      rightMotor.Backward();
      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 {
      fd.Connect(IFTypen.ftROBO_first_USB, 0);
      fd.Start();
      cmdAction.Enabled = false;
      lblStatus.Text = "--- MR2 gestartet, Ende : ESC ---";
      do {
            RunForward();
            if(leftSensor.IsTrue) {
                  RunBack(BackCount);
                  LeftTurn(TurnCount);}
            else if(rightSensor.IsTrue) {
                  RunBack(BackCount);
                  RightTurn(TurnCount);}
      } while(!fd.Finish());
      fd.DisConnect();
      this.Close();}
catch (FishDevException efd) {lblStatus.Text = efd.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 FishDevices40.DLL. Betrieb über das ROBO RF Datalink. Zusätzlich – für manuelle Eingriffe in "verfahrenen" Situationen – eine IR-Fernbedienung. Außerdem werden die aktuelle Aktion und die Versorgungsspannung ständig angezeigt.

Klassendeklarationen

Wie gehabt, hinzukommen :

```
backLeft      = new IRInput(fd, IRCode.Code1, IRKeys.M1BW);
backRight     = new IRInput(fd, IRCode.Code1, IRKeys.M1FW);
powerSupply   = new VoltageInput(fd, true, Inp.AV, 2.50, 4.00);
BusyInTurn    = false;
```

Damit bei einem zentralen Kontakt, wo beide Taster nahezu gleichzeitig getätigt werden nicht wild durcheinander nach rechts und links gedreht wird auch noch eine Sperre (BusyInTurn), die die automatische Vorrangsteuerung des linearen Betriebes ersetzt.

Instanzierung, Ereignisse

Instanzierung wie gehabt, hinzukommt die Anmeldung der genutzten Ereignisroutinen für den Betrieb (TurnLeft / TurnRight), manuelle Korrekturen (BackLeft / BackRight) und die Anzeige der Versorgungsspannung (PowerFail / PowerNormal) :

```
leftSensor.ChangedToTrue    += new BinaryInput.Changed(TurnLeft);
rightSensor.ChangedToTrue   += new BinaryInput.Changed(TurnRight);
backLeft.ChangedToTrue      += new IRInput.Changed(BackLeft);
backRight.ChangedToTrue     += new IRInput.Changed(BackRight);
powerSupply.ChangedToLow    += new VoltageInput.Limit(PowerFail);
powerSupply.ChangedToNormal += new VoltageInput.Limit(PowerNormal);
```

MR2 : Betriebsroutinen

Wie gehabt & Anzeige der aktuellen Aktion und der Versorgungsspannung bei RunForward.

```
private void RunForward() {
   lblStatus.Text = "RunForward";
   lblPower.Text  = powerSupply.ActualValue.ToString("0.00V");
   leftMotor.Forward();
   rightMotor.Forward();}
```

FishDevices Ereignis-Routinen

```
private void TurnLeft(object sender) {
  if (BusyInTurn) return;
   BusyInTurn = true;
   RunBack(BackCount);
   LeftTurn(TurnCount);
   RunForward();
   BusyInTurn = false;}
private void TurnRight(object sender) {
   if (BusyInTurn) return;
   BusyInTurn = true;
   RunBack(BackCount);
   RightTurn(TurnCount);
   RunForward();
   BusyInTurn = false;}
private void BackLeft(object sender) {
   RunBack(BackCount);
   LeftTurn(TurnCount);
   RunForward();}
```

```
private void BackRight(object sender) {
   RunBack(BackCount);
   RightTurn(TurnCount);
   RunForward();}
private void PowerFail(object sender, int actValue) {
   lblPowerFail.BackColor = Color.Red; }
private void PowerNormal(object sender, int actValue) {
   lblPowerFail.BackColor = Color.GreenYellow;}
```

TurnLeft/Right (aufgerufen bei left/rightSensor == true). Hinzugekommen ist jeweils ein RunForward, das bisher im Endlos do zentral stand und eine Vorrangsteuerung, die verhindert, daß bei frontalem Kontakt, beide Ereigniss-Routinen ausgeführt werden.

