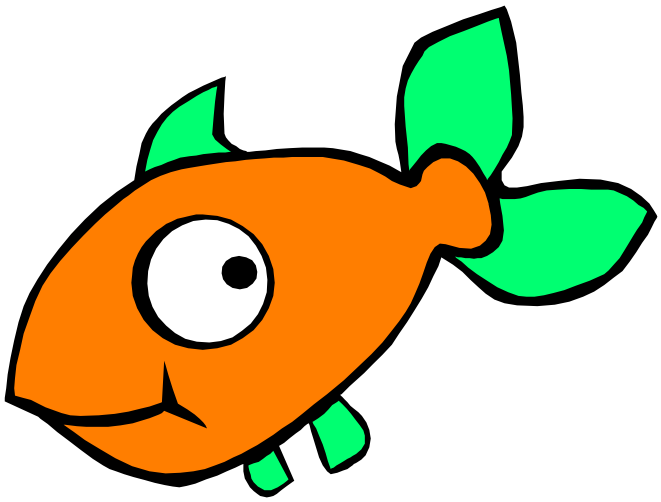
Notizen und Übersichten zu

FishFa30 für Perl

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 : FishFa30PL.Doc, Druckdatum : 06.01.2004
Titelbild : Einfügen | Grafik | AusDatei | Office | Fischchn.WMF

Übersichten

Allgemeines

Mit umFish30.DLL und der darauf aufbauenden ActiveX.DLL FishFa30.FishFace wird die Möglichkeit geboten, die fischertechnik Interfaces auf Win32-Systemen unter Perl zu betreiben. FishFace erlaubt die Ansteuerung der parallelen (Universal) 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 (s.a. OpenInterface).

Die ActiveX FishFa30.DLL kann unter Perl über dessen OLE-Package betrieben werden. Beschrieben wird hier der Betrieb mit der use Win32::OLE Variante. Der Betrieb mit der use OLE Variante ist genauso möglich, es kann aber kleinere Differenzen in der Kontext-sensibiltät geben. In der Instanzierung bestehen formale Unterschiede.

Installation Perl und FishFa30

Die Installation ist erfolgt mehrstufig :

1. **Perl-System**
   Wenn nicht schon vorhanden : Kostenloser Download von http://www.activestate.com/ActivePerl. Hier wurde mit der MSI Binär-Distribution für Win32 "Active Perl 5.6.1 build 635" gearbeitet. Diese Distribution stellt auch das erforderliche OLE-Package zur Verfügung. Die Perl-Installation erfolgt sehr einfach über den MSI-Installer, der auf dem System vorhanden sein muß.
2. **FishFa30.DLL**
   Sollte mit dem Installprogramm www.ftcomputing.de/zip/vbfish30setup.exe installiert werden. Das Installprogramm erlaubt wahlweise die Installation eines Treiber, der zum Betrieb des älteren (parallelen) Universal-Interfaces erforderlich ist. Außerdem wird ein "Interface Panel" (ein Tool zum Test der an das Interface angeschlossenen Modelle) installiert. Das beiliegende VB6-Handbuch kann ebenfalls gewinnbringend genutzt werden, da die FishFace.Methoden gleich sind (man muß sich nur die $, @, -> ; dazu denken) .
3. **FishFa30PL.PDF**
   Dieses Handbuch mit einer Befehlsreferenz und umfangreichen Tips & Tricks in Perl.
4. **Entwicklungsumgebung**
   Im einfachsten Fall Editieren mit NotePad (Editor) und Testen im MS-DOS-Fenster. Alternativ ein Editor mit Syntax-Highlighting und Kommando zum Ausführen des Programms z.B. Uli Meybohms (www.meybohm.de) Proton.

Literatur zu Perl

Hier ein paar sehr subjektive Hinweise und Anmerkungen :

O'Reilly : *Einführung in Perl für Win32 Systeme* (ISBN 3-89721-106-8, scheint nicht mehr lieferbar zu sein, ISBN der amerikanischen Ausgabe : 1-56592-324-3).
Enthält ein kurzes Kapitel "OLE-Automatisierung".

