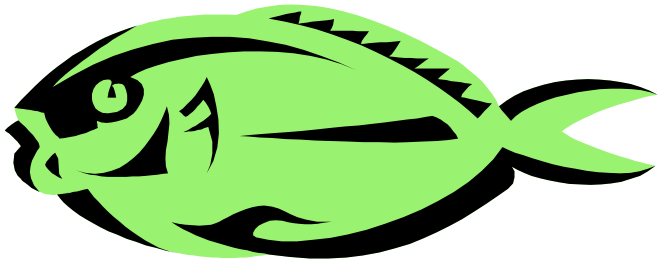
Notizen und Übersichten zu

FishFa30 für Python

Ulrich Müller



Inhaltsverzeichnis

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

Dokument : FishFa30Py.Doc, Druckdatum : 05.09.2003
Titelbild : Einfügen | Grafik | AusDatei | Office | Fish5.WMF

Übersichten

Allgemeines

MIt umFish30.DLL und der darauf aufbauenden Klasse FishFace wird die Möglichkeit geboten, die fischertechnik Interfaces unter Python zu programmieren. FishFace erlaubt die Ansteuerung der parallelen (Unversal) und der seriellen (Intelligent) Interface jeweils wahlweise mit Slave/Extension Module. Es können mehrere Interfaces innerhalb eines Programmes simultan betrieben werden.

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.

Alternativen

fischertechnik Interfaces können auf Basis von umFish30.DLL auf zweierlei Weise betrieben werden :

1. Über die **ActiveX.DLL FishFa30.DLL auf Basis von umFish30.DLL**
   Dies ist in Verbindung mit der PythonWin Entwicklungsumgebung eine recht komfortable Lösung, die auch mit GUI-Programmen (Programmen mit Python/Tk-Forms) vernünftig betrieben werden kann, da durch FishFa30.DLL die Unterbrechbarkeit des Programmes gewährleistet wird und so die Programm-Oberfläche bedienbar bleibt.
   Nachteil : es wird allerhand installiert.
2. Über das **Python Modul FishFa30.py auf Basis von umFish30.DLL**
   Dies ist in Verbindung mit ctypes eine eher minimalistische Lösung. Sie hat den Vorteil transparant zu sein und kann den eigenen Wünschen angepaßt werden, da sie als Source vorliegt. Als Editor empfiehlt sich IDLE, es sind genauso auch andere möglich.
   Nachteil : FishFa30.py eignet sich weniger für GUI-Programme, da mit FishFa30.py geschriebene Programme von Haus aus nicht unterbrechbar (aber abbrechbar) sind, hier ist dann der Ablauf in einem Thread gefragt bzw. müssen Tk.update() eingestreut werden. FishFa30.py besitzt die zusätzlichen Klassen FishRobot zur Programmierung von Robots, die 'RobotMotoren' einsetzen, und FishStep zur Programmierung von Schrittmotoren.

Hier wird der Einsatz des Moduls FishFa30.PY mit den Klassen FishFace, FishFaceException, FishRobot und FishStep beschrieben. Gelegentlich werden Hinweise zu FishFa30.DLL gegeben.

Installation Python und FishFa30

Die Installation ist ein wenig unübersichtlich – wie das so manchmal bei Freeware vorkommt

Immer erforderlich :

Python-2.2.3.exe
Enthält das Python-System mit Dokumentation einschließlich der Entwicklungsumgebung IDLE und der Python Command Line
www.python.org/download

PythonFish30.ZIP
Enthält dieses Handbuch, FishFa30.py und Beispiele
FishFa30.py sollte nach C:\Programme\Python\Modules (Standardinstallation von Python) kopiert werden. Die restlichen Programme nach Belieben.
www.ftcomputing.de/zip/pythonfish30.zip

Für Alternative 1 :

win32all-154.exe
Enthält Mark Hammonds Python Extension for Windows einschließlich der Entwicklungsumgebung PythonWin
http://starship.python.net/crew/mhammond

vbFish30Setup.exe
Enthält die erforderlichen FishFa30.DLL und umFish30.DLLnebst Zubehör sowie ein Handbuch und Tutorial für FishFa30.DLL (ausgerichtet auf Visual Basic, es ist also etwas Phantasie gefragt, meist dürfte auch dieses Handbuch reichen).
www.ftcomputing.de/zip/vbfish30setup.exe

Für Alternative 2 :

ctypes-0.6.2.win32-py2.2.exe
Enthält T.Hellers Zugriffsfunktionen auf (in C/C++ geschriebene) systemkonforme DLLs
http://starship.python.net/crew/theller/ctypes/

umFish30.ZIP
Enthält umFish30.DLL mit Zubehör.
www.ftcomputing.de/zip/umfish30.zip

Die angegebenen Versionen sind die Versionen mit denen getestet wurde. Ist wurde berichtet, daß Python 2.3 und ctypes-0.6.2a ebenfalls laufen. Es ist zu erwarten, daß auch höhere Versionen nutzbar sind.

Hinweise

Auf langsamen Rechnern (200 MHz und darunter) kann Python in Zusammenarbeit mit FishFace mit dem voreingestellten PollInterval u.U. nicht zurechtkommen. Das gilt nur für parallele Interfaces. Hier also den Standardwert 1 auf z.B. 12 erhöhen. Das OpenInterface von FishFa30.py tut das schon.

Der Befehlsumfang der Klasse FishFace von FishFa30.py und FishFa30.DLL ist nahezu gleich. Hier wird nur FishFa30.py beschrieben. FishFa30.DLL bietet ein paar mehr Möglichkeit, die im Bedarfsfall dem Handbuch von vbFish30Setup.EXE zu entnehmen sind.

Bei FishFa30.py kommen außerdem noch die Klassen FishRobot zur Programmierung von Robots und FishStep zur Programmierung von Schrittmotoren hinzu. Sie sind von FishFace abgeleitet und erben deswegen alle FishFace-Methoden.

Literatur zu Python

Hier ein paar sehr subjektive Hinweise und Anmerkungen :

O'Reilly : Python in a Nutshell

Gute knappe Einführung mit gutem Referenzteil (engl.). Eher etwas für Profis
ISBN 0-596-00188-6 Preis : 38 Euro

Galileo Computing : Einstieg in Python

Recht gemütliche Angelegenheit für Anfänger und Umsteiger.
Schneidet viele relevante Themen an, ein Referenzteil fehlt.
ISBN 3-89843-227-5 Preis : 34,90 Euro

mitp : Python für Kids

Für Programmier-Anfänger jeden Alters, die sich nicht an der Du-Anrede stören und für Lehrer, die die Logo-Turtle in die Gegenwart retten wollen.
ISBN 3-8266-0951-4 Preis 19,95 Euro

Informationen zu ctypes und win32all findet man allerdings nur in der mit den Paketen gelieferten Dokumentation.

Referenz

Importe und Instanzierung

FishFa30.DLL

Bei Einsatz von FishFa30.DLL ist der folgende import erforderlich :

```
import win32com.client
```

FishFa30.DLL muß ordnungsgemäß installiert sein (geschieht automatisch bei der Installation).

Die Instanzierung erfolgt über :

```
ft = win32com.client.Dispatch("FishFa30.FishFace")
```

FishFa30.PY

Bei Einsatz von FishFa30.py ist der folgende import erforderlich :

```
from FishFa30 import *
```

Es wird dabei vorausgesetzt, das FishFa30.py in einem zugreifbaren Pfad liegt (siehe Installierung).

Die Instanzierung erfolgt über :

```
ft = FishFace()
```

bzw. :

```
ft = FishRobot(motorenliste)
```

Die Motorenliste/-Tupel enthält die MotorNr (Nr des zugehörenden M-Ausgangs) und den maximalen Fahrweg in Impulsen ab Endtaster (siehe auch Anmerkungen zu den Rob-Funktionen). z.B.: ((2,120), (3, 96), (1,180).

bzw.:

```
ft = FishStep(motorenliste)
```

Die Motorenliste/-Tupel enthält die MotorNr(Nr des ersten zugehörenden M-Ausgangs) und den maximalen Fahrweg in Steppenzyklen (12 Zyklen ergeben einen Motorumdrehung) ab Endtaster (siehe auch Anmerkungen zu den Step-Funktionen). z.B.: ((1, 680), (3, 500)).

Anstelle von ft kann natürlich ein beliebiger anderer Name gewählt werden.

Fehlerbehandlung

Es wird empfohlen FishFace-Fehler über ein try ... except-Konstrukt abzufangen :

```
try:
  ft.OpenInterface("COM2")
except FishFaceException:
  print "------- Interface Problem ------"
  sys.exit(0)
```

Für sys ist ein zusätzliches `import sys` erforderlich. Bei FishFa30.DLL nur except.

FishFa30.PY

Allgemeines

bool 0 = false, 1 = true

int 32bit Integer mit Vorzeichen, Python Standard Wert

Verwendete Variablenbezeichnungen

Die Variablen sind durchweg vom Typ integer (32bit-Wert)

ActPosition Aktuelle Position in Impulsen ab 0 (positiv)

AnalogNr Nummer des Analog-Einganges (0 – 1)

AnalogScan Scannen der AnalogEingänge (1 = true)

Counter Impulszähler (der Wechsel der E-Eingänge zwischen true und false wird automatisch gezählt).

Direction Drehrichtung eines Motors (Ein, Aus, Links, Rechts)

InputNr Nummer eines E-Einganges (1 – 8(16))

LampNr Nummer eines 'halben' M-Ausganges (1-8(16))

LPTAnalog AnalogScalierung. Begrenzung des Analogwertes nach oben.

LPTDelay Ausgabeverzögerung, nur bei LPT1 – LPT3

ModeStatus Status der Betriebsmodi aller M-Ausgänge
Jeweils vier bit pro Ausgang
Begonnen bei 0-3 für M1 (0000 normal, 0001 RobMode)

MotorNr, MotNr Nummer eines M-Ausganges (1-4(8))

MotorNrs Liste von zu überwachenden MotorNummern

MotorStatus Status aller M-Ausgänge. Jeweils 2 bit pro Ausgang.
Begonnen bei 0-1 für M1 (00 = Aus, 01 = Links, 10 = Rechts).

mSek Zeitangabe in MilliSekunden

NrOfChange Anzahl Impulse (Wechsel des Schaltzustandes am zugehörenden E-Eingäng).

OnOff logischer Wert (1 = true, 0 = false)

PollInterval Zeit in MilliSekunden in denen das Interface abgefragt wird

PortName Name des Ports an den das Interface angeschlossen ist
"LPT", "COM1", ... "COM8", "LPT1", "LPT2", "LPT3"

PosHook Name einer CallBack-Funktion zur Verarbeitung (Anzeige) der aktuellen Position

Slave Anschluß eines Extension Modules / Slave Modules (1 = true)

Speed Geschwindigkeit mit der ein M-Ausgang betrieben werden soll
(0, steht – 15, voll)

SpeedStatus Status der Geschwindigkeit aller M-Ausgänge. Jeweils 4bit pro Ausgang.
Begonnen beit 0-3 für M1. Werte 0000-1111.

TargetPosition Zielposition in Impulsen ab 0 (positiv)

TermInputNr Nummer eines E-Einganges mit der die Methode (vorzeitig) beendet wird.

Wert Beliebiger 32bit Integerwert

Symbolische Konstanten

Fehler	Allgemeiner FishFace-Fehler
Ein	Einschalten M-Ausgang
Aus	Ausschalten M-Ausgang
Links	Einschalten M-Ausgang linksdrehend
Rechts	Einschalten M-Ausgang rechtsdrehend
Ende	Ende WaitForMotors, Aufgabe vollständig erfüllt
Time	Ende WaitForMotors, vorgegebene Zeit ist abgelaufen
ESC	Ende WaitForMotors, Abbruch durch ESC-Taste

Eigenschaften

bool	**AnalogScan** (Lesen) Angabe ob auch die Analogeingänge gescannt werden sollen (default = 0 (false))
int	**LPTAnalog** (Lesen) Lesen Analogscalierung
int	**LPTDelay** (Lesen)
int	**PollInterval** (Lesen) Interval (in Millisekunden) in dem der Status des Interfaces abgefragt(gepollt) und aufgefrischt(refresh) wird.
bool	**SILT**(Lesen) Betrieb im SILT-Modus (true) sonst fischertechnik-Modus
bool	**Slave** (Lesen) Mit/ohne Extension Module

FishFace Methoden

- **ClearCounter**(InputNr)
  Löschen(0) des angegebenen Counters

- **ClearCounters**()
  Löschen (0) aller Counter

- **ClearMotors**()
  Abschalten aller M-Ausgänge

- **CloseInterface**()
  Schließen der Verbindung zum Interface

bool **Finish**(InputNr)
Feststellen eines Endewunsches (ESC-Taste, E-Eingang (optional))
Beispiel :

```
while not ft.Finish():
  ft.SetMotor(1, ft.Ein)
  ft.Pause(555)
  ft.SetMotor(1, ft.Aus)
  ft.Pause(333)
```

Die while-Schleife (eine blinkende Lampe) läuft solange bis die ESC-Taste gedrückt wird

int **GetAnalog**(AnalogNr)
Lesen Analog-Wert

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

```
print ft.AnalogDirect(0)
```

Der aktuelle Wert von EX wird ausgelesen und auf der Konsole angezeigt

int **GetCounter**(InputNr)
Auslesen des Wertes des angebenen Counters
Beispiel

```
print "Turm Position : ", ft.GetCounter(2)
```

bool **GetInput**(InputNr)
Auslesen des Wertes des angegebene E-Einganges
Beispiel

```
if ft.GetInput(1):
  .....
else:
  ....
```

Wenn der E-Eingang 1 (Taster, Phototransistor ...) true ist (!= 0), wird der "then"-Zweig durchlaufen

int **GetInputs**()
Lesen der Werte aller E-Eingänge

Beispiel

```
e = ft.GetInputs()
if e & 0x1 or e & 0x40: print "TRUE"
```

Wenn die E-Eingänge E1 oder E7 true sind, wird "TRUE" ausgegeben

int **GetOutputs**()
Lesen der Werte aller M-Ausgänge

- **OpenInterface**(PortName, AnalogScan, Slave, PollInterval, LPTAnalog, LPTDelay)
PortName = "LPT", "COM1" ... "COM8", "LPT1" .. "LPT3"
--- ab hier optional ---
AnalogScan = mit Scannen der Analog-Eingänge (default = 0, ohne)
Slave = Betrieb mit Extension Module (default = 0, ohne)
PollInterval = Zeit in MilliSekunden in denen das Interface abgefragt werden soll (default = 0, Wert wird durch OpenInterface bestimmt)
diesen Wert sorfältig wählen, da sich der Rechner bei zu kleinen Werten "aufhängen" kann.
LPTAnalog = AnalogScalierung (default = 0, setzen durch OpenInterface)
LPTDelay = Ausgabeverzögerung (default = 0, setzen durch OpenInterface), wird nur bei LPT1 – LPT3 ausgewertet.
Exception FishFaceException bei Openfehler.

- **Pause**(mSek)
Anhalten des Programmablaufs für mSek MilliSekunden
Beispiel

```
ft.SetMotor(1, ft.Links)
ft.Pause(1000)
ft.SetMotor(1, ft.Aus)
```

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

- **SetCounter**(InputNr, Wert)
Setzen des Counters InputNr auf Wert

- **SetLamp**(LampNr, OnOff)
Setzen eines 'halben' M-Ausganges
Beispiel:

```
ft.SetLamp(1, ft.Ein)
ft.Pause(2000)
ft.SetLamp(1, ft.Aus)
ft.SetLamp(2, ft.Ein)
```

Die Lampe an M1vorn und Masse wird für 2 Sekunden eingeschaltet und anschließend die an M1 hinten ...

- **SetOutputs**(MotorStatus)
Schreiben des neuen Motorstatus

- **SetMotor**(MotorNr, Direction, Speed, Counter)
setzen eines M-Ausganges, optional mit Geschwindigkeitsangabe (default = 15, full)und Angabe der zurückzulegenden Strecke in Impulsen (Count), die Motoren beenden sich hier selbständig.
Beispiel 1 :

```
ft.SetMotor(1, ft.Rechts, 15)
ft.Pause(1000)
ft.SetMotor(1, ft.Links, 8)
ft.Pause(1000)
ft.SetMotor(1, ft.Aus)
```

Der Motor an M1 wird für 1000 MilliSekunden rechtsdrehend,
volle Geschwindigkeit eingeschaltet und anschließend für 1 Sekunde rechtsdrehend, halbe Geschwindigkeit

Beispiel 2 :

```
ft.SetMotor(1, ft.Links, 12, 123)
```

Der Motor an M1 wird für 123 Impulse an E2 oder E1 = true mit der Geschwindigkeitsstufe 12 eingeschaltet. Das Abschalten erfolgt selbsttätig. Das Programm läuft währenddessen weiter.

- **SetMotors**(MotorStatus, SpeedStatus, ModeStatus)
  Setzen des Status aller M-Ausgänge, optional mit Geschwindigkeitsangabe(SpeedStatus) und des Betriebsmodes (ModeStatus, default = 0). Bei Betriebsmodus RobMode (1) sind vor dem Aufruf der Methode die entsprechenden Counter zu setzen(SetCounter(m)), die Motoren beenden sich sich selbständig.
  Beispiel :

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

  Der Motor an M1 wird auf links geschaltet und gleichzeitig den an M4 auf rechts. Alle anderen Ausgänge werden ausgeschaltet. Nach 1 sekunde werden alle M-Ausgänge abgeschaltet.

- **WaitForChange**(InputNr, NrOfChanges, TermInputNr)
  Warten auf NrOfChanges Impulse an InputNr oder TermInputNr(optional)
  Beispiel :

```
ft.SetMotor(1, ft.Links)
ft.WaitForChange(2, 123, 1)
ft.SetMotor(1, ft.Aus)
```

  Der Motor an M1 wird linksdrehend geschaltet, es wird auf 123 Impulse an E2 oder E1 = true gewartet, der Motor wird abgeschaltet. Vergleiche mit dem Beispiel bei SetMotor. Hier wird das Programm solange angehalten.
  Siehe auch Beispiel bei WaitForMotors.

- **WaitForHigh**(InputNr)
  Warten auf einen false/true Durchgang an InputNr
  Beispiel :

```
ft.SetMotor(1, ft.Ein)
ft.SetMotor(2, ft.Links)
ft.WaitForHigh(1)
ft.SetMotor(2, ft.Aus)
```

  Eine Lichtschranke mit Lampe an M1 und Phototransistor an E1 wird eingeschaltet. Ein Förderband mit Motor 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**(InputNr, OnOff)
  Warten auf InputNr = OnOff. OnOff ist optional (default=1)
  Beispiel :

```
ft.SetMotor(1, ft.Links)
ft.WaitForInput(1)
ft.SetMotor(1, ft.Aus)
```

  Der Motor an M1 wird gestartet, es wird auf E1 = true gewartet, dann wird der Motor wieder abgeschaltet : Anfahren einer Endposition.

- **WaitForMotorLow**(InputNr)
  Warten auf einen true/false Durchgang an InputNr
  Beispiel :

```
ft.SetMotor(1, ft.Ein)
ft.SetMotor(2, ft.Links)
ft.WaitForLow(1)
ft.SetMotor(2, ft.Aus)
```

  Eine Lichtschranke mit Lampe an M1 und Phototransistor an 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).

wait **WaitForMotors**(mSek, MotorNrs)
Warten auf ein MotorReady-Ereignis oder den Ablauf der Zeit (mSek). Ein MotorReady-Ereignis wird durch SetMotor mit Parameter Count bzw. ModeStatus (0001....) gestartet und tritt bei Counter = 0 aller MotorNrs ein.
mSek = 0 : unbegrenztes Warten.
MotorNrs = Liste, Tuple der zu überwachende Motoren (z.B.[4,2,3])
bei FishFa30.DLL sind Einzelwerte anzugeben.
wait = Time : Ablauf der durch mSek vorgegebenen Wartezeit
wait = Ende : Alle Motoren stehen
wait = ESC : Abbruch durch ESC-Taste
Beipiel :

```
ft.SetMotor(4, ft.Links, 8, 50)
ft.SetMotor(3, ft.Rechts, 15, 40)
while ft.WaitForMotors(500, (4, 3)) == ft.Time :
  print ft.GetCounter(6), " - ", ft.GetCounter(8)
```

Der Motor an M4 wird linksdrehend mit halber Geschindigkeit für 50 Impulse gestartet, der an M3 rechtdrehend mit voller Geschwindigkeit für 40 Impulse. Die while-Schleife wartet auf das Ende der Motoren (WaitForMotors). Alle 500 MilliSekunden wird in der Schleife die aktuelle Position angezeigt (500 .... == ft.Time). Wenn die Position erreicht ist (!= ft.Time), ist der Auftrag abgeschlossen, die Motoeren haben sich selber beendet. Achtung : hier wurde nicht auf ESC-Taste abgefragt, es könnte also auch vor Erreichen der Zielposition abgebrochen worden sein.

ActPosition **WaitForPositionDown**(InputNr, ActPosition, TargetPosition, TermInputNr)
Warten auf Erreichen einer vorgegebenen Position (TargetPosition, in Impulsen), ausgehend von der aktuellen Position (ActPosition) durch Herunterzählen der festgestellten Impulse an InputNr.
TermInputNr ist optional.
return = tatsächlich erreichte Position (ActPosition)
bei FishFa30.DLL wird keine ActPosition zurückgegeben.
Beispiel :

```
IstPosition = 12
ft.SetMotor(3, ft.Links)
IstPosition = ft.WaitForPositionDown(6, IstPosition, 0)
ft.SetMotor(3, ft.Aus)
```

Die aktuelle Position ist 12 (IstPosition), der Motor an M3 wird linksdrehend gestartet. WaitForPositionDown wartet dann auf Erreichen der Position 0, der Motor wird dann ausgeschaltet.

