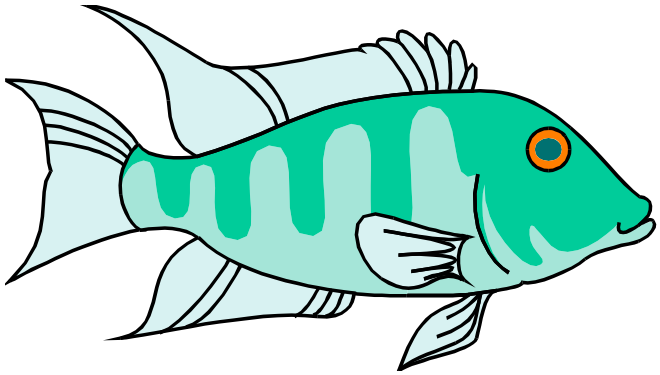
ftComputing

FishFa30 für C#

Ulrich Müller



Inhaltsverzeichnis

Ulrich Müller, D-33100 Paderborn, Lange Wenne 18. Fon 05251/56873, Fax 05251/55709
eMail : UM@ftComputing.de
HomePage : www.ftcomputing.de

Dokumentname : FishFa30CS.doc. Druckdatum : 16.06.2004
Titelbild : Einfügen | Grafik | AusDatei | Office | Fish11.WMF

Einführung

Allgemeines

MIt der in C# geschriebenen Assembly FishFa30.DLL wird die Möglichkeit geboten, die fischertechnik Interfaces unter einer .NET Sprache zu programmieren (beschrieben wird hier der Einsatz von C#). FishFa30.DLL setzt auf umFish30.DLL (Systemkonforme.DLL) auf. Die zentrale Klasse FishFace von FishFa30.DLL erlaubt die Ansteuerung der parallelen (Universal) und der seriellen (Intelligent) Interfaces jeweils wahlweise mit Slave/Extension Module. Es können mehrere Interfaces innerhalb eines Programmes simultan betrieben werden.

Von der Basisklasse FishFace abgeleitet sind die Klassen FishRobot für den Betrieb von Robot-Motoren und FishStep für den Betrieb von Schrittmotoren.

Angeboten werden Befehle zur Schaltung der M-Ausgänge und zur Abfrage der Eingänge eines Interfaces. Dazu wird das Interface in einem besonderen Thread von umFish30.DLL in regelmäßigen Abständen abgefragt (gepollt -> PollInterval). Zusätzlich werden die Veränderungen (Ein/Aus) an den E-Eingängen gezählt, sie werden außerdem zur Bestimmung der Schaltdauer der M-Ausgänge herangezogen. Dazu ist eine feste Zuordnung des an einen M-Ausgang angeschlossenen Motors und der an die E-Eingänge angeschlossenen Ende- und Impulstaster notwendig(RobMotoren). Die M-Ausgänge können außerdem mit verschiedener "Geschwindigkeit" betrieben werden, dazu werden sie in Intervallen ein- und ausgeschaltet (PWM).

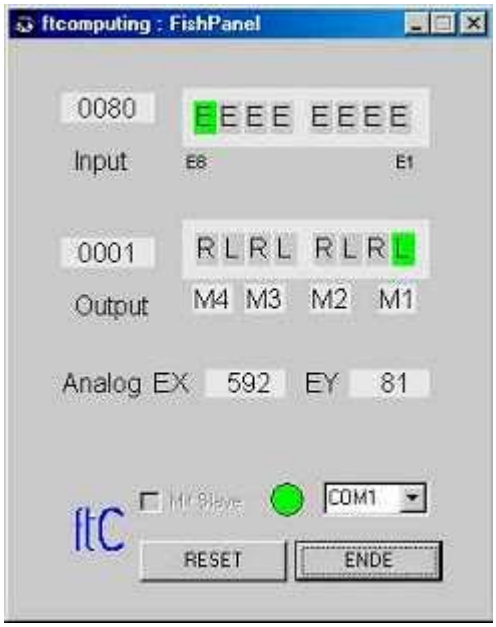
Die A-Eingänge (Analog-Eingänge) liefern Raw-Werte im Bereich von 0 – 1024. Der Betrieb eines Interfaces mit Auslesen der A-Eingänge benötigt ein größeres PollInterval als ohne, das trifft besonders für das parallele Interface zu (Instanzierung mit Parameter AnalogScan = True ist erforderlich).

Getestet wurde mit : VS.NET Final auf Windows 2000 mit .NET Framework v1.0 und VS.NET v1.1 auf Windows XP mit .NET Framework v1.1

und SharpDevelop Fidalgo v1.0 Beta auf Windows XP mit .NET Framework v1.1

Ein praktischer Unterschied beim Einsatz von FishFa30.DLL wurde nicht festgestellt. Jedoch verwenden die IDEs unterschiedliche Projekt-Dateien. Eine Übernahme via Datei | Projekt importieren geht problemlos. In den getesteten Fällen mußte lediglich die Referenz auf FishFa30.DLL nachgetragen werden. Im Falle Riese6CS mußte auch noch das Hintergrundbild nachgetragen werden.

Interface Panel



Das Interface Panel dient zur Anzeige der Werte eines fischertechnik Interfaces und zum Schalten der M-Ausgänge (Output).

Nach Start des Panels kann eingestellt werden, ob mit Slave (Extension Module) gearbeitet werden soll.

Über die ComboBox kann der Interface-Anschluß gewählt werden (PortName). Bei Wahl von LPT erscheint ein weiterer Button LPT-Optionen, mit denen die Einstellungen des zuvor installierten LPT-Treibers modifiziert werden können (das ist etwas mühlselig und erfordert Kenntnisse über die vorhandene Hardware, ist aber meist nicht erforderlich).

Neben der ComboBox wird nach Klick auf START die Betriebsbereitschaft angezeigt.

Die Input-Zeile zeigt den Status aller E-Eingänge an, links als Hexa-Wert.

Die Output-Zeile zeigt den Status der M-Ausgänge an, links wieder als Hexa-Wert. Ein Klick auf L bzw. R schaltet den entsprechenden Ausgang für die Dauer des Klicks ein, wird gleichzeitig die Strg-Taste gedrückt auch dauerhaft. Das Ausschalten erfolgt dann durch Klick auf M1 ... Alle M-Ausgänge können durch Klick auf den RESET-Button gleichzeitig ausgeschaltet werden.

L legt gleichzeitig die Richtung Links (Dir.Links, Dir.Left) fest, also nach dem Modellaufbau testen in welche Richtung es beim L-Klick geht und bei der Programmierung dann berücksichtigen (bei Nichtgefallen : die Motoren umpolen). Analoges gilt für einen R-Klick.

Die Analog-Zeile zeigt die (dezimalen) Werte an die an den Eingängen EX und EY gemessen werden.

Literatur zu C#

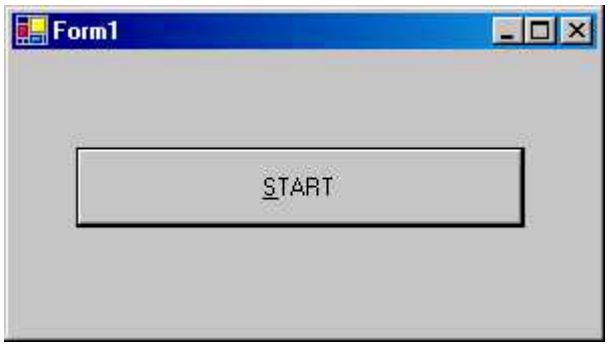
- Eric Gunnarson : C#, Galileo, zweite Auflage, ISBN 3-89842-183-X (deutsch) als fundierte Einführung
- Nitty Gritty C#, Addison-Wesley (deutsch). Handfeste und preiswerte Einführung/Übersicht für Programmierer mit Erfahrung. ISBN 3-8273- 1856-4
- [dpunkt] Mössenböck : Software Entwicklung mit C# - Ein kompakter Lehrgang. dpunkt.verlag, ISBN 3-89864-126-0. Eher Kurzreferenz für erfahrene Programmierer mit Kenntnissen in anderen Sprachen.
- O'Reilly : Programming C# 2. Auflage, ISBN 0-596-00309-9, als Übersicht.
- O'Reilly : C# in a Nutshell, 0-596-00181-9, als Referenz neben der recht ansprechenden Hilfe des Visual Studio.NET

Und dann gibt es jetzt auch von den altbekannten VB-Autoren eine VB.NET Version :

- Frank Eller, Michael Kofler : Visual C# - Grundlagen, Programmiertechniken, Windows – Programmierung - Addison-Wesley ISBN 3-8273-2073-9
- Doberenz / Kowalski : Visual C#.NET - Grundlagen und Profiwissen , Hanser ISBN 3-446-22021-6

In den beiden letztgenannten Büchern wird auch ausführlich auf die Programmierung mit Windows.Forms eingegangen.

Die StartAmpel



So geht's los :

- Interface anschließen, Funktion mit dem Interface Panel testen
- An das Interface anschließen M1 : grüne, M2 gelbe, M3 rote Lampe
- Neues Projekt anlegen (Windows Anwendung mit einer einfachen Form)
- In der Projektübersicht Verweis (Referenz / Verweise) FishFa30.DLL (Assemblies) eintragen.
- Ganz zu Beginn der Source der ebenangelegten Form `using FishFa30` eintragen.
- Einen Button cmdAction mit der Beschriftung START anlegen
- Für cmdAction eine Click-Routine anlegen und den unten angegebenen Text eingeben. Port.COM1 ggf. anpassen.
- START Drücken : Es geht los – und das wars denn auch schon.
- Wenn mans genauer wissen will gleich noch mal im Einzelschritt durchlaufen lassen.

Code der cmdAction – Click Routine :

```
FishFace ft = new FishFace();
ft.OpenInterface(Port.COM1);
ft.SetMotor(Nr.M3, Dir.Ein);
ft.Pause(1000);
ft.SetMotor(Nr.M2, Dir.Ein);
ft.Pause(500);
ft.SetMotor(Nr.M3, Dir.Aus);
ft.SetMotor(Nr.M2, Dir.Aus);
ft.SetMotor(Nr.M1, Dir.Ein);
ft.Pause(2000);
ft.SetMotor(Nr.M1, Dir.Aus);
ft.CloseInterface();
```

Das Programm steht in einer Datei mit der Endung .CS (Form1 / MainForm / eigener Name).

Hinzu kommen noch die Projekt-Dateien.

Sub cmdAction_Click ist die einzige Nutzroutine des Programmes, die die M-Ausgänge schaltet.

Zu den Elementen :

- `FishFace ft = new FishFace();` : anlegen einer neuen Instanz der Klasse FishFace (Teil von FishFa30.DLL) mit dem Name ft. Unter diesem Namen werden dann die Methoden (Funktionen) der Klasse angesprochen.
- `ft.OpenInterface(Port.COM1);` : Herstellen einer Verbindung zum Interface, hier dem Intelligent Interface an COM1 Es werden außerdem eine Reihe von Eigenschaften gesetzt.
- `ft.SetMotor(Nr.M3, Dir.Ein);` : Einschalten der roten Lampe
  Die Methoden von FishFace bieten meist einen Auswahlliste (Enum) möglicher Parameterwerte. Hier aus der Aufzählung Nr und Dir. Es können aber auch einfache Zahlen oder eigene Konstanten angegeben werden.
- `ft.Pause(1000);` : Das Programm wird für 1000 MilliSekunden (1 Sekunde) angehalten.
- `ft.SetMotor(Nr.M2, Dir.Ein);` die gelbe Lampe wird für 500 MilliSekunden zugeschaltet. und dann werden beide aus und die grüne Lampe an M1 wird für 2000 MilliSekunden angeschaltet
- und der Ordnung halber : `ft.CloseInterface();`, die Verbindung zum Interface gekappt.

Das wars denn auch schon für den Anfang. Weiter kann es mit dem Durcharbeiten des Abschnitts Riesenrad. Zu empfehlen, wenn die eigenen Programmierkünste noch nicht besonders ausgeprägt sind. Der Abschnitt Referenz beschreibt Eigenschaften und Methoden von FishFace im Detail. Im Abschnitt Tips & Tricks werden kleine Problemlösungen vorgestellt. Will man noch mehr Beispiele, sollte man sich auf www.ftcomputing.de/csecke.htm umsehen.

Beispiel Riesenrad

Allgemeines

Es wird die schrittweise Entwicklung eines Betriebsprogrammes für das Modell Riesenrad aus dem fischertechnik Kasten "Fun Park" (57 484, Anleitung allein 62 959) mit der Programmiersprache C# unter der Enwicklungsumgebung VS.NET, sowie der Assembly FishFa30.DLL beschrieben.

Der angebene Code für das Betriebsprogramm ist unabhängig von der Entwicklungsumgebung. Ein Import der VS.NET Final C# Programme in SharpDevelop Fidalgo beta 1 verlief problemlos.

Zum Anlegen eines VS.NET bzw. SharpDevelop Projektes siehe Abschnitt. Referenz.

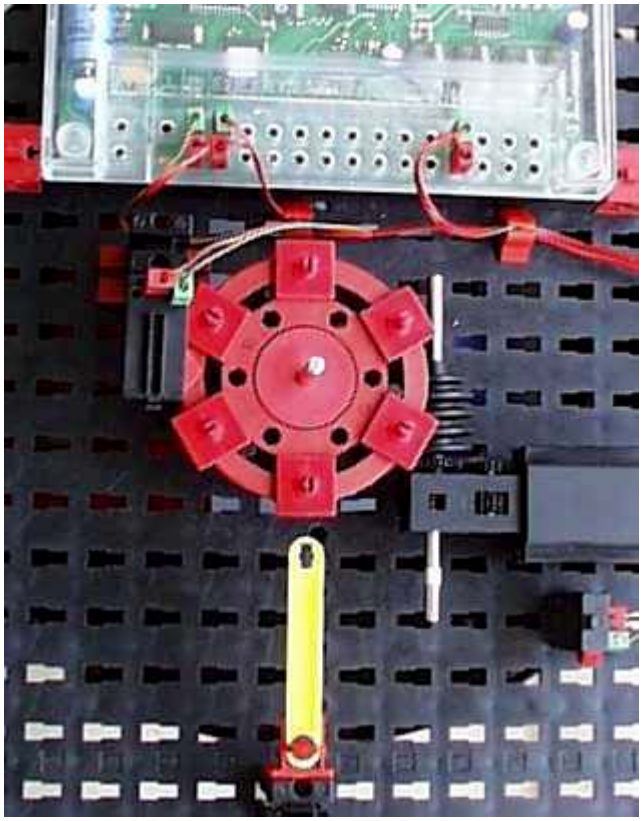
Das Modell

Das mit Motorantrieb ausgerüstete Modell entspricht weitgehend dem Original des Kastens Fun Park. Es wurde auf den Betrieb mit dem Intelligent Interface umgerüstet :

- Antriebsmotor an M1
- Im Fußbereich wurden das Intelligent Interface und der Taster E1 untergebracht.

- Am hinteren Scheibenrad wurden Schaltnocken (siehe Bild rechts) und der Taster E2 montiert.

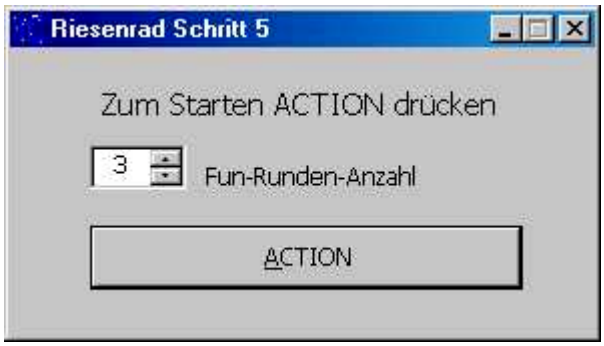
Wenn der Wunsch der Einarbeitung in C# im Vordergrund steht, kann alternativ zum realen Modell kann aber auch ein Simulationsmodell eingesetzt werden. Es ist bei weitem nicht so sperrig :



Es kann die gleichen Steuerfunktionen ausführen wie das Original. Das Zeitverhalten entspricht nicht ganz dem großen Modell. Besonders beim Anfahren und Anhalten macht sich das bemerkbar. Die verwendete Zeitkonstante bewegt sich im Bereich 1500 (Riesenrad mit 12 Männeken / Simulations Modell) bis zu 3200 (leeres Riesenrad). Der gelbe Zeiger markiert die Position der unteren Gondel.

Programmrahmen

Anlegen des Projektes siehe FishFa30 Handbuch Abschnitt Referenz



Das Windows Form Programm ist sehr einfach gehalten, hier der Aufbau von Schritt 5, die vorhergehenden Schritte enthalten teilweise noch weniger Elemente :

- lblStatus : Label Control zur Anzeige des aktuellen Status

- nudRunden bzw. nudFaktor : Spin Control zur Eingabe der Rundenzahl bzw. des Rundenfaktors.
- cmdAction : Button Control zum Start des Programms. Zugehörende Ereignis-Routine : cmdAction_Click (C#) bzw. CmdActionClick bei SharpDevelop.

Das Betriebsprogramm läuft überwiegend in der Action-Ereignis-Routine ab.

Schritt 1 : Aus- und Einsteigen

```
private void cmdAction_Click(object sender, System.EventArgs e)
{
   try  {
      ft.OpenInterface(Port.COM2);
      lblStatus.Text = "--- gestartet ---";
      for(int i = 1; i <= 6; i++) {
         ft.SetMotor(Nr.M1, Dir.Links);
         ft.WaitForHigh(Nr.E2);
         ft.Pause(1500);
         ft.SetMotor(Nr.M1, Dir.Aus);
         ft.Pause(4000); }
      lblStatus.Text = "--- das war's ---";  }
   catch(FishFaceException eft)  {lblStatus.Text = eft.Message;}
   finally {ft.CloseInterface(); }
}
```

Alle Gondeln werden nacheinander zur Einsteigeposition gefahren. Zum Aus- und Einsteigen wird eine feste Zeit gehalten.

Kernpunkt der Positionserkennung ist der Befehl WaitForHigh (Warten auf einen false/true-Durchgang). ein schlichtes Warten auf E2 = true reicht nicht, da der Taster noch auf true stehen kann. Da mit E2 = true die exakte Einsteigeposition nicht gewährleistet ist (Lage der Schaltnocken, Bremsverhalten des Modells), wird nach E2 = true noch ein paar (1,5 Sekunden) weitergefahren und dann erst abgeschaltet. Danach folgt die Pause für das Aus- und Einsteigen. Das ganze immer schön in einer 6er-Schleife.

ACHTUNG : Die Länge der Pause (hier Pause 1500) hängt beim Original Riesenrad von der Beladung und der Befestigung der Hauptachse ab. Sie lag bei mir zwischen 1500 (12 Männeken) und 3200(leer).

Schritt 2 : Die Fun-Runden

```
private void cmdAction_Click(object sender, System.EventArgs e)
{
   try
   {
      ft.OpenInterface(Port.COM2);
      while (!ft.Finish(Nr.E1)) {
         lblStatus.Text = "--- Aus- und Einsteigen ---";
         for(int i = 1; i <= 6; i++) {
            ft.SetMotor(Nr.M1, Dir.Links);
            ft.WaitForHigh(Nr.E2);
            ft.Pause(1500);
            ft.SetMotor(Nr.M1, Dir.Aus);
            lblStatus.Text = "Gondel : " + i.ToString();
            ft.Pause(4000);
         }
         lblStatus.Text = "--- Tour linksrum ---";
         ft.SetMotor(Nr.M1, Dir.Links);
         ft.Pause(1000 * (int)nudFaktor.Value);
         ft.SetMotor(Nr.M1, Dir.Aus);
         ft.Pause(1000);
         lblStatus.Text = "--- Tour rechtsrum ---";
         ft.SetMotor(Nr.M1, Dir.Rechts);
         ft.Pause(1000 * (int)nudFaktor.Value);
```

```
        ft.SetMotor(Nr.M1, Dir.Aus);
        ft.Pause(1000);
      }
      lblStatus.Text = "--- Demo beendet ---";
    }
    catch(FishFaceException eft)
    {
      lblStatus.Text = eft.Message;
    }
    finally
    {
      ft.CloseInterface();
    }
}
```

Das ganze Programm wird in eine while Schleife gepackt, so erhält man ein schönes Demo-Programm, das durch E1 = true oder die ESC-Taste abgebrochen werden kann. Zusätzlich wird in lblStatus die aktuelle Funktion angezeigt.

Nach dem Aus-/Einsteigen (wie bisher) wird 15 Sekunden links und dann 15 rechts gedreht. Da es bei sofortiger Richtungsumschaltung richtig knirschen kann, wird dazwischen 1 Sekunde Pause eingelegt.

Wenn man die Runden gerne länger hätte, kann man im Spin Control einen Rundenfaktor eingeben (Vorgabe 10, Rundenzeit also 10 Sekunden).

Schritt 3a : Echt-Betrieb

```
private void cmdAction_Click(object sender, System.EventArgs e)
{
  try
  {
    ft.OpenInterface(Port.COM2);
    while (!ft.Finish(Nr.E1)) {
      lblStatus.Text = "--- Aus- und Einsteigen ---";
      for(int i = 1; i <= 6; i++) {
        ft.SetMotor(Nr.M1, Dir.Links);
        ft.WaitForHigh(Nr.E2);
        ft.Pause(1500);
        ft.SetMotor(Nr.M1, Dir.Aus);
        lblStatus.Text = "Gondel : " + i.ToString();
        ft.WaitForLow(Nr.E1);
      }
      lblStatus.Text = "Fahrbetrieb : Quittungs-Taster";
      ft.Pause(3000);
      if (!ft.GetInput(Nr.E1)) ft.NotHalt = true;
..... weiter wie gehabt .....
```

Bei einem Echt-Betrieb sind die genauen Zeiten für Aus- und Einsteigen nicht vorhersebar, die Pause(4000) durch eine WaitForLow(Nr.E1) ersetzt. Das Programm wartet bie E1 gedrückt und wieder freigegeben wird.

Nach dem Aus- und Einsteigen muß der Betrieb durch erneutes Drücken der E1-Taste innerhalb von 3 Sekunden freigegeben werden, sonst wird das Programm beendet : Feierabend. Zum Abbruch wurde hier `ft.NotHalt` auf `true` gesetzt. Das läßt alle FishFace-Methoden "durchrauschen". Bei ft.Finish führt das dann zum Beenden der while-Schleife. Eine bequeme Methode für den Abbruch in Notfälle (das Modell läuft "gegen die Wand") oder zum Beenden einer Endlosschleife. Hier hätte es auch ein schlichtes `break` getan.

Man kann sich zum Betrieb natürlich locker noch mehr einfallen lassen.

Schritt 3b : Symbolische Namen

```
// --- Globale Daten ----------------------------------------
private FishFace ft = new FishFace(true, false, 0);
const int mRadMotor = (int)Nr.M1, eRadPos = (int)Nr.E2,
          eQuittung = (int)Nr.E1,
          cLinks = (int)Dir.Links, cRechts = (int)Dir.Rechts,
          cAus = (int)Dir.Aus;
```

Anstelle der allgemeinen Bezeichnungen (enums) für die Ein- und Ausgänge des Interfaces sollte man, wenn's was größeres wird, besser spezielle symbolische Namen setzen. Hier mRadMotor, eRadPos, eQuittung. Da beim Aufruf der Methoden ein EntwederOder gilt, sind auch für die eigentlich passenden enums eigene symbolische Namen erforderlich : cLinks ... (zusammen mit dem TypeCasting kann man sie allerdings weiterverwenden – s.o.). Sie können dann überall verwendet werden, wo jetzt die Enums stehen.

Aussehen tut das dann so (Ausschnitt) :

```
for(int i = 1; i <= 6; i++) {
   ft.SetMotor(mRadMotor, cLinks);
   ft.WaitForHigh(eRadPos);
   ft.Pause(1500);
   ft.SetMotor(mRadMotor, cAus);
   lblStatus.Text = "Gondel : " + i.ToString();
   ft.WaitForLow(eQuittung);
}
```

Schritt 4 : Zählen statt warten

```
lblStatus.Text = "Fahrbetrieb : Quittungs-Taster";
ft.Pause(3000);
if (!ft.GetInput(eQuittung)) ft.NotHalt = true;
   ft.SetMotor(Nr.M1, Dir.Links);
   for(int i = 1; i <= (int)nudRunden.Value; i++) {
      lblStatus.Text = "--- Runde : " + i.ToString() + " linksrum";
      ft.WaitForChange(eRadPos, 12);
   }
   ft.SetMotor(mRadMotor, cAus);
   ft.Pause(1000);
   ft.SetMotor(mRadMotor, cRechts);
   for(int i = 1; i <= (int)nudRunden.Value; i++) {
      lblStatus.Text = "--- Runde : " + i.ToString() + " rechtsrum;
      ft.WaitForChange(eRadPos, 12);
   }
   ft.SetMotor(mRadMotor, cAus);
   ft.Pause(1000);
}
lblStatus.Text = "--- Betrieb beendet ---";
```

Die Pause(1000 * nudFaktor.Value) wurde durch eine Schleifenkonstruktion ersetzt. Die Schleife selber enthält ein WaitForChange(eRadPos, 12), das ist genau eine Runde, da WaitForChange die Flanke d.h. den Übergang von true/false und false/true zählt. Die Schleife wird sooft duchlaufen, wie es der Spin Control nudRunden.Value (ex nudFaktor) angibt. Das bringt zum Einen eine angebbare Runddnzahl, zum Anderen die Möglichkeit, die aktuelle Rundennummer auch auszugeben.

Schritt 5 : Überkreuz Beladen

In der Praxis werden bei einem Riesenrad die Gondeln selten in ihrer direkten Reihenfolge "beladen". Das Beladen übernimmt hier ein gleichnamige Unterprogramm. Das Unterprogramm selber entspricht weitgehend dem bisherigen Belade-Code, lediglich die Ansteuerung der Postion findet jetzt in einer Schleife statt.

```
private void Beladen(int Position, int Runde) {
   ft.SetMotor(mRadMotor, cLinks);
   for(int n = 1; n <= Position; n++) ft.WaitForHigh(eRadPos);
   ft.Pause(1500);
   ft.SetMotor(mRadMotor, cAus);
   lblStatus.Text = "Gondel : " + Runde.ToString();
   ft.WaitForLow(eQuittung);
}
```

Anstelle des Belade-Codes in cmdAction_Click findet man dort eine for-Schleife, die Beladen aufruft. Der erste Parameter gibt die relative Nummer der nchsten Beladeposition an (Anzahl Gondeln, die zu überspringen sind, + 1). Der zweite Prarameter gibt an die wievielte Gondel zu beladen ist, sie wird aus der äußeren for-Schleife abgeleitet.

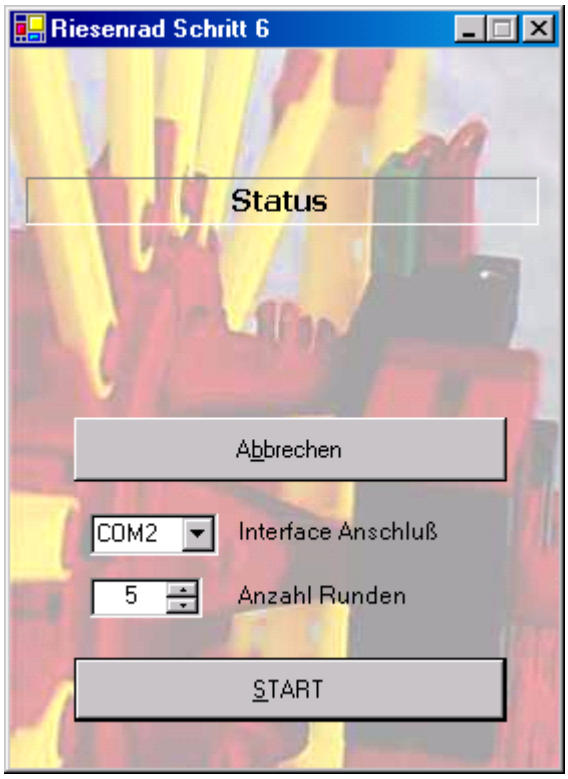
```
try
{
   ft.OpenInterface(Port.COM2);
   while (!ft.Finish(Nr.E1)) {
      lblStatus.Text = "--- Aus- und Einsteigen ---";
      for(int i = 1; i <= 3; i++) {
         Beladen(1, i * 2 -1);
         Beladen(3, i * 2);
      }
  lblStatus.Text = "Fahrbetrieb : Quittungs-Taster";
```

Beladen wird in der Reihenfolge 1 – 4 – 5 – 2 – 3 – 6. Also die gegenüberliegende Gondel und deren Nachbar. Denkbar sind natürlich auch noch andere Beladepläne.

Man könnte auch noch durch Anbau eines zusätzlichen Tasters samt Nocken die Gondel Nr. 1 identifizieren. Dann kann man auch noch die einzelnen Gondeln mit der zugehörenden Nummer beschriften, so kann die man über die Position jede Gondel genau Buchführen.

Schritt 6a : Betriebsicherheit – Verriegeln der Buttons

Nur für diejenigen, denen die bisherigen Programmversionen zu langweilig geworden sind, das Riesenrad läuft auch ohne sie, der Betrieb wird sicherer und komfortabler (6b).



Dazu wird erstmal ein neuer Button cmdEnde mit wechselder Beschrifttung : Abbrechen – HALT – ENDE eingeführt. Das Control cboPortNr kommt erst bei 6b.

Gleich nach erfolgreichem OpenInterface wird dann der START-Button verriegelt (Enabled = false) und der Abbrechen-Button in HALT umgetauft, er bekommt dann auch den Fokus.

Im finally-Block wird der HALT-Button dann in ENDE umgetauft und der START-Button reaktiviert (Enabled = true).

In der cmdEnde_Click Routine wird die Beschriftung des cmdEnde-Buttons auf "&HALT" abgefragt. In diesem Fall läuft das Programm, es wird deswegen nur eine "Abbruchwunsch" (siehe auch Schritt 3a) angemeldet, das Programm läuft weiter, bis es auf FishFace-Methoden stößt, die NotHalt auswerten. Schließlich landet das Programm bei ft.Finish, das führt dann zum Abbruch der while-Schleife und damit zum Betriebs-Ende (nicht Programm-Ende).

Hört sich recht kompliziert an, ist es auch und es geht noch weiter : In der Riese6CD_Closing Routine (Aufruf z.B. bei Klick auf das Kreuz auf der Form rechts oben) wird abgefragt (`if(cmdEnde.Text == "&HALT") e.Cancel=true`) ob der Betrieb beendet wurde, wenn das nicht der Fall ist, wird das Schließen der Form abgelehnt.

Die Verriegelung ist sinnvoll, weil bei einem Schließen der Form die Betriebsschleife munter weiterläuft. Bislang wurde sie (vorher oder nachher) durch Drücken der ESC-Taste beendet. Das geht auch weiterhin.

Schritt 6b : Serialisierung – Persistenz

Wenn man Betriebsdaten des Programms über den Programmablauf hinaus erhalten will, sie also persitent machen will, muß man sie in eine Datei speichern und beim erneuten Programmstart wieder einlesen. Das kann man nach Altväterart mit handgestrickten Dateizugriffen lösen oder man kann sich der Serialisierung bedienen. Hier werden alle aktuellen Daten einer Klasseninstanz mit System gerettet (binär (geht schneller) oder XML(kann mit NotePad geändert werden)). Für das Riesenrad soll die Lage der Form auf dem Bildschirm und der verwendete Portname gespeichert werden. Die Angelegenheit sieht etwas vertrackt aus und ist es auch, wenns dann aber mal läuft, geht es verblüffend einfach.

1. Erstellen einer Klasse mit den zu rettenden Daten (mit der Direktive Serializable) :

```
[Serializable()]
public class PersDaten
{
   public int FormLeft = 0;
   public int FormTop = 0;
   public int PortNr = 2;
}
```

2. Deklarieren des Dateinamens (IniName) in die gespeichert werden soll und einer Variablen (gp) für eine Klasseninstanz von PersDaten (Globale Daten).

```
private string IniName = new System.IO.DirectoryInfo(@".\").FullName
                         + "Riese6CS.DAT";
private PersDaten gp;
```

Gespeichert wird der volle Pfadname von Riese6CS.DAT, das im Verzeichnis der Anwendung liegen soll.

3. Plazieren der Serialisierungsfunktionen in Ries6CS.Closing :

```
private void Riese6CS_Closing(object sender,
                              System.ComponentModel.CancelEventArgs e)
{
  if(cmdEnde.Text == "&HALT") e.Cancel=true;
  else
  {
     System.Runtime.Serialization.Formatters.Binary.BinaryFormatter
     ser = new
  System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
    System.IO.FileStream bin = new System.IO.FileStream(IniName,
                                       System.IO.FileMode.Create);
    gp.FormLeft = this.Left;
    gp.FormTop  = this.Top;
    gp.PortNr   = cboPortName.SelectedIndex;
    ser.Serialize(bin, gp);
    bin.Close();
  }
}
```

Die zugehörenden Methoden haben wahre Bandwurmnamen, man könnte sie durch ein passendes using verkürzen, hier sind nun aber alle "Schuldigen" besammen. Zunächst werden Formatter (ser) und FileStream (bin) erstellt (im Falle, daß das Programm beendet werden kann), dann die zur rettenden Daten in der Klasse PersDaten upgedated (wenn man die PersDaten direkt benutzen kann, geht's auch ohne Update). Dann steigt mit `ser.Serialize(bin, gp);` die Serialisierung der Instanz gp der Klasse PersDaten. Es folgt `bin.Close();`

4. Plazieren der Deserialisierungsfunktionen in Riese6CS_Load :

```
private void Riese6CS_Load(object sender, System.EventArgs e)
{
  if(System.IO.File.Exists(IniName))
  {
    System.Runtime.Serialization.Formatters.Binary.BinaryFormatter
    ser = new
System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
    System.IO.FileStream bin = new
            System.IO.FileStream(IniName, System.IO.FileMode.Open);
    gp = (PersDaten)ser.Deserialize(bin);
    bin.Close();
  }
  else gp = new PersDaten();
  this.Left = gp.FormLeft;
  this.Top  = gp.FormTop;
  cboPortName.SelectedIndex = gp.PortNr;
  }
```

Hier wird auch "das erste Mal" abgehandelt : es existiert keine entsprechende Datei. Dann wird eine Klasseninstanz angelegt und mit Werten gefüllt (das kann entfallen, da dieses Programm nicht darauf zugreift). Wenn die Datei bereits vorhanden ist, werden Formatter (ser) und FileStream (bin) angelegt. Durch die nachfolgende Deserialisierung `gp = (PersDaten)ser.Deserialize(bin);` wird eine Klasseninstanz von PersDaten mit den Daten des vorhergehenden Programmlaufs angelegt. Es folgt noch `bin.Close();`

Referenz

Programmrahmen

Projekt in der VS.NET Version (Konsolen-Anwendung)

Anlegen eines neuen Projektes mit Menü :
Datei | Neu | Projekt | C#-Projekte | KonsolenAnwendung

- Auswahl des Verzeichnisses in das das Projekt gespeichert werden soll
- Festlegen des Projektnamens

Ergebnis :

Eine Projektmappe (*.SLN) mit dem Projekt (*.CSPROJ) und einer *.CS Source. Zusätzlich noch weitere Dateien, die hier nicht weiter interessieren.

- Beim eben angelegenten Projekt (im Projektmappen Explorer) wird ein Verweis auf die Assembly FishFa30.DLL angelegt :

Projekt | Verweise | Verweise hinzufügen | Durchsuchen
FishFa30.DLL auswählen. FishFa30.DLL kann in einem beliebigen (zentralen)Verzeichnis liegen.

Projekt in der SharpDevelop Version

Anlegen eines neuen Projektes mit Menü :

Datei | Neu | Combine

C# | KonsolenAnwendung

- Auswahl des Verzeichnisses in das das Projekt gespeichert werden soll
- Festlegen des Projektnamens

Ergebnis :

Eine Projektmappe (*.CMBX) mit dem Projekt (*.PRJX) und der Klasse Main.CS.

Zusätzlich noch weitere Dateien, die hier nicht weiter interessieren.

- Beim eben angelegten Projekt (im Projektmappen Explorer) wird ein Verweis auf die Assembly FishFa30.DLL angelegt :

Projekte | combine | ..project.. | Referenzen | Referenz hinzufügen

.NET AssemblyBrowser | Browse

FishFa30.DLL auswählen. FishFa30.DLL kann in einem beliebigen (zentralen)Verzeichnis liegen.

Die Source Class1.CS | Main.CS

Nach der vorhandenen using System; Zeile :

```
using cn = System.Console;
using FishFa30;
```

eintragen.

Zu Beginn der class Class1 (Standardname) eine Instanz der Klasse FishFace anlegen.

```
static FishFace ft = new FishFace();
```

Wenn man die Analog-Eingänge EX/EY nutzen will :

```
static FishFace ft = new FishFace(true, false, 0);
```

Die Parameter bedeuten : mit Auswertung der Analog-Eingänge, kein Extension Module, Standard-Werte für das PollInterval.

In die vorhandene static void Main Routine weitere Source-Zeilen einfügen, die gesamte Source sollte dann so aussehen :

```
class Class1
{
  static FishFace ft = new FishFace(true, false, 0);

  [STAThread]
  static void Main(string[] args)
  {
    try
    {
      ft.OpenInterface(Port.COM2);
      cn.WriteLine("--- InputStatus ---");
      .... weiterer Code hier .......
    }
    catch(FishFaceException eft)
    {
      cn.WriteLine(eft.Message);
    }
    finally
    {
      ft.CloseInterface();
      cn.WriteLine("--- Finito ---");
      cn.Read();
    }
  }
}
```

Der try – Block fängt mögliche Fehler aus der Klasse FishFace ab (aber nur diese), sie werden ggf. auf der Console angezeigt.

Mit OpenInterface wird die Verbindung zum Interface hergestellt (hier natürlich, den eigenen Anschluß angeben). ft.CloseInterface() hebt die Verbindung wieder auf.

Auf Basis dieses Programmrahmens kann man nette kleine Testprogramme erstellen, z.B. zum probieren mit den Beispielen der Referenz. Ein Programmrahmen für eine Windows-Anwendung ist bei den Tips & Tricks angegeben. Bei richtigen Anwendungen sollte man sich etwas mehr einfallen lassen. Der Programmrahmen sollte aber eingehalten werden.

Verwendete Variablenbezeichnungen

Die Variablen sind durchweg vom Typ Integer. Parallel dazu gibt es eine Aufrufvariante (overload), die Enums verwendet. Hier werden zur Beschreibung des Wertebereichs einer Variablen die beschreibende Namen angegeben und in Klammern das entsprechende Enum bzw. der Datentyp.

Die Parameter-Angaben erfolgen – soweit nicht extra notiert – By Value

AnalogNr	Nummer eines Analog-Einganges (Nr) EX = 0, EY = 1
Counter	Wert eines ImpulsCounters (int)
Direction	Drehrichtung eines Motors (Dir) Aus = 0, Ein = 1, Links = 1, Rechts = 2
InputNr	Nummer eines E-Einganges (Nr) E1 = 1 ... E16 = 16
InputStatus	Aktueller Status aller E-Eingänge. Jeweils ein bit pro Ausgang, begonnen bei 0 (0 = E1, 1 = E2 .... 15 = E16)
LampNr	Nummer eines "halben"-M-Ausganges (int) Werte 1 - 16
ModeStatus	Status der Betriebsmodi aller M-Ausgänge. Jeweils 4 bit pro Ausgang. Begonnen bei 0-3 für M1, Werte 0000 normal, 0001 RobMode
MotorNr	Nummer eines M-Ausganges (Nr) M1 = 1 ... M8 = 8
MotorStatus	SollStatus aller M-Ausgänge. Jeweils 2 bit pro Ausgang. Begonnen bei 0-1 für M1 (00 = Aus, 01 = Links, 10 = Rechts).
mSek	Zeitangabe in MilliSekunden
NrOfChanges	Anzahl Impulse (int)
OnOff	Ein/Ausschalten eines M-Ausganges (Dir) Off = 0, On = 1
PortNr	Nummer der wählbaren Ports (Port) LPT = 0, COM1 = 1 ... COM8 = 8
Position	Positionsangabe in Impulsen (int)
Speed	Geschwindigkeit mit der ein M-Ausgang betrieben werden soll (Speed) Off = 0, 1 – 15 (Full)
SpeedStatus	Status der Geschwindigkeiten aller M-ausgänge. Jeweils 4 bit pro Ausgang. Begonnen bei 0-3 für M1. Werte 0000 – 1111 (Full).
TermInputNr	Nummer eines E-Einganges mit der die Methode beendet werden soll (Nr) E1 = 1 ... E16 = 16
Value	allgemeiner Integer-Wert

Enums

Verwendung zur Eingaben von Parametern bei den FishFace-Methoden und den darauf aufbauenden Klassen FishRobot und FishStep.

Dir	Angabe der Drehrichtung ...
Nr	Angabe der Ein-/Ausgangsnr
Port	Angabe des zu nutzenden Ports
Speed	Geschwindigkeitsangabe
Wait	Return-Werte von WaitForMotors

Parallel dazu können auch entsprechende numerische (int) Angaben gemacht werden. Dazu siehe "Verwendete Variablenbezeichungen". Zu beachten ist, daß es hier ein Entweder/Oder gibt : in einer Methode können nur entweder Enums oder eigene Konstanten verwendet werden.

Klasse FishFace

Enthalten in der Assembly FishFa30.DLL mit der Source FishFa30.CS.
FishFace ist die Basisklasse der Assembly FishFa30.

Konstruktor

FishFace()
Mit default-Werten für AnalogScan = false, Slave = false,
PollInterval = 0, LPTAnalog = 0, LPTDelay = 0

FishFace(bool AnalogScan, bool Slave, int PollInterval)
PollInterval = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTAnalog = 0, LPTDelay = 0

FishFace(bool AnalogScan, bool Slave, int PollInterval,
int LPTAnalog, int LPTDelay)
PollInterval = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTAnalog = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTDelay = 0 Bestimmung durch OpenInterface, >0 wird übernommen

AnalogScan : mit/ohne Auswerten der Analog-Eingänge EX/EY

Slave : mit/ohne Slave/Extension Module

PollInterval : Zeitinterval in MilliSekunden in dem das Interface abgefragt und upgedated werden soll. Wert = 0 : Der Wert wird in intern beim OpenInterface bestimmt. Der aktuelle Wert kann (nach OpenInterface) mit der Eigenschaft PollInterval abgefragt werden und ggf. bei der nächsten Instanzierung – vorsichtig – verändert werden. ACHTUNG zu kleine Werte führen zum "Aufhängen" des Rechners.

LPTAnalog : Skalierfaktor für Analogwerte.

LPTDelay : Ausgabeverzögerung bei Universal(LPT) Interface (Werte 50 – 1000)

LPTAnalog / LPTDelay werden nur der vollständigkeithalber aufgeführt, da sie nur auf Win9x-Systemen (mit Direktzugriff auf den LPT-Port) zum Einsatz kommen können. Und da gibt es kein .NET Framework.

Eigenschaften

bool	**AnalogScan** (get) Angabe ob auch die Analogeingänge gescannt werden sollen (default = false)
int	**AnalogsEX** (get) Lesen EX-Wert
int	**AnalgosEY** (get) Lesen EY-Wert
int	**Inputs** (get) Lesen der Werte aller E-Eingänge
int	**LPTAnalog**(get) Lesen Analogskalierung
int	**LPTDelay**(get) Lesen Ausgabeverzögerung
bool	**NotHalt** Anmelden eines Abbruchwunsches (Default = false).
int	**Outputs** Lesen/Schreiben der Werte aller M-Ausgänge (MotorStatus)
int	**PollInterval** (get) Interval (in MilliSekunden) in dem der Status des Interfaces abgefragt(gepollt) und aufgefrischt (refresh) wird.
string	**PortName** (get) Aktueller PortName
bool	**Slave** (get) Mit/ohne Extension Module
string	**Version** (get, static) Version der DLL

get : der Wert kann nur ausgelesen, nicht aber verändert werden.

Methoden

ClearCounter

Löschen (0) des angegebenen Counters

ft.**ClearCounter**(InputNr)

Siehe auch : ClearCounters, GetCounter, SetCounter

ClearCounters

Löschen (0) aller Counter

ft.**ClearCounters**()

Siehe auch : ClearCounter, GetCounter, SetCounter

ClearMotors

Abschalten aller M-Ausgänge

ft.**ClearMotors**()

Exception : InterfaceProblem, KeinOpen

Siehe auch : SetMotor, SetMotors, SetLamp Outputs

CloseInterface

Schließen der Verbindung zum Interface

ft.**CloseInterface**()

Siehe auch : OpenInterface

Finish

Feststellen eines Endewunsches (NotHalt, Escape, E-Eingang(optional))

bool = ft.**Finish**(Optional InputNr)

Exception : InterfaceProblem, KeinOpen. DoEvents

Siehe auch : GetInput, GetInputs, Inputs

Beispiel :

```
do {
  cn.WriteLine("läuft");
  ft.Pause(2345);
} while (!ft.Finish(Nr.E1));
```

Die do .. while-Schleife wird solange duchlaufen, bis entweder ft.NotHalt = true, die ESC-Taste gedrückt oder E1 = true wurde. Die Schleife wird mindestens einmal durchlaufen.

Alternativ :

```
while (ft.Finish(Nr.E1) == false) {
   cn.WriteLine("läuft");
  ft.Pause(2345);
}
lblStatus.Text = "--- FINIS ---";
```

Die Schleife wird ggf. übersprungen (!ft.Finish(Nr.E1) ginge hier natürlich auch.

GetAnalog

Feststellen eines Analogwertes(EX / EY).
Es wird der intern vorliegende Wert ausgegeben. Der Parameter AnalogScan muß bei der Instanzierung auf true gesetzt werden.

Value = ft.**GetAnalog**(AnalogNr)

Exception : InterfaceProblem, KeinOpen

Siehe auch : GetAnalogDirect, AnalogsEX, AnalogsEY, AnalogScan, Instanzierung

Beispiel

```
cn.WriteLine(" EX : " + ft.GetAnalog(Nr.EX).ToString());
```

WriteLine gibt den aktuellen Wert von EX aus.

GetAnalogDirect

Direktes Auslesen der Werte von EX / EY. Dazu wird das Pollen vorübergehend abgeschaltet. Sinnvoll nur, wenn die Motoren stehen. Vorteil : Das PollInterval kann auf dem kleineren Wert von AnalogScan = false (bei der Instanzierung) bleiben.

Value = ft.**GetAnalogDirect**(AnalogNr)

Exception : InterfaceProblem, KeinOpen

Siehe auch : GetAnalog, AnalogsEX, AnalogsEY, AnalogScan, Instanzierung

Beispiel

```
cn.WriteLine(" EX : " + ft.GetAnalogDirect(Nr.EX).ToString());
```

WriteLine gibt den aktuellen Wert von EX aus.

GetCounter

Auslesen des Wertes des angegebenen Counters

Value = ft.**GetCounter**(InputNr)

Siehe auch : SetCounter, ClearCounter, ClearCounters

Beispiel

```
cn.WriteLine("Counter für E2 : " + ft.GetCounter(Nr.E2).ToString());
```

Der aktuelle Zählerstand, der dem E-Eingang E2 zugeordnet ist, wird ausgegeben.

GetInput

Auslesen des Wertes des angegebenen E-Einganges

bool = ft.**GetInput**(InputNr)

Exception : InterfaceProblem, KeinOpen. DoEvents

Siehe auch : GetInputs, Inputs, Finish, WaitForInput

Beispiel

```
if (ft.GetInput(Nr.E1)) {
  ...
}
else {
  ...
}
```

Wenn der E-Eingang E1 (Taster, PhotoTransistor, Reedkontakt ...) = true ist, wird der erste Block durchlaufen. Bei `!ft.GetInput(Nr.E1)` wird der else-Zweig durchlaufen. Möglich

ist auch `if (ft.GetInput(Nr.E1) == false)` { ... } oder
`if(!ft.GetInput(Nr.E1)) {...}`

GetInputs

Auslesen der Werte aller E-Eingänge

IntputStatus = ft.**GetInputs**()

Exception : InterfaceProblem, KeinOpen; DoEvents

Siehe auch : GetInputs, Inputs, Finish, WaitForInputs

Beispiel

```
int E13;
  E13 = ft.GetInputs();
  if ((E13 & (0x1 + 0x4)) > 0) cn.WriteLine("TRUE");
```

Der Block wird ausgeführt, wenn die E-Eingänge E1 oder E3 true sind.
Alternativ :

```
  if ((E13 & 0x1) > 0 || (E13 & 0x4) > 0) cn.WriteLine("TRUE");
```

OpenInterface

Setzen von FishFace-Eigenschaften, Herstellen der Verbindung zum Interface

ft.**OpenInterface**(PortNr, Optional DoEvents)
DoEvents : mit/ohne DoEvents (die Mehrzahl der Methoden kann wahlweise intern durch den Befehl DoEvents unterbrochen werden um eine Unterbrechbarkeit der Anwenderoberfläche sicherzustellen. Default : true.

Exception : InterfaceProblem

Siehe auch : CloseInterface

Beispiel

```
try {
  ft.OpenInterface(Port.COM2)
......
}
catch(FishFaceException eft) {
  cn.WriteLine(eft.Message);
}
finally  {
  ft.CloseInterface();
}
```

Herstellen der Verbindung zum Interface an COM2, DoEvents wird ausgeführt. Im Fehlerfall wird der Text `'InterfaceProblem.Open'` ausgegeben

Pause

Anhalten des Programmablauf für mSek MilliSekunden

ft.**Pause**(mSek)

Exception : InterfaceProblem, KeinOpen; DoEvents; Abbrechbar

Siehe auch : WaitForTime

Beispiel

```
ft.SetMotor(Nr.M1, Dir.Links);
ft.Pause(1000);
ft.SetMotor(Nr.M1, Dir.Aus);
```

Der Motor am M-Ausgang M1 wird für eine Sekunde (1000 MilliSekunden) eingeschaltet.

SetCounter

Setzen des Counters für den angegebenen E-Eingang

ft.**SetCounter**(InputNr, Value)

Siehe auch : GetCounter, ClearCounter, ClearCounters

SetLamp

Setzen eines 'halben' M-Ausganges. Anschluß einer lampe oder eines Magneten ... an einen Kontakt eines M-Ausganges und Masse.

ft.**SetLamp**(LampNr, OnOff)

Exception : InterfaceProblem, KeinOpen

Siehe auch : SetMotors, SetMotors, ClearMotors

Beispiel

```
const int lGruen = 1, lGelb = 2, lRot = 3;

  ft.SetLamp(lGruen, Dir.Ein);
  ft.Pause(2000);
  ft.SetLamp(lGruen, Dir.Aus);
  ft.SetLamp(lGelb, Dir.Ein);
```

Die grüne Lampe an M1-vorn und Masse wird für 2 Sekunden eingeschaltet und anschließend die gelbe an M1-hinten.

SetMotor

Setzen eines M-Ausganges (Motor). Die Motordrehzahl kann gewählt werden (Default = Full), ebenso die Fahrstrecke in Anzahl Impulsen. Siehe auch "Anmerkungen zu den Rob-Funktionen.

ft.**SetMotor**(MotorNr, Direction, Optional Speed, Counter)

Exception : InterfaceProblem, KeinOpen; DoEvents; Counter (bei Parameter Counter)

Siehe auch : SetMotors, ClearMotors, SetLamp, Outputs.

Beipiel 1

```
ft.SetMotor(Nr.M1, Dir.Rechts, Speed.Full);
ft.Pause(1000);
ft.SetMotor(Nr.M1, Dir.Links, Speed.Half);
ft.Pause(1000);
ft.SetMotor(Nr.M1, Dir.Aus);
```

Der Motor am M-Ausgang M1 wird für 1000 Millisekunden rechtsdrehend, volle Geschwindigkeit eingeschaltet und anschließend für 1000 MilliSekunden linksdrehend, halbe Geschwindigkeit.

Beispiel 2

```
ft.SetMotor(Nr.M1, Dir.Links, 12, 123);
```

Der Motor am M-Ausgang M1 wird für 123 Impulse am E-Eingang E2 oder E1 = true mit Geschwindigkeitsstufe 12 eingeschaltet. Das Abschalten erfolgt selbsttätig, das Programm läuft solange weiter. Siehe Auch Beispiel WaitForMotors.

SetMotors

Setzen des Status aller M-Ausgänge, optional mit Geschwindigkeitsangabe (SpeedStatus) und des Betriebsmodes (ModeStatus, default = 0). Bei Betriebsmodus RobMode sind vor dem Aufruf der Methode die entsprechenden Counter zu setzen (SetCounter[m]) Siehe auch "Anmerkungen zu den Rob-Funktionen"

ft.**SetMotors**(MotorStatus, Optional SpeedStatus, ModeStatus)

Exception : InterfaceProblem, KeinOpen; DoEvents, Counter(bei Parameter Counter)

Siehe auch : ClearMotors, SetMotors, SetLamp, Outputs

Beispiel

```
ft.SetMotors(0x1 + 0x80);
ft.Pause(1000);
ft.ClearMotors();
```

Der M-Ausgang (Motor) M1 wird auf links geschaltet und gleichzeitig M4 auf rechts. Alle anderen Ausgänge werden ausgeschaltet. Nach 1 Sekunde werden alle Ausgänge abgeschaltet.

WaitForChange

Warten auf NrOfChanges Impulse an InputNr oder TermInputNr = True

Intern wird der zu InputNr gehörende Counter genommen, der dazu zu Beginn zurückgesetzt wird.

ft.**WaitForChange**(InputNr, NrOfChanges, Optional TermInputNr)

Exception : InterfaceProblem, KeinOpen; DoEvent; Abbrechbar.

Siehe auch : WaitForPositionDown, WaitForPositionUp, WaitForInput, WaitForLow, WaitForHigh.

Beispiel

```
ft.SetMotor(Nr.M1, Dir.Links);
ft.WaitForChange(Nr.E2, 123, Nr.E1);
ft.SetMotor(Nr.M1, Dir.Aus);
```

Der M-Ausgang (Motor) M1 wird linksdrehend geschaltet, es wird auf 123 Impulse an E-Eingang E2 oder E1 = true gewartet, der Motor wird abgeschaltet. Solange wird der Programmablauf angehalten. Siehe auch Beispiel bei SetMotors : dort läuft das Programm weiter.

WaitForHigh

Warten auf einen false/true-Durchgang an einem E-Eingang

ft.**WaitForHigh**(InputNr)

Exception : InterfaceProblem, KeinOpen; DoEvent, Abbrechbar

Siehe auch : WaitForLow, WaitForChange, WaitForInput.

Beispiel

```
ft.SetMotor(Nr.M1, Dir.Ein);
ft.SetMotor(Nr.M2, Dir.Links);
ft.WaitForHigh(Nr.E1);
ft.SetMotor(Nr.M2, Dir.Aus);
```

Eine Lichtschranke mit Lampe an M-Ausgang M1 und Phototransistor an E-Eingang E1 wird eingeschaltet. Ein Förderband mit Motor an M2 wird gestartet, es wird gewartet bis ein Teil auf dem Förderband aus der Lichtschranke ausgefahren ist (die Lichtschranke wird geschlossen), dann wird abgeschaltet. Die Lichtschranke muß vorher false sein (unterbrochen).

WaitForInput

Warten, daß der angegebene E-Eingang den vorgegebenen Wert annimmt. (Default = true)

ft.**WaitForInput**(InputNr, Optional OnOff)

Exception : InterfaceProblem, KeinOpen; DoEvent; Abbrechbar

Siehe auch : WaitForChange, WaitForLow, WaitForHigh.

Beispiel

```
ft.SetMotor(Nr.M1, Dir.Links);
ft.WaitForInput(Nr.E1);
ft.SetMotor(Nr.M1, Dir.Aus);
```

Der Motor an M-Ausgang M1 wird gestartet, es wird auf E-Eingang = true gewartet, dann wird der Motor wieder abgeschaltet : Anfahren einer EndPosition.

WaitForLow

Warten auf einen true/false-Durchgang an einem E-Eingang

ft.**WaitForLow**(InputNr)

Exception : InterfaceProblem, KeinOpen; DoEvent, Abbrechbar

Siehe auch : WaitForChange, WaitForInput, WaitForHigh.

Beispiel

```
ft.SetMotor(Nr.M1, Dir.Ein);
ft.SetMotor(Nr.M2, Dir.Links);
ft.WaitForLow(Nr.E1);
ft.SetMotor(Nr.M2, Dir.Aus);
```

Eine Lichtschranke mit Lampe an M-Ausgang M1 und Phototransistor an E-Eingang E1 wird eingeschaltet. Ein Förderband mit Motor an M2 wird gestartet, es wird gewartet bis ein Teil auf dem Förderband in die Lichtschranke einfährt (sie unterbricht), dann wird abgeschaltet. Die Lichtschranke muß vorher true sein (nicht unterbrochen).

WaitForMotors

Warten auf ein MotorReadyEreignis oder den Ablauf von Time

WaitWert = ft.**WaitForMotors**(Time, MotorNr, ...)

Time (int) : Zeit in MilliSekunden. Bei Time = 0 wird endlos gewartet.

MotorNr(Nr) : Liste von M-Ausgängen in beliebiger Reihenfolge auf die gewartet werden soll. Gewartet wird auf MotorStatus = Aus für die betreffenden M-Ausgänge gewartet.

WaitWert(Wait) : Grund warum die Methode beendet wurde
Wait.Ende : Alle betroffenen M-Ausgänge = Dir.Aus
Wait.Time : Die vorgegebene Wartezeit ist abgelaufen
Wait.NotHalt : Die Eigenschaft NotHalt = True, alle betroffenen Motoren wurden angehalten
Wait.ESC : Die ESC-Taste wurde betätigt, alle betroffenen Motoren wurden angehalten.

Exception : InterfaceProblem, KeinOpen; DoEvents; Abbrechbar.

Siehe auch : SetMotor

Beispiel

```
ft.SetMotor(Nr.M4, Dir.Links, Speed.Half, 50);
ft.SetMotor(Nr.M3, Dir.Rechts, Speed.Full, 40);
do {
  cn.WriteLine(ft.GetCounter(Nr.E6).ToString() + " - " +
               ft.GetCounter(Nr.E8).ToString());
} while (ft.WaitForMotors(300, 4, 3) == Wait.Time);
cn.WriteLine(ft.GetCounter(Nr.E6).ToString() + " - " +
             ft.GetCounter(Nr.E8).ToString());
```

Der Motor am M-Ausgang M4 wird linksdrehend mit halber Geschwindigkeit für 50 Impulse gestartet, der an M3 rechtsdrehen mit voller Geschwindigkeit für 40 Impulse. Die do .. while-Schleife wartet auf das Ende der Motoren (ft.WaitForMotors). Alle 300 MilliSekunden wird in der Schleife die aktuelle Position angezeigt (300 ... = Wait.Time). Wenn die Position erreicht ist (<> Time), ist der Auftrag abgeschlossen, die Motoren haben sich selber beendet. Achtung hier wurde nicht auf NotHalt, oder ESC abgefragt, es könnte also auch vor Erreichen der Zielposition abgebrochen worden sein. In der Schleife wird auf Label lblPos die aktuelle Position angezeigt. Zusätzlich nach Ende der Schleife (Differenz zu 300 MSek).

WaitForPositionDown

Warten auf Errreichen einer vorgegebenen Position durch Abwärtszählen von der aktuellen

ft.**WaitForPositionDown**(InputNr, ByRef Counter, Position, Optional TermInputNr)

Ausgegangen wird von der aktuellen Position, die in Counter gespeichert ist, es werden solange Impulse von Counter abgezogen, bis der in Position angegebenen Stand erreicht ist. Counter enthält zusätzlich die dann tatsächlich erreichte postion (kann um einen Wert höher liegen, wenn der Motor nochmal "geruckt" hat). Alternativ wird die Methode durch E-Eingang TermInputNr = True beendet. Counter und Position mpssen immer postive Werte (einschl. 0) enthalten.

Exception : InterfaceProblem, KeinOpen; DoEvents; Abbrechbar

Siehe auch : WaitForPositionUp, WaitForChange

Beispiel

```
int Zaehler = 12;
  ft.SetMotor(Nr.M1, Dir.Links);
  ft.WaitForPositionDown(Nr.E2, ref Zaehler, 0, Nr.E1);
  ft.SetMotor(Nr.M1, Dir.Aus);
  cn.WriteLine("Zählerstand : " + Zaehler.ToString());
```

Die aktuelle Position ist 12 (Zaehler), der Motor an M1 wird linksdrehend gestartet. WaitForPositionDown wartet dann auf Erreichen der Postion 0, der Motor wird dann ausgeschaltet. Wenn vorher E1 = true wird, wird ebenfalls abgeschaltet.

WaitForPositionUp

Warten auf Erreichen einer vorgegebenen Position durch Aufwärtszählen von der aktuellen.

ft.**WaitForPositionUp**(InputNr, ByRef Counter, Position, Optional TermInputNr)

Ausgegangen wird von der aktuellen Postion in Counter, es werden solange Impulse auf Counter aufaddiert, bis der in Position angegebene Stand erreicht ist. Counter enthält zusätzlich die dann tatsächlich erreichte Position (kann einen Wert höher liegen, wenn der Motor nochmal "geruckt" hat). Alternativ wird die Methode durch E-Eingang TermInputNr = True beendet. Counter und Position müssen immer postive Wrte (einschl. 0) enthalten.

Exception : InterfaceProblem, KeinOpen; DoEvents; Abbrechbar

Siehe auch : WaitForPositionDown, WaitForChange

Beipiel

```
int Zaehler = 0;
  ft.SetMotor(Nr.M1, Dir.Rechts);
  ft.WaitForPositionUp(Nr.E2, Zaehler, 24);
  ft.SetMotor(Nr.M1, Dir.Aus);
  cn.WriteLine("Zähler : " + Zaehler.ToString());
```

Die aktuelle Position ist 0 (Zaehler), der Motor an M1 wird rechtdrehend gestartet. WaitForPositionUp wartet dann auf Erreichen der Position 24, der Motor wird dann ausgeschaltet. Siehe auch Beispiel zu WaitForPositionDown, hier wird die Gegenrichtung gefahren.

WaitForTime

Anhalten des Programmablaufes für mSek MilliSekunden.

ft.**WaitForTime**(mSek)

Synonym für Pause

Exception : InterfaceProblem, KeinOpen; DoEvents; Abbrechbar.

Siehe auch : Pause

Beispiel

```
do {
  ft.SetMotors(0x1);
  ft.WaitForTime(555);
  ft.SetMotors(0x4);
  ft.WaitForTime(555);
} while (!ft.Finish());
```

In der Schleife do .. while wird erst M-Ausgang (Lampe) M1 eingeschaltet und alle anderen abgeschaltet (binär : 0001), dann gewartet, M2 (Lampe) eingeschaltet (Rest aus, binär 0100) und gewartet. Ergebnis ein Wechselblinker. Ende der Schleife durch ESC-Taste.

Anmerkungen

Die Methoden erwarten ein vorhergehendes OpenInterface. Ggf. wird eine entsprechende Exception ausgelöst. Sie enthalten meist ein **DoEvents** um das Programm unterbrechbar zu machen. Wird im Ablauf ein InterfaceProblem festgestellt, wird eine entsprechende **Exception** ausgelöst. Die Wait-Methoden setzen bei Bedarf den zugehörenden **Counter** zurück.

Die SetMotor(s)-Methoden sind **asynchron** d.h. der oder die angesprochenen Motoren (Lampen) werden mit der Methode gestartet. Sie laufen dann unabhängig vom Programm weiter. Sie werden durch ein weiteres SetMotors mit Direction = 0 (Aus) beendet. Ausnahme : SetMotor mit Count-Parameter. Diese Methode beendet sich nach Erreichen der vorgegebenen Position selber.

Die Wait-Methoden koordinieren – meist in Verbindung mit End- bzw. ImpulsTastern den asynchronen Motorlauf mit dem Ablauf des Programms.Sie halten den weiteren Progammablauf an, bis das Waitziel (Ablauf Zeit, erreichte Position, Tasterstellung ...) erreicht ist d.h. sie synchronisieren den Programmablauf wieder.

Die längerlaufenden Methoden sind abbrechbar. Das geschieht manuell durch Drücken der ESC-Taste oder im Programm durch Setzen der Eigenschaft NotHalt = True (z.B. über einen Button).

Bei der Beschreibung der Methoden wird das unter dem Stichwort Exception angegeben.

Klasse FishRobot

Die Klasse FishRobot ist von FishFace abgeleitet und bietet zusätzlich zu den FishFace Eigenschaften und Methoden eine Reihe von Methoden, die speziell für den Betrieb von Robots des Typs "Industry Robot" geeignet sind. Charakteristika : Der Antriebsmotor treibt auf einer geeigneten Welle noch ein zusätzliches Impulsrad an, das einen Taster betätigt. Die Schaltvorgänge (Einschalten, Ausschalten separat) werden ab Robot Null gezählt. Der Robot Null wird durch einen weiteren Taster markiert, der bei einer Linksdrehung (Dir.Links) vom Robot angefahren wird. Die Taster sind dem M-Ausgang fest zugeordnet (M1 : E1 Endtaster, E2 Impulszähler). Siehe auch Abschnitt Anmerkungen | Rob-Funktionen.

Test-Programmrahmen

Erstellung des Projekts wie bei FishFace als Console Projekt. Die Source sieht in Anlehnung an FishFace wie folgt aus :

```
using System;
using cn = System.Console;
using FishFa30;

namespace ManRefRCons {
  class ManRef {
    static FishRobot ft =
                new FishRobot(new int[,]{{3,222},{4,88}});
    [STAThread]
    static void Main(string[] args) {
      ft.PositionChange +=
                  new FishRobot.CommonDelegate(PositionAusgabe);
      try {
        ft.OpenInterface(Port.COM2);  // --- ggf. anpassen
        cn.WriteLine("--- ManRef gestartet ---");
        // --- Hier Code einfügen ----
      }
      catch(FishFaceException eft) {
        cn.WriteLine(eft.Message);
      }
      finally {
        ft.CloseInterface();
        cn.WriteLine("---Finito---");
        cn.Read();
      }
    }
    static void PositionAusgabe(object sender, int[] actPos) {
      cn.WriteLine(actPos[0].ToString() + " - " +
                                          actPos[1].ToString());
    }
  }
}
```

Verändert hat sich die Instanzierung, jetzt mit FishRobot und der Liste der beteiligten Motoren an M3 und M4 mit einer Fahrstrecke von max 222 bzw. 88 Impulsen.

Hinzugekommen ist die Deklaration für die Ereignisroutine PositionAusgabe und die Ereignisroutine selber. Zu beachten : Die Ereignis Routine ist static um vom static Main bequem zugreifen zu können (ohne eine Instanzierung der umgebenden Klasse ManRef).

Die Ereignisroutine ist nicht zwingend.

Konstruktor

FishRobot(int[,] MotList)

MotList : int[,] Liste der für den Robot-Betrieb eingesetzten Motoren. Jeweils Nummer des M-Ausganges und max. Fahrweg ab Endtaster in Impulsen.
Mit default-Werten für AnalogScan = false, Slave = false,
PollInterval = 0, LPTAnalog = 0, LPTDelay = 0

```
private FishRobot ft = new FishRobot(new int[,]{{1,123},{4,456}});
```

oder :

```
private int[,] MotList = new int[,] {{1,123),{4,456}});
private FishRobot ft = new FishRobot(MotList);
```

In beiden Fällen werden die Motoren an M1 (mit E1 Endtaster, E2 Impulstaster) und M4(mit E7 Endtaster, E8 Impulstaster) in den Robbetrieb einbezogen. Der Fahrweg beträgt 123 bzw. 456 Impulse ab Endtaster. Zu beachten ist, daß die anzufahrenden Positionen bei den Methoden MoveTo/MoveDelta in dieser Reihenfolge anzugeben sind.

FishRobot(int[,] MotList, bool AnalogScan, bool Slave, int PollInterval)

MotList : wie oben
PollInterval = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTAnalog = 0, LPTDelay = 0

Fishrobot(int[,] MotList, bool AnalogScan, bool Slave, int PollInterval,
int LPTAnalog, int LPTDelay)

MotList : wie oben
PollInterval = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTAnalog = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTDelay = 0 Bestimmung durch OpenInterface, >0 wird übernommen

AnalogScan : mit/ohne Auswerten der Analog-Eingänge EX/EY

Slave : mit/ohne Slave/Extension Module

PollInterval : Zeitinterval in MilliSekunden in dem das Interface abgefragt und upgedated werden soll. Wert = 0 : Der Wert wird in intern beim OpenInterface bestimmt. Der aktuelle Wert kann (nach OpenInterface) mit der Eigenschaft PollInterval abgefragt werden und ggf. bei der nächsten Instanzierung – vorsichtig – verändert werden. ACHTUNG zu kleine Werte führen zum "Aufhängen" des Rechners.

LPTAnalog : Skalierfaktor für Analogwerte.

LPTDelay : Ausgabeverzögerung bei Universal(LPT) Interface (Werte 50 – 1000)

LPTAnalog / LPTDelay werden nur der vollständigkeithalber aufgeführt, da sie nur auf Win9x-Systemen (mit Direktzugriff auf den LPT-Port) zum Einsatz kommen können. Und da gibt es kein .NET Framework.

Eigenschaften

MotCntl

Liste mit den Daten der bei der Instanzierung übergebenen Motordaten

ft.**MotCntl**[n].Nr Nummer des zugehörenden M-Ausganges

ft.**MotCntl**[n].maxPos Maximaler Fahrweg in Impulsen

ft.**MotCntl**[n].actPos Aktuelle Position ab Home (0) in Impulsen

Alle Werte vom Typ int. Zusammengefaßt in der struct MotWerte.

n bezieht sich auf die Motor-Position in der MotList der Instanzierung.

Methoden

MoveDelta

Simultanes Anfahren einer vorgegebenen Position relativ zur aktuellen.

ft.**MoveDelta**(params int[] DeltaList)

int DeltaList : Liste der anzufahrenden Positionen gezählt ab der aktuellen Positionen, bei negativen Werten wird die Fahrrichtung umgekehrt. Die Werte können als Aufzählung einzelner Werte oder alternativ als Array angegeben werden. Während der jeweils letzten 6 Impulse wird gebremst.

Exception : InterfaceProblem; DoEvent, Abbrechbar
Ereignis : PositionChange.

Siehe auch : MoveTo

Beispiel

```
FishRobot ft = new FishRobot(new int[,]{{3,234}, {4,123}});
.....
ft.MoveDelta(34, -12);
```

Der Motor an M3 fährt 34 Impulse nach rechts, der Motor an M4 fährt 12 Impulse nach links.

MoveHome

Simultanes Anfahren der Home Position

ft.**MoveHome**()

Betroffen sind die Motoren, die bei der Instanzierung angegeben wurden. Es wird nach links (Dir.Links) gefahren, bis der zugehörende Endtaster erreicht wird. Die erreichte Position wird auf 0 gesetzt.

Exception : InterfaceProblem; DoEvent, Abbrechbar.

Beispiel : siehe MoveTo

MoveTo

Simultanes Anfahren einer vorgegebenen Position bezogen auf die Home Position.

ft.**MoveTo**(params int[] PosList)

int PosList : Liste der anzufahrenden Positionen gezählt ab Home Position (zug. Endtaster bei Dir.Links). Wertebereich 0 – max. (Angabe bei der Instanzierung). Die Werte können als Aufzählung einzelner Werte oder alternativ als Array angegeben werden. Während der jeweils letzten 6 Impulse wird gebremst.

Exception : InterfaceProblem; DoEvent, Abbrechbar
Ereignis : PositionChange.

Siehe auch : MoveDelta

Beispiel

```
FishRobot ft = new FishRobot(new int[,]{{3,234}, {4,123}});
int [] PosList = new int[] {100, 200);
.....
ft.MoveHome();
ft.MoveTo(PosList);
```

Der gesamte Robot fährt mit den Motoren an M3 und M4 zunächst die Home Position an und dann die durch MoveTo vorgegebene. Der Motor an M3 fährt auf Position 100, der an M4 auf Position 123, da er bei Erreichen der max. zul. Position gestoppt wird.

Alternative Schreibweise :

```
FishRobot ft = new FishRobot(new int[,]{{3,234}, {4,123}});
.....
ft.MoveTo(100, 200);
```

Schreibweise 1 ist sinnvoll, wenn die Werte bestehenden Tabellen entnommen werden können (Abarbeiten einer Instruktionsliste). Schreibweise 2, wenn Einzelwerte vorliegen (z.B. beim TeachIn).

Ereignisse

PositionChange

Aufruf bei einer Veränderung der Position eines Motors durch die Methoden MoveTo / MoveDelta

ft.**PositionChange** += new **FishRobot.CommonDelegate**(name_event_ routine);

void name_event_routine(object sender, int[] actPos)

{ .... }

actPos : Liste mit den aktuellen Positionen ab Home für die Motoren nach der MotList der Instanzierung.

Beispiel

```
FishRobot ft = new FishRobot(new int[,]{{3,222},{4,88}});
.....
ft.PositionChange += new FishRobot.CommonDelegate(PositionsAusgabe);
.....
static void PositionsAusgabe(object sender, int[] actPos) {
  cn.WriteLine(actPos[0].ToString() + " - " + actPos[1].ToString());
}
```

ft.PositionChange : Für das Ereignis PositionChange wird in die Liste des zuständigen Delegate CommonDelegate die Ereignisroutine PositionsAusgabe eingetragen.

static void ... : Die Ereignisroutine. sender enthält, wie gewohnt, einen Hinweis auf das rufende Objekt. actPos eine Liste mit den aktuellen Positionen der laut Instanzierung übergebenen MotList. Die Positionen zählen in Impulsen ab Home positiv.

Klasse FishStep

Die Klasse FishStep ist von FishFace abgeleitet und bietet zusätzlich zu den FishFace Eigenschaften und Methoden eine Reihe von Methoden, die speziell für den Betrieb von Schrittmotoren geeignet sind. Dabei wird zwischen dem Betrieb einzelner Schrittmotoren (Anschlußbelegung : zwei aufeinanderfolgende M-Ausgänge, Methoden Step...) und dem Simultanbetrieb zweier zusammenhängender Schrittmotoren im XY-Verbund (Anschlußbelegung : drei aufeinanderfolgende M-Ausgänge für die beiden Motoren, Methoden Plot) unterschieden. Die Positionierung erfolgt in Zyklen (in der Regel vier Schaltvorgänge von 7,5°). Zu den M-Ausgängen gehören fest vorgegeben E-Eingänge zur Feststellung der Home Position. Dazu siehe auch "Anmerkungen zu den Step-Funktionen" am Ende des Dokumentes.

Test-Programmrahmen

Erstellung des Projekt wie bei FishFace als Console Projekt. Die Source sieht (in Anlehnung an FishFace) wie folgt aus :

```
using System;
using cn = System.Console;
using FishFa30;

namespace ManRefRCons {
  class ManRef {
    static FishStep ft = new FishStep(new int[,]{{1,456},{3,456}});

    [STAThread]
    static void Main(string[] args) {
      ft.StepChange += new FishStep.StepDelegate(StepPosition);
      ft.PlotChange += new FishStep.PlotDelegate(PlotPosition);

      try {
        ft.OpenInterface(Port.COM2);  // --- ggf. anpassen
        cn.WriteLine("--- ManRef gestartet ---");
        // --- Hier Code einfügen ----
      }
      catch(FishFaceException eft) {
        cn.WriteLine(eft.Message);
      }
      finally {
        ft.CloseInterface();
        cn.WriteLine("---Finito---");
        cn.Read();
      }
    }
    static void StepPosition(object sender, int MotNr, int actPos){
      cn.WriteLine("Aktuelle Step-Position : " +
                                          actPos.ToString());
    }
    static void PlotPosition(object sender, int MotNr,
                                           int xPos, int yPos) {
      cn.WriteLine("Akt. Plot-Position : " + xPos.ToString() +
                                        " / " + yPos.ToString());
    }
  }
}
```

Verändert hat sich die Instanzierung, jetzt mit FishStep und der Liste der beteiligten Motoren an M1/M2 (Endtaster E1) und M3/M4 (Endtaster E5) bei Nutzung durch Step-Methoden bzw.

M1/M2 (Endtaster E1) und M3/M1 (Endtaster E5) bei Nutzung durch Plot-Methoden. Jeweils mit einer Fahrstrecke von 456 Zyklen.

Hinzugekommen sind Deklarationen für die Ereignisroutinen StepPosition und PlotPosition und die Ereignisroutinen selber. Zu beachten : Die Ereignisroutinen wurden static deklariert um einen bequemen Zugriff von static Main zu bieten (ohne eine Instanzierung der umgebenden Klasse ManRef).

Der gezeigte Testrahmen läßt nur den alternativen Betrieb mit Step- bzw. Plot-Methoden zu. Dementsprechend können die jeweils nicht benötigten Ereignisse entfallen. Bei Einsatz eines Extension-Modules (Slave) können Step- und Plot-Methoden aber gemeinsam genutzt werden, wenn sie bei der Instanzierung entsprechend auf die Interfaces verteilt werden.

Die Ereignisroutinen können auch ganz entfallen.

Konstruktor

FishStep(int[,] MotList)

MotList : int[,] Liste der für den Step-Betrieb eingesetzten Motoren. Jeweils Nummer des M-Ausganges und max. Fahrweg ab Endtaster in Zyklen.
Mit default-Werten für AnalogScan = false, Slave = false,
PollInterval = 0, LPTAnalog = 0, LPTDelay = 0

```
private FishStep ft = new FishStep(new int[,]{{1,123},{3,456}});
```

oder :

```
private int[,] MotList = new int[,] {{1,123),{3,456}});
private FishStep ft = new FishStep(MotList);
```

In beiden Fällen werden die Motoren an M1/M2 (Endtaster E1) und M3/M4(Endtaster E5) in den Betrieb mit Step-Methoden einbezogen. Der Fahrweg beträgt 123 bzw. 456 Zyklen ab Endtaster. Beim Betrieb mit Plot-Methoden im XY-Verbund werden die Motoren an M1/M2 (Endtaster E1) und M3/M1(Endtaster E5) in den Betrieb einbezogen. Eine fließende Verteilung auf Interface und Extension Module ist möglich.

FishStep(int[,] MotList, bool AnalogScan, bool Slave, int PollInterval)

MotList : wie oben
PollInterval = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTAnalog = 0, LPTDelay = 0

FishStep(int[,] MotList, bool AnalogScan, bool Slave, int PollInterval,
int LPTAnalog, int LPTDelay)

MotList : wie oben
PollInterval = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTAnalog = 0 Bestimmung durch OpenInterface, >0 wird übernommen
LPTDelay = 0 Bestimmung durch OpenInterface, >0 wird übernommen

AnalogScan : mit/ohne Auswerten der Analog-Eingänge EX/EY

Slave : mit/ohne Slave/Extension Module

PollInterval : Zeitinterval in MilliSekunden in dem das Interface abgefragt und upgedated werden soll. Wert = 0 : Der Wert wird in intern beim OpenInterface bestimmt. Der aktuelle Wert kann (nach OpenInterface) mit der Eigenschaft PollInterval abgefragt werden und ggf. bei der nächsten Instanzierung – vorsichtig – verändert werden. ACHTUNG zu kleine Werte führen zum "Aufhängen" des Rechners.

LPTAnalog : Skalierfaktor für Analogwerte.

LPTDelay : Ausgabeverzögerung bei Universal(LPT) Interface (Werte 50 – 1000)

LPTAnalog / LPTDelay werden nur der vollständigkeithalber aufgeführt, da sie nur auf Win9x-Systemen (mit Direktzugriff auf den LPT-Port) zum Einsatz kommen können. Und da gibt es kein .NET Framework.

Eigenschaften

MotCntl

Liste mit den Daten der bei der Instanzierung übergebenen Motordaten

ft.**MotCntl**[n].maxPos	int Maximaler Fahrweg in Zyklen
ft.**MotCntl**[n].actPos	int Aktuelle Position ab Home (0) in Zyklen
ft.**MotCntl**[n].outPos	bool Angabe, ob einer der Motoren die actPos überschritten hat nur bei PlotTo/PlotDelta

n bezieht sich auf die Motor-Position in der MotList der Instanzierung.

Methoden

PlotDelta

Fahren eines Motorenpaares im XY-Verbund um Xrel / Yrel Zyklen bezogen auf die aktuelle Position

ft.**PlotDelta**(MotNr, int Xrel, int Yrel)

MotNr : Nummer (Nr. oder int) des ersten M-Ausganges, der dem XY-Verbund bei der Instanzierung zugeordnet wurde.
Xrel / Yrel : Increment in Zyklen. Positive Werte rechtsdrehend (weg vom Endtaster, begrenzt durch den Wert der maxPos), negative linksdrehend (in Richtung Home Position / Endtaster).

Exception : InterfaceProblem; DoEvent, Abbrechbar.
Ereignis : PlotChange.

Siehe auch : PlotTo

Beispiel :
```
ft.PlotDelta(Nr.M1, 100, -50);
```

Von der aktuellen Position wird um 100 Zyklen in X- und um 50 Zyklen (hin zum Endtaster) Y-Richtung gefahren.

PlotHome

Anfahren der HomePosition für einen XY-Verbund von zwei Schrittmotoren.

ft.**PlotHome**(MotNr)

MotNr : Nummer (Nr. oder int) des ersten M-Ausganges, der dem XY-Verbund bei der Instanzierung zugeordnet wurde.

Gefahren wird in Richtung der zugeordneten Endtaster. Nach Erreichen der Endtaster wird um zwei Zyklen in Gegenrichtung freigefahren.

Exception : InterfaceProblem; DoEvent, Abbrechbar.

Beispiel :
```
private FishStep ft = new FishStep(new int[,]{{1,123},{3,456}});
ft.Home(Nr.M1);
```

Bei der Instanzierung werden die M-Ausgänge, die von den Motoren genutzt werden sollen, und die max. Fahrwege angegeben. Hier wird der X-Motor an M1/M2 (Endtaster E1, Fahrweg 123 Zyklen) und der Y-Motor an M3/M1(Endtaster E5, Fahrweg 456 Zyklen) angeschlossen. Anschließend wird auf die Home Position gefahren.

PlotTo

Fahren eines Motorenpaares im XY-Verbund auf die Position Xabs / Yabs.

ft.**PlotTo**(MotNr, int Xabs, int Yrel)

MotNr : Nummer (Nr. oder int) des ersten M-Ausganges, der dem XY-Verbund bei der Instanzierung zugeordnet wurde.
Xabs / Yabs : Position ab Home Position (0) in Zyklen. Begrenzung durch Endtaster.

Exception : InterfaceProblem; DoEvents, Abbrechbar.
Ereignis : PlotChange

Beispiel :
```
ft.PlotTo(Nr.M1, 150, 333);
```

Gefahren wird auf Position 123 / 273, wenn die Instanzierung (max 123 / 456)des Beispiels von PlotHome angenommen wird. Die Fahrwegbegrenzung hat hier also zugeschlagen. Der Fahrbefehl wird mit Erreichen von Xmax abgebrochen, daraus ergibt sich dann die erreichte Y-Position (123/150 * 333 = 273).

StepDelta

Fahren eines einzelnen Schrittmotors um Xabs Zyklen bezogen auf die aktuelle Position

ft.**StepDelta**(MotNr, int Xabs)

MotNr : Nummer (Nr. oder int) des ersten M-Ausganges, der dem Motor bei der Instanzierung zugeordnet wurde.

Bei positiven Werten wird weg vom Endtaster gefahren bei negativen Werten hin zum Endtaster. Fahrwegbegrenzung bzw. Endtaster werden beachtet.

Exception : InterfaceProblem; DoEvent, Abbrechbar.
Ereignis : StepChange.

Beispiel :
```
ft.StepDelta(Nr.M5, 123);
```

Der Schrittmotor an M5/M6 fährt von der aktuellen Position rechtsdrehend (weg vom Endtaster E9) 123 Zyklen.

StepHome

Anfahren der Home Position des über MotNr angegebenen Schrittmotors.

ft.**StepHome**(MotNr)

MotNr : Nummer (Nr. oder int) des ersten M-Ausganges, der dem Motor bei der Instanzierung zugeordnet wurde.

Gefahren wird in Richtung des zugeordneten Endtasters (E9).

Exception : InterfaceProblem; DoEvents, Abbrechbar:

Beispiel :
```
FishStep ft = new FishStep(new int[,]{{1,123},{3,456}});
ft.StepHome(Nr.M3);
```

Der Motor an M3/M4 wird in Richtung des Endtasters (E9) gefahren, die aktuelle Position wird auf 0 gesetzt.

StepTo

Fahren eines einzelnen Schrittmotors auf Position Xabs.

ft.**StepTo**(MotNr, int Xabs)

MotNr : Nummer (Nr. oder int) des ersten M-Ausganges, der dem Motor bei der Instanzierung zugeordnet wurde.

Endtaster und Fahrwegbegrenzung werden beachtet.

Exception : InterfaceProblem; DoEvent, Abbrechbar.
Ereignis : StepChange.

Beispiel :

```
const int mAufzug = 1;
ft.StepTo(mAufzug, 123);
```

Der Motor an M1/M2 fährt auf Position 123.

Ereignisse

PlotChange

Aufruf bei einer Veränderung der Position des Motorenpaares des XY-Verbundes durch die Methoden PlotTo/PlotDelta.

ft.**PlotChange** += new **FishStep.PlotDelegate**(name_event_routine)

void name_event_routine(object sender, int xPos, int yPos)
{ ... }

Beispiel :

```
FishStep ft = new FishStep(new int[,]{{1,123},{3,456}});
....
ft.PlotChange += new FishStep.PlotDelegate(PlotPosition);
....
static void PlotPosition(object sender, int MotNr,
                                          int xPos, int yPos) {
  cn.WriteLine("Position : " + xPos.ToString() + " / " +
                               yPos.ToString());
}
```

ft.PlotChange : Für das Ereignis PlotChange wird in die Liste des zuständigen Delegate – PlotDelegate – die Ereignisroutine PlotPosition eingetragen.

static void : Die Ereignisroutine. sender enthält, wie gewohnt, einen Hinweis auf das rufende Objekt. xPos / yPos die aktuelle Position des XY-Verbundes.

StepChange

Aufruf bei einer Veränderung der Position eines einzelnen Schrittmotors durch die Methoden StepTo/StepDelta.

ft.**StepChange** += new **FishStep.StepDelegate**(name_event_routine)

void name_event_routine(object sender, int actPos)

{ ... }

Beispiel :

```
FishStep ft = new FishStep(new int[,]{{1,123},{3,456}});
....
ft.StepChange += new FishStep.StepDelegate(StepPosition);
....
static void StepPosition(object sender, int MotNr, int actPos) {
  cn.WriteLine("Position : " + actPos.ToString() yPos.ToString());
}
```

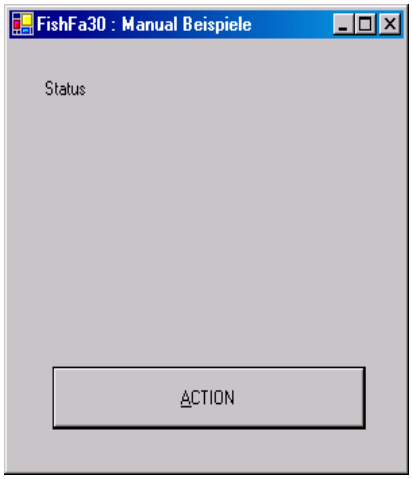
ft.StepChange : Für das Ereignis StepChange wird in die Liste des zuständigen Delegate – StepDelegate – die Ereignisroutine StepPosition eingetragen.

static void : Die Ereignisroutine. sender enthält, wie gewohnt, einen Hinweis auf das rufende Objekt. actPos die aktuelle Position des Schrittmotors.

Tips & Tricks

Programmrahmen

Die im Kapitel Techniken angeführten Programmausschnitte benötigen einen Programmrahmen innerhalb dessen sie ablaufen können. Hier wird – im Gegensatz zum Abschnitt Referenz – eine Windows Anwendung genutzt. Der Rahmen wird im Kapitel Techniken dann nicht mehr extra angegeben.



Elemente :

Label lblStatus

Button cmdAction

ACHTUNG : Der Programmrahmen unterscheidet sich ein wenig bei mit C# bzw. SharpDevelop erstellen Forms. Hier die VS.NET Schreibweise. Zu SharpDevelop siehe Einführung.

```
using System;
.......
using FishFa30;

namespace ManRef
{
  public class ManRef : System.Windows.Forms.Form
  {
    private System.Windows.Forms.Button cmdAction;
    private System.Windows.Forms.Label lblStatus;

// --- FishFace : Klassen-Instanz --------------
    private FishFace ft =
            new FishFace(true, false, 0);
.........

  private void cmdAction_Click(object sender,
                               System.EventArgs e)
  {
      try {
        ft.OpenInterface(Port.COM2);
        lblStatus.Text = "--- gestartet ---";
        ..... Code .....
        lblStatus.Text = "--- FINIS ---";
      }
      catch(FishFaceException eft) {
        lblStatus.Text = eft.Message;
      }
      finally {
        ft.CloseInterface();
      }
  } }
```

Erstellung des Programmrahmens erfolgt durch Anlegen eines neuen WindowsForm-Projektes. Zusätzlich ist dann ein Verweis auf FishFa30.DLL in der Projektmappe anzulegen. Auf diesen bezieht sich dann der `using FishFa30;`
Mit `private FishFace ft = new FishFace(true, false, 0)` wird eine Instanz der Klasse FishFace der Assembly FishFa30.DLL angelegt. `true` : Zugriff auf die Analog-Eingänge EX / EY ist möglich, `false` : kein Extension Module, `0` : Standard PollInterval.

Für den Button `cmdAction` wird eine Click-Routine angelegt. Sie enthält mit `ft.OpenInterface(Port.COM2)` und `ft.CloseInterface` die Methoden, die die Verbindung zum Interface kontrollieren. Sie sind in einen `try ... catch ... finally`-Block eingebettet um auftretende InterfaceProbleme abzufangen und in `lblStatus` anzuzeigen.

Allgemeine Techniken

Diese Techniken basieren auf der Klasse FishFace. Sie lassen sich natürlich auch bei Nutzung der davon abgeleiteten Klassen FishRobot und FishStep einsetzen.

Blinker/Schleife

Lampe an M1 blinkt im Sekundentakt :

```
const int mGelb = 1, Ein = 1, Aus = 0;

do {
  ft.SetMotor(mGelb, Ein);
  ft.Pause(555);
  ft.SetMotor(mGelb, Aus);
  ft.Pause(444);
} while (!ft.Finish());
```

Die Parameter für die FishFace Methoden sollten benannt werden. Für einige Standardwerte gibt es bereits Namen (Enums) : Nr.M1 ... Nr.E16, Dir.Ein, Dir.Aus, Dir.Links, Dir.Rechts ... und Wait.Ende, Wait.Time, Wait.ESC für die Methode WaitForMotors. Weitere sollte man selber erfinden. Zu beachten ist, daß es hier ein entweder/oder gibt : Verwenden der Enums oder eigener Namen.

Meistens enthält das Programm ein große Schleife in der alle Befehle wiederholt durchlaufen werden. Hier ist das `do .. while (!ft.Finish());` : Die Methode Finish prüft, ob ein Abbruchwunsch vorliegt und meldet dann true zurück.
Beenden der Schleifte duch ESC-Taste. Es kann auch zusätzlich ein E-Eingang angegeben werden : `ft.Finish(Nr.E8).` Beendigung durch ESC-Taste oder E8.

WechselBlinker

Lampen an M1 und M2 blinken im Wechsel.

Alternative 1 :

```
do {
  ft.SetMotor(Nr.M2, Dir.Aus);
  ft.SetMotor(Nr.M1, Dir.Ein);
  ft.Pause(444);
  ft.SetMotor(Nr.M1, Dir.Aus);
  ft.SetMotor(Nr.M2, Dir.Ein);
  ft.Pause(444);
} while (!ft.Finish());
```

Alternative 2 kompakter:

```
do {
  ft.SetMotors(0x1);
  ft.Pause(444);
  ft.SetMotors(0x4);
  ft.Pause(444);
} while (!ft.Finish());
```

Hier werden alle M-Ausgänge gleichzeitig geschaltet. Jeweils zwei bit pro M-Ausgang. Also 00000001 für M1 Ein und 00000100 für M2 ein. Alle anderen Ausgänge sind Aus.

Abfrage eines E-Einganges

Wenn E1 geschaltet ist lblStatus = "---EIN---" sonst "---AUS---" :

```
if (ft.GetInput(Nr.E1)) lblStatus.Text = "--- EIN ---";
else lblStatus.Text =  "--- Aus ---";
```

Warten auf einen E-Eingang

Wenn E1 geschlossen ist, wird in lblStatus "--- Es geht los ---" ausgegeben :

```
lblStatus.Text = "--- Zum Programmstart E1 drücken ---";
ft.WaitForInput(Nr.E1);
lblStatus.Text = "--- Es geht los ---";
```

Anzeige des Status der E-Eingänge

Status von E1 :

```
lblStatus.Text = "E1 : " & ft.GetInput(Nr.E1)
```

Laufende Anzeige des Status aller E-Eingänge :

```
do {
   string EWerte = "";
   for(int i = 0x80, E = ft.Inputs; i > 0; i >>= 1)
       EWerte += (E & i) > 0 ? "1" : "0";
   lblStatus.Text = EWerte.ToString();
   ft.Pause(1234);
} while (ft.Finish());
```

Die Angelegenheit benötigt nur ein paar Zeilen, sieht aber etwas vertrackt aus – und ist es auch : In EWerte werden die Schaltzustände der (hier 8) E-Eingänge gesammelt. Das geschieht in einer for-Schleife mit der Laufvariablen i, die gleichzeitig auch noch eine logische Maske ist. i wird auf den Ausgangswert 0x80 (das bit für E8 im InputStatus) gesetzt, gleichzeitig auch E auf den aktuellen InputStatus. Die Schleife läuft solange i > 0 ist. Bei jedem Schleifendurchlauf wird das Maskenbit in i um eine Position nach rechts geschoben (z.B. von E8 -> E7). An die Variable EWerte wird bei jedem Schleifendurchlauf der Status eines E-Einganges ("1" bzw. "0") gehängt. Der Status wird durch ein logische Und (&) von E und i bestimmt, die Auswertung geschieht mit einem Bedingungsoperator (?) und einer verkürzten Zuweisung.