BackLeft/Right : für manuelle Korrekturen mit dem IR-Sender.

PowerFail/Normal : zur Signalisierung des Zustandes der Versorgungsspannung. Man muß sich hier natürlich nicht aufs Signalisieren beschränken, man kann den Saft auch ganz abdrehen.

MR2 : Hauptprogramm

```
private void cmdAction_Click(object sender, System.EventArgs e) {
   try {
     if(cmdAction.Text == "&START") {
       fd.Connect(IFTypen.ftROBO_first_USB, 0);
       DeviceData dd = fd.ActDevice;
       lblActDevice.Text =
           String.Format("{0} ({1} / {2} ) Firmware : {3}",
            dd.Name, dd.Type, dd.SerialNr, dd.Firmware);
       fd.Start();
       cmdAction.Text = "&HALT";
       lblStatus.Text = "--- MR2 gestartet ---";
       RunForward();
       }
     else {
       lblStatus.Text = "--- MR2 wird beendet ---";
       fd.DisConnect();
       this.Close();
     }
   }
   catch (FishDevException efd) {lblStatus.Text = efd.Message;}
}
```

Die Rahmenteile wie gehabt. Hinzugekommen ist ein RunForward um "das Ding in Gang zubringen". Weggefallen ist die gesamte, endlose, do Schleife. Das eigentliche Programm findet zwischen den Ereignissen statt : Instanzierung, Ereignisse, FishDevices Ereignis-routinen. s.o. Neu ist die Anzeige des genutzten Interfaces samt Firmwareversion.
Ende hier über den Button cmdAction, der dann die Beschriftung HALT trägt.

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 FishDevices40.DLL. Bei diesem Beispiel werden für die Lichtschranke (Lampe und Phototransitor) und den Schrankenmotor (Motor und zwei Endtaster) ComplexOutputs genutzt.

Klassendeklarationen

```
private FishDevices     fd;
private LimitedMotor    Schranke;
private LightBarrier    Durchfahrt;
private Sensor          Anforderung;
private DLamp           Rot;
private DLamp           Gruen;
```

Instanzierung, Ereignisse

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

Betriebsroutinen

Keine, es konnte alles im Hauptprogramm untergebracht werden.

Hauptprogramm

```
try {
      fd.Connect(IFTypen.ftROBO_first_USB, 0);
      fd.Start();
      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();
            fd.Pause(555);
            Rot.On();
            Gruen.Off();
      } while(!fd.Finish());
      fd.DisConnect();
      this.Close();
}
catch (FishDevException efd) {
      lblStatus.Text = efd.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 FishDevices40.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) {
      fd.Pause(555);
      Rot.On();
      Gruen.Off();
      Schranke.GoLeft();
}
private void SchrankeWarten(object sender) {
      SchrankeBusy = false;
      lblStatus.Text = "--- Schranke betriebsbereit ---";
}
```

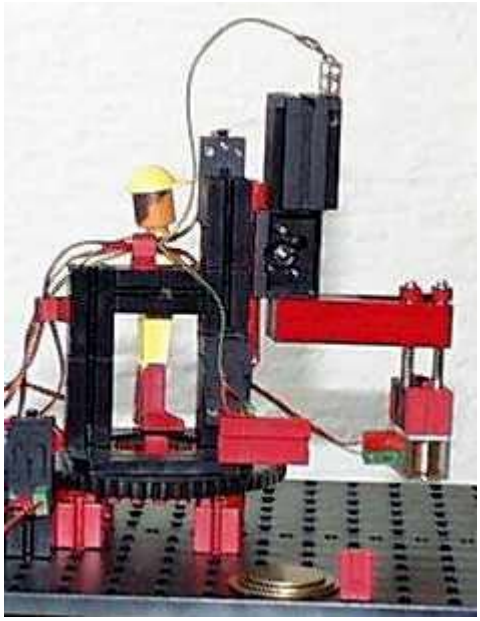
Die Ereignisroutinen sind in der Reihenfolge aufgeführt, wie sie in einem Betriebszyklus genutzt werden. SchrankeWarten tritt zusätzlich auch bei Start des Programms in Aktion.

Hauptprogramm

```
try {
      if(cmdAction.Text == "&START") {
            fd.Connect(IFTypen.ftROBO_first_USB, 0);
            fd.Start();
            cmdAction.Text = "&HALT";
            Durchfahrt.On();
            Rot.On();
            Schranke.GoLeft();
      }
      else {
      lblStatus.Text = "--- Schrankenbetrieb wird beendet ---";
            fd.Pause(1234);
            fd.DisConnect();
            this.Close();
      }
}
catch (FishDevException efd) {
      lblStatus.Text = efd.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 FishDevices  fd;
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;
fd       = new FishDevices();
mSaule   = new RobMotor(fd,     false, Out.M1, 180);
mArm     = new LimitedMotor(fd, false, new Motor(fd, Out.M2),
                                       new Sensor(fd, Inp.I4),
                                      new Sensor(fd, Inp.I3));
mGreifer = new DMagnet(fd, Out.M3);

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

Die Eigenschaften und Methoden

```
public int PosLinks  {get{return PosA;} set{PosA = value;}}
public int PosMitte  {get{return PosB;} set{PosB = value;}}
public int PosRechts {get{return PosC;} set{PosC = value;}}

public void Start() {
      fd.Connect(PortNr);
      Grundstellung();}
public void Stop() {
      fd.Pause(555);
      fd.DisConnect();}
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 :

```
InitializeComponent();
hr = new HanoiRobot(IFTypen.ftROBO_first_USB, 0);

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 FishDevices  fd;
private RobMotor     mSaule;
private RobMotor     mArmV;
private RobMotor     mArmH;
private RobMotor     mGreifer;
```

Instanzierung

```
fd       = new FishDevices();
mSaule   = new RobMotor(fd, Out.M1, 222);
mArmV    = new RobMotor(fd, Out.M3, 111);
mArmH    = new RobMotor(fd, Out.M2, 88);
mGreifer = new RobMotor(fd, Out.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 {
            fd.Connect(cboPortName.SelectedIndex);
            cmdAction.Enabled = false;
            lblStatus.Text    = "läuft";
      }
      catch(FishDevException efd) {lblStatus.Text = efd.Message;}
}
```

Verbindung zum Interface über fd.Connect, 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;
}
```

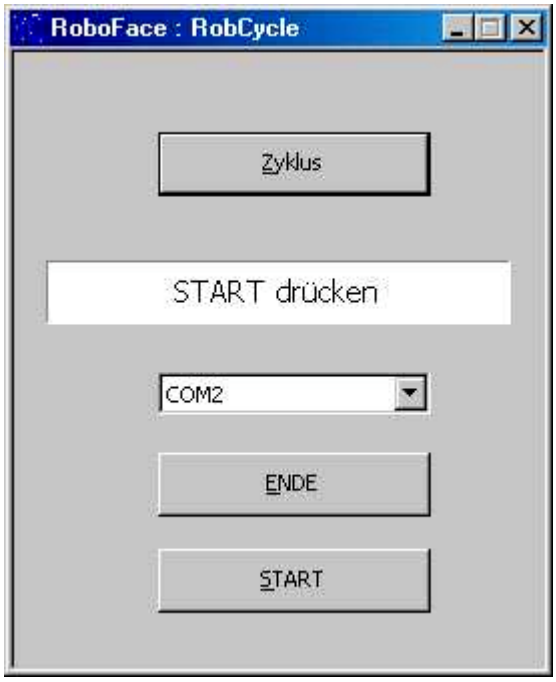
Zuerst wird das Panel, auf dem der zugehörende Button plaziert ist, Disabled um weitere Clicks zu unterbinden. Am Schluß der Routine wird das Panel wieder Enabled.

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 FishDevices fd;
private RobMotors   iRob;
```

Instanzierung

```
fd    = new FishDevices();
iRob  = new RobMotors(fd, true,
      new RobMotor(fd, Out.M1, 222),     // --- Säule
      new RobMotor(fd, Out.M3, 111),     // --- Arm vertikal
      new RobMotor(fd, Out.M2, 88),      // --- Arm horizontal
      new RobMotor(fd, Out.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{
            fd.Connect(cboPortName.SelectedIndex);
            fd.Start();
            cmdAction.Enabled = false;
            lblStatus.Text = "Nach Hause";
            iRob.MoveHome();
            iRob.WaitForDone();
            iRob.ChangedPosition +=
                  new RobMotors.PositionS(RobPosition);
            cmdZyklus.Enabled = true;}
      catch(FishDevException efd) {
            lblStatus.Text = efd.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;
}
```

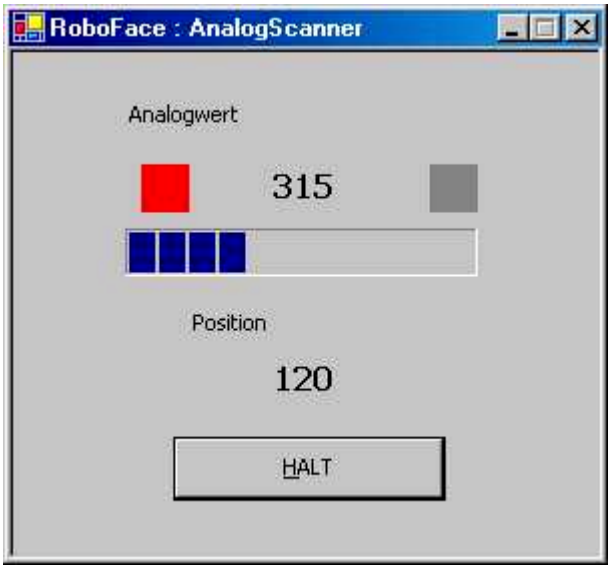
Die Clickroutine cmdZyklus_Click enthält den vollständigen Code zur Steuerung des angegebenen Zyklus. Verwendet wird hier die Methode MoveTo und die Routine WaitForDone um die Ausführung des Fahrbefehl abzuwarten. Bei MoveTo werden anzufahrenden Positionen in Form von Positionen für die einzelnen Motoren 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 FishDevices   fd;
private RobMotor      Turm;
private PhotoResistor Photo;
```

Instanzierung

```
fd    = new FishDevices();
Turm  = new RobMotor(fd, Out.M1, 175);
Photo = new PhotoResistor(fd, Inp.AX, 333, 555);
```

Bei FishDevices 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") {
  fd.Connect(IFTypen.ftROBO_first_USB, 0);
  fd.Start();
  cmdAction.Text = "&HALT";
  Turm.DriveHome();
  Turm.WaitForDone();
  Turm.ChangedPosition  += new RobMotor.Position(ScanAnalog);
  Photo.ChangedToHigh   += new AnalogInput.Limit(Hoch);
  Photo.ChangedToLow    += new AnalogInput.Limit(Niedrig);
  Photo.ChangedToNormal += new AnalogInput.Limit(Normal);
```