ActPosition **WaitForPositionUp**(InputNr, ActPosition, TargetPosition, TermInputNr)
Warten auf Erreichen einer vorgegebenen Position(TargetPosition, in Impulsen), ausgehend von der aktuellen Position(ActPosition) durch Heraufzählen der festgestellten Impulse an InputNr.
TermInputNr ist optional.
return = tatsächlich erreichte Position(ActPosition)
bei FishFa30.DLL wird keine ActPosition zurückgegeben.
Beispiel :

```
IstPosition = 0
ft.SetMotor(3, ft.Rechts)
IstPosition = ft.WaitForPositionUp(6, IstPosition, 24)
ft.SetMotor(3, ft.Aus)
```

Die aktuelle Positon ist 0 (IstPosition), der Motor an M3 wird rechtsdrehend gestartet. WaitForPositionUp wartet dann auf Erreichten der Position 24, der Motor wird dann ausgeschaltet. Siehe auch Beispiel zu WaitFor PositionDown, hier wird in Gegenrichtung gefahren.

- **WaitForTime**(mSek)
  Anhalten des Programmablaufs (wie Pause)
  Beispiel :

```
while not ft.Finish():
  ft.SetMotors(0x1)
  ft.WaitForTime(555)
  ft.SetMotors(0x4)
  ft.WaitForTime(555)
```

  In der Schleife wird erst die Lampe an M1 eingeschaltet und alle anderen abgeschaltet (binär : 0001), dann gewartet, Lampe an M2 eingeschaltet (Rest aus, binär : 0100) und gewartet. Ergebnis : Ein Wechselblinker.

Die Methoden erwarten ein vorhergehendes OpenInterface. Eine entsprechende Exception wird z.Zt. nicht ausgelöst. Sie enthalten kein **DoEvents** (Abarbeiten der Messagequeue) um das Programm unterbrechbar zu machen. Ggf. sind sie (besonders bei GUI-Progammen) in eigene Thread zu verpacken. Wird im Ablauf ein InterfaceProblem festgestellt, wird keine entsprechende **Exception** ausgelöst. Die Wait-Methoden setzen bei Bedarf den zugehörenden **Counter** zurück. Das Auslösen von Exceptions in den Methoden wäre möglich, wurde aber mit Rücksicht auf Stil und Geist von Python unterlassen. Ein Einstieg in die Messagequeue wurde noch nicht gefunden.

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 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.

FishRobot-Methoden

Die FishRobot-Methoden nutzen 'RobMotoren'. Dazu siehe 'Anmerkungen zu den Rob-Funktionen'.

- **MotCntl**
  Liste der unterstützen Motoren und zugehörenden Positionen maximal zulässigen Position ab Endtaster (maxPos) und der aktuellen Position (actPos).
  Beispiel
  ```
  [[3,222,0],[4,88,0]]
  ```
  Der Roboter besitzt zwei Motoren an M3 und M4 mit maxPos = 222 bzw. 88 und der aktuellen Position actPos 0 / 0.

- **MoveHome**()
  Anfahren der Home-Position
  Beispiel
  ```
  ft = FishRobot([[3,222],[4,88]])
  ft.MoveHome()
  ```
  Die bei der Instanzierung aufgelisteten Motoren an M3 und M4 werden linksdrehend gegen die zugehörenden Endtaster an E6 und E8 gefahren (siehe auch "Anmerkungen zu den Rob-Funktionen".

- **MoveTo**(PosHook, PosList)
  Anfahren der in PosList vorgegebenen Position. Dabei wird die in PosHook angegebene Funktion in regelmäßigen Abständen aufgerufen.
  Beispiel
  ```
  ft = FishRobot([[3,222],[4,88]])
  ft.MoveHome()
  ft.MoveTo(ft.PosPrint, [23,34])
  ```
  Der Robot besitzt zwei Motoren an M3 und M4. Zunächst wird die Home-Position angefahren (Position [0, 0], anschließend wird auf die Position [23, 34] gefahren. Die Positionswerte werden in der Reihenfolge der Instanzierungsliste interpretiert, also M3 nach 23 und M4 nach 34. Ist keine PosHook gewünscht, wird None angegeben.

- **PosPrint**(PosList)
  Standard-Funktion, die während des Ablaufs von MoveTo aufgerufen werden kann. MoveTo übergibt in PosList die Liste der aktuellen Position.
  ```
  ft.MoveTo(ft.PosPrint, [23,34])
  ```
  In PosList stehen die Werte für die aktuelle Position, zum Schluß die tatsächlich erreichte Position hier sollte es [23,34] sein.
  Eine eigene PosHook Funktion könnte so aussehen :
  ```
  def PosAnzeige(PosList):
    for p in PosList: print p

  ft.MoveTo(PosAnzeige, [23,34])
  ```
  Die erreichten Positionen werden sehr schlicht untereinander an der Konsole ausgegeben.

FishStep-Methoden

Die FishStep-Methoden betreiben einzelne Schrittmotoren (StepHome, StepTo, StepDelta) und zwei Schrittmotoren simultan im XY-Verbund (PlotHome, PlotTo, PlotDelta). Dazu siehe 'Anmerkungen zu den Step-Funktionen'.

- **MotCntl**
  Liste aller acht möglichen M-Ausgänge mit der maximal möglichen Position in Stepperzyklen ab Endtaster, der aktuellen Position und der Angabe, ob die aktuelle Position im gültigen Bereich liegt.
  Beispiel
  ```
  ((680,100,0), (999,234,0), (500,0,0), (100,0,0))
  ```
  M1-M3 werden durch die zwei Schrittmotoren eines XY-Verbundes belegt. Maximaler Fahrweg 680/500 für X/Y, die aktuellePosition ist 100/234 für X/Y. Die X/Y-Positionen sind beide gültig.

- **PlotHome**(MotNr)
  Anfahren der Home-Position
  Beispiel
  ```
  ft = FishStep(((1,680),(3,500)))
  ft.MoveHome(1)
  ```
  Bei der Instanzierung werden die M-Ausgänge angegeben, die von den FishStep-Methoden genutzt werden sollen : M-Ausgang, maximaler Fahrweg. MoveHome(1) veranlaßt, daß die beiden Schrittmotoren(XY-Verbund), die an den M-Ausgängen, beginnend mit M1, angeschlossen sind gegen die zugehörenden Endtaster E1 und E5 gefahren werden. Anschließend werden noch 2 Zyklen weg vom Endtaster freigefahren.

- **PlotDelta**(PosHook, MotNr, Xrel, Yrel)
  Die Motoren des XY-Verbundes ab MotNr fahren von der aktuellen Position um Xrel/Yrel Zyklen weg vom Endtaster (positiver Wert) oder hin zum Endtaster (negativer Wert). Dabei kann über PosHook eine CallBack-Funktion angegeben werden, die während des Fahrens regelmäßig aufgerufen wird. Bei Angabe None wird keine aufgerufen.
  Beispiel
  ```
  ft.PlotDelta(ft.PosPrintXY, 1, 100, -50)
  ```
  Von der aktuellen Position wird um 100 Zyklen in X- und um -50 Zyklen (hin zum Endtaster) gefahren. Dabei wird über die Standard-Funktion PosPrintXY regelmäßig die aktuelle Position angezeigt.

- **PlotTo**(PosHook, MotNr, Xabs, Yabs)
  Die Motoren des XY-Verbundes ab MotNr fahren auf die in Xabs/Yabs angegebenen Position (Zyklen). Dabei kann über PosHook eine CallBack-Funktion angegeben werden, die während des Fahrens regelmäßig aufgerufen wird. Bei Angabe None wird keine aufgerufen.
  Beipiel
  ```
  def PosAnzeige(PosX, PosY): print PosX, PosY
  ft.PlotTo(PosAnzeige, 1, 234, 234)
  ```
  Der XY-Verbund ab M1 fährt auf die Position 234/234 für X/Y. Dabei wird über die eigene CallBack-Funktion PosAnzeige die aktuelle Position angezeigt.

- **StepHome**(MotNr)
  Anfahren der Home-Position des über MotNr angegebenen Schrittmotors.
  Beispiel

```
ft = FishStep(((1,345),(3,456),(5,432)))
ft.StepHome(5)
```

  Bei der Instanzierung werden die M-Ausgänge angegeben, die von den FishStep-Methoden benutzt werden sollen : M-Ausgang, maximaler Fahrweg. M1-M3 könnten z.B. Motoren im XY-Verbund sein, M5-M6 ein einzelner Schrittmotor am Extension Module. StepHome(5) veranlaßt, daß der Schrittmotor an M5-M6 gegen den zugehörenden Endtaster E9 gefahren wird.

- **StepDelta**(PosHook, MotNr, Xrel)
  Der Motor an MotNr fährt von der aktuellen Position um Xrel Zyklen weg vom Endtaster (positiver Wert) oder hin zum Endtaster (negativer Wert). DAbei kann über PosHook eine CallBAck-Funktion angegeben werden, die während des Fahrens regelmäßig aufgerufen wird. Bei Angabe None wird keine aufgerufen.
  Beispiel

```
ft.StepDelta(None, 5, 123)
```

  Von der aktuellen Position wird um 123 Zyklen weg vom Endtaster gefahren. Dabei wird keine CallBack-Funktion aufgerufen.

- **StepTo**(PosHook, MotNr, Xabs)
  Der Motor MotNr fährt zu der in Xabs angegebenen Position (Zyklen ab 0). Dabei kann über PosHook eine CallBack-Funktion angegeben werden, die während des Fahrens regelmäßig aufgerufen wird. Bei Angabe None wird keine aufgerufen.
  Beispiel

```
def PosAnzeige(PosX): print "Hier bin ich : ", PosX
ft.StepTo(PosAnzeige, 1, 345)
```

  Der Motor an M1-M2 fährt auf Position 345. Dabei wird über die eigene CallBack-Funktion PosAnzeige die aktuelle Position angezeigt (Die entsprechende Standard-Funktion heißt : PosAnzeigeX).

Tips & Tricks

Programmrahmen

Die im Kapitel Techniken angeführten Programmausschnitte benötigen einen Programmrahmen innerhalb dessen sie ablaufen können. Er wird im Kapitel Techniken dann nicht mehr extra angegeben.

Import

Die Programmausschnitte nutzen den Modul FishFa30.py, der in einem Python Zugriffspfad liegen muß. z.B. C:\Programme\Python\Modules (bei Standardinstallation von Python) :

```
from FishFa30 import *
ft = FishFace()
```

Open/Close

Vor einem Zugriff auf ein Interface ist die Verbindung dazu herzustellen z.B. :

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

Der Portname muß den eigenen Erfordernissen angepaßt werden. Das Beispiel zeigt die minimal erforderlichen Angaben. Das schließt einen Zugriff auf die Analog-Eingänge und das Extension Module aus. Weitere Parameter siehe Referenz.

