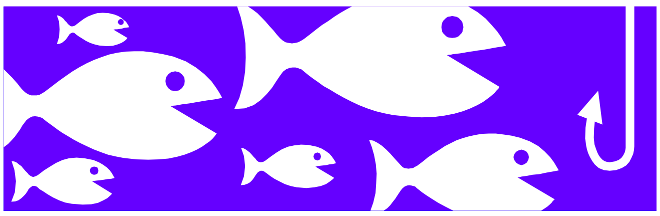
ftComputing

CFishFace for VC++

Ulrich Müller



Content

umFish30.DLL

Common

This manual describes using the functions from umFish30.DLL to access the fischertechnik interfaces by VC++ 6.0. There are two different ways to do it :

- The direct access to the umFish30 functions (cs style) via umFish30VC.H.
  This is of interest, if you want to write your own class library.
- Using the class CFishFace based on umFish30 via FishFa30.H (and csFish30.H, FishFa30.CPP). This is the usual way.

The sources and examples are contained in www.ftcomputing.de/zip/ccfish30.zip.

In both cases you need umFish30.DLL with umFish30.LIB. For some more details to umFish30.DLL look to www.ftcomputing.de/zip/umfish30.zip and www.ftcomputing.de/ccfishe.htm .

umFish30.ZIP contains a TFishFace class for C++Builder, which is very similar to CFishFace. Because of the documentation for TFishFace is a little short, you can use this documentation too.

The Access to the Interface

The access to the interface is done indirect via a poll routine. That routine reads out in constant distances the values of the interface and writes in the same turn the status of the outputs (M-Outlets). That fullfills the refresh conditions of the interface, which must be contacted in interval of 300 msec. Therfore the is no switch off of the interface in times of inactivity of the program. To have a constant time base of typical 10 msec the MultMediaTimer is used. The poll routine triggered by him is running in a separate thread.

The values read out by the poll routine are stored in a control block and vice verse the value for the outputs are read from there. The control block contains in addition all values used for running the interface (a slave included). The simultan running of several interfaces therefore is possible.

The poll routine does beside the pure control of the interface some more services. That are counting impulses on the inputs (changing from true to false or false to true) and controlling the "speed" of the outputs (done by PWM Pulse Wide Modulation). Running in Rob Mode single outputs are switched off, if the impulse counter belonging to that output reaches 0. Short time before the output is slowed down.

The offered access functions are a mix from necessity and comfort. Open/CloseInterface makes a connection to the interface (or ends it), is setting default values (especially for the poll interval) and will start the MultiMediaTimer. The GetInput/GetAnalog function reads out the input value from the control block belonging to the input called. Same is done by SetMotor / SetLamp in the opposite direction. There is no direct acces to the interface.

GetAnalogDirect makes an exception. It accesses the analog input direct, in that time the poll routine is stopped. Reason : the read out of the analog values (especially with the parallel interface) will last much longer than a normal poll interval for read/write the digitalinputs/outputs only. Using the this function you can run the program with a much shorter poll interval and only at time the motors are stopped (recommended) you access the analog inputs.

SetMotor(s)Ex and RobMotor(s) don't access the interface direct, but do more sophisticated function.

Using the um style functions, the more complex functions can be done via direct access to the control block too.

Some programming languages will do hard with using a static structure in connection with a DLL

HelloFish for VC++ 6.0 – Console Project

Setting up the C++ Project

Easiest way : copy the whole ccFish30.ZIP in a new directory

Or : build yourself a new project

- New Workspace : Console
- Copy umFish30.DLL to \Debug or \Release or alternatively to \WinNT\System32
- Insert umFish30VC.h, umFish30.lib, umFish30.VC.cpp
- Compile (F7)

To connect the Interface

The interface is suggested to be on COM2. In other case change csOpenInterface(2, 1, 0, 0); to csOpenInterface(1, 1, 0, 0); for connection on COM1.

The HelloFish project expects 3 lamps on M1 to M3 and an switch on E1.

The Source

```
#include <windows.h>
#include <iostream.h>
#include "umFish30VC.h"

int ft;

void main()
{
  cout << "---- HelloFish started ----" << endl;
  cout << "umFish30 v" << csVersion() << endl;
  ft = csOpenInterface(2, 1, 0, 0);
  if(ft == ftiError) {
    cout << "Interface Problem, exit" << endl;
    return;
  }
  cout << "OpenInterface succeeded" << endl;

  cout << "--- Loop for the three lamps on M1 to M3" << endl;
  for(int j=1; j<=4; j++)
  {
    cout << "Round : " << j << endl;
    csSetMotors(ft, 0);
    Sleep(300);
    for(int i=1; i<=3; i++)
    {
      csSetMotor(ft, i, ftiOn);
      Sleep (500);
    }
  }

  csSetMotors(ft, 0);
  cout << "END : Input E1 = TRUE" << endl;
  while(!csGetInput(ft, 1));
  csCloseInterface(ft);
}
```

Explanations :

`#include <windows.h>` : Standard include

`#include <iostream.h>` : include for cout / cin

`#include "umFish30VC.h"` : umFish30.DLL declarations.

`int ft;` : Handle to umFish30

```
ft = csOpenInterface(2, 1, 0, 0);
if(ft == ftiFehler) {
  cout << "Interface Problem, exit" << endl;
  return;
}
```

Connection to the interface. Parameters : COM2, with AnalogScan, no Slave, Poll default.
Returns the Handle to umFish30. If it is == ftiError the connection failed.

```
for(int j=1; j<=4; j++) {...}
```

Repeat lamp switching 4 times

```
csSetMotors(ft, 0);
Sleep(300);
for(int i=1; i<=3; i++)
{
  csSetMotor(ft, i, ftiOn);
  Sleep (500);
}
```

Clear all M-Outputs (Lamps) and pause for 0.3 secs.
Switch on lamps on M1 – M2 – M3, pause after each switch for 0.5 secs.

```
  csSetMotors(ft, 0);
  cout << "END : Input E1 = TRUE" << endl;
  while(!csGetInput(ft, 1));
  csCloseInterface(ft);
```

Ending the program :
- Switch off all M-Outputs
– Write message
– Wait for E-Input E1 to be true.
– Cancel the connection to the interface.

Some more Examples

Some more examples for using umFish30.DLL (cs style) can be found in ccfish30.zip, directory \TipsTriCