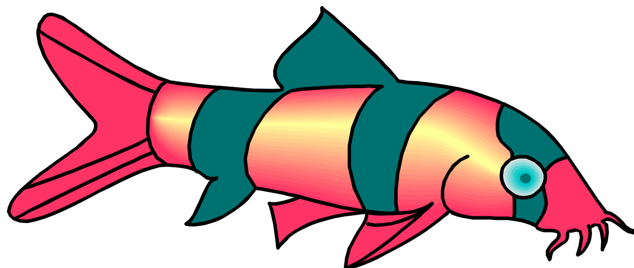
ftComputing

FishFa30.DCU

Handbuch zu Version 3.0

Ulrich Müller



Inhaltsverzeichnis

Einführung 4

Allgemeines 4
Installation 5
Interface Panel 6
Die StartAmpel 6
FisFa30.DPR : Das Beispielprojekt 9
  Allgemeines 9
  Das Testmodell 9
  FiFa30.DPR : Das UserInterface 10
  Sub Blinken : Wechselblinken mit zwei Lampen 11
  Sub Fahren : Fahren eines Motors zu einem Endtaster 12
  Sub Warten 13
  Sub Positionieren : Fahren zu einer vorgegebenen Position 14
  FisFa30.DPR : Der Programmrahmen 16

Modelle & FishFace-Erweiterungen 19

Kleine Modelle 19
  Händetrockner : Templates, externe Units (DCUs) 19
  FussAmpel : Über den Umgang mit SetMotors und Listen 21
Industry Robots 24
  RobStep : Steuerung eines Robots durch Einzelschritte 24
  RobCycle : Robotsteuerung durch Abruf eines Funktionszyklus 27

Referenz 33

Allgemeines 33
  Verwendete Parameterbezeichnungen 33
  Konstanten 34
Eigenschaften 34
Methoden 36
  Allgemeines 36
  Speed 37
  Counter 37
  RobMotoren 38
  Lampen am Interface 38
  Liste der Methoden 39

Copyright © 1998 – 2003 für Software und Dokumentation :

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

Freeware : Eine private Nutzung ist kostenfrei gestattet.

Dokumentname : FishFa30Delphi.DOC. Druckdatum : 03.02.2003
Titelbild : Einfügen | Grafik | AusDatei | Office | Fisch.WMF

Einführung

Allgemeines

FishFa30.DCU ist ein Visual Basic-Klassenmodul der als zentrale Klasse FishFace enthält. FishFa30 basiert auf der in VC++ 6.0 geschriebenen umFish30.DLL. FishFa30 selber liegt als Source und in übersetzter Form als FishFa30.DCU vor.

Dies Dokument soll eine Einführung in die Programmierung von ftComputing-Modellen mit Visual Basic und FishFace sein. Es berücksichtigt dabei besonders die Programmieranfänger. Einstiegsvoraussetzung ist der erfolgreiche Betrieb eines (selber) erweiterten "HelloWorld". Das heißt erste Kenntnisse werden vorausgesetzt. Es ist auch nicht beabsichtigt, Delphi Lehrbücher überflüssig zu machen.

Zum Einstieg empfiehlt sich ein Durcharbeiten des Dokumentes in der Reihenfolge der Kapitel :

- Installation mit delphiFish30Setup.EXE
- Test der erfolgreichen Installation mit dem Interface Panel
- Erstes Projekt : StartAmpel.DPR mit eigenen Modifikationen.
- Aufbau des TestModells
- Durcharbeiten des Beispielprojektes mit eigenen Modifikationen. Bei LLWin-Erfahrung vielleicht parallel dazu auch eine Erprobung der LLWin-Beispiele.
  - Blinken : Programmschleife und der Abbruch, Analogwerte
  - Fahren : Warten auf E-Eingänge, Motorbetrieb
  - Warten : Das Warten geht weiter, Abbruch
  - Positionieren : Anfahren frei vorgegebener Positionen
- Bauen und Programmieren eigener Modelle
  - HändeTrockner : Templates, externe Units
  - FußgängerAmpel : Listen mit Anweisungen abarbeiten
- Wenn Kasten Industry Robots vorhanden : Durcharbeiten der Robots-Beispiele
  - EinzelAktionen
  - Tätigkeitsablauf
  - Lernen, Speichern und Ausführen
- ?!?Weitere Modelle und FishFace-Erweiterungen können bei Bedarf konsultiert werden
  - ?!?Analog.CTL : Analog-Anzeige von EX und EY
  - ?!?KurveS30 : Erfassen von Meßwerten als Kurven
  - ?!?Step30 : Steuern von Schrittmotoren
- Eine Referenz der Eigenschaften und Methoden bildet den Abschluß des Handbuchs.

Installation

Vorausgesetzt wird ein Windows System ab Windows 95 mit einem installierten Delphi ab Version 4.
Verwendet wurde Windows 2000 SR2 und Delphi 4 SP3.

Das Setup-Programm delphiFish30Setup.EXE (www.ftcomputing.de/zip/delphifishsetup.exe) enthält alles was man zum Arbeiten mit FishFace benötigt
{app} gewählter Installationspfad
(default : C:\Programme\ftComputing), {sys} Windows\System-Verzeichnis :

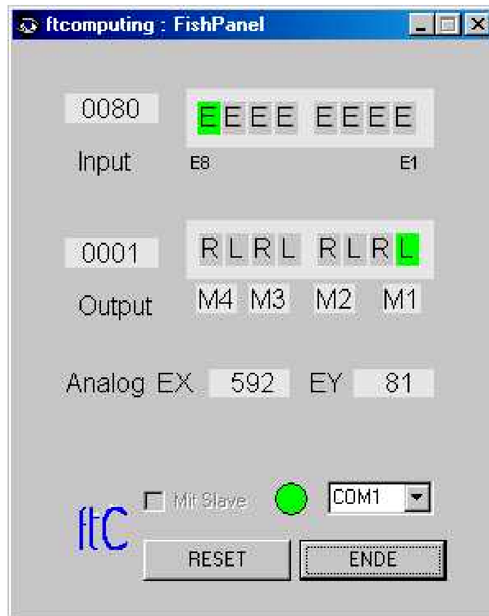
- {app} : dieses Dokument (FishFa30Delphi.PDF, auch über das Start-Menü erreichbar) und ein delphiFish30.TXT (ReadMe).
- {app} : Das Interface Panel umFishDP30.EXE
- {app}\FishFa30\Delphi4\DLL : die FishFa30.PAS Source, FishFa30.DCU
- {app}\FishFa30\Delphi4\Sample : das Beispielprojekt FisFa30.DPR.
- {app}\FishFa30\LLWin30: die LLWin 3.0 Beispiele dazu
- {sys} : umFish30.DLL und ggf. WinRT.SYS bzw. WRTdev0.VxD in \Driver

Das Setup-Programm läuft, wie üblich, weitgehend automatisch. Installationspfad, anzulegende Desktop-Icons, Einträge ins Start-Menü können gewählt werden. Ebenso die Installation eines Treibers für das (alte) Universal Interface an einem LPT-Port.

FishFa30.PAS/DCU liegt jedem Beispielprogramm bei, es sollte aber besser in ein Verzeichnis verschoben werden, das im Suchpfad von Delphi liegt z.B. :
C:\Programme\Borland\Delphi4\Imports.
Wenn man die Templates nutzen will, müssen sie in die Objektablage aufgenommen werden. (entsprechendes Projekt öffnen und über Menü | Der Objektablage hinzufügen aufnehmen, ggf. vorher noch eigene Anpassungen machen (z.B. den PortNamen festlegen).

Eine Deinstallation kann über Start **|** Einstellungen **|** Systemsteuerung **|** Software erfolgen.

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 mühselig 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 (ftiLinks, ftiLeft) 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.

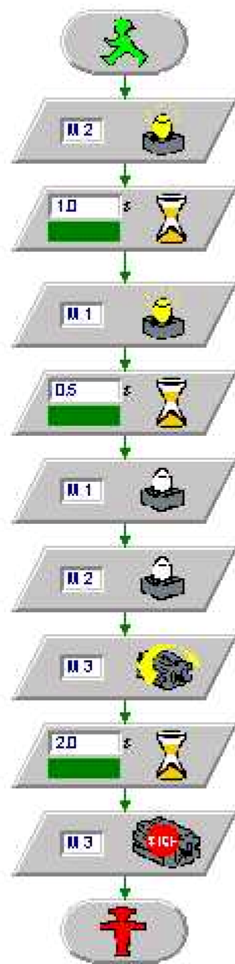
Die Analog-Zeile zeigt die (dezimalen) Werte an die an den Eingängen EX und EY gemessen werden.

Die StartAmpel

So geht's los :

- Interface anschließen, Funktion mit dem Interface Panel testen
- An das Interface anschließen M1 : gelbe Lampe, M2 rote Lampe, M3 MiniMotor
- im Verzeichnis {app}\Modelle30 z.B. C:\ftComputing\Modelle30\Delphi4\StartAmpel auf StartAmpel.DPR klicken.

- die Source von StartLights.PAS kurz ansehen (es sind nur ein paar Zeilen)
- F9 Drücken : das Programm läuft an
- Kontrollieren, ob COM1 der richtige Anschluß ist
  (es muß der Wert vom Interface Panel sein)
- START Drücken : Es geht los – und das wars denn auch schon
- Und nochmal :
  in StartLights.PAS in TfrmMain.cmdActionClick eine Stop-Adresse setzen auf
  ft := TFishFace.Create; (Zeile markieren, F5), Programm mit F9 wie gewohnt starten, den START-Button drücken
- Und dann mit F8 im Einzelschritt durch die Befehle gehen. Bei ft.Pause dauerts ein wenig.
- Mit F7 gerät man dann auch noch in FishFa30.PAS, das ist für den Anfang zu viel).



StartAmpel

```
procedure TfrmMain.cmdActionClick(Sender:
TObject);
var ft: TFishFace;
begin
  ft := TFishFace.Create;
  ft.OpenInterface(txtPortName.Text);
  ft.SetMotor(ftiM2, ftiEin);
  ft.Pause(1000);
  ft.SetMotor(ftiM1, ftiEin);
  ft.Pause(500);
  ft.SetMotor(ftiM1, ftiAus);
  ft.SetMotor(ftiM2, ftiAus);
  ft.SetMotor(ftiM3, ftiLinks);
  ft.Pause(2000);
  ft.SetMotor(ftiM3, ftiAus);
  ft.CloseInterface;
  Self.Close;
end;
```

Das Projekt StartAmpel.DPR besteht aus der gleichnamigen Projekt-Datei und weiteren StartAmpel.* Dateien.

Das Programm ist in StartLight.PAS enthalten. Ausßerdem sind in StartLight.PAS noch die Daten der Klasse TfrmMain (der einzigen und damit Hauptform des Projekts), zur Form gehört auch noch StartLights.DFM.

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

Hinzu kommt der Klassenmodul FishFa30.PAS mit den Interface-Funktionen.

Zu den Elementen :

- `var ft: TFishFace;` : Variable mit einem Zeiger auf eine Instanz der Klasse TFishFace (Teil von FishFa30.PAS). Die Instanz (oder auch das Objekt) wird mit `ft := TFishFace.Create;` angelegt.
- `ft.OpenInterface('COM1');` : Herstellen einer Verbindung zum Interface, hier dem Intelligent Interface an COM1 (txtPortName.Text liefert den aktuellen Inhalt des Textfeldes auf der Form). Es werden außerdem eine Reihe von Eigenschaften gesetzt.
- `ft.SetMotor(ftiM2, ftiEin);` : Einschalten der roten Lampe
  ftiM2 und ftiEin sind symbolische Konstanten, die von FishFa30 bereit gestellt werden. Sie sollen den Sinn der Variablen verdeutlichen. Es können aber auch einfache Zahlen oder eigene Konstanten angegeben werden.
- `ft.Pause` : Das Programm wird für 1000 MilliSekunden (1 Sekunde) angehalten.
- `ft.SetMotor(ftiM1, ftiEin);` die gelbe Lampe wird für 500 MilliSekunden zugeschaltet. und dann werden beide aus und der Motor an M3 wird für 2000 MilliSekunden angeschaltet
- und der Ordnung halber : `ft.CloseInterface`, die Verbindung zum Interface gekappt und mit `Self.Close;` entladen.

Das wars denn auch schon für den Anfang. Etwas ausführlicher geht’s dann mit dem Beispielprojekt weiter. Vorher sollte man aber noch etwas mit dem Programm "herumspielen".

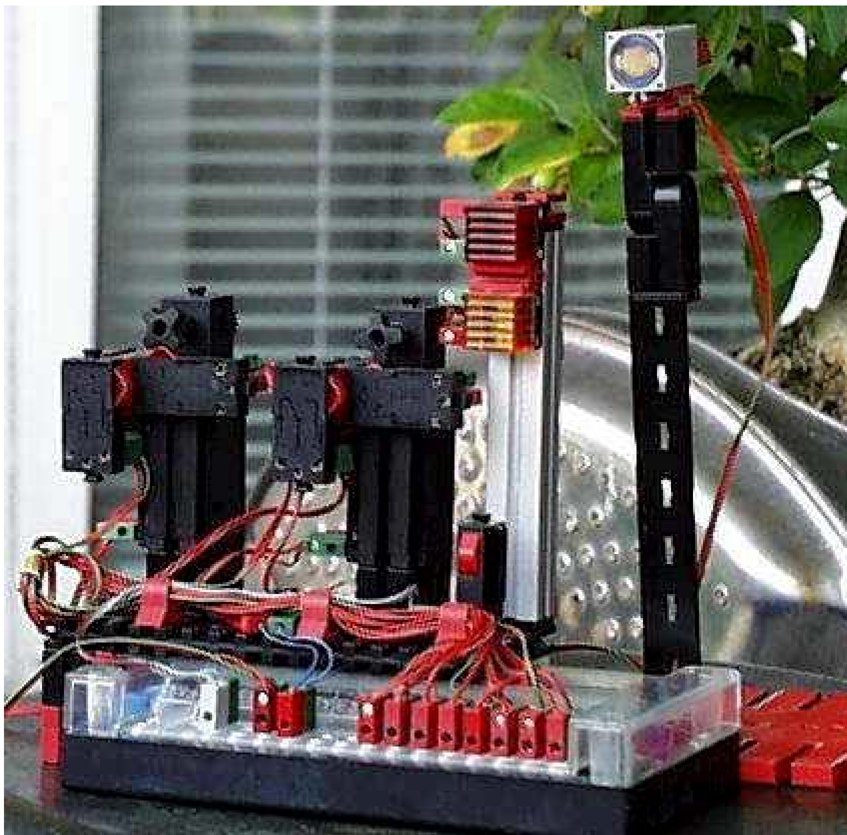
FisFa30.DPR : Das Beispielprojekt

Allgemeines

Der Einsatz von FishFace-Methoden und Eigenschaften wird anhand eines Beispielprogrammes vorgestellt. Die Methoden werden in mehreren Subs zu Gruppen zusammengefaßt und einer LLWin3.0-Lösung gegenübergestellt. Das erleichtert den Umstieg von LLWin und bietet gleichzeitig noch ein Ablauf-Diagramm. Die LLWin-Lösungen wurden bewußt knapp gehalten, insbesondere wurde auf den Einsatz des Terminalbausteins verzichtet, der in manchen Punkten der Bedienoberfläche des Delphi-Programms entspricht, aber letztlich vom Thema ablenkt. Das sollte LLWin-Kenner nicht davon abhalten, die gezeigten LLWin-Fragmente entsprechend zu erweitern (z.B. durch Abbildung der lblStatus-Ausgaben auf entsprechende des Terminal-Bausteins).

Das Beipielprojekt benötigt ein einfaches Testmodell :

Das Testmodell



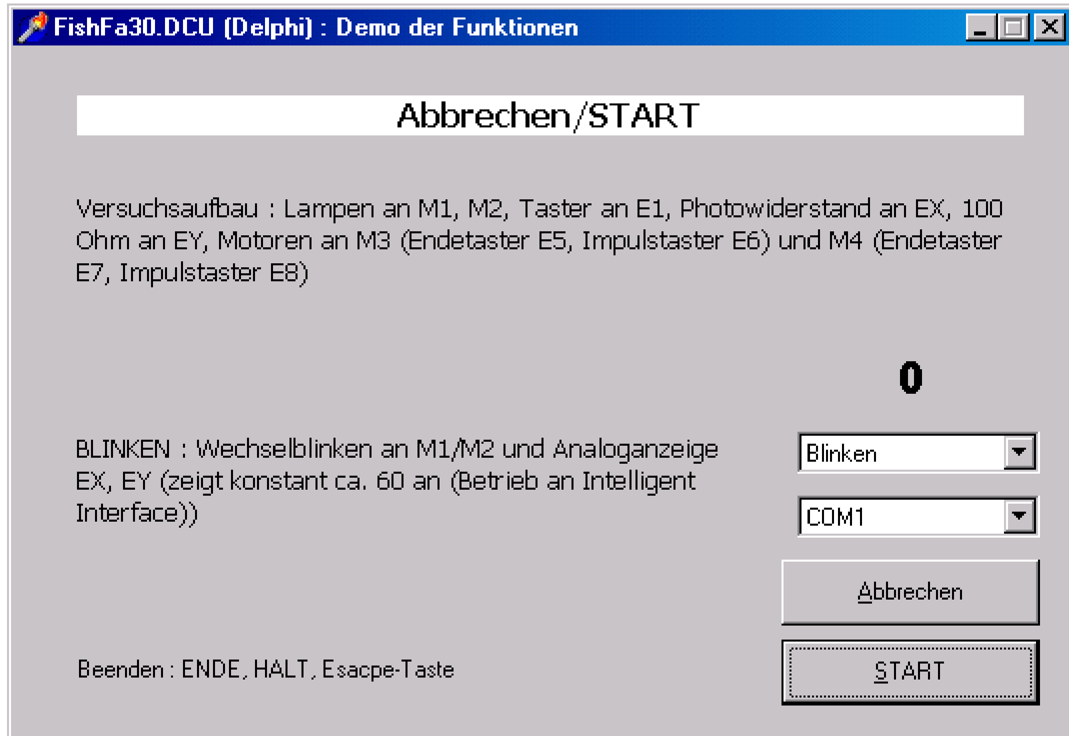
Das Testmodell hat nur die Aufgabe, die FishFace-Methode zu demonstrieren, es führt keine bestimmten Funktionen aus. Es besteht aus folgenden Elementen (von links) :

- Motor mit Getriebe und Impulsrad an M3 mit Endtaster an E5 und Impulstaster an E6
- Motor mit Getriebe und Impulsrad an M4 mit Endtaster an E7 und Impulstaster an E8
- Taster an E1
- Rote Lampe an M2 und Gelbe an M1
- Photowiderstand an EX (schwenkbar auf bewegbarer Säule)

- Festwiderstand 100 Ohm an EY (zu Vergleichszwecken, sollte beim Intelligent Interface Werte von ca. 60 anzeigen). Ist entbehrlich.

Alle Taster werden als Schließer geschaltet (Kontakte 1 und 3, Betätigung : Schließen)

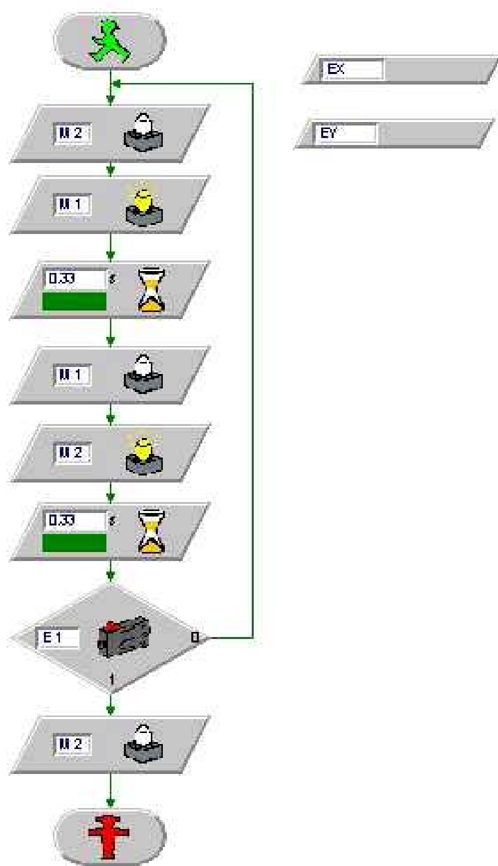
FiFa30.DPR : Das UserInterface



Das ist die einzige Form des Projektes FisFa30.DPR. Sie bietet die Kommunikation mit dem Benutzer und hat folgende Elemente :

- lblStatus : die weiße Zeile oben, in ihr werden der aktuelle Status des Programms und Anweisungen an den Benutzer angezeigt
- lblAnalog : Anzeige der aktuellen Werte von EX – EY und auch von Positionswerten.
- cboProgName : KomboBox mit den verfügbaren Demo Sub's. Eine Kurzbeschreibung wird jeweils links daneben angezeigt.
- cboPortName : KomboBox mit den zulässigen PortNamen, gleichzeitig Anzeige des aktuell eingestellten.
- ENDE/HALT/Abbrechen Button (cmdEnde) zum Beenden des Programms oder einer gerade laufenden DemoRoutine. Die Beschriftung wechselt entsprechend. DemoRoutinen können auch durch die Escape-Taste angehalten werden.
- START Button (cmdAction) zum Start der aktuell angezeigten DemoRoutine
  Bei laufender DemoRoutine ist der START Button verriegelt (Enabled = false) um einen erneuten Start der DemoRoutine zu verhindern. (man könnte auch noch cboProgName / cboPortName sperren).

Sub Blinken : Wechselblinken mit zwei Lampen



Blinken : M1/M2 bis E1 gedrückt

```
procedure TfrmMain.Blinken;
begin
  repeat
    ft.SetMotor(ftiM2, ftiAus);
    ft.SetMotor(ftiM1, ftiEin);
    lblAnalog.Caption :=
    IntToStr(ft.GetAnalog(ftiEX)) + ' - ' +
    IntToStr(ft.GetAnalog(ftiEY));
    ft.Pause(333);
    ft.SetMotor(ftiM1, ftiAus);
    ft.SetMotor(ftiM2, ftiEin);
    lblAnalog.Caption :=
    IntToStr(ft.GetAnalog(ftiEX)) + ' - ' +
    IntToStr(ft.GetAnalog(ftiEY));
    ft.Pause(333);
  until ft.Finish(ftiE1);
end;
```

Geblinkt wird hier mit zwei Lampen (M2 Rot und M1 Gelb) im Wechsel in jeweils 333 MilliSekunden Abstand und das in einer Endlosschleife.

Lampe M2 aus : `ft.SetMotor(ftiM2, ftiAus)` .... auch hier als Parameter die Const ftiM2 ...

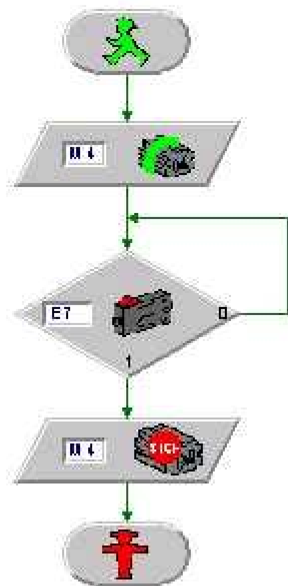
`ft.Pause(333)` : Programm für 0,3 Sekunden anhalten. Aber so ganz halten tut es nicht. `Pause` reagiert auf die Esc-Taste mit Abbruch.

Neu ist hier die Schleife `repeat until ft.Finish(ftiE1);` : die Befehle dazwischen werden solange (until) durchlaufen bis die Methode `ft.Finish` true zurückgibt. Das tut sie hier wenn auf die Esc-Taste gedrückt wurde oder der Taster an E1 geschlossen wurde (bei dem TestModell manuell). Außerdem wird die Eigenschaft NotHalt ausgewertet, true bedeutet auch Abbruch. Im Beispiel wird `ft.NotHalt` bei HALT gesetzt.

Bei der LLWin Codierung wurde anstelle von Finish der Eingang-Baustein verwendet, es wird hier nur E1 abgefragt, mehr ist da auch nicht erforderlich, da die Laufzeitumgebung genügend Abbruchmöglichkeiten bietet. Die reine Abfrage eines E-Einganges geschieht bei FishFace mit `ft.GetInput`.

Außerdem werden mit `ft.GetAnalog(ftiEX)` und `ft.GetAnalog(ftiEY)` die aktuellen Analogwerte abgefragt und in `lblAnalog` angezeigt. LLWin macht die Abfrage in der Laufzeitumgebung und zeigt sie mit dem WerteAnzeigen-Baustein an (rechts oben).

Sub Fahren : Fahren eines Motors zu einem Endtaster



Fahren : Motor läuft bis E1 gedrückt wird

```
procedure TfrmMain.Fahren;
begin
  lblStatus.Caption :=
                'Beenden : E7 drücken';
  ft.SetMotor(ftiM4, ftiRight);
  ft.WaitForInput(ftiE7);
  ft.SetMotor(ftiM4, ftiOff);
end;
```

Eine ganz einfache Übung : Fahren bis das Ende in Form eines geschlossenen E7 Tasters kommt.

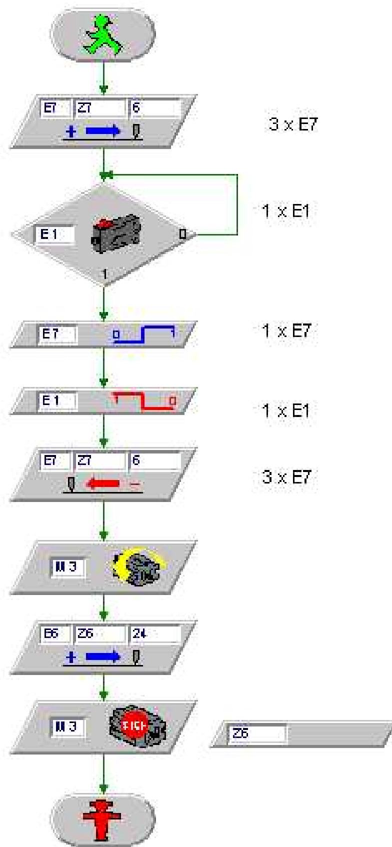
Hier gibt es im Gegensatz zu Blinken keine Schleife, da es hier nichts zu Wiederholen gibt.

Zur Erkennung von E7 = true wird `ft.WaitForInput` eingesetzt, der die gleichen "Nebenwirkungen" wie `ft.Finish` hat.

Bei LLWin wieder der Eingabe-Baustein. und danach folgt dann noch ein Motor Aus, bei Delphi aber nicht, Grund : im rufenden `cmdActionClick` gibt es nach dem Aufruf für Fahren noch ein `ft.ClearMotors` (Gilt natürlich auch für Blinken).

in lblStatus wird eine Bediener-Anweisung "Beenden : E7 drücken" angezeigt

Sub Warten



Warten : Demo der Warte-Befehle

```
procedure TfrmMain.Warten;
var Zahler: LongInt;
begin
  lblStatus.Caption :=
               'WaitForChange : 3 x E7 ' +
               'drücken & loslassen';
  ft.WaitForChange(ftiE7, 6);
  lblStatus.Caption :=
                'WaitForInput : E1 drücken';
  ft.WaitForInput(ftiE1);
  lblStatus.Caption :=
                  'WaitForHigh : E7 drücken';
  ft.WaitForHigh(ftiE7);
  lblStatus.Caption :=
                    'WaitForLow : E1 drücken';
  ft.WaitForLow(ftiE1);
  lblStatus.Caption :=
        'WaitForPositionDown : 3 x E7 drücken';
  Zahler := 6;
  ft.WaitForPositionDown(ftiE7, Zahler, 0);
  lblAnalog.Caption := IntToStr(Zahler);
  lblStatus.Caption :=
      'WaitForPositionUp : mit halber ' +
      'Motorkraft 24 Impulse';
  ft.SetMotor(ftiM3, ftiLeft, ftiHalf);
  ft.WaitForPositionUp(ftiE6, Zahler, 24);
  ft.SetMotor(ftiM3, ftiOff);
  lblAnalog.Caption := IntToStr(Zahler);
end;
```

Die Sub Warten zeigt restlichen Möglichkeiten auf etwas zu warten (zwei sind ja schon bekannt : WaitForInput, Pause/WaitForTime). Gewartet wird immer auf eine oder mehrere Veränderungen an einem der E-Eingänge.

WaitForChange wartet hier auf 6 Impulse (Pegelwechsel) an E7, dazu muß man im Beispielprogramm 3 mal den Taster an E7 drücken (und wieder loslassen).
Nutzen kann man diese Methode z.B. zum Verfahren eines Modells um eine festgelegte Anzahl von Impulsen, die Impulse werden dann über ein Impulsrad durch den Motor ausgelöst. Siehe auch WaitForPosition.

WaitForInput wartet auf E1 true
z.B. beim Anfahren einer Endposition

WaitForHigh wartet auf einen false/true Durchgang an E7. Das heißt : E7 muß vor dem Drücken des Taster offen gewesen sein.
Beispiel : Ausfahren aus einer Lichtschranke.

WaitForLow dito : aber true/false Durchgang
Beispiel Einfahren in eine Lichtschranke.

WaitForPositionDown : ähnlich WaitForChange, es wird aber mit den tatsächliche Positionen gearbeitet, Zahler enthält die aktuelle Position und der Parameter Position (hier 0) ist die ZielPosition, Zahler enthält nach Ende der Methode die tatsächlich erreichte Position, dabei werden die festgestellten Impulse von Zahler abgezogen.

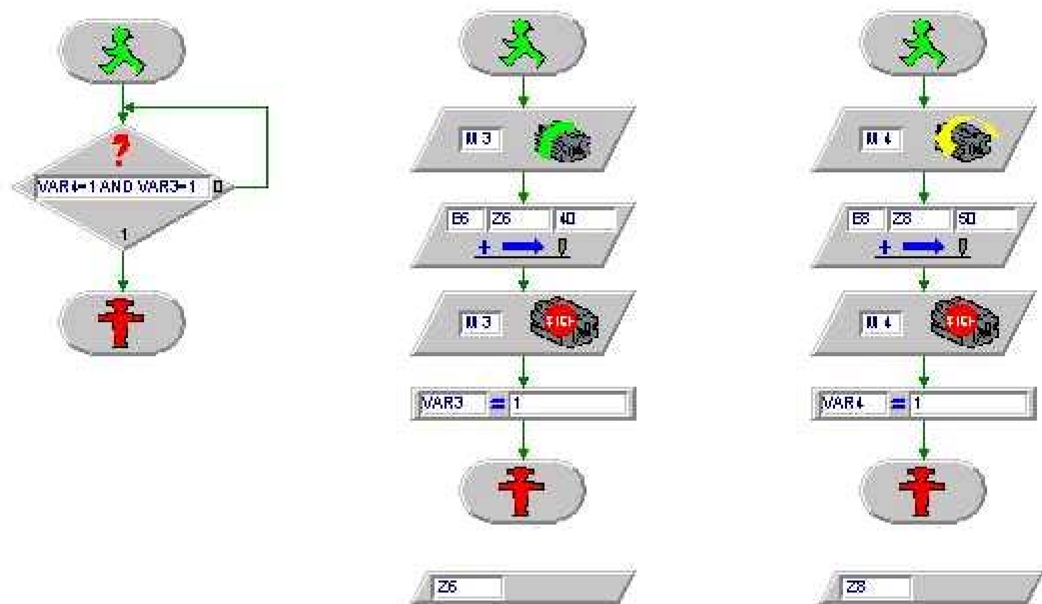
WaitForPositionUp zur Abwechslung mit Motorkraft und aufwärts ZielPosition ist hier 24, AusgangsPosition 0, da Zahler seit WaitForPosiionDown nicht verändert wurde.
In `lblAnalog` wird in beiden Fällen die tatsächlich erreichte Position angezeigt.

Optionale Parameter : Werte, die der Methode wahlweise übergeben werden können, die Methode ändert dadurch ihr Verhalten. ft.SetMotor ftiM3, ftiLeft, ftiHalf. ftiHalf steht als Wert für den Parameter Speed, also mit halber Geschwindigkeit, default ist ftiFull.

Die WaitFor... Methoden können alle vorzeitig durch die Esc-Taste oder die Eigenschaft ft.NotHalt beendet werden. WaitForChange, WaitForPositionUp/Down kennen außerdem noch die TermInputNr(optionaler Parameter), die Nummer eines E-Einganges, der bei true ebenfalls ein vorzeitiges Ende bewirkt, üblicherweise ein EndTaster.

lblStatus enthält immer die entsprechende Bedienanweisung

Sub Positionieren : Fahren zu einer vorgegebenen Position



Simultanes Anfahren von Position 40 (M3) und 50 (M4)

```
procedure TfrmMain.Positionieren;
begin
  lblStatus.Caption :=
            'Motor an M4 fährt 50 links und Motor an M3, 40 rechts';
  ft.SetMotor(ftiM4, ftiLeft,   ftiHalf, 50);
  ft.SetMotor(ftiM3, ftiRechts, ftiFull, 40);

  while ft.WaitForMotors(100, ftiM4, ftiM3) = ftiTime do
        lblAnalog.Caption := IntToStr(ft.GetCounter(ftiE6)) +
                             ' - ' + IntToStr(ft.GetCounter(ftiE8));

  lblAnalog.Caption := IntToStr(ft.GetCounter(ftiE6)) +
                       ' - ' + IntToStr(ft.GetCounter(ftiE8));
end;
```

Hier fahren zwei Motoren gleichzeitig (simultan) unterschiedliche Ziele an. Der Motor an M4 fährt mit halber Geschwindigkeit 50 Impulse nach links und der Motor an M3 mit voller Geschwindigkeit 40 Impulse nach rechts. Dazu hat die bekannte Methode `ft.SetMotor` einen weiteren (optionalen) Parameter Counter der die angegebenen Werte enthält. Die Methode ist asynchron, d.h. der Motor läuft unabhängig vom übrigen Programm weiter. Das ist ansich nichts neues, denn die anderen Varianten von `ft.SetMotor` verhalten sich

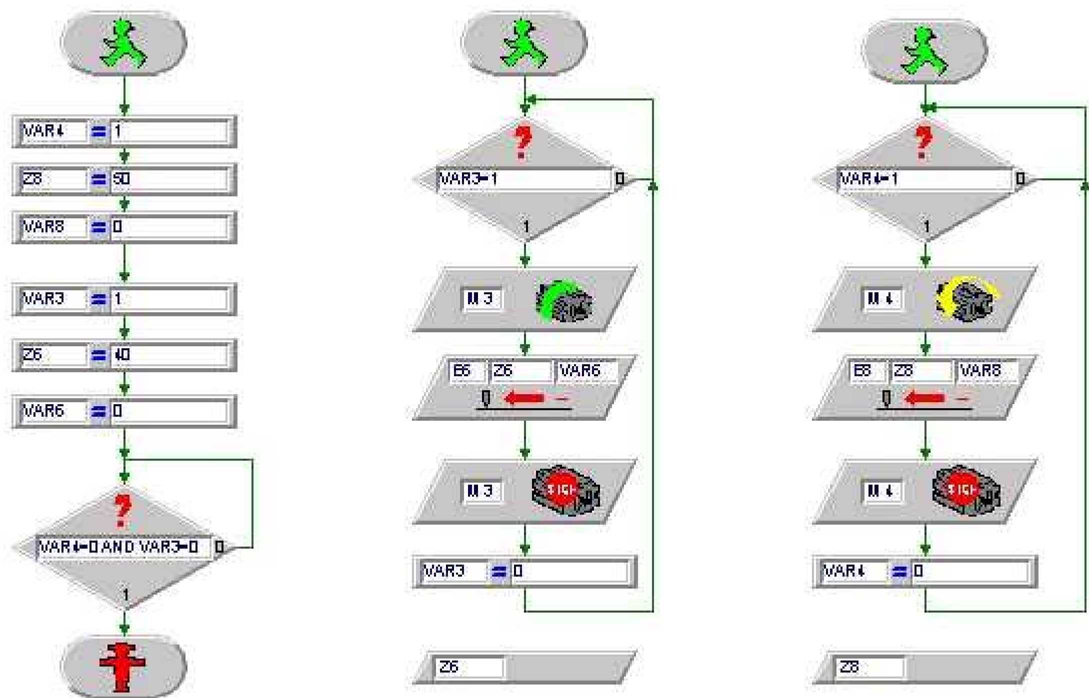
ebenso, neu ist hier, das die Motoren intern bei Erreichen der vorgegebenen Impulszahl abgeschaltet werden. Das Programm, das sie über die ft.SetMotor Methoden gestartet hat, muß deswegen (bei Gelegenheit) nachfragen, ob die Zielwerte schon erreicht sind, in der Zwischenzeit kann etwas anderes erledigt werden.

Die Methode `ft.WaitForMotors` kann das tun. Der erste Parameter hier 100, gibt die Zeit in Millisekunden an, die jeweils gewartet werden soll, spätestens dann kehrt sie zurück und das rufende Programm fragt ab, warum. Die zweiten Parameter sind eine Liste von M-Ausgängen (Motoren) auf die gewartet werden soll.

Hier wird eine `while` – Schleife verwendet in der alle 100 MilliSekunden (der WarteParameter) der aktuelle Counterstand angezeigt wird. Die Schleife wird solange durchlaufen, wie der Rückkehrwert = ftiTime ist d.h. die Motoren sind noch nicht am Ziel. weitere Rückkehrwerte sind ftiEnde : Ziel erreicht, ftiEsc und ftiNotHalt : vorzeitiges Ende.

Damit die ganze Zählerei funktioniert ist ein fester ModellAufbau erforderlich zu jedem Motor gehört ein Endtaster und ein Impulstaster (M1 / E1 / E2, M2 / E3 / E4, M3 / E5 / E6 ....). Der Endtaster dient zum einen als fester Bezugspunkt für die Modell Home-Position und ist eine "Notbremse" bei dessen Erreichen der entsprechende Motor ggf. vorzeitig abgeschaltet wird. Aus Gründe der Verfügbarkeit gibt es nur für eine Drehrichtung einen Endtaster, der Endtaster wird bei Linkslauf ausgewertet.

LLWin kennt keinen entsprechenden simultanen Befehl, hat dafür aber die Möglichkeit (simultaner) Task, sprich mehrere "Grüner Männchen" in einem Projekt. Oben eine knappe Lösung mit drei Grünen Männchen. Unten eine Lösung, die auch Wiederholungen zuläßt.



Simultanes Anfahren von Position 40 (M3) und 50 (M4)

FisFa30.DPR : Der Programmrahmen

Hinweise

Das Programm nutzt die Unit FishFa30, deswegen muß unter uses ein entsprechender Eintrag vorhanden sein :

```
interfaces
uses
  ...., FishFa30;
```

Solange Programme innerhalb ihrer Entwicklungsumgebung (IDE – Interactive Development Environment) gestestet werden, können auch deren Möglichkeiten zu Start und Abbruch genutzt werden (Bei Delphi : Das Start Menü bzw. die entsprechenden Icons oder Shortcuts) Bei der Ausführung eines übersetzten Programmes über ein Desktop oder Explorer-Icon steht sie nicht mehr zur Verfügung, in diesem Fall muß das Programm selber entsprechende Möglichkeiten bieten.

Methoden, die den Programmablauf anhalten oder steuern (Finish, WaitForxxx, Pause) sind abbrechbar. Dazu werten sie die Esc-Taste und die Eigenschaft NotHalt aus. Die Methoden beenden sich dann ohne besondere Hinweise. Die Esc-Taste kann jederzeit durch den Bediener betätigt werden, die Eigenschaft NotHalt muß im Programm z.B. beim Betätigen des HALT-Buttons gesetzt werden

Ebenso muß ein wiederholter Start bzw. ein Schließen der Form während das Programm noch (in einer Schleife) läuft (und auch nach Schließen der Form weiterläuft) verhindert werden. Das geschieht hier durch Verriegeln der entsprechenden Buttons.

cmdActionClick : Steuern des Modellbetriebes

```
procedure TfrmMain.cmdActionClick(Sender: TObject);
begin
  try
    ft.OpenInterface(cboPortName.Text);
    cmdAction.Enabled := false;
    cmdEnde.Caption   := '&HALT';
    lblStatus.Caption := 'läuft';
    picFish.Visible   := true;

    case cboProgName.ItemIndex of
    0: Blinken;
    1: Fahren;
    2: Warten;
    3: Positionieren;
    end;

    lblStatus.Caption := 'START/ENDE';
    ft.ClearMotors;
  except
    on e: EFishFace do lblStatus.Caption := e.Message;
  end;

  ft.CloseInterface;
  cmdAction.Enabled := true;
  cmdEnde.Caption   := '&ENDE';
  picFish.Visible   := false;
  cmdEnde.SetFocus;
end;
```

`ft := TFishFace.Create(true)` (im globalen Bereich zu Programmanfang) erzeugt eine **FishFace Instanz**. Sie wurde in den globalen Bereich gestellt, da ft von mehreren Subs (die ja alle ihren eigenen Gültigkeitsbereich für dort deklarierte Variablen haben) zugegriffen wird. Die Analogeingänge werden abgefragt (true), es ist kein Extension Module vorhanden (default).

`ft.OpenInterface` stellt **Verbindung** zu einem Intelligent Interface am COM1-Port her. Die Verbindung wird durch ft.CloseInterface wieder beendet.

Während der Laufzeit des Programmes können **Fehler** auftreten, sie sollten abgefangen werden. Das geschieht hier durch einen `try except end` Block. Alle Statements, die logisch zwischen try except – also auch die in procedures -, springen bei Auftreten eines Fehlers in den `except end` Teil. Die Fehlerbehandlung besteht hier in der Anzeige eines Fehlertextes für alle erkannten FishFace Fehler in der Status-Box. Weitere Fehler führen zum Standard Systemverhalten (MessageBox).

Die von FishFace erkannten Fehler werden in der Form Fehlergrund.verusachendeMethode angezeigt. z.B. : "InterfaceProblem.OpenInterface (d.h. man hat mal wieder die Stromversorgung am Interface vergessen).

`case cboProgName.ItemIndex` ist lediglich ein **Sprungverteiler** über den die durch die KomboBox cboProgName ausgewählte procedure aufgerufen wird.

`cmdAction.Enabled := false` **sperrt** den erneuten Aufruf von START (cmdActionClick). Nach dem Ablauf einer Demo-Sub wird es dann wieder entsperrt : `cmdAction.Enabled := true`. Parallel dazu wechselt die Beschriftung von cmdEnde von ENDE in HALT und wieder in ENDE.

Nach dem Ablauf einer Demo-Sub werden außerdem alle M-Ausgänge (Motoren) **ausgeschaltet** und die Interface-Verbindung geschlossen, deswegen können sie am Ende der Demo-Subs fehlen.

cmdEndeClick : Beenden des Modellbetriebes

```
procedure TfrmMain.cmdEndeClick(Sender: TObject);
begin
  if cmdEnde.Caption = '&HALT' then ft.NotHalt := true else
Self.Close;
end;

procedure TfrmMain.FormCloseQuery(Sender: TObject;
                                  var CanClose: Boolean);
begin
  if cmdEnde.Caption = '&HALT' then CanClose := false;
end;
```

Während des **Ablaufs** einer Demo-Sub ist der cmdEnde-Button mit HALT beschriftet, das wird hier abgefragt : `if cmdEnde.Caption = '&HALT' then`. Das Programm kann an dieser Stelle nicht einfach beendet werden, da das Modell noch läuft, in diesem Fall wird die Eigenschaft `ft.NotHalt` gesetzt, um einen Ende-Wunsch anzumelden (er wird z.B. von `ft.Finish` erkannt), Nach Ende einer Demo-Sub wechselt die Beschriftung wieder auf ENDE und die Form kann entladen und damit das Programm beendet werden.

Da über das **x** rechts oben im Rahmen der Form auch ein Programmende herbeigeführt werden kann,wird das konsequenterweise in FormCloseQuery bei `cmdEnde.Caption = '&HALT'` verhindert.

FormCreate : Startwerte

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  ft := TFishFace.Create(true);
```

```
  cboPortName.ItemIndex := 1;
  cboProgName.ItemIndex := 0;
end;
```

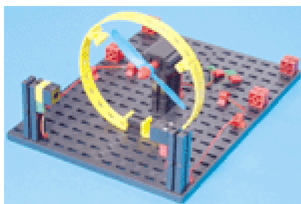
Gleich beim Start des Programm, dem Laden der Form, wird eine Instanz von TFishFace gebildet und es werden die in den KomboBoxen anzuzeigenden Texte bestimmt : `cboPortName.ItemIndex := 1;` und `cboProgName.ItemIndex := 0;`. Besonders cboPortName sollte hier auf den eigenen Anschluß angepaßt werden.

Modelle & FishFace-Erweiterungen

Kleine Modelle

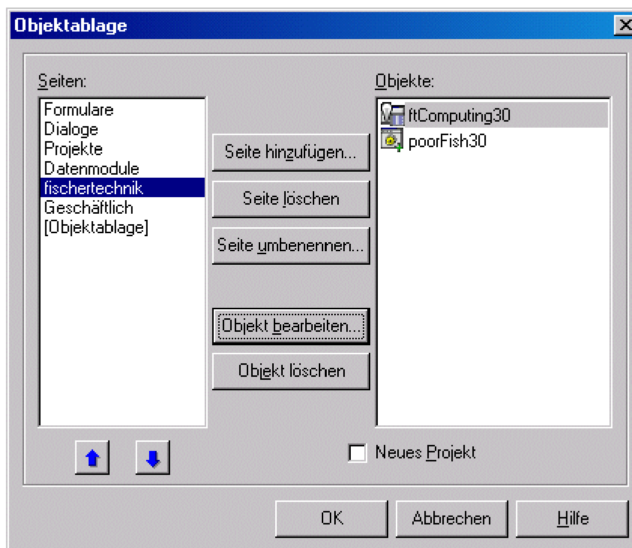
Eine Folge kleiner Modelle, die primär Visual Basic-Programmiertechniken und den Umgang mit FishFace zeigen sollen.

Händetrockner : Templates, externe Units (DCUs)



Modell aus dem Computing Starter Kit 16 553 (Handbuch einzeln : 30 434). Aufgabe laut Handbuch : "Der Händetrockner soll so programmiert werden, daß, sobald die Lichtschranke unterbrochen wird, der Lüfter ein- und nach 5 Sekunden wieder ausgeschaltet wird."

Templates



Bisher war das Programmgerippe eine "Einzelanfertigung" zu der die FishFa30.PAS und die eigentliche Anwendung hinzugefügt wurde. Jetzt soll ein Programmgerippe "von der Stange" genutzt werden, ein sogenanntes Template. Das ist ein vorgefertigtes Programmgerippe in das nur noch die eigentliche Anwendung einzufügen ist. In diesem Fall geht es um ftComputing30.DPR. Das ist ein lauffähiges Projekt mit START und ENDE Button, FishFa30 und den den erforderlichen ft.OpenInterface / CloseInterface Methoden-Aufrufen.

Zu finden ist es im Verzeichnis {app}\Templates30\Delphi4. Vor der ersten Benutzung ist es in die Objektablage einzutragen. Dazu ist das Template Projekt normal zu öffnen, ggf. an eigene Bedürfnisse anzupassen (z.B. PortNamen fest eintragen, ResourceFile erstellen) und dann über Menü | Projekt | Der Objektablage hinzuzufügen.

Nutzen kann man Template dann beim Anlegen eines neuen Projektes : (Menü : Datei | Neues Projekt) in der Objektgalerie erscheinen jetzt unter fischertechnik auch die Projekte ftCompting30 und poorFish30, da ist dann eins zu wählen.Nach der Auswahl muß noch das Zielverzeichnis angegeben werden in das es gespeichert werden soll. Gespeichert wird unter den alten Namen, bei Bedarf also gleich mit Speichern unter ... Umbenennen.

Externe Units

Bei den bisherigen Projekten wurde der Klassenmodul FishFa30.PAS genutzt, man kann es sich auch leichter machen und auf die kompilierte Version von FishFa30.PAS : FishFa30.DCU zugreifen. FishFa30.DCU muß dazu in einem BibliotheksPfad (Menü | Tools | Umgebungsoptionen) liegen. FishFa30.PAS/DCU also z.B. nach C:\Programme\Borland\Delphi4\Imports verschieben. Die Angabe in **uses** ist weiterhin erforderlich. Da die Source FishFa30.PAS vorliegt, kann man sie natürlich auch selber ändern.

Die Modellfunktionen

Wenn das Template ftComputing30 genutzt wird (und das ist bei dem mitgelieferten Programm in {app}\Modelle30\Delphi4\HandTrockner der Fall) gibt es da eine **`procedure`** `TfrmMain.Action();` :

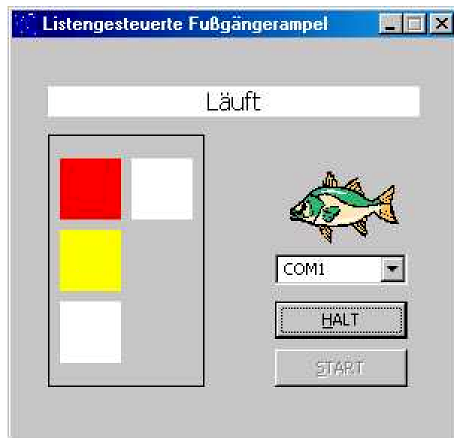
```
procedure TfrmMain.Action();
const mVentilator = 1; mLampe = 2; ePhototransistor = 1;
begin
  ft.SetMotor(mLampe, ftiEin);
  ft.WaitForTime(1000);
  repeat
    if not ft.GetInput(ePhototransistor) then begin
      ft.SetMotor(mVentilator, ftiEin);
      lblStatus.Caption := '--- trocknet ---';
      tmrAction.Enabled := true;
      ft.WaitForTime(5000);
      ft.SetMotor(mVentilator, ftiAus);
      lblStatus.Caption := '--- bereit ---';
      tmrAction.Enabled := false;
      imgFish.Top := 24;
      end;
  until ft.Finish;
end;
```

Viel passiert da nicht mehr : Die `repeat until` Schleife ist bekannt. Vorher wird noch die Lampe für die Lichtschranke angewärmt. In der Schleife gibt es nur ein `if then` in der abgefragt wird, ob die Lichtschranke offen ist (`not ft.GetInput(ePhototransistor)`), dann wird der Ventilator angeworfen, die Nachricht "trocknet" ausgegeben und der Fisch angestellt und 5 Sekunden gewartet. Anschließend das Ganze rückwärts. Wenn man den Fisch in Schwung halten will braucht man auch noch einen Timer :

```
procedure TfrmMain.tmrActionTimer(Sender: TObject);
begin
  if imgFish.Top = 24 then imgFish.Top := 136
                      else imgFish.Top := 24;
end;
```

Die Lage des Image-Controls wird hier wechselnd verändert. Beim Testen sollte man den Fisch aber lieber abstellen `tmrAction.Enabled = false;`, er stört da nur. Aber bei einer Demo muß man schon sichtbar machen, das das Programm läuft, bei dem Krach, den der Ventilator macht --- .

FussAmpel : Über den Umgang mit SetMotors und Listen



Mal wieder eine Ampel, diesmal als Fußgängerampel mit vier Lampen an den M-Ausgängen und einem Taster an E1.

Bei gleichzeitiger Steuerung mehrerer Motoren oder Lampen über SetMotor ist ein Folge vom Befehlen zum Ein- und Aussschalten erforderlich. Mit der Methode SetMotor**s** können alle M-Ausgänge auf einmal geschaltet werden. Und wenn man dann noch die einzelnen AmpelTakte in eine Tabelle stellt, kommt miteinmal dabei ein richtig interessantes Programm heraus :

SetMotors

Die Methode SetMotors hat in seiner einfachsten Form nur den Parameter MotorStatus in dem alle M-Ausgänge (M1 – M4, bei angeschlossenem Slave auch M1 – M8) geschaltet werden können. Dazu wird der Parameter Direction von SetMotor für jeden M-Ausgang als 2bitWert (ftiLinks = 01, ftiRechts = 10, ftiEin = 01, ftiAus = 00) in MotorStatus abgestellt.

MotorStatus ist eine Dezimalzahl mit Vorzeichen und hat eine Länge von 32bit. Im Rechner wird sie aber nur als bitFolge gesehen. Im Anwendungsprogramm kann man sie wahlweise als Dezimalzahl, als Hexa- oder Binärwert verwenden. Um die Direction-Wert Anordnung zu sehen, wird erstmal die Binärdarstellung genutzt

M8	M7	M6	M5	M4	M3	M2	M1

ganz rechts sind die bits 0-1. M1 ftiLinks, Rest ftiAus heißt dann 0000000000000001 oder schlicht dezimal 1. M2 hat seine Position 2 bit weiter links ftiLinks hieße da 0000000000000100 oder dezimal 4, M1 und M2 ftiLinks : 0000000000000101 oder dezimal 4 + 1 = 5. M3 ist dann 0000000000010000 oder dezimal 16, das ist Hexa 10 (&H10&).

Für die Lampen der Ampel werden jetzt solche (Hexa) Konstanten angelegt :

```
const lGruen = $1; lGelb = $4; lRot = $10&; lFuss = $40;
```

entsprechend der Belegung der M-Ausgänge am Interface.

Ein `ft.SetMotors(lRot + lFuss);` schaltet dann die rote Autoampel und die grüne Fußgängerampel ein, alle anderen Ampeln (M-Ausgänge) werden ausgeschaltet (auf den entsprechenden Positionen stehen ja Nullen).

Die FussTabelle

Man könnte jetzt eine Folge von ft.SetMotors schreiben, die die Fußgängerphase bilden, etwa so:

```
ft.SetMotors(lGruen);
ft.SetMotors(lGelb);
ft.SetMotors(lRot);
ft.SetMotors(lRot + lFuss);
ft.SetMotors(lRot);
ft.SetMotors(lRot + lGelb);
```

und dazwischen noch jeweils eine `ft.Pause`. Das ist schon eine rechte Vereinfachung gegenüber dem Einsatz von ft.SetMotor. Eleganter wird es noch, wenn diese Werte in eine

Tabelle gestellt werden, dann kann man sie in einer einfachen Schleife abarbeiten (und könnte die Tabelle auch flexibel aus einer Datei lesen).

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
// --- Besetzen der FussTabelle --------------
  FussTabelle[1].AmpelStatus := lGruen;
  FussTabelle[1].Dauer       := 1000;
  FussTabelle[2].AmpelStatus := lGelb;
  FussTabelle[2].Dauer       := 500;
  FussTabelle[3].AmpelStatus := lRot;
  FussTabelle[3].Dauer       := 500;
  FussTabelle[4].AmpelStatus := lRot + lFuss;
  FussTabelle[4].Dauer       := 2000;
  FussTabelle[5].AmpelStatus := lRot;
  FussTabelle[5].Dauer       := 500;
  FussTabelle[6].AmpelStatus := lRot + lGelb;
  FussTabelle[6].Dauer       := 500;
end;
```

Die FussTabelle enthält als Elemente eine Struktur (record) mit den Werten für einen Ampeltakt.

```
type
  FussSatz = record
    AmpelStatus: LongInt;
    Dauer:       LongInt;
  end;
```

Die Tabelle selber ist im interface Teil des Programm deklariert worden.

FussPhase

Die Schleife für die Abarbeitung der Takte einer Fußgängerphase befindet sich in

```
procedure TfrmMain.FussPhase;
var i: Cardinal;
begin
  for i := 1 to 6 do begin
    ft.SetMotors(FussTabelle[i].AmpelStatus);
    Anzeige(FussTabelle[i].AmpelStatus);
    ft.Pause(FussTabelle[i].Dauer);
  end;
end;
```

Da gibt es dann den (einzigen) `ft.SetMotors` Aufruf und eine `ft.Pause` in einer Schleife.

Damit es nicht zu einfach wird :

Anzeige

Eine Anzeige des aktuellen Ampeltaktes in entsprechenden Labels auf der Form :

```
procedure TfrmMain.Anzeige(AmpelStatus: Integer);
begin
  if (AmpelStatus and lGruen) > 0 then lblGruen.Color := clLime
                                   else lblGruen.Color := clWhite;
  if (AmpelStatus and lGelb)  > 0 then lblGelb.Color  := clYellow
                                   else lblGelb.Color  := clWhite;
  if (AmpelStatus and lRot)   > 0 then lblRot.Color   := clRed
                                   else lblRot.Color   := clWhite;
  if (AmpelStatus and lFuss)  > 0 then lblFuss.Color  := clLime
                                   else lblFuss.Color  := clWhite;
end;
```

Hier wird der AmpelStatus wieder auseinandergenommen und dann auf die einzelnen Label farblich verteilt.

AmpelStatus And lGruen ist ein bitweises Verarbeiten einer Variablen (AmpelStatus) und einer Maske (lGruen), wobei im Ergebnis alle bits stehen bleiben, die sowohl in Variable wie auch Maske eine Endsprechung haben :

```
0000000001010101   ' hier wären alle M-Ausgänge an
and
0000000000000001
=
0000000000000001
```

Abgefragt wird das Ergebnis auf größer Null, weil nur interessiert, ob lGruen an ist. Es gibt übrigens auch noch or und xor ... einfach auf F1-Hilfe drücken um mehr zu erfahren.

Action

Für die Hauptroutine Action bleibt dann nicht mehr viel über :

```
procedure TfrmMain.Action();
begin
  repeat
    ft.SetMotors(lGruen);
    Anzeige(lGruen);
    if ft.GetInput(eFussWunsch) then FussPhase;
    ft.Pause(111);
  until ft.Finish;
end;
```

Eine Endlosschleife in der auf Fußgänger und deren Wünsche gewartet wird. In der Schleife wird erstmal die Autoseite auf Grün geschaltet (und nur die : ftSetMotor**s**), das wird dann angezeigt, danach die Abfrage des FussWunsch-Taster mit dem Aufruf der FussPhase, genau alle 111 MilliSekunden.

Industry Robots

Einige Beispielprogramme, die – aufeinander aufbauend – den Einsatz von FishFace bei den Industry Robots zeigen sollen.

RobStep : Steuerung eines Robots durch Einzelschritte

Bändigen eines ganz normalen Industry Robots (Säulen- oder Knickarm-Robot). Aufbau nach Handbuch.

Das Programm bringt, neben dem bekannten Rahmen des Templates ftComputing30, eine Reihe von Buttons, über die die Robotfunktionen abgerufen werden können, am besten von unten nach oben :

- Anfahren der Home-Position (muß die erste Funktion sein).
- Fahren zur Position A (dem Magazin)
- Greifen eines Teils
- Fahren zur Position B (der Ablage)
- und Ablegen
- HALT/ENDE hat noch eine besondere Parkfunktion, das Anfahren einer Mittelstellung in der man den Robot gut wegstellen kann.

Die ausgeführten Funktionen greifen ins "Leere", d.h. es wurde keine festinstalliertes Magazin bzw. eine besondere Ablage vorgesehen. Hier ist Phantasie und Basteltalent gefragt

Die Sub Home

Zur Positionierung des Robots sind Bezugspunkte erforderlich, das ist in diesem Fall die Position an den Endtastern, die durch ein Fahren in Richtung ftiLinks erreicht werden. Von hier werden dann die weiteren Positionen bestimmt.

```
procedure TfrmMain.Home;
begin
  lblStatus.Caption := 'Home';
  ft.SetMotor(mSaule,   ftiLinks, ftiFull, 999);
  ft.SetMotor(mArmV,    ftiLinks, ftiFull, 999);
  ft.SetMotor(mArmH,    ftiLinks, ftiFull, 999);
  ft.SetMotor(mGreifer, ftiLinks, ftiFull, 999);
  ft.WaitForMotors(0, mSaule, mArmV, mArmH, mGreifer);
  ft.SetMotor(mSaule, ftiRechts, ftiFull, 10);
  ft.WaitForMotors(0, mSaule);
end;
```

Mit `ft.SetMotor` werden alle Motoren in Richtung `ftiLinks` mit voller Geschwindigkeit um 999 Impulse gestartet. Die 999 Impulse entsprechen etwa drei Umdrehungen der Säule, werden also nie erreicht. Erreicht wird aber der Endtaster, der ebenfalls den `ft.SetMotor` beendet. Die Motoren fahren alle gleichzeitig (simultan) bis sie ihre vorgegebene Position erreicht haben, hier ist es eigentlich eine Ersatzposition : die jeweiligen Endtaster.

Mit `ft.WaitForMotors` wird (= 0 endlos) auf das Erreichen der Home-Position gewartet. Anschließend wird die Position B angefahren (Säule 10 Schritte nach rechts).

Relative Positionierung

Die Methode ft.SetMotor erwartet bei dem Parameter Counter eine Angabe um wieviele Impulse nach ftiRechts oder ftiLinks (Angabe im Parameter Direction) zu fahren ist, immer bezogen auf die aktuelle Position. Es ist hier also eine Buchführung erforderlich mit der die absolute Position (gerechnet ab Endtaster) nachgehalten wird. In diesem Kapitel geschieht das mit Papier und Bleistift, das ändert sich im nächsten.

```
procedure TfrmMain.NachA;
begin
  lblStatus.Caption := 'Nach A';
  ft.SetMotor(mSaule, ftiRechts, ftiFull, 100);
  ft.WaitForMotors(0, mSaule);
  ft.SetMotor(mArmV, ftiRechts, ftiFull, 45);
  ft.SetMotor(mArmH, ftiRechts, ftiFull, 80);
  ft.WaitForMotors(0, mArmV, mArmH);
end;
```

Nach dem Klick auf Home steht die Säule auf Position 10, die anderen Komponenten auf Position 0 (Arm hinten/oben, Zange offen). Hier fährt die Säule jetzt um 100 Schritte nach `ftiRechts` auf Position 110, es wird endlos (Parameter 0) auf Erreichen der Position gewartet.Anschließend fährt der Arm um 45 Schritte nach `ftiRechts` auf Positition 45 und um 80 Schritte `ftiRechts` nach vorn auf Position 80. Anschließen wird auf das Erreichen der Positionen gewartet. Man könnte alle Befehle in einem Wait zusammen "abwarten". Aber manchmal stößt man mit zuviel Simultanität die zur greifenden Teile vom Sockel.

Das Greifen in procedure Greifen ist dann auch nur noch eine reines Zupacken (Schließen des Greifers um 24 Impulse).

Dann geht’s zur Ablage (procedure NachB), auch wieder erst heben und dann `ftiFull` nach `ftiLinks` wieder um 100 Schritte auf Position 10.

Das (procedure) Ablegen ist dann auch nur ein Greifer aufreißen `ft.SetMotor( mGreifer, ftiLinks, ftiFull, 999)`; bis zum Anschlag.

Der Programmrahmen

Das Programm basiert auf dem Template ftComputing30.DPR, ist also bekannt. Die Sub Action dient hier als Funktionsverteiler :

```
procedure TfrmMain.Action();
begin
  repeat
    case (NrAction) of
    1: begin
         NrAction := 0;
         Home;
       end;
    2: begin
         NrAction := 0;
         NachA;
       end;
    3: begin
         NrAction := 0;
         Greifen;
       end;
    4: begin
         NrAction := 0;
         NachB;
```

```
        end;
    5: begin
         NrAction := 0;
         Ablegen;
       end;
    else ft.Pause(111);
    end;
  until ft.Finish(0);
end;
```

Gesetzt wird die FunktionsNummer (NrAction) durch den entsprechenden Button. Hier wird sie gleich wieder gelöscht, um einen erneuten Aufruf durch die Schleife zu verhindern. Das Sperren der Buttons könnte noch verbessert werden.

Sub cmdAction : Das Ende

Der Programmrahmen des Templates wurde beibehalten, deswegen auch die Konstruktion mit der Schleife in Sub Action. Nach dem Action-Aufruf wurde aber noch ein `ft.NotHalt := false` und der Aufruf von `procedure Parken` hinzugefügt.

Bei Klick auf den HALT-Button wird `ft.NotHalt` auf `true` gesetzt, das beendet die Schleife in Action bei der nächsten Auswertung von `ft.Finish`, denn `ft.Finish` meldet bei `ft.NotHalt = true` seinerseits true. Damit ist Action beendet. ftNotHalt bleibt aber weiterhin true.

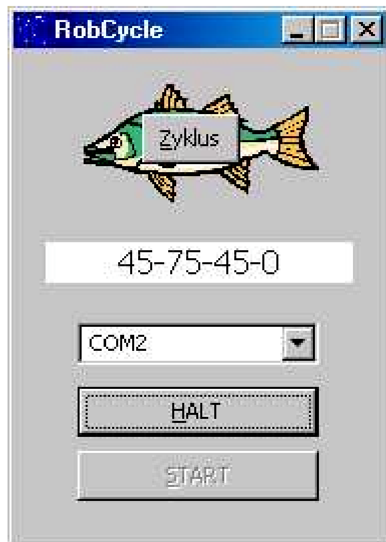
Außer Finish werten noch alle Wait-Befehle (einschl. Pause) NotHalt und ebenso die ESC-Taste aus und beenden sich im positiven Fall schlagartig. Das Modell kommt so recht schnell zum Stand.

Soll nun nach einem solchen NotHalt nochmal mit Wait-Befehlen gearbeitet werden (`procedure Parken` tut das), so ist ft.NotHalt wieder auf false zu setzen (beim START tut das `ft.OpenInterface`).

```
procedure TfrmMain.Parken;
begin
  Home;
  lblStatus.Caption := 'Parken';
  ft.SetMotor(mSaule, ftiRechts, ftiFull, 100);
  ft.WaitForMotors(0, mSaule);
end;
```

`Parken` selber fährt erstmal auf Home-Position, um eine gesichterte Ausgangslage zu schaffen (der HALT-Button ist auch während der Funktionsabläufe aktiv) und von da aus den die Säule auf eine platzsparende Mittelposition, reif für die Ablage. Bei den ersten Tests sollte man Parken lieber auskommentieren, es kann auch zuviel des Guten sein.

RobCycle : Robotsteuerung durch Abruf eines Funktionszyklus



Eine Ein-Button-Lösung zu wiederholten Aufruf eines Funktionszyklus.

Das Programm hat einen leicht modifizierten Rahmen auf Basis von Template ftComputing30 und zeigt, wie man sichs mit der absoluten Positionierung leichter machen kann.

Außerdem ein kleiner Exkurs zum Umgang mit Klassen und deren Instanzen und die Vererbung.

Hinzu kommt die Anzeige der aktuellen Position in einer Event-Routine.

Die absolute Positionierung

Ist schon etwas vertrackter als die von SetMotor freiHaus gelieferte relative Positionierung. Hier mit MoveTo eine zentrale Routine dafür :

```
procedure TFishRobot.MoveTo(P1, P2, P3, P4: LongInt);
begin
// --- Starten aller Motoren, wenn nicht nullPos ---
  Starten(ftiM1, P1);
  Starten(ftiM2, P2);
  Starten(ftiM3, P3);
  Starten(ftiM4, P4);

// --- Warten auf Ready aller Motoren und laufende Positionsangabe
  while WaitForMotors(8 * PollInterval, ftiM1, ftiM2, ftiM3, ftiM4)
                     = ftiTime do
    if Assigned(FOnFishPosition) then FOnFishPosition(Self,
        WoIst(ftiM1, P1), WoIst(ftiM2, P2),
        WoIst(ftiM3, P3), WoIst(ftiM4, P4));

// --- Aktualisieren PosTab -------------------------------
  if P1 <> nullPos then if P1 > PosTab[1].aktPosition
                 then PosTab[1].aktPosition := P1 + GetCounter(2)
                 else PosTab[1].aktPosition := P1 - GetCounter(2);
  ..........
  if P4 <> nullPos then if P4 > PosTab[4].aktPosition
                 then PosTab[4].aktPosition := P4 + GetCounter(8)
                 else PosTab[4].aktPosition := P4 - GetCounter(8);

// --- Abschließende Positionsangabe ---------------------
  if Assigned(FOnFishPosition) then FOnFishPosition(Self,
                                     PosTab[1].aktPosition,
                                     PosTab[2].aktPosition,
                                     PosTab[3].aktPosition,
                                     PosTab[4].aktPosition);
```

```
end;
```

MoveTo hat bis zu vier Parameter mit den (absoluten, ab Endtaster) Positionsangaben für die Motoren M1 – M4 (das Extension Module erfordert dann einen entsprechenden Ausbau), die letzten, nicht benötigten können weglassen werden. Werden vordere Motoren nicht verfahren, so können sie durch `nullPos` gekennzeichnet werden.

Ablauf : Starten aller beteiligter Motoren über `procedure Starten` und Warten auf das Ready aller Motoren. Diesmal aber mit Unterbrechungen (`8*PollInterval`) um sich auch noch um die aktuelle Position kümmern zu können, sie wird über `FOnFishPosition` weitergegeben.

Nach dem **Warten** das Aufräumen (hier fehlt etwas vom Code) : Feststellen der Abschlußpostition in einer PosTab. Dabei wird auch ein eventueller zusätzlicher Impuls mitgezählt. Für die Fälle normales Ende (ftiEnde) und Abbruch (ftiESC, ftiNotHalt). Das wird dann auch noch mal über `FOnFishPosition` nach außen gegeben.

PosTab ist eine Tabelle, die für jeden Motor einen Eintrag `aktPosition` und `maxPosition` enthält (abs. Positionen, Deklaration über einen `record` zu Anfang der Source). Starten rechnet nun die abs. Positionen in die relativen um und starten dann den jeweiligen Motor :

```
procedure TFishRobot.Starten(MotNr, zielPos: LongInt);
var ct: LongInt;
begin
  if zielPos <> nullPos then begin
    if zielPos > PosTab[MotNr].maxPosition then
                              zielPos := PosTab[MotNr].maxPosition;
    if zielPos > PosTab[MotNr].aktPosition then
      SetMotor(MotNr, ftiRechts, ftiFull,
                                 zielPos-PosTab[MotNr].aktPosition)
    else begin
      ct := PosTab[MotNr].aktPosition - zielPos;
      if ct > 0 then SetMotor(MotNr, ftiLinks, ftiFull, ct);
      end;
  end;
end;
```

Start nur wenn nötig : `if zielPos <> nullPos`.

Korrektur der `zielPos` auf `maxPosition`, wenn sie zu groß ist.

Bei Rechtslauf (`zielPos > aktPosition` : SetMotor mit `zielPos – aktPosition`

Sonst : gar nicht, wenn die Differenz zur aktPosition <= 0 ist
bei >0 SetMotor mit `aktPosition – zielPos` (also linksrum)

Und dann gibt es noch WoIst :

```
function TFishRobot.WoIst(i, Pos: Integer): LongInt;
var j: LongInt;
begin
  j := (i-1) * 2 + 2;
  if Pos = nullPos then Result := PosTab[i].aktPosition
  else if Pos > PosTab[i].aktPosition
                                then Result := Pos - GetCounter(j)
                                else Result := Pos + GetCounter(j);
end;
```

Da die `aktPosiition` erst am Ende von MoveTo upgedated wird, muß die derzeitige Position aus der gar nicht so aktuellen aktPosition und dem Counter bestimmt werden, den umFish30.DLL aktuell über GetCounter(j) liefert.

Die Klasse TFishRobot in der Unit FishRob30.PAS

Wo bringt man eine so schöne Routine nun unter, einfach so in der Form? Da sich ohnehin schon ein paar Routinen angesammelt haben und noch ein paar dazu kommen, tauft man die Routinen jetzt Methoden und bringt sie in einer Klasse in einer separaten Unit unter. Damit hat man eine abgeschlossene (gekapselte) Einheit mit allem was dazu gehört und kann es (vielleicht?) im nächsten Projekt wieder verwenden. Man könnte die Routinen natürlich auch in der Unit FishFa30 unterbringen. Aber eine separate Lösung ist für den ersten Versuch übersichtlicher.

```
unit FishRob30;
interface
uses
  Windows, Messages, SysUtils, Classes, FishFa30;
const
  nullPos = 999;
type
  PosSatz = record
    maxPosition: LongInt;
    aktPosition: LongInt;
  end;
  TFishRobot = class;
  TFishPosition = procedure(Sender: TFishRobot; Pos1, Pos2,
                                 Pos3, Pos4: LongInt) of object;
  TFishRobot = class(TFishFace)
  private
    FOnFishPosition: TFishPosition;
    function  GetmaxPosition(MotorNr: LongInt): LongInt;
    procedure SetmaxPosition(MotorNr, Wert: LongInt);
    procedure Starten(MotNr, zielPos: LongInt);
    function  WoIst(i, Pos: LongInt): LongInt;
    function  GetaktPosition(MotorNr: LongInt): LongInt;
    procedure SetaktPosition(MotorNr, Wert: LongInt);
  public
    property OnFishPosition: TFishPosition read  FOnFishPosition
                                           write FOnFishPosition;
    property aktPosition[MotorNr: LongInt]: LongInt
                                      read  GetaktPosition
                                      write SetaktPosition;
    property maxPosition[MotorNr: LongInt]: LongInt
                                      read  GetmaxPosition
                                      write SetmaxPosition;
    procedure MoveHome;
    procedure MoveTo(P1: LongInt=nullPos; P2: LongInt=nullPos;
                     P3: LongInt=nullPos; P4: LongInt=nullPos);
    constructor Create(a,b: Boolean);
  end;
implementation
var
  PosTab: array[1..8] of PosSatz;
......... .......
constructor TFishRobot.Create;
begin
  inherited Create(a, b);
  PosTab[1].maxPosition := 180;
  PosTab[2].maxPosition := 100;
  PosTab[3].maxPosition := 80;
  PosTab[4].maxPosition := 24;
end;
.....................
procedure TFishRobot.MoveTo(P1, P2, P3, P4: LongInt);
```

```
begin
   ................
end;

procedure TFishRobot.Starten(MotNr, zielPos: Integer);
begin
   ............
end;

function TFishRobot.WoIst(i, Pos: Integer): LongInt;
begin
   ...
end;

end.
```

Die Unit FishRob30.PAS mit der Klasse TFishRobot ist Bestandteil des Projektes (Anlegen, wie gewohnt : Neu | Unit);

Die Klasse TFishRobot wird von TFishFace abgeleitet : TFishRobot = class(TFishFace), d.h. TFishRobot erbt von TFishFace alle Eigenschaften und Methoden, hinzukommen die eigenen Eigenschaften und Methoden. In RobCMain muß dann bei uses ein entsprechender Verweis aufgenommen werden. Der Verweis auf FishFa30 muß erhalten bleiben, da die Klasse EFishFace der Unit benötigt wird. Das war hier die ganze Veerbung.

PosTab enthält die aktPosition und die maxPosition (record PosSatz) für alle Motoren.

Die Maximalpositionen des Robots in PosTab können durch die Eigenschaft maxPosition gesetzt werden, brauchen sie aber nicht, da rein zufällig im constructor die Werte schon beim Anlegen der Klasse (Instanzieren) passend für den Säulenrobot gesetzt werden. Und sonst gibt es nur noch alte Bekannte (OnFishPosition kommt später).

Jetzt muß die Klasse nur noch genutzt werden. D.h. es muß eine Instanz der Klasse erzeugt werden :

RobCycle : Der Programmrahmen

Ganz oben (im interface-Teil) :

```
uses .... FishFa30, FishRob30;
```

FishFa30 ist nur noch für die Nutzung von EFishFace im try except von cmdActionClick erforderlich, sonst läuft alles über FishRob30. D.h. zu Beginn des implementation-Teils steht jetzt :

```
var ft: TFishRobot;
```

Bei cmdActionClick und cmdEndeClick wurde ein wenig umverteilt, da bei Action die repeat until-Schleife entfallen ist. Action enthält dann nun endlich den namensgebenden Zyklus :

procedure Action : Der Zyklus

```
procedure TfrmMain.Action();
begin
  ft.MoveTo(145);
  ft.MoveTo(nullPos, 45, 45);
  ft.MoveTo(nullPos, nullPos, nullPos, 24);
  ft.MoveTo(nullPos, 0);
  ft.MoveTo(45);
  ft.MoveTo(nullPos, 75);
  ft.MoveTo(nullPos, nullPos, nullPos, 0);
end;
```

Immer schön der Reihe nach :

- Zum Magazin auf Position 145 (der Arm wird nicht verfahren)
- Greifen : dazu erstmal Fahren des Arms auf ftiNull (Säule bleibt auf Postion), 45, 45
  Schließen der Greifers,
  Einziehen des Arms auf Position 0
- Zur Ablage auf Säulenposition 45
- Zum Ablegen Arm wieder ausfahren auf Positon 75
  und Greifer aufmachen.

Ein Zyklus wird mit Button **Zyklus** gestartet.

Das Ereignis OnFishPosition

```
procedure TfrmMain.OnFishPosition(Sender: TFishRobot;
                                  Pos1, Pos2, Pos3, Pos4; LongInt);
begin
  lblStatus.Caption := IntToStr(Pos1) + '-' + IntToStr(Pos2) + '-' +
                       IntToStr(Pos3) + '-' + IntToStr(Pos4);
end;
```

Eine Ereignisroutine die die jeweils aktuelle Position des Robots im Label lblStatus anzeigt. Deklariert wurde sie, wie üblich, bei der Klassen Definition von TfrmMain (der Form). Die Zuweisung zur aktuellen Instanz von TFishRobot erfolgt in cmdActionClick :

```
ft.OnFishPosition := OnFishPosition;
```

Und jetzt nochmal zurück zu FishRob30. Dort wird bei type am Anfang ein Type

```
TFishPosition = procedure(Sender: TFishRobot; Pos1, Pos2,
                          Pos3, Pos4: LongInt) of object;
```

definiert, der die Ereignisroutine beschreibt.

Im private-Teil der Definition von TFishRobot wird dann die erforderliche Ereignis-Variable `FOnFishPosition: TFishPosition`; definiert und im public-Teil dann endlich die property für das Ereignis

```
property OnFishpostion: TFishPosition read  FOnFishPosition
                                      write FOnFishPosition;
```

Die Methode MoveTo nutzt das Ereignis (wenn denn in der Anwendung seinerseits eine Ereignisroutine zugewiesen wurde) :

```
if Assigned(FOnFishPosition) then  FOnFishPosition(Self,
                                     WoIst(ftiM1, P1), .......);
```

für die laufende Postionsangabe und

```
if Assigned(FOnFishPositon)  then FOnFishPosition(Self,
                                   PosTab[1].aktPosition, ....);
```

für die abschließende Position.

Referenz

Allgemeines

Verwendete Parameterbezeichnungen

In der Referenz werden für Parameter und Returnwerte besondere Bezeichnungen verwendet um deren Bedeutung zu charakterisieren. Sie stehen gleichzeitig für einen Variablentyp bzw. alternativ eine Enum.

AnalogNr	Nummer eines Analog-Einganges (LongInt 0-1, ftiNr)
AnalogWert	Rückgabewert beim Auslesen von EX/EY (0-1024)
Counter	Wert eines ImpulsCounters (LongInt)
Direction	Schaltzustand eines M-Ausganges (LongInt 0-2, ftiDir)
InputNr	Nummer eines E-Einganges (LongInt 0-8(16), ftiNr)
InputStatus	Rückgabewert beim Auslesen aller E-Eingänge (0 - $FFFF)
LampNr	Nummer eines "halben"-M-Ausganges (LongInt 0-8(16), ftiNr)
ModeStatus	Status der Betriebsmodi aller M-Ausgänge(LongInt). Jeweils 4 bit pro Ausgang. Begonnen bei 0-3 für M1 (0000 normal, 0001 RobMode).
MotorNr	Nummer eines M-Ausganges (LongInt 0-4(8), ftiNr)
MotorStatus	SollStatus aller M-Ausgänge (LongInt). Jeweils 2 bit pro Ausgang Begonnen bei 0-1 für M1 (00 = Aus, 01 = Links, 10 = Rechts)
mSek	Zeitangabe in MilliSekunden (LongInt)
NrOfchanges	Anzahl Impulse (LongInt)
OnOff	Ein/Ausschalten eines M-Ausganges (Boolean, ftiDir)
PortName	Name des wählbaren Anschlußports (String) ("LPT", "COM1" – "COM8", "LPT1" – "LPT3")
Position	Position in Impulsen ab Endtaster (LongInt)
Speed	Geschwindigkeitsstufe mit der ein M-Ausgang (Motor) betrieben werden soll (LongInt 0-15, ftiSpeed)
SpeedStatus	Status der Geschwindigkeiten aller M-Ausgänge (LongInt) Jeweils 4 bit pro Ausgang. Begonnen bei 0-3 für M1 Werte 0000 (stop) – 1111 (full)
TermInputNr	Nummer eines E-Einganges mit der die (Wait)Methode beendet werden soll (LongInt, ftiNr)
Value	Allgemeiner LongInt Wert
WaitWert	Rückgabewert von WaitForMotors (LongInt, ftiWait)

Die Aufrufparameter werden ByVal (Ausnahme WaitForPosition, Parameter Counter) übergeben.

Eine Reihe von Parametern sind optional, sie werden dann meist im Sinne einer Funktions-Überladung (Overload). Das wird dann bei der betreffenden Methode besondersbeschrieben.

Konstanten

fti...	zur symbolischen Kennzeichnung von Parametern. Sie sind in der Beschreibung zu Gruppen zusammengefaßt :
ftiDir	Angabe des Schaltzustandes (Drehrichtung, Ein/Aus)
ftiNr	Angabe der Nummer eines Ein- bzw. Ausganges
ftiSpeed	Angabe einer Geschwindigkeitsstufe
ftiWait	Returnwerte der Methode WaitForMotors

Es können auch einfache numerische Werte oder eigene Konstanten benutzt werden.

Eigenschaften

AnalogScan

Angabe ob auch die Analog-Eingänge gescannt werden sollen.

read, Boolean, Default = false

AnalogsEX

Lesen eines EX-Wertes. Im Gegensatz zur Methode GetAnalog erfolgt hier nur ein Zugriff auf den aktuellen Wert, keine weiteren Aktionen wie DoEvents.

read, LongInt | AnalogWert

AnalogsEY

Lesen eines EY-Wertes. Sonst siehe AnalogsEX

read, LongInt | AnalogWert

Inputs

Lesen der Werte aller E-Eingänge

read, LongInt | InputStatus

LPTAnalog

Lesen AnalogSkalierung. Begrenzung des Analogwertes nach oben und damit der für die Messung erforderlichen Zeit (nur LPT-Interfaces)

read, LongInt, Default = 5

LPTDelay

Lesen der Ausgabeverzögerung. Bei LPT-Interface auf schnellen Rechnern erforderlich.

read, LongInt, Default = 10

NotHalt

Anmelden eines Abbruchwunsches, Auswertung durch die Wait-Methoden und Finish

read | write, Boolean, Default = false

Outputs

Lesen Werte aller M-Ausgänge. Siehe auch Bemerkung AnalogsEX

read, LongInt | MotorStatus

PollInterval

Lesen des Intervals in dem der Status des Interfaces abgefragt (gepollt) und aufgefrischt (refreshed) wird.

read, LongInt | mSek

Slave

Lesen ob ein Slave (Extension Module) bearbeitet wird.

read, Boolean

Version

Lesen der Version von FishFa30.PAS/DCU

read, String

Methoden

Allgemeines

Lampen am Interface

Hinweis : Beispiele siehe auch "FisFa30.DPR : Das Beispielprojekt"

ft

Bezeichnung für die zugehörende Klasseninstanz

Application.ProcessMessages

Herstellen der Unterbrechbarkeit durch den Befehl Application.ProcessMessages. Bei Einsatz von engen Schleifen, z.B. Abfrage E-Eingang, kann die Bedieneroberfläche blockieren (Button-Click wird nicht ausgeführt, Label werden nicht aktualisiert). Um das zu verhindern wurde in die Mehrzahl der Methoden der Delphi-Befehl ProcessMessages eingebaut.

Wird bei den betroffenen Methoden extra angegeben.

Abbrechbar

Länger laufende Methoden (Wait...) können von außen durch Setzen der Eigenschaft NotHalt = true oder durch Drücken der Esc-Taste abgebrochen werden.

Wird bei den betroffenen Methoden besonders angegeben.

raise

Werden Ausnahmebedingungen festgestellt, wird ein entsprechendes Ereignis ausgelöst (raise) :

Es wird jeweils ein String nach dem Muster "fehlertext.verursachendeMethode" bereitgestellt. z.B. InterfaceProblem.OpenInterface.

Beispiel

```
try
   ft.SetMotor(ftiM1, ftiLinks);
   ft.WaitForChange(ftiE2, 100);
   ft.SetMotor(ftiM1, ftiAus);
   ....
except
  on e: EFishFace do lblStatus.Caption := e.Message;
end;
```

Motor an M-Ausgang M1 wird linksdrehend gestartet, es wird auf 100 Impulse an E-Eingang E2 gewartet und dann der Motor wieder abgeschaltet. Tritt in dieser Zeit eine Ausnahme auf (z.B. Ausfall der Spannungsversorgung), wird das in lblStatus angezeigt.

NotHalt

Die Eigenschaft NotHalt (siehe auch "Abbrechbar") wird intern genutzt um langlaufende Funktionen ggf. Abzubrechen. NotHalt wird von OpenInterface auf false gesetzt. Es kann im Programm (z.B. über einen HALT-Button) genutzt werden um den Programmlauf abzubrechen oder auch eine Endlosschleife (z.B. repeat .. until ft.Finish;) zu beenden. Soll das Programm anschließend weiterlaufen, so ist NotHalt wieder auf false zu setzen.

Parameter

Siehe auch "Verwendete Parameterbezeichnungen"

Die Methoden sind meist procedures, bei functions wird der Ergebniswert vorangestellt.

Als Parameter werden in der Regel LongInt-Werte angegeben Davon abweichende Typen werden extra angegeben. Für viele Zwecke gibt es vorgefertigte Konstanten (siehe dort), die zu Gruppen zusammengefaßt sind, bei der Parameterbeschreibung werden diese Gruppennamen angegeben.

Speed

Die Geschwindigkeitssteuerung beruht auf einem zyklischen Ein- und Ausschalten der betroffenen M-Ausgänge (Motoren) im Takt des PollIntervals. Dazu wird intern für jede Geschwindigkeitsstufe eine entsprechende Schaltliste vorgehalten. Die Geschwindigkeitsstufe wird durch die Parameter Speed bzw. SpeedStatus für einen bzw. alle Motoren bei der Methode SetMotor(s) bestimmt.

Counter

Zu jedem E-Eingang wird ein Counter geführt, in dem die Impulse (Umschalten von true auf false und umgekehrt) gezählt werden. Die Counter werden bei OpenInterface auf 0 gesetzt. Sie werden außerdem von einigen Methoden interngenutzt (SetMotor, WaitForxxx). Sie können mit GetCounter abgefragt und mit SetCounter / ClearCounter(s) gesetzt werden. In der Regel werden sie zur Positionsbestimmung eingesetzt.

RobMotoren

Unter RobMotoren wird eine Kombination von einem M-Ausgang und zwei E-Eingängen mit den Funktionen Endtaster / Impulstaster verstanden, die im Betrieb eine Einheit bilden. Dabei muß am M-Ausgang ein Motor angeschlossen sein und an den E-Eingängen Taster mit Schließfunktion (Kontakte 1-3). Auf der Motorwelle muß ein "Impulsrädchen" montiert sein, das den Impulstaster betätigt. Der Motor muß linksdrehend angeschlossen werden. D.h. er läuft bei ftiLinks auf den Endtaster zu. Folgende Kombinationen sind zulässig

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 RobMotoren können über die Methode
`SetMotor(MotorNr, Direction, Speed, Counter`);
betrieben werden. Die Methode erlaubt das simultane Anfahren vorgegebener Positionen mit bis zu 8 Motoren. Bei Erreichen einer Position wird der zugehörende Motor abgeschaltet. Die Methode ist asynchron. D.h.Das Erreichen der vorgegebenen Positionen wird von der Methode nicht abgewartet (der Ausführungsteil der Methode läuft in einem separatem Thread). Die Synchronität zum Programm kann durch die Methode WaitForMotors wieder hergestellt werden.

Beispiel

```
ft.SetMotor(ftiM1, ftiLeft, ftiHalf, 50);
ft.SetMotor(ftiM2, ftiRight, ftiFull, 60);
ft.WaitForMotors(0, ftiM1, ftiM2);
```

Motoren M1 und M2 werden gestartet, anschließend wird auf das Erreichen der Positionen gewartet.

Lampen am Interface

Die vier M-Ausgänge des Interfaces sind primär zum Schalten von Motoren in zwei Laufrichtungen vorgesehen. Doch bietet sich zusätzlch die Möglichkeit, Geräte, die nur ein- und ausgeschaltet werden müssen, an nur einem Pol eines M-Ausganges und Masse anzuschließen. Auf diese Weise ist es möglich z.B. acht Lampen mit einem Interface zu schalten. Hiefür gibt es die Methode SetLamp.

Sollen Lampen gemeinsam (z.B. bei einer Verkehrampel) geschaltet werden, können sie auch über die Methode SetMotors geschaltet werden. Dazu sind die einzelnen Lampenbits im MotorStatus zu setzen.

Es gibt Unterschiede zwischen den Interfaces :

Universal (paralleles) Interface : im nicht geschalteten Zustand sind die Lampen aus. M1 vorderer (gelber) Kontakt : Lampe 1, hinterer (orange) Kontakt Lampe 2 ...

Intelligent (serielles) Interface : im nicht geschalteten Zustand sind die Lampen **an**. M1 vorderer Kontakt : Lampe 1, hinterer Kontakt Lampe 2 ...

Liste der Methoden

constructor

Anlegen einer Instanz von TFishFace.

ft := TFishFace.**Create**(AnalogScan: Boolen=false; Slave: Boolean=false;
PollInterval: Cardinal=0;
LPTAnalog: LongInt=0; LPTDelay: LongInt=0);

- AnalogScan (Boolean) : Angabe, ob auch die Analogeingänge (EX / EY) gepollt werden sollen. Es ist dann ein höheres PollInterval erforderlich. Default : false
- Slave (Boolean) : Angabe ob an das primäre Interface ein weiteres angeschlossen ist. Default = false
- PollInterval (LongInt) : Angabe in mSek in welchen Intervallen auf das Interface zugegriffen werden soll. Default = 0 : Bestimmung durch die Methode OpenInterface in Abhängigkeit vom Kontext.
- LPTAnalog (LongInt) : AnalogSkalierung. Begrenzung des Analogwertes nach oben und damit der für die Messung erforderlichen Zeit (nur LPT-Interface). Default = 5
- LPTDelay (LongInt) : Ausgabeverzögerung. Bei LPT-Interface auf schnellen Rechnern erforderlich. Default = 10

ClearCounter

Löschen des angegebenen Counters (setzen auf 0)

ft**.ClearCounter(InputNr)**

Siehe auch ClearCounters, GetCounter, SetCounter

ClearCounters

Löschen aller Counter (setzen auf 0)

ft.**ClearCounters**

Siehe auch ClearCounter, GetCounter, SetCounter

ClearMotors

Abschalten aller M-Ausgänge (ftiAus)

ft.**ClearMotors**

raise KeinOpen, InterfaceProblem

Siehe auch SetMotor, SetMotors, SetLamp, Outputs

CloseInterface

Schließen der Verbindung zum Interface

ft.**CloseInterface**

Siehe auch OpenInterface

Finish

Feststellen eines Endewunsches (NotHalt, Esc-Taste, E-Eingang)

Boolean = ft.**Finish**(InputNr)

raise KeinOpen, InterfaceProblem; ProcessMessages

Siehe auch GetInput, GetInputs, Inputs

Beispiel

```
repeat
   ....
until ft.Finish(ftiE1);
```

Die repeat-Schleife wird solange durchlaufen, bis entweder ft.NotHalt = true, die Esc-Taste gedrückt oder E1 = true wurde.