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

Beim Start wird wie gewohnt die Verbindung zum Interface hergestellt und die EventThread 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.ActualValue.ToString();
        pgbPhoto.Value = Photo.ActualValue;
        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 Unterschied 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 {
```

Dreipunkt-Regelung

Von dem ursprünglichen Programm, das in www.ftcomputing.de/dreip.htm vorgestellt wurde ist hier – mangels echtem Modell - nicht mehr viel übrig geblieben. Dieser Programmkomplex soll vorwiegend den Umgang mit Ereignissen und hier besonders mit langlaufenden Ereignisroutinen (länger als ein EventPollInterval) zeigen. FishDevices40 ist aus Effizienzgründen so konzipiert, daß pro Objekt im Bedarfsfall für den EventLoop (nicht alle Objekte haben EventLoops, eine Abwahl der Eventverarbeitung (Parameter WithEvents = false) ist ebenfalls möglich) ein eigener Thread gestartet wird in dem die auftretenden Ereignisse des Objekts ermittelt und ggf. auch ausgelöst werden. Wenn für ein Objekt mehrere Ereignisse auftreten können (beim PhotoResistor in den nachfolgenden Beispielen sind es gleich vier), werden einzelne Ereignisse (wie z.B. beim System.Timer auch) "verschluckt". Man sollte sich also kurz fassen :

DreipunktLampen

Ein Photowiderstand zur Messung der Helligkeit im Umfeld der Schreibtischlampe und zwei Lampen, die ein Unter- bzw. Überschreiten des vorgegebenen Helligkeitsbereich signalisieren. Zusätzlich wird auf der Form der aktuell gemessene Wert angezeigt. Die Regelung erfolgt manuel durch Bewegen der Schreibtischlampe. Bei der Anzeige ist zu beachten, daß die kleineren Werte eine größere Helligkeit signalisieren :

Klassendeklarationen

```
private FishDevices   fd;
private DLamp         LampeHigh;
private DLamp         LampeLow;
private PhotoResistor Photo;
```

Instanzierung, Ereignisse

```
fd     = new FishDevices();
LampeHigh = new DLamp(fd, Out.M1);
LampeLow = new DLamp(fd, Out.M2);
Photo  = new PhotoResistor(fd, Inp.AX, 333, 555);

Photo.ChangedToHigh   += new AnalogInput.Limit(Hoch);
Photo.ChangedToLow    += new AnalogInput.Limit(Niedrig);
Photo.ChangedToNormal += new AnalogInput.Limit(Normal);
Photo.ShowActualValue += new AnalogInput.Limit(ShowValue);
```

Ereignisroutinen

```
private void Hoch(object sender, int Wert) {
      LampeHigh.On();
}
private void Niedrig(object sender, int Wert) {
      LampeLow.On();
}
private void Normal(object sender, int Wert) {
      LampeHigh.Off();
      LampeLow.Off();
}
private void ShowValue(object sender, int Wert) {
      lblStatus.Text = Wert.ToString();
}
```

Kurz sind sie und – vor allem – sie bergen keine langlaufenden Schleifen (wie z.B. die WaitForxxx-Methoden).

Was passiert : In dem EventLoop des Objektes Photo wird laufend (Photo.ThreadInterval = 234 mSek.) der aktuelle Analogwert an EX gemessen und mit LimitHigh / LimitLow verglichen. Bei Veränderung im Wertebereich wird die zugehörende Eventroutine (Hoch, Niedrig, Normal) aufgerufen : LampeHigh, LampeLow an oder gleich beide aus (vorsichtshalber, weil hier nicht klar ist, woher der Wind weht). Zusätzlich immer ShowValue.

Start / Ende

```
private void cmdAction_Click(object sender, System.EventArgs e) {
      try{
            if(cmdAction.Text == "&START") {
                  fd.Connect(Port.COM2);
                  fd.Start();
                  Photo.ThreadInterval = 234;
                  cmdAction.Text = "&ENDE";
            }
            else {
                  fd.DisConnect();
                  this.Close();
            }
      }
      catch(FishDevException efd) {
            lblStatus.Text = efd.Message;
      }
}
```

Ist eigentlich ein wenig mager geraten. Es wird die Verbindung zum Interface hergestellt, die Eventloops gestartet und das Photo.ThreadInterval festgelegt und dann folgt schon das Ende. Der eigentliche Programmablauf spielt sich im EventLoop von Photo ab.

DreipunktBlinker

Schöner ist's natürlich, wenn die Lampen nicht nur Leuchten, sondern auch noch Blinken. Doch das kann schnell eine langlaufende Ereignisroutine werden. Hier wird das durch Nutzung des Photo.ThreadInterval und durch eine Kommunikation zwischen den Eventroutinen umgangen :

Deklarationen