Nach der Nutzung ist er Port wieder freizugeben :

```
ft.CloseInterface
```

Will man mögliche Fehler beim Open abfangen, ist das OpenInterface in einen try / except Block zustellen. Der Standard Programmrumpf sieht dann so aus :

```
# ---------- Programmname.py : Kurztext ---------------

from FishFa30 import *
import sys
ft = FishFace()

try:
  ft.OpenInterface("COM2")
except FishFaceError:
  print "--- Interface Problem ---"
  sys.exit(0)

# --- Beginn des eigentlichen Programms ---


.....


ft.CloseInterface()
```

Techniken

Blinker/Schleife

Lampe an M1 blinkt im Sekundentakt :

```
mGelb = 1

while not ft.Finish():
  ft.SetMotor(mGelb, ft.Ein)
  ft.Pause(555)
  ft.SetMotor(mGelb, ft.Aus)
  ft.Pause(333)
```

Die Parameter für die FishFace Methoden sollten benannt werden. Für einige Standardwerte gibt es bereits Name : ft.Ein, ft.Aus, ft.Links, ft.Rechts und ft.Ende, ft.Time, ft.ESC für die Methode WaitForMotors. Weitere sollte man selber erfinden.

Meistens enthält das Programm ein große Schleife in der alle Befehle wiederholt durchlaufen werden. Hier ist das `while not ft.Finish():` Die Methode Finish prüft, ob ein Abbruchwunsch vorliegt und meldet dann true (1) zurück, deswegen in der while-Schleife die Verneinung durch `not`
Beenden der Schleifte duch ESC-Taste. Es kann auch zusätzlich ein E-Eingang angegeben werden : `ft.Finish(8)`. Beendigung durch ESC-Taste oder E8.

WechselBlinker

Lampen an M1 und M2 blinken im Wechsel.

Alternative 1 :

```
while not ft.Finish():
  ft.SetMotor(2, ft.Aus)
  ft.SetMotor(1, ft.Ein)
  ft.Pause(444)
  ft.SetMotor(1, ft.Aus)
  ft.SetMotor(2, ft.Ein)
  ft.Pause(444)
```

Alternative 2 kompakter:

```
while not ft.Finish():
  ft.SetMotors(0x1)
  ft.Pause(444)
  ft.SetMotors(0x4)
  ft.Pause(444)
```

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 print "---EIN---" sonst "---AUS---" :

```
if ft.GetInput(1):
  print "--- EIN ---"
else:
  print "--- Aus ---"
```

Warten auf einen E-Eingang

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

```
print "--- Zum Programmstart E1 drücken ---"
ft.WaitForInput(1)
print "--- Es geht los ---"
```

Anzeige des Status der E-Eingänge

Status von E1 :

```
e1 = ft.GetInput(1)
if e1: print "E1 : ", e1
```

Laufende Anzeige des Status aller E-Eingänge :

```
while not ft.Finish():
  print "Status der E-Eingaenge : ", hex(ft.GetInputs())
  ft.Pause(1234)
```

Analog-Anzeige

Laufende Anzeige der beiden Analog-Eingänge EX und EY :

```
while not ft.Finish():
  print "Werte EX : ", ft.GetAnalog(0), " EY : ", \
        ft.GetAnalog(1)
  ft.Pause(1111)
```

ACHTUNG : Hier muß beim `OpenInterface("COM1", 1)` das AnalogScan eingestellt werden. Außerdem ist zu beachten, daß ein größeres PollInterval erforderlich ist. Hier wird es durch OpenInterface bestimmt. 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) :

```
print "Wert von EY : ", ft.GetAnalog(1)
```

Fahren für eine bestimmte Zeit

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

```
ft.SetMotor(3, ft.Links)
ft.Pause(3500)
ft.SetMotor(3, ft.Aus)
```

Fahren zum Endtaster

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

```
ft.SetMotor(3, ft.Links)
while not ft.GetInput(5): pass
ft.SetMotor(3, ft.Aus)
```

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

```
ft.SetMotor(3, ft.Links)
ft.WaitForInput(5)
ft.SetMotor(3, ft.Aus)
```

Fahren um eine vorgegebene Anzahl von Schritten

WaitForChange

Motor an M3 mit Impulstaster an E6

```
ft.SetMotor(3, ft.Links)
ft.WaitForChange(6, 12)
ft.SetMotor(3, ft.Aus)
```

WaitForPositionDown

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

```
IstPosition = 12
ft.SetMotor(3, ft.Links)
IstPosition = ft.WaitForPositionDown(6, IstPosition, 0)
```

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

WaitForPositionUp

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

```
IstPosition = 12
ft.SetMotor(3, ft.Rechts)
IstPosition = ft.WaitForPositionUp(6, IstPosition, 24)
```

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(3, ft.Links, 15, 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 : 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(3, ft.Links, 15, 121)
ft.SetMotor(4, ft.Rechts, 7, 64)
while ft.WaitForMotors(300, (3,4)) == ft.Time:
  print "Position M3 - M4 : ", ft.GetCounter(6), " - ", \
        ft.GetCounter(8)
print "Position M3 - M4 : ", ft.GetCounter(6), " - ", \
      ft.GetCounter(8), " --- 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, ft.Ein)
ft.SetLamp(4, ft.Ein)
ft.Pause(1000)
ft.SetLamp(1, ft.Aus)
ft.SetLamp(4, ft.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 :

```
mLicht, ePhoto, false = 1,1,0

ft.SetMotor(mLicht, ft.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 :

```
mBand, ePhoto = 2,1

ft.SetMotor(mBand, ft.Links)
ft.WaitForLow(ePhoto)
ft.SetMotor(mBand, ft.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 :

```
mBand, ePhoto = 2,1

ft.SetMotor(mBand, ft.Links)
ft.WaitForHigh(ePhoto)
ft.SetMotor(mBand, ft.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.