O'Reilly : *Perl in a Nutshell* (ISBN 3-89721-338-9) Referenz und Kurzeinführung.
Ebenfalls mit einem Kapitel OLE-Automatisierung.

O'Reilly : Programmieren mit Perl (Larry Wall et. al. ISBN 3-89721-144-0)
Das Buch über Perl und seine Philosophie.

Addison-Wesley : Nitty Gritty Perl (ISBN 3-8273-18886-6) Für erfahrene Programmierer, die einen kurzen Überblick gewinnen möchten. Zudem das (mit Abstand preiswerteste Buch).
Kein OLE-Automatisierungs Kapitel.

Wenns nur um OLE-Automatisierung geht und man Perl einigermaßen sicher kann und den Beispielen dieses Dokumentes traut, geht’s auch ohne.

Referenz

use und Instanzierung

Bei Einsatz von FishFa30.DLL ist der folgendes use erforderlich :

```
use Win32::OLE;
```

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

Die Objekterstellung erfolgt über :

```
$ft = Win32::OLE->new("FishFa30.FishFace", 'Quit');
```

Alternative kann auch mit use OLE; gearbeitet werden (in diesem Dokument wird mit use Win32::OLE gearbeitet) :

```
use OLE;
$ft = CreateObject OLE "FishFa30.FishFace" || die "CreateObject: $!;
```

Der Rest ist dann wie oben. Gelegentlich wird aber mehr Wert auf {} und "" in der Schreibweise gelegt.

FishFa30.DLL

Symbolische Konstanten

Folgende symbolische Konstanten sollten im Programm verwendet werden :

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

Da FishFa30.DLL sie sich nicht entlocken läßt tuns auch zwei Hashes :

```
%Dir  = (Aus => 0,  Ein => 1,  Links => 1,   Rechts => 2);
%Wait = (Ende => 0, Time => 1, NotHalt => 2, ESC => 3);
```

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	**Slave** (Lesen) Mit/ohne Extension Module
string	**Version** (Lesen) Version von FishFa30.DLL

Schreibweise : `$ft->{AnalogScan}`. z.B.
Gesetzt werden diese Eigenschaften beim OpenInterface

FishFace Methoden

Für das Objekt von FishFace wird einheitlich $ft verwendet. Es wird die Existenz der oben angegebenen Hashes %Dir und %Wait angenommen.

- **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 :

```
until($ft->Finish) {
  $ft->SetMotor(1, $Dir{Ein});
  $ft->Pause(555);
  $ft->SetMotor(1, $Dir{Aus});
  $ft->Pause(333);
}
```

Die until-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->GetAnalogDirect(0), "\n";
```

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), "\n";
```

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\n"}
```

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

- **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 sorgfältig 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, $Dir{Links});
$ft->Pause(1000);
$ft->SetMotor(1, $Dir{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, $Dir{Ein});
$ft->Pause(2000);
$ft->SetLamp(1, $Dir{Aus});
$ft->SetLamp(2, $Dir{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, $Dir{Rechts})
$ft->Pause(1000);
$ft->SetMotor(1, $Dir{Links}, 8);
$ft->Pause(1000);
$ft->SetMotor(1, $Dir{Aus});
```

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

Beispiel 2 :

```
$ft->SetMotor(1, $Dir{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 der 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, $Dir{Links});
$ft->WaitForChange(2, 123, 1);
$ft->SetMotor(1, $Dir{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, $Dir{Ein});
$ft->SetMotor(2, $Dir{Links});
$ft->WaitForHigh(1);
$ft->SetMotor(2, $Dir{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, $Dir{Links});
$ft->WaitForInput(1);
$ft->SetMotor(1, $Dir{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, $Dir{Ein});
$ft->SetMotor(2, $Dir{Links});
$ft->WaitForLow(1);
$ft->SetMotor(2, $Dir{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 = zu überwachende Motoren (z.B. 4,2,3)
wait = Time : Ablauf der durch mSek vorgegebenen Wartezeit
wait = Ende : Alle Motoren stehen
wait = ESC : Abbruch durch ESC-Taste
Beipiel :

```
$ft->SetMotor(4, $Dir{Links}, 8, 50);
$ft->SetMotor(3, $Dir{Rechts}, 15, 40);
while($ft->WaitForMotors(500, 4, 3) eq $Wait{Time}) {
  print $ft->GetCounter(6),
        " - ", $ft->GetCounter(8), "\n";
}
```

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 .... eq $Wait{Time}). Wenn die Position erreicht ist (ne $Wait{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.

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

```
until($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. Wird im Ablauf ein InterfaceProblem festgestellt, wird eine entsprechende **Exception** ausgelöst. Die Wait-Methoden setzen bei Bedarf den zugehörenden **Counter** zurück.