```
private FishDevices   fd;
private DLamp         LampeHigh;
private DLamp         LampeLow;
private DLamp         Blinker;
private PhotoResistor Photo;

bool   FlipFlop;
string Limit;
```

Hinzugekommen ist eine dritte (virtuelle) Lampe Blinker, ein FlipFlop und, damit die Anzeige instruktiver wird, ein string Limit.

Die Instanzierung ist geblieben, wie gehabt.

Aber die Ereignisroutinen haben deutlich zugelegt :

Ereignisroutinen

```
private void Hoch(object sender, int Wert) {
      if(Blinker != null) Blinker.Off();
      Blinker = LampeHigh;
      Limit   = "Zu dunkel : ";
}
private void Niedrig(object sender, int Wert) {
      if(Blinker != null) Blinker.Off();
      Blinker = LampeLow;
      Limit   = "Zu hell : ";
}
private void Normal(object sender, int Wert) {
      if(Blinker != null) {
            Blinker.Off();
            FlipFlop = true;
            Limit    = "Normal : ";
            Blinker  = null;
      }
}
private void ShowValue(object sender, int Wert) {
      lblStatus.Text = Limit + ((1024-Wert)/10).ToString();
      if(Blinker != null) {
            if(FlipFlop) {Blinker.On();  FlipFlop = false;}
            else {Blinker.Off(); FlipFlop = true;}}}
```

Kern der Veranstaltung ist die "virtuelle" Lampe Blinker, der in Hoch bzw. Niedrig das jeweils zuständige Lampen-Objekt zugewiesen wird. Da ShowValue ohnehin ständig am werkeln ist, ist hier der Ort zum Blinken. Aber ohne Schleife, deswegen nach Maßgabe des FlipFlop Blinker.On() oder Blinker.Off(). Die Dauer wird durch das eingestellte Photo.ThreadInterval (hier 222 mSek.) bestimmt. Außerdem wird mit Limit zusätzlich andeutet, was man von dem angezeigten Wert halten soll. Und schöner ist er (der Analogwert) auch geworden : aufsteigende Werten von klein (dunkel) nach groß (hell) und nicht mehr so unanständig große Zahlen (aber Lux/Lumen sind es immernoch nicht).

Und nun das gleiche nochmal mit VB.NET (natürlich unter Nutzung der gleichen, in C# geschriebenen, Assembly FishDevices40.DLL) :

DreipunktBlinker.VB

Deklarationen, Instanzierung

```
Dim fd As New FishDevices()
Dim LampeHigh As New DLamp(fd, Out.M1)
Dim LampeLow As New DLamp(fd, Out.M2)
Dim Blinker As DLamp
Dim WithEvents Photo As New PhotoResistor(fd, Inp.AX, 333, 555)

Dim FlipFlop As Boolean
Dim Limit As String
```

erfolgt hier in einem Aufwasch. Das Anmelden der Ereignisroutinen ist nicht erforderlich, es geschieht automatisch mit der Klausel WithEvents vom Objekt Photo. Die Zuordnung der Ereignisroutinen wird dann direkt an der Routine in der Handles-Klausel notiert.

Ereignisroutinen

```
Private Sub Photo_ChangedToHigh(ByVal sender As Object, _
  ByVal Value As Integer) Handles Photo.ChangedToHigh
    If Not Blinker Is Nothing Then Blinker.Off()
    Blinker = LampeHigh
    Limit = "Zu dunkel : "
End Sub
Private Sub Photo_ChangedToLow(ByVal sender As Object, _
  ByVal Value As Integer) Handles Photo.ChangedToLow
    If Not Blinker Is Nothing Then Blinker.Off()
    Blinker = LampeLow
    Limit = "Zu hell : "
End Sub
Private Sub Photo_ChangedToNormal(ByVal sender As Object, _
  ByVal Value As Integer) Handles Photo.ChangedToNormal
    If Not Blinker Is Nothing Then
      Blinker.Off()
      FlipFlop = False
      Limit = "Normal : "
      Blinker = Nothing
    End If
End Sub
Private Sub tmrAnalog_Tick(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles tmrAnalog.Tick
    lblStatus.Text = Limit & Photo.ActualValue.ToString()
    If Not Blinker Is Nothing Then
      If FlipFlop Then
        Blinker.On()
        FlipFlop = False
      Else
        Blinker.Off()
        FlipFlop = True
      End If
    End If
  End Sub
```

Der Ablauf ist wie bei der C#-Lösung. Im Unterschied dazu wird jedoch nicht das ShowActualValue von Photo genutzt, sondern ein ganz normales Timer-Control, das geht genauso schön, vor allem auch dann, wenn das Objekt über kein passendes Ereignis verfügt. Hier muß man sich dann selber um den aktuellen Analogwert kümmern : lblStatus.Text = Limit & Photo.ActualValue.ToString()

DreipunktLampenBlinker

Wenn man denn Methoden zur Verfügung hat, oder selber schreibt, die in einem eigenen Thread laufen, darf es ruhig auch ein wenig länger dauern (mit den Folgen eines Ereignisse). Jetzt wieder in C# :

Deklarationen / Instanzierung / Ereignisse

Wie gehabt, der Blinker entfällt.

Ereignisroutinen

```
private void Hoch(object sender, int Wert) {
      LampeHigh.BlinkingOn(333);}
private void Niedrig(object sender, int Wert) {
      LampeLow.BlinkingOn(234, 123);}
private void Normal(object sender, int Wert) {
      LampeHigh.BlinkingOff();
      LampeLow.BlinkingOff(); }
private void ShowValue(object sender, int Wert) {
      lblStatus.Text = Wert.ToString();}
```

So schön einfach wie bei DreipunktLampen ganz am Anfang. Der Unterschied : statt LampeHigh.On() steht da jetzt LampeHigh.BlinkingOn(333) .. und LampeLow.BlinkingOff()

Die Methoden BlinkingOn läuft in einem eigenen Thread und stört so die anderen Ereignisse nicht.

DreipunktLampenEvent

Wenn keine geeignete Methode dieser Art zur Verfügung steht oder die vorhandene nicht gefällt, muß man sie selber stricken :

Deklarationen

```
private FishDevices   fd;
private DLamp         LampeHigh;
private DLamp         LampeLow;
private DLamp         Blinker;
private PhotoResistor Photo;

private bool          blinkingOn;
```

Instanzierung, Ereignisse

wie gehabt

Ereignisroutinen

```
private void Hoch(object sender, int Wert) {
      Thread bt = new Thread(new ThreadStart(Blinking));
      bt.IsBackground = true;
      blinkingOn      = true;
      Blinker         = LampeHigh;
      bt.Start();}
private void Niedrig(object sender, int Wert) {
      Thread bt = new Thread(new ThreadStart(Blinking));
      bt.IsBackground = true;
      blinkingOn      = true;
      Blinker         = LampeLow;
      bt.Start();}
private void Normal(object sender, int Wert) {
      blinkingOn = false;}
private void ShowValue(object sender, int Wert) {
      lblStatus.Text = Wert.ToString();}
private void Blinking() {
      bool FlipFlop = true;
      while(blinkingOn) {
            if(FlipFlop) {Blinker.On();   FlipFlop = false;}
            else {Blinker.Off(); FlipFlop = true;}
            Thread.Sleep(333);}
      Blinker.Off();}
```

In den Ereignisroutinen Hoch und Niedrig wird jetzt ein neuer Thread definiert und gestartet dessen Nutzroutine Blinking dann das Blinken übernimmt. Das Blinken wird wie bisher in der Ereignisroutine Normal beendet. Hier durch Setzen der Loopbedingung blinkingOn = false. Der Thread beendet sich dann selber.