```
mGruen, mGelb, mRot = 1,4,16
while not 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 Ampeltak vorgibt, kann man den Ablauf auch listengesteuert machen :

```
mGruen, mGelb, mRot = 1,4,16
Phase = [mGruen, mGruen, mGruen, mGruen, mGelb,
         mRot,   mRot,   mRot,   mRot,   mRot+mGelb]

while not ft.Finish():
  for 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 man rein zufällig am parallelen Interface gerade acht Lampen angeschlossen hat (siehe oben Tip Lampen), kann man auch ganz einfach ein Lauflicht programmieren :

```
while not ft.Finish():
  Phase = 3
  while Phase < 256:
    ft.SetMotors(Phase)
    ft.Pause(555)
    Phase = Phase<<1
```

Robot-Fahren

Bei Verwendung der Klasse FishRobot anstelle von FishFace stehen zusätzich noch einige Robot-Methoden zur Verfügung :

```
from FishFa30 import *
ft = FishRobot([[3,222],[4,88]])

ft.OpenInterface("COM2")

ft.Home()

ft.MoveTo(ft.PosPrint, [23,34])
ft.MoveTo(None, [23,34])
ft.MoveTo(ft.PosPrint, [23,23])

ft.CloseInterface()
```

Bei der Instanzierung wird die Roboter-Konfiguration mitgeteilt Motore an M3 und M4 mit maximalem Fahrweg ab Endtaster von 222 bzw. 88 Impulsen.
Nach dem Open wird die Home-Position (die Position an den Endtastern) angefahren und der interne Positionszähler in ft.MotCntl auf 0 gesetzt.
Anschließend wird auf die Position M3 = 23 und M4 = 34 gefahren. Die laufende Position wird über PosPrint auf der Konsole ausgegeben. Dann wird nochmal die gleiche Position angefahren – um zu zeigen, daß da nichts ruckelt – und dann die Position M4 = 23, also wieder in Richtung Endtaster. Siehe auch "Anmerkungen zu den Rob-Funktionen".

Schrittmotoren

Bei Verwendung von FishStep anstelle von FishFace stehen zusätzlich noch einige Methoden für Schrittmotorern zur Verfügung :

Einzelner Schrittmotor

Fahrstuhl aus einem Schrittmotor mit einer (langen) Schneckenwelle senkrecht nach oben und einem "Korb" an der Schneckenmutter. Den Endtaster E1 nicht vergessen :

```
from FishFa30 import *

def ShowEtage(PosX):
  print "Laufende Etage : ", PosX/50

Fahrstuhl = 1
ft = FishStep(((1,222),))
ft.OpenInterface("COM2")

ft.StepHome(Fahrstuhl)
Etage = 0
while Etage >= 0:
  print "Haelt in Etage : ", Etage
  ft.StepTo(ShowEtage, Fahrstuhl, \
            (0,50,100,150,200)[Etage])
  Etage = input("StepFahrstuhl : Bitte Etage eingeben : ")
  if Etage > 4: Etage = 4

ft.CloseInterface()
```

Bei der Instanzierung wird ein Schrittmotor an M1-M2 festgelegt, der von 0 bis 222 Zyklen fahren darf. Nach dem Open (hier COM2) wird die Home-Position angefahren (Endtaster E1), dann wird Etage = 0 vorgegeben und innerhalb der while-Schleife auch gleich mit StepTo angefahren. Anschließend wird die nächste Etage erfragt.

In StepTo werden die Etagen-Positionen (Xabs) in Form eines Tupels vorgegeben. Auf das dann über Etage zugegriffen wird. Mit ShowEtage wird die gerade durchfahrene Etage angezeigt.

Zwei Schrittmotoren im XY-Verbund

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.

```
from FishFa30 import *

def Vieleck(RelList):
  for xy in RelList: ft.PlotDelta(None, 1, xy[0], xy[1])

Plotter = 1
ft = FishStep(((1,333),(3,222)))
ft.OpenInterface("COM2")

ft.PlotHome(Plotter)
ft.PlotTo(None, Plotter, 50, 50)
Vieleck(((50,0),(0,50),(-50,0),(0,-50)))

ft.CloseInterface()
```

Bei der Instanzierung werden zwei XY-Verbund Schrittmotoren an M1-M3 angemeldet. Die Zeichenfläche hat eine Größen von 333/222 für X/Y. Nach dem Open (hier COM2) wird die Home-Position angefahren (Endtaster E1 und E5) und daran anschließend mit PlotTo die Position 50/50. In der Funktion Vieleck wird dann ein Quadrat mit 50er Kantenlänge gezeichnet. Genutzt wird die Methode PlotDelta. Die erforderlichen Werte werden als Liste/Tupel übergeben.

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 Counters. Sie sind den E-Eingängen zugeordnet (Achtung : E1 wird je nach Sprache und Implementierung auf Counter 0 bzw. 1 abgebildet). In den Countern wird jede Veränderung des Zustandes der E-Eingänge gezählt. Also z.B. das Öffnen oder auch das Schließen eines Tasters.

Die Counter sind Teil des Kontrollblocks und können dort abgefragt bzw. gesetzt werden. Wenn der Kontrollblock nicht direkt zugänglich ist, werden entsprechende Funktionen/Methoden angeboten. Die Counter werden auch intern von einigen Funktionen/Methoden (z.B. SetMotor mit Parameter Counter und den meisten Wait-Methoden) genutzt – Bei umFish30-Funktionen sind es nur um/csRobMotor(s).

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

Die Rob-Funktionen laufen in einem bsonderen 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 des Python Moduls FishFa30.py 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)).