GetAnalog

Feststellen eines Analogwertes (EX / EY).
Es wird der intern vorliegende Wert ausgegeben. AnalogScan beim OpenInterface ist erforderlich.

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

raise KeinOpen, InterfaceProblem; ProcessMessages

Siehe auch GetAnalogs, AnalogsEX, AnalogsEY, AnalogScan, OpenInterface

Beispiel

```
lblAnalog.Caption := ft.Analog(ftiEX);
```

Dem Label lblAnalog wird der aktuellen Wert von EX zugewiesen.

GetAnalogDirect

Feststellen eines Analogwertes (EX / EY).
Es wird direkt auf das Interface zugegegriffen. AnalogScan beim OpenInterface ist nicht erforderlich. Sinnvoll besonders beim Universal (parallelen) Interface um die sonst erforderlichen hohen Werte für das PollInterval zu vermeiden. Die M-Ausgänge sollten abgeschaltet sein, da während der Methode keine Veränderungen an E-Eingängen erkannt werden können.

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

raise KeinOpen, InterfaceProblem; ProcessMessages

Siehe auch GetAnalog, AnalogsEX, AnalogsEY, AnalogScan, OpenInterface

Beispiel

```
ft.ClearMotors;
lblAnalog.Caption := ft.AnalogDirect(ftiEX);
```

Dem Label lblAnalog wird der aktuellen Wert von EX zugewiesen, alle M-Ausgänge wurden vorher abgeschaltet.

GetCounter

Feststellen des Wertes des angegebenen Counters

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

Siehe auch SetCounter, ClearCounter, ClearCounters

Beipiel

```
lblPosTurm.Caption := ft.GetCounter(ftiE2);
```

Dem Label lblPosTurm wird der aktuelle Zählerstand der dem E-Eingang E2 zugeordnet ist, zugewiesen.

GetInput

Feststellen des Wertes des angegebenen E-Einganges

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

raise KeinOpen, InterfaceProblem; ProcessMessages

Siehe auch GetInputs, Inputs, Finish

Beispiel

```
if ft.GetInput(ftiE1) then ... else ...;
```

Wenn der E-Eingang E1 (Taster, PhotoTransistor, Reedkontakt ...) = true ist, wird der then-Zweig durchlaufen.

GetInputs

Feststellen der Werte aller E-Eingänge

InputStatus = ft.**GetInputs**()

raise KeinOpen, InterfaceProblem; ProcessMessages

Siehe auch GetInput, Inputs, Finish

Beispiel

```
var e: LongInt;
  e := ft.GetInputs
  if (e and $1) or (e and $4) then ...
```

Wenn die E-Eingänge E1 oder E3 true sind, wird der Then-Zweig ausgeführt.

OpenInterface

Herstellen der Verbindung zum Interface und Setzen/Bestimmen von dazu erforderlichen Parametern. OpenInterface muß deswegen als erste Methode aufgerufen werden.

ft.**OpenInterface**(PortName, ProcessMessages: Boolean = true)

- PortName (String) : LPT | COM1 – COM8 | LPT1 – LPT3
  Zuordnung des zuverwendeden Interfaces durch Angabe des Ports an dem es angeschlossen ist.
- ProcessMessages (optional, default = true). Aufruf von Application.ProcessMessages in den dafür vorgesehenen Methoden.

raise InterfaceProblem

Siehe auch CloseInterface, AnalogScan, LPTAnalog, LPTDelay, PollInterval, Slave

Beispiel

```
try
   ft.OpenInterface(COM1");

   ....
except
  on e: EFishFace do lablStatus.Caption := e.Message;
end;
```

Herstellen einer Verbindung zum Interface an COM1. Im Fehlerfall wird '`Interface Problem.OpenInterface`' ausgegeben.

Pause

Anhalten des Programmablaufs.

ft.**Pause** mSek

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar

Siehe auch WaitForTime

Beispiel

```
ft.SetMotor(ftiM1, ftiLinks);
ft.Pause(1000);
ft.SetMotor(ftiM1, ftiAus);
```

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

SetCounter

Setzen eines Counters

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. Siehe auch "Lampen am Interface"

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

raise KeinOpen, InterfaceProblem; ProcessMessages

Siehe auch SetMotor, SetMotors, ClearMotors

Beispiel

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

   ft.SetLamp(lGruen, ftiEin);
   ft.Pause(2000);
   ft.SetLamp(lGruen, ftiAus);
  ft.SetLamp(lGelb, ftiEin);
```

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)

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

Die Parameter ab Speed sind optional

MotorNr (ftiNr) : Nummer des zu schaltenden M-Ausganges

Direction (ftiDir) : Schaltzustand (ftiLinks, ftiRechts, ftiEin, ftiAus)

Speed (ftiSpeed) : Geschwindigkeitsstufe, Default : ftiFull

Counter (ftiNr) : Begrenzung der Einschaltzeit. Default = 0, unbegrenzt. Werte > 0 geben die Anzahl Impulse an, die der M-Ausgang eingeschaltet sein soll (Fahren eines Motors um n Impulse). Siehe auch "RobMotoren"

raise KeinOpen, InterfaceProblem; ProcessMessages; Counter (bei Parameter Counter)

Siehe auch SetMotors, ClearMotors, SetLamp, Outputs

Beispiel 1

```
ft.SetMotor(ftiM1, ftiRight, ftiFull);
ft.Pause(1000);
ft.SetMotor(ftiM1, ftiLeft, ftiHalf);
ft.Pause(1000);
ft.SetMotor(ftiM1, ftiOff);
```

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

Beispiel 2

```
ft.SetMotor(ftiM1, ftiLeft, 12, 123);
```

Der Motor am M-Ausgang M1 wird für 123 Impulse am E-Eingang E2 oder E1 = true mit Geschwindigkeitstufe 12 eingeschaltet. Das Abschalten erfolgt selbsttätig.

SetMotors

Setzen des Status aller M-Ausgänge

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

Die Parameter ab SpeedStatus sind optional

MotorStatus (LongInt) : Schaltzustand der M-Ausgänge

SpeedStatus (LongInt) : Geschwindigkeitsstufen der M-Ausgänge. Default : ftiFull

ModeStatus (LongInt) : Betriebsmodus der M-Ausgänge. Default = 0, normal
Bei ModeStatus RobMode sind vorher mit SetCounter die zugehörenden Counterstände zu setzen.

raise KeinOpen, InterfaceProblem; ProcessMessages; Counter (bei Parameter Counter)

Siehe auch ClearMotors, SetMotor, SetLamp, Outputs

Beispiel

```
ft.SetMotors($1 + $80);
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 M-Ausgänge abgeschaltet.

WaitForChange

Warten auf eine vorgegebene Anzahl von Impulsen

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

Der Parameter TermInputNr ist optional

InputNr (ftiNr) : E-Eingang an dem die Impulse gezählt werden.

NrOfChanges (LongInt) : Anzahl Impulse

TermInputNr (ftiNr) : Alternatives Ende. E-Eingang = true

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar; Counter

Intern wird Counter (InputNr) verwendet, der zu Beginn der Methode zurückgesetzt wird

Siehe auch WaitForPositionDown, WaitForPositionUp, WaitForInput, WaitForLow, WaitForHigh

Beispiel

```
ft.SetMotor ftiM1, ftiLeft);
ft.WaitForChange(ftiE2, 123, ftiE1);
ft.SetMotor(ftiM1, ftiOff);
```

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. Vergleiche mit dem Beispiel bei SetMotor. Hier wird das Programm angehalten solange der Motor läuft.

WaitForHigh

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

ft.**WaitForHigh**(InputNr)

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar

Siehe auch WaitForLow, WaitForChange, WaitForInput

Beispiel

```
ft.SetMotor(ftiM1, ftiOn);
ft.SetMotor(ftiM2, ftiLeft);
ft.WaitForHigh(ftiE1);
ft.SetMotor(ftiM2, ftiOff);
```

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.

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

OnOff (Boolean) : Endebedingung für E-Eingang InputNr, Default = true

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar

Siehe auch WaitForChange, WaitForLow, WaitForHigh

Beispiel

```
ft.SetMotor(ftiM1, ftiLeft);
ft.WaitForInput(ftiE1);
ft.SetMotor(ftiM1, ftiOff);
```

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)

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar

Siehe auch WaitForChange, WaitForInput, WaitForHigh

Beispiel

```
ft.SetMotor(ftiM1, ftiOn);
ft.SetMotor(ftiM2, ftiLeft);
ft.WaitForLow(ftiE1);
ft.SetMotor ftiM2, ftiOff
```

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 (LongInt) : Zeit in MilliSekunden. Bei Time = 0 wird endlosgewartet.

MotorNr (ftiNr) : Nummern der M-Ausgänge auf die gewartet werden soll. Es wird auf MotorStatus = ftiAus für die angegebenen M-Ausgänge gewartet. MotorNr ftiM1 – ftiM8 in beliebiger Reihenfolge. Die nicht betroffenen Motoren müssen nicht angegeben werden. Bei den Return-Werten ftiNotHalt und .ftiESC werden alle betroffen Motoren angehalten.

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar

Beispiel

```
ft.SetMotor(ftiM4, ftiLeft, ftiHalf, 50);
ft.SetMotor(ftiM3, ftiRight, ftiFull, 40);
while ft.WaitForMotors(100, ftiM4, ftiM3) = ftiTime do
  lblPos = IntToStr(ft.GetCounter(ftiE6)) + " - " +
           IntToStr(ft.GetCounter(ftiE8));
```

Der Motor am M-Ausgang M4 wird linksdrehend mit halber Geschwindigkeit für 50 Impulse gestartet, der an M3 rechtsdrehend mit voller Geschwindigkeit für 40 Impulse. Die while-Schleife wartet auf das Ende der Motoren (ft.WaitForMotors). Alle 100 MilliSekunden wird in der Schleife die aktuelle Position angezeigt (100 .... = ftiTime). Wenn die Position erreicht ist <> ftiTime, ist der Auftrag abgeschlossen, die Motoren haben sich selber beendet. Achtung hier wurde nicht auf NotHalt (ftiNotHalt) oder Esc-Taste (ftiEsc) abgefragt, es könnte also auch vor Erreichen der Zielposition abgebrochen worden sein.

WaitForPositionDown

Warten auf Erreichen einer vorgegebenen Position.

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

Ausgegangen wird von der aktuellen Position, die in Counter gespeichert ist, es werden solange Impulse von Counter abgezogen, bis der in Position angegebene Stand erreicht ist. Counter enthält zusätzlich die dann tatsächlich erreichte Position (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 müssen immer positive Werte (einschl. 0) enthalten.

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar; Counter

Siehe auch WaitForPositionUp, WaitForChange

Beispiel

```
var Zaehler: LongInt;
  Zaehler := 12;
  ft.SetMotor(ftiM1, ftiLinks);
  ft.WaitForPositionDown(ftiE2, Zaehler, 0);
  ft.SetMotor(ftiM1, ftiAus);
```

Die aktuelle Position ist 12 (Zaehler), der Motor an M-Ausgang M1 wird linksdrehend gestartet. WaitForPositionDown wartet dann auf Erreichen der Position 0, der Motor wird dann ausgeschaltet.

WaitForPositionUp

Warten auf Erreichen einer vorgegebenen Position.

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

Ausgegangen wird von der aktuellen Position in Counter, es werden solange Impulse auf Counter addiert, bis der in Position angegebene Stand erreicht ist. Counter enthält zusätzlich die dann tatsächlich erreichte Position (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 müssen immer positive Werte (einschl. 0) enthalten.

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar; Counter

Siehe auch WaitForPositionDown, WaitForChange

Beispiel

```
var Zaehler: LongInt;
  Zaehler := 0;
  ft.SetMotor(ftiM1, ftiRechts)
  ft.WaitForPositionUp(ftiE2, Zaehler, 24);
```

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

WaitForTime

Anhalten des Programmablaufs.

ft.**WaitForTime**(mSek)

Synonym für Pause

raise KeinOpen, InterfaceProblem; ProcessMessages; Abbrechbar

Siehe auch Pause

Beispiel

```
repeat
  ft.SetMotors($1);
  ft.WaitForTime(555);
  ft.SetMotors($4);
  ft.WaitForTime(555);
until Finish;
```

In der Schleife repeat until Finish 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.