Die SetMotor(s)-Methoden sind **asynchron** d.h. der oder die angesprochenen Motoren (Lampen) werden mit der Methode gestartet. Sie laufen dann unabhängig vom Programm weiter. Sie werden durch ein weiteres SetMotors mit Direction = 0 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.

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.

use

Die Programmausschnitte nutzen use Win32::OLE; Alternativ dazu kann ebenso use OLE; eingesetzt werden. :

```
use warnings
use Win32::OLE;
$ft = Win32::OLE->new("FishFa30.FishFace", 'Quit');
```

alternativ :

```
use warnings
use OLE;
$ft = CreateObject OLE "FishFa30.FishFace" ||
      die "CreateObject: $!";
```

Die erste Variante bietet insgesamt mehr Möglichkeiten, sie wird hier eingesetzt. Properties(eigenschaften) erfordern je nach Kontext eine Schreibweise in "{}", hier werden für Properties immer "{}" verwendet.

Zusätzlich wird bei Bedarf der Einsatz folgender Hashes angenommen :

```
%Dir  = (Aus  => 0, Ein  => 1, Links   => 1, Rechts => 2);
%Wait = (Ende => 0, Time => 1, NotHalt => 2, ESC    => 3);
```

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;
```

Der komplette Programmrahmen Win32::OLE

```
# --- Manual.PL ------------------ Ulrich Müller   04.01.04
#                                 PCS1700 ActivePerl 5.6.1
#                                 use Win32::OLE Version
# Testrahmen für die Manual-Beispiele

use warnings;
use Win32::OLE;

%Dir  = (Aus  => 0, Ein  => 1, Links   => 1, Rechts => 2);
%Wait = (Ende => 0, Time => 1, NotHalt => 2, ESC    => 3);


$ft = Win32::OLE->new("FishFa30.FishFace", 'Quit');
print "FishFace-Version : $ft->{Version} \n";

$ft->OpenInterface("COM2");

until($ft->Finish) {
  ............
}

$ft->CloseInterface;
print "--- FINIS ---\n";
```

Werden %Dir, %Wait nicht benutzt, gibt's entsprechende Warnungen. Die until-Schleife ist hier eigentlich überflüssig, da sie im Beispielcode (bei Bedarf) enthalt ist, hier soll sie nur die Einfügestelle markieren.

Techniken

Blinker/Schleife

Lampe an M1 blinkt im Sekundentakt :

```
$mGelb = 1;
while(! $ft->Finish) {
  $ft->SetMotor($mGelb, $Dir{Ein});
  $ft->Pause(555);
  $ft->SetMotor($mGelb, $Dir{Aus});
  $ft->Pause(333);
}
```

Die Parameter für die FishFace Methoden sollten benannt werden. Für einige Standardwerte werden hier mit Hashes definierte Namen verwendet : $Dir{} : Ein, Aus, Links, Rechts und $Wait{} : Ende, Time, ESC für die Methode WaitForMotors.

Meistens enthält das Programm ein große Schleife in der alle Befehle wiederholt durchlaufen werden. Hier ist das `while (! $ft->Finish)` : Die Methode Finish prüft, ob ein Abbruchwunsch vorliegt und meldet dann true (1) zurück, deswegen in der while-Schleife die Verneinung durch `!`
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 :

```
($mGelb, $mRot) = (1, 2);
until($ft->Finish) {
  $ft->SetMotor($mRot,  $Dir{Aus});
  $ft->SetMotor($mGelb, $Dir{Ein});
  $ft->Pause(444);
  $ft->SetMotor($mGelb, $Dir{Aus});
  $ft->SetMotor($mRot,  $Dir{Ein});
  $ft->Pause(444);
}
```

Alternative 2 kompakter:

```
until($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---" :

```
until($ft->Finish) {
  if ($ft->GetInput(1)) {
    print "--- EIN ---\n";
    $ft->Pause(333);
  } else {
    print "--- Aus ---\n";
    $ft->Pause(333);
  }
}
```

Warten auf einen E-Eingang

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

```
print "--- Zum Programmstart E1 druecken ---\n";
$ft->WaitForInput(1);
print "--- Es geht los ---\n";
```

Anzeige des Status der E-Eingänge

Status von E1 :

```
print "Status E1 : ", $ft->GetInput(1), "\n";
```

man beachte die Kommata im print, im "" Kontext mag er nicht.

Laufende Anzeige des Status aller E-Eingänge :

```
until($ft->Finish) {
  print "Status der E-Eingaenge : ",
        hex($ft->GetInputs), "\n";
  $ft->Pause(1234);
}
```

Analog-Anzeige

Laufende Anzeige der beiden Analog-Eingänge EX und EY :

```
until($ft->Finish) {
  print "Werte EX : ", $ft->GetAnalog(0), " EY : ",
        $ft->GetAnalog(1), "\n";
  $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->GetAnalogDirect(1), "\n";
```

Fahren für eine bestimmte Zeit

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

```
$ft->SetMotor(3, $Dir{Links});
$ft->Pause(3500);
$ft->SetMotor(3, $Dir{Aus});
```

Fahren zum Endtaster

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

```
$ft->SetMotor(3, $Dir{Links});
until ($ft->GetInput(5)) {}
$ft->SetMotor(3, $Dir{Aus});
```

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

```
$ft->SetMotor(3, $Dir{Links});
$ft->WaitForInput(5);
$ft->SetMotor(3, $Dir{Aus});
```

Fahren um eine vorgegebene Anzahl von Schritten

WaitForChange

Motor an M3 mit Impulstaster an E6

```
$ft->SetMotor(3, $Dir{Links});
$ft->WaitForChange(6, 12);
$ft->SetMotor(3, $Dir{Aus});
```

WaitForMotors

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

```
$ft->SetMotor(3, $Dir{Links}, 5, 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, $Dir{Links}, 15, 121);
$ft->SetMotor(4, $Dir{Rechts}, 7, 64);
while($ft->WaitForMotors(300, 3, 4) eq $Wait{Time}}) {
  print "Position M3 - M4 : ", $ft->GetCounter(6), " - ",
        $ft->GetCounter(8), "\n";
}
print "Position M3 - M4 : ", $ft->GetCounter(6), " - ",
      $ft->GetCounter(8), " --- Final ---\n";
```

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.

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, $Dir{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, $Dir{Links});
$ft->WaitForLow($ePhoto);
$ft->SetMotor($mBand, $Dir{Aus});
```

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

Warten auf Ausfahrt aus einer Lichtschranke

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

```
($mBand, $ePhoto) = (2, 1);

$ft->SetMotor($mBand, $Dir{Links});
$ft->WaitForHigh($ePhoto);
$ft->SetMotor($mBand, $Dir{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, $mRot) = (1, 4, 16);
until($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);

until($ft->Finish) {
  foreach $p (@Phase) {
    $ft->SetMotors($p);
    $ft->Pause(250);
  }
}
```

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

Lauflicht

An M1 – M4 sind jeweils eine Lampe angeschlossen, sie werden nacheinander eingeschaltet :

```
until($ft->Finish) {
  $Phase = 1;
  while($Phase < 256) {
    $ft->SetMotors($Phase);
    $ft->Pause(555);
    $Phase <<= 2;
  }
}
```

Wenn man 8 Lampen jeweils einpolig an Mx und an Masse anschließt, ists noch schöner und wenn man dann $Phase eingangs auf 3 setzt laufen zwei eingeschaltete Lampen übers Tableau.

Anmerkungen zum Verständnis

Zugriff auf das Interface

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

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

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

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

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

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

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

Anmerkungen zu den Counters

Ein wesentliches Element zur Positionsbestimmung sind die Counter. Sie sind den E-Eingängen zugeordnet (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.