Es geht auch einfacher, wenn man sich mit einer schlichten Hexa-Ausgabe begnügt :

```
do {
   lblStatus.Text = ft.Inputs.ToString("X");
   ft.Pause(1234);
} while (ft.Finish());
```

Analog-Anzeige

Laufende Anzeige der beiden Analog-Eingänge EX und EY :

```
do {
  lblStatus.Text = "EX : " + ft.GetAnalog(Nr.EX).ToString()
                 + " EY : " + ft.GetAnalog(Nr.EY).ToString();
  ft.Pause(1111);
while (!ft.Finish());
```

ACHTUNG : Hier muß beim Instanzieren das AnalogScan eingestellt werden (siehe Programmrahmen). Außerdem ist zu beachten, daß ein größeres PollInterval erforderlich ist. Hier wird ein Defaultwert eingesetzt. Kann man das nicht gebrauchen, kann man man GetAnalogDirect benutzten. Das greift jedesmal direkt auf das Interface zu (und hält das Programm solange an, der Parameter AnalogScan beim OpenInterface ist also nicht erforderlich) :

```
lblStatus.Text = "EY : ", ft.GetAnalog(Nr.EY).ToString();
```

Fahren für eine bestimmte Zeit

Der Motor an M3 soll 3,5 Sekunden nach links laufen :

```
ft.SetMotor(Nr.M3, Dir.Links);
ft.Pause(3500);
ft.SetMotor(Nr.M3, Dir.Aus);
```

Fahren zum Endtaster

Der Motor an M3 soll den Endtaster E5 anfahren und dann abschalten :

```
ft.SetMotor(Nr.M3, Dir.Links);
while(!ft.GetInput(Nr.E5));
ft.SetMotor(Nr.M3, Dir.Aus);
```

Das Beispiel ist nicht durch ESC-Taste abbrechbar und auch etwas umständlich.
besser :

```
ft.SetMotor(Nr.E3, Dir.Links);
ft.WaitForInput(Nr.E5);
ft.SetMotor(Nr.M3, Dir.Aus);
```

Fahren um eine vorgegebene Anzahl von Schritten

WaitForChange

Motor an M3 mit Impulstaster an E6 soll um 12 Impluse fahren.

```
ft.SetMotor(Nr.M3, Dir.Links);
ft.WaitForChange(Nr.E6, 12);
ft.SetMotor(Nr.M3, Dir.Aus);
```

WaitForPositionDown

Motor an M3 fährt von IstPosition = 12 auf ZielPosition = 0,
Impulszählung an E6 in Richtung 0 (Endtaster = E5) :

```
int IstPosition = 12;
ft.SetMotor(Nr.M3, Dir.Links);
ft.WaitForPositionDown(Nr.E6, ref IstPosition, 0, Nr.E5);
lblStatus.Text = "Istposition : " + IstPosition.ToString();
```

Die tatsächlich erreichte Position (kann um einen Impuls von der Vorgabe abweichen) steht nach dem Vorgang in IstPosition. Wird der Endtaster E5 vorher true, wird vorzeitig abgebrochen.

WaitForPositionUp

Motor an M3 fährt von IstPosition = 12 auf ZielPosition = 24,
Impulszählung an E6 in Richtung weg von Endtaster :

```
int IstPosition = 12;
ft.SetMotor(Nr.M3, Dir.Rechts);
ft.WaitForPositionUp(Nr.E6, ref IstPosition, 24);
lblStatus.Text = "Istposition : " + IstPosition.ToString();
```

Die tatsächlich erreichte Position (kann um einen Impuls von der Vorgabe abweichen) steht nach dem Vorgang in IstPosition.

WaitForMotors

Der Motor an M3 fährt für 12 Impulse an E6 mit verminderter Geschwindigkeit nach Links.

```
ft.SetMotor(Nr.M3, Dir.Links, Speed.L5, 12);
ft.WaitForMotors(0, 3);
```

Es wird gewartet, bis das Ziel erreicht wurde. Es geht auch ohne WaitForMotors, wenn das Programm anderweitig beschäftigt ist (Die Motoren schalten bei Erreichen der Zielposition selbsttätig ab). Siehe auch "Anmerkungen zu den Rob-Funktionen".

Zwei Motoren simultan mit laufender Positionsanzeige

Zwei Motoren (M3, M4) fahren simultan (gleichzeitig), die Impulszählung erfolgt an E6 und E8 (siehe auch Rob-Funktionen). Parallel dazu wird die aktuelle Position angezeigt :

```
ft.SetMotor(Nr.M3, Dir.Links, Speed.Full, 121);
ft.SetMotor(Nr.M4, Dir.Rechts, Speed.L8, 64);
do {
  lblStatus.Text = "Position M3 - M4 : " +
         ft.GetCounter(Nr.E6).ToString() + " - " +
         ft.GetCounter(Nr.E8).ToString();
} while (ft.WaitForMotors(300, 3, 4) == Wait.Time);
lblStatus.Text = "Position M3 - M4 : " +
         ft.GetCounter(Nr.E6).ToString() + " - " +
         ft.GetCounter(Nr.E8).ToString() + "--- Final ---";
```

Motor M3 fährt mit voller Geschwindigkeit um 121 Impulse nach Links
Motor M4 fährt mit halber Geschwindigkeit um 64 Impulse nach Rechts
WaitForMotors wartet auf beide, alle 0,3 Sekunden wird die aktuelle Position angezeigt. Zum Schluß wird die tatsächlich erreichte Position angezeigt. Zur Positionsanzeige wird mit GetCounter die aktuelle Position ausgelesen.

Lampen

Lampen werden meistens genauso behandelt wie Motoren (mit zwei Polen an einem M-Ausgang, z.B. `ft.SetMotor(1, ft.Ein))`, da sie aber nur ein oder ausgeschaltet werden können, ist auch die Schaltung an einem Pol eines M-Ausganges und Masse möglich man kann so bis zu acht Lampen an ein Interface anschließen :

```
ft.SetLamp(1, Dir.Ein);
ft.SetLamp(4, Dir.Ein);
ft.Pause(1000);
ft.SetLamp(1, Dir.Aus);
ft.SetLamp(4, Dir.Aus);
```

Die Lampen an Pin 1 und 4 (M1 vorn, M2 hinten) werden für 1 Sekunde eingeschaltet.
Hinweis : vernünftig geht das nur mit dem parallelen Interface.

Lichtschranken

Warten auf Lichtschranke

Lampe an M1, Phototransistor an E1. Es wird auf eine Unterbrechung der Lichtschranke gewartet :

```
const int mLicht = 1, ePhoto = 1, Ein = (int)Dir.Ein;
ft.SetMotor(mLicht, Ein);
ft.Pause(555);
```

```
ft.WaitForInput(ePhoto, False);
```

Lampe wird eingeschaltet, danach 0,5 Sekunden Pause um den Phototransistor "anzuwärmen", dann wird auf eine Unterbrechung der Lichtschranke gewartet.

Warten auf Einfahrt in eine Lichtschranke

Lampe an M1, Förderbandmotor an M3, Phototransistor an E1 :

```
const int mBand = 2, ePhoto = 1;
ft.SetMotor(mBand, (int)Dir.Links);
ft.WaitForLow(ePhoto);
ft.SetMotor(mBand, (int)Dir.Aus);
```

Der Motor M1 läuft solange bis ein Teil auf dem Band in die vorher nicht unterbrochene Lichtschranke einfährt. Die Lichtschranke wurde bereits vorher eingeschaltet.

Warten auf Ausfahrt aus einer Lichtschranke

Lampe an M1, Förderbandmotor an M3, Phototransistor an E1 :

```
const int mBand = 2, ePhoto = 1,
          Links = (int)Dir.Links, Aus = (int)Dir.Aus;
ft.SetMotor(mBand, Links);
ft.WaitForHigh(ePhoto);
ft.SetMotor(mBand, Aus);
```

Der Motor M1 läuft solange bis ein Teil auf dem Band,das die Lichtschranke unterbricht, aus der Lichtschranke herausgefahren ist.

Gleichzeitiges Schalten aller M-Ausgänge

Mit SetMotors können alle M-Ausgänge mit einem Befehl geschaltet werden. Dazu muß der Parameter MotorStatus entsprechend besetzt werden. Im MotorStatus sind pro M-Ausgang jeweils 2bit resierviert : 00 00 00 00 (bei Einsatz des Extension Modules nochmal 4). 00 bedeutet ausgeschaltet, 01 Drehrichtung links bzw. Ein, 10 Drehrichtung rechts.
00 01 00 00 demnach M3 links und 01 00 00 00 M4 links.

Einfache Ampel

Ein einfaches Ampelspiel sieht so aus : Grün – Gelb – Rot – RotGelb.
Die Lampen dazu M1 : Grün, M2 : Gelb, M3 : Rot und die Konstanten dazu :
mGruen = 00 00 00 01, mGelb = 00 00 01 00, mRot = 00 01 00 00,
dezimal = 1, 4, 16.

```
const int mGruen = 1, mGelb = 4, mRot = 16;
while (!ft.Finish()) {
  ft.SetMotors(mGruen);
  ft.Pause(1000);
  ft.SetMotors(mGelb);
  ft.Pause(250);
  ft.SetMotors(mRot);
  ft.Pause(1000);
  ft.SetMotors(mRot+mGelb);
  ft.Pause(250);
}
```

Listengesteuerte Ampel

Wenn man einen festen Ampeltakt vorgibt, kann man den Ablauf auch listengesteuert machen :

```
const int mGruen = 1, mGelb = 4, mRot = 16;
int[] Phase = {mGruen, mGruen, mGruen, mGruen,
               mGelb, mRot, mRot, mRot, mRot, mRot+mGelb};
while (!ft.Finish()) {
  foreach (int p in Phase) {
    ft.SetMotors(p);
    ft.Pause(250);
  }
}
```

Hier wird mit einer festen Taktung von 250 MilliSekunden gearbeitet. Das Verfahren lohnt bei komplexeren Steuerungen.

Lauflicht

Wenn an einem Interface gerade 4 Lampen angeschlossen sind, kann man ganz einfach ein Lauflicht programmieren :

```
while (!ft.Finish()) {
  for (int Phase = 1; Phase < 0xFF; Phase <<= 2) {
    ft.SetMotors(Phase);
    ft.Pause(555);
  }
}
```

Phase ist gleichzeitig Laufvariable und MotorStatus. Es wird jeweils ein M-Ausgang eingeschaltet. Dazu werden zyklisch 2 bit durch die Phase geschoben.

Betrieb eines Robots

Die Klasse FishRobot ist speziell auf den Betrieb von Robot-Motoren ausgerichtet. Als Robot-Motor wird ein Motor dann bezeichnet, wenn auf einer Motorwelle ein Impulsrad mitläuft, das einen Taster betätigt über den die Umdrehungen der Motorwelle in Form von Impulsen gezählt werden. Hinzu kommt ein Endtaster zur Bestimmung der Home-Position. Endtaster und Impulstaster sind dem jeweiligen M-Ausgang fest zugeordnet. Außerdem kann der max. Fahrweg vorgegeben werden (s.a. Anmerkungen zu den Rob-Funktionen). Es können bis zu vier (mit Extension Module bis zu acht) Motoren simultan (gleichzeitig) betrieben werden.

Die Testroutine ist eine Windows.Form, sie entspricht der im Kapitel Programmrahmen angeführten, die Instanzierung wird auf FishRobot umgestellt.

Robot-Fahren

Das Robot-Fahren geschieht über die Methoden MoveTo und MoveDelta. Als Parameter enthalten sie eine Liste der anzufahrenden Positionen (absolut von Home oder relativ zur aktuellen Position). Die Liste der zugehörenden M-Ausgänge und die Begrenzung wird bei der Instanzierung festgelegt. Die Home-Position (die Position an den Endtastern) wird mit MoveHome angefahren. Die Motoren müssen so gepolt sein, daß sie linksdrehen (Dir.Links). Bei Erreichen der Endposition werden die aktuellen Motorpositionen auf 0 gesetzt.

```
private FishRobot ft = new FishRobot(new int[,]{{3,222},{4,88}});
.......
private void cmdAction_Click(object sender, System.EventArgs e){
  try {
    ft.OpenInterface(Port.COM2);
     lblStatus.Text = "--- gestartet ---";
     ft.MoveHome();
     lblStatus.Text = "M3 auf Pos : " +
                              ft.MotCntl[0].actPos.ToString();
     ft.Pause(1234);
     ft.MoveTo(23, 34);
     ft.MoveDelta(-13, 6);
     ft.MoveTo(50,80);
   }
   catch(FishFaceException eft) {
     lblStatus.Text = eft.Message;
   }
   finally {
     ft.CloseInterface();
   }
}
```

Bei der Instanzierung wird die Roboter-Konfiguration festgelegt : Motoren an M3 und M4 mit Fahrwegbegrenzung auf 222 bzw. 88 Impulse.

Nach dem OpenInterface wird die Home-Position angefahren und die aktuelle Position auf 0 gesetzt. Das wird kurz angezeigt.

Anschließend wird auf die Position M3 = 23 und M4 = 34 gefahren. Dann wird M3 um 13 Impulse zurück auf Position 10 und M4 um 6 Positionen vor auf Position 40 gefahren. Zum Schluß wird die Position 50/80 angefahren.

Positionsanzeige

Die aktuelle Position kann nach einer Move-Methode den entsprechenden Werten von MotCntl entnommen werden, wie im vorhergehenden Beispiel geschehen. Die aktuelle Position kann aber auch während der Ausführung der Methoden MoveTo/MoveDelta über eine Ereignis-Routine angezeigt werden.

Dazu ist nach der Instanzierung (am besten noch im Konstruktor der Form-Klasse) eine entsprechende Routine anzumelden :

```
ft.PositionChange += new FishRobot.CommonDelegate(PositionsAusgabe);
```

In die Liste des von MoveTo/MoveDelta ausgelösten Ereignisses PositionChange wird der Delegate CommonDelegate mit dem Namen der zugehörenden Ereignisroutine (PositionsAusgabe) eingetragen :

```
private void PositionsAusgabe(object sender, int[] actPos) {
  lblStatus.Text = "Position : " + actPos[0].ToString() + " - " +
         actPos[1].ToString();
}
```

Die Ereignisroutine zeigt die aktuelle Position der beiden Motoren in lblStatus an. Natürlich könnten hier auch noch weitere Aufgaben wahrgenommen werden.

Betrieb von Schrittmotoren

Die Klasse FishStep ist von FishFace abgeleitet und unterstützt zusätzlich besonders den Einsatz von Schrittmotoren. Mit den Methoden StepHome / StepTo / StepDelta werden einzelne Schrittmotoren, die an zwei aufeinander folgende M-Ausgänge angeschlossen sind, unterstützt. Und mit den Methoden PlotHome / PlotTo / PlotDelta der Betrieb von zwei Schrittmotoren im XY-Verbund unterstützt. Die Schrittmotoren belegen 3 aufeinanderfolgende M-Ausgänge. Jedem Schrittmotor ist ein Endtaster fest zugeordnet.

Die hier verwendete Testroutine ist eine einfache Windows.Form, sie entspricht der im Kapitel Programmrahmen angeführten, die Instanzierung wird auf FishStep umgestellt.

Einzelner Schrittmotor

Beispiel : Fahrstuhl aus einem Schrittmotor mit einer (langen) Schneckenwelle senkrecht nach oben und einem "Korb" an der Schneckenmutter. Der Endtaster liegt auf E1.

```
private const int mFahrstuhl = 1;
private FishStep  ft = new FishStep(new int[,]{{mFahrstuhl,456}});
......
ft.StepChange += new FishStep.StepDelegate(PositionsAusgabe);
......
private void cmdAction_Click(object sender, System.EventArgs e) {
   try {
      ft.OpenInterface(Port.COM2);
      lblStatus.Text = "--- fährt auf Home (Keller) ---";
      ft.StepHome(mFahrstuhl);
      lblStatus.Text = "--- fährt auf Etage 4 ---";
      ft.StepTo(mFahrstuhl, 200);
      ft.Pause(1234);
      lblStatus.Text = "--- fährt eine Etage tiefer ---";
      ft.StepDelta(mFahrstuhl, -50);
      lblStatus.Text = "Das war's : Die Rufknöpfe nachrüsten";
   }
   catch(FishFaceException eft) {
      lblStatus.Text = eft.Message;
   }
   finally{
      ft.CloseInterface();
   }
}
```

Und die Routine zur Positionsausgabe :

```
private void PositionsAusgabe(object sender, int MotNr, int actPos) {
  lblStatus.Text = "Fahrstuhl auf Etage : " +  _
                   (actPos/50).ToString()+
                    " | " + (actPos%50).ToString();
}
```

Bei der Instanzierung der Instanzierung wird der Fahrstuhlmotor mFahrstuhl (M1/M2) mit einem max. Fahrweg von 456 Zyklen der Instanz zugeordnet.

Im Konstruktor der Form-Klasse wird noch die Routine für die PositionsAusgabe in die Ereignisliste von StepChange eingeklinkt.

In der Klick-Routine für den ACTION-Button läuft das Steuer-Programm :

- Nach OpenInterface : Anfahren der Home Position (StepHome)
- Fahren zu Etage 4 (StepTo)
- Eine Etage tiefer mit StepDelta
- Laufende Anzeige der Position mit Routine PositionsAusgabe.

Zwei Motoren im XY-Verbund : Plotten

Plotter mit zwei Schrittmotoren an M1 – M3 und den Endtastern E1 und E5. Für Testzwecke reichen die nackten Motoren mit Scheibenrädern drauf, damit man etwas sehen kann.

Instanzierung :

```
private const int mPlotter = 1;
private FishStep  ft = new FishStep(new
                                    int[,]{{mPlotter,456},{3,456}});
.....
```

Zuordnung der Ereignisroutine zur PostionsAusgabe :

```
ft.PlotChange += new FishStep.PlotDelegate(PositionsAusgabe);
......
```

Das Steuer-Programm in der ACTION-Button Klickroutine :

```
private void cmdAction_Click(object sender, System.EventArgs e) {
   try  {
      ft.OpenInterface(Port.COM2);
      lblStatus.Text = "--- fährt auf Home (linke untere Ecke) ---";
      ft.PlotHome(mPlotter);
      lblStatus.Text = "--- Anfahren 50/50 ----";
      ft.PlotTo(mPlotter, 50, 50);
      lblStatus.Text = "--- Zeichnen eines Quadrats ---";
      Vieleck(new int[,] {{50,0},{0,50},{-50,0},{0,-50}});
      lblStatus.Text = "--- Das war's ---";
   }
   catch(FishFaceException eft) {
      lblStatus.Text = eft.Message;
   }
   finally {
      ft.CloseInterface();
   }
}
```

Die laufende Positionsanzeige :

```
private void PositionsAusgabe(object sender, int MotNr,
                                          int xPos, int yPos)  {
   lblStatus.Text = "Position X/Y : " + xPos.ToString()+
                                            " / " + yPos.ToString();
}
```

Zeichnen des Quadrats :

```
private void Vieleck(int[,] RelList) {
   for(int i = 0; i < RelList.GetLength(0); i++)
          ft.PlotDelta(mPlotter, RelList[i,0], RelList[i,1]);
}
```

Diesmal sind die Kommentare in der Source.

Anmerkungen zum Verständnis

Zugriff auf das Interface

Der Zugriff auf das Interface erfolgt indirekt über eine Poll-Routine, die in regelmäßigen Abständen die Werte des Interface ausliest und gleichzeitig den Status der M-Ausgänge setzt (schaltet). Damit sind auch die Refresh-Bedingungen (ca. alle 300 mSek ein Zugriff) erfüllt, ein Abschalten des Interfaces erfolgt nicht mehr. Eine konstante Zeitbasis (typisch : 10 mSek) wird durch den MultiMediaTimer des Systems gewährleistet, der die PollRoutine in einem eigenen Thread betreibt.

Die ausgelesenen Werte werden in einem internen Kontrollblock abgestellt bzw.die Werte für die M-Ausgänge werden dort entnommen. Der Kontrollblock enthält darüberhinaus alle Werte die für den Betrieb eines Interfaces (mit Slave) erforderlich sind. Ein Parallel-Betrieb mehrerer Interfaces (z.B. eins an LPT, ein weiteres an COM1) ist somit möglich.

Die Poll-Routine erledigt über den reinen Verkehr mit dem Interface hinaus noch weitere Aufgaben. Das sind die Zählung der Impulse an den E-Eingängen (Veränderung am True/False-Status eines Einganges), die Geschwindigkeitssteuerung (durch zyklisches Ein/Ausschalten der M-Ausgänge) und im RobMode das Abschalten eines M-Ausganges, wenn der zugehörende Impuls-Counter den Wert null erreicht hat. Kurz vor Erreichen des Wertes null wird der M-Ausgang "gebremst".

Die angebotenen Zugriffsfunktionen sind ein Mix aus Notwendigkeit und Komfort. Open/CloseInterface stellen die Verbindung zum Interface her, setzen default-Parameter, starten den MultiMediaTimer und beenden die Verbindung wieder. Die GetInput-Funktion liest lediglich den Wert für einen E-Ausgang aus dem Kontrollblock. Dabei wird das zutreffend bit maskiert, ähnliches gilt für SetMotor und SetLamp in der Gegenrichtung. Es erfolgt auch hier keine direkte Ansteuerung des Interfaces.

Das tut dagegen die Funktion GetAnalogDirect, die für die Dauer eines (direkten) Zugriff auf einen Analog-Eingang die Poll-Routine abschaltet. Grund : besonders das Auslesen der Analog-Eingänge des parallelen Interfaces dauert wesentlich länger als das Lesen/Schreiben der E-Eingänge/M-Ausgänge. So kann das PollInterval auf die Bedürfnisse des Motorbetriebes eingestellt werden und ein gelegentliches Lesen der Analog-Eingänge (typischerweise bei Stillstand der Motoren) ermöglicht werden.

SetMotor(s) arbeiten nur mit dem Kontrollblock zusammen, führen aber (über das reine Setzen der M-Ausgänge hinaus) etwas komplexere Operationen aus.

Die Komfort-Funktionen und weitere Operationen auf den Kontrollblock können auch durch die Anwendung direkt vorgenommen werden.

Anmerkungen zu den Counters

Ein wesentliches Element zur Positionsbestimmung sind die Counter. Sie sind den E-Eingängen zugeordnet. In den Countern wird von einer zentralen Routine in umFish30.DLL jede Veränderung des Zustandes der E-Eingänge gezählt. Also z.B. das Öffnen oder auch das Schließen eines Tasters, der z.B. durch ein Impulsrad betätigt wird.

Die Counter sind Teil eines internen Kontrollblocks. Sie können mit entsprechenden Methoden gesetzt und abgefragt werden. Die Counter werden auch intern von einigen Funktionen/Methoden (z.B. SetMotor mit Parameter Counter und den meisten Wait-Methoden) genutzt, es kann also nicht damit gerechnet werden, daß sie über den Programmablauf Bestand haben.

Anmerkungen zur Geschwindigkeitssteuerung

Die Geschwindigkeitssteuerung beruht auf einem zyklischen Ein- und Ausschalten der betroffenen M-Ausgänge (Motoren). Dazu wird intern für jede Geschwindigkeitsstufe eine entsprechende Schaltliste vorgehalten. Die Geschwindigkeit wird durch den Parameter Speed für einen Motor und den Parameter SpeedStatus für alle Motoren angewählt. Die Geschwindigkeitssteuerung erfolgt in einem separaten Thread von umFish30.DLL, der die Motoren bis zu ihrem Ausschalten durch SetMotor(s) so steuert.

Anmerkungen zu den Rob-Funktionen

Hier werden allgemeine Anmerkungen zu den Rob-Funktionen und deren Nutzung durch Methoden der Klasse FishFace gemacht. Die spezielle Klasse FishRobot wird separat beschrieben.

Die Rob-Funktionen laufen in einem besonderen Betriebsmodus, dem RobMode. In diesem Modus werden die betroffenen Counter decrementiert. Bei Erreichen des Wertes 0 wird der betroffen Motor abgeschaltet. Während der letzten 6 Impulse fahren sie nur noch mit halber Geschwindigkeit um ein sicheres Erreichen der Endposition zu erreichen. Gelegentlich kann es trotzdem vorkommen, daß noch um einen Impuls weiter gefahren wird. Das kann man durch Abfrage des entsprechenden ImpulsCounters (wert > 0) feststellen und bei der Speicherung der aktuellen Position entsprechend berücksichtigen.

Der Betrieb eines Motors mit den Rob-Funktionen setzt ein festes Anschlußkonzept voraus. Zum jeweiligen Motor gehören je ein Impulstaster und ein Endtaster. Dazu folgende Tabelle :

Motor	**Endtaster**	**Impulstaster**
1	1	2
2	3	4
3	5	6
4	7	8
5	9	10
6	11	12
7	13	14
8	15	16

Die Motoren sind „linksdrehend“ d.h. sie drehen bei ftiLinks in Richtung Endtaster.

Die Motoren können einzeln über SetMotor oder alle gemeinsam über SetMotors geschaltet werden. Das Argument Counter gibt die Anzahl der zu fahrenden Impulse an.Die Argumente

ActPosition und ZielPosition beschreiben den Fahrauftrag. GetCounter /SetCounter greifen direkt auf den intern verwendeten Counter zu.

Die Motoren können auch alle mit einem Befehl geschaltet werden : SetMotors. Dazu müssen vorher die Parameter aufbereitetet werden.
MotorStatus : pro Motor 2bit, mit M1 : bit 0 und 1 beginnend.
00 : aus, 01 links, 10 rechts.
SpeedStatus : pro Motor 4bit, mit M1 : bit 0-3 beginnend,
0000 aus, 1000 halbe Kraft, 11111 voll.
ModeStatus : proMotor 4 bit, mit M1 : bit 0-3 beginnend,
0000 Normal-Mode, 0001 Rob-Mode, Rest z.Zt. nicht besetzt
(vorgesehen z.B. für Schrittmotorenbetrieb).

Beispiel : SetMotors(0x9, 0xF6, 0x11);
0x steht für Hexa, binär : 1001 | 11110110 | 10001 -> M2 = rechts, Speed 15 im Rob-Mode, M1 = links, Speed 6 im RobMode. Der Rest steht. Die zugehörenden Counter sind vorher mit SetCounter auf die gewünschte Fahrstrecke zu setzen.

Direction = 0 bzw. die Angabe im MotorStatus hält den Motor unabhängig von den Speed-Werten an.

Die Motoren laufen simultan (ggf. auch alle acht), sie können der Reihe nach mit SetMotor geschaltet werden. Sie starten dann beim nächsten Pollzyklus (Abfragezyklus) automatisch und laufen asynchron (d.h. unabhängig von den Aktionen des rufenden Programms) bis sie die vorgegebene Position erreicht haben. Sie werden dann ebenfalls (einzeln) während des Pollens abgeschaltet.

Um Festzustellen, ob die Motoren ihr Ziel erreicht haben und um das Programm mit den durch durch die Rob-Funktionen ausgelösten Aktionen wieder zu synchronisieren ist ein WaitForRobMotor(s) erforderlich.

Anmerkungen zu den Step-Funktionen

Der Betrieb von Schrittmotoren über ein fischrtechnik Interface ist möglich. Dazu ist die Klasse FishStep der Assembly FishFa30.DLL vorgesehen.

Schrittmotoren könnenmit FishStep **einzeln** oder im **XY-Verbund** paarweise betrieben werden. In beiden Fällen werden sie synchron betrieben, d.h. das Programm wartet, bis die vorgegebene Position erreicht ist. Im Falle des XY-Verbundes werden die beiden dazugehörenden Schrittmotoren simultan (gleichzeitig) betrieben.

Die Schrittmotoren erfordern zum Anschluß zwei aufeinanderfolgende M-Ausgänge (Einzel-Motoren) bzw. drei aufeinanderfolgende M-Ausgänge (XY-Verbund). Die M-Ausgänge können sich über Master und Slave (Extension Module) erstrecken. Der Betrieb erfolgt in Zyklen zu vier Schritten. Ein Zyklus ist damit auch die Positioniereinheit für einen Motor. Da die Motoren pro Schritt eine 7,5° Drehung machen, ergeben 48 Schritte eine volle Umdrehung. Das entspricht dann 12 Zyklen.

Diese Betriebsart wurde besonders in Hinblick auf die beschränkte Anzahl von M-Ausgängen am Interface gewählt. So ist ein Plotterbetrieb mit nur einem (Master) Interface möglich. Der freie M-Ausgang wird hier für den Stift-Antrieb genutzt. Da führt zu einem "Zittern" des Motors, der gerade nichts zu tun hat (Vor-/Rückschritt im Wechsel). Diese Zittern stört den Betrieb aber nicht weiter, da es von dem üblichen Spiel (Schneckenantrieb) des Modells aufgefangen wird.

Zeiten

Bei Schrittmotoren werden häufig Schnecken zum Modellantrieb genutzt. Die 'große' Schnecke hat eine Steigung von ca. 4,77 mm. d.h. bei einer Umdrehung der Motorwelle (mit der aufgesteckten Schnecke) legt die Schneckenmutter einen Weg von 4,77 mm zurück. Bei 12 Zyklen/Umdrehung sind das pro Zyklus 0,4 mm.

Bei einem Win2000 Rechner mit 1700 MHz und einem eingestellten PollInterval von 10 MilliSekunden ergeben sich folgenden Zeiten und Geschwindigkeiten :

200 Zyklen : 12,5 Sekunden. Pro Zyklus also 62.5 MilliSekunden.

100 mm Weg entsprechen 250 Zyklen. Das sind dann ca. 15 Sekunden für die 100 mm.

Anschluß der Schrittmotoren

Die verwendeten Schrittmotoren haben vier Kabelanschlüsse : rot, grün, schwarz, grau.

Zwei Schrittmotoren im XY-Verbund

		vorn	hinten
Motor X-Achse	Ma	rot	schwarz
	Mb	grün	grau
Motor Y-Achse	Ma	rot	schwarz
	Mc	grün	grau

vorn heißt die Stiftreihe an der Außenkante.
Mit Ma-Mc sind aufeinanderfolgende M-Ausgänge gemeint.
z.B. M1-M3. Aber M4-M6 sind auch möglich.
Die Motoren drehen bei PlotHome in Richtung 0 auf den zugehörenden Endtaster (Schließer, Kontakte 1 und 3). Für X ist der zu Ma gehörende, für Y der zu Mc gehörende
z.B. M1 : E1, M3 : E5. (Alle von M1 : E1, E3, E5, E7, E9, E11, E13, E15 (für M8)).

Ein einzelner Schrittmotor

		vorn	hinten
Motor A	Ma	rot	schwarz
	Mb	grün	grau

vorn heißt die Stiftreihe an der Außenkante.
Mit Ma-Mb sind aufeinanderfolgende M-Ausgänge gemeint.
z.B. M1-M2. Aber M4-M5 sind auch möglich.
Der Motor dreht bei StepHome in Richtung 0 auf den zugehörenden Endtaster (Schließer, Kontakte 1 und 3). Das ist der zu Ma gehörende.
z.B. M1 : E1 (Alle von M1 : E1, E3, E5, E7, E9, E11, E13, E15 (für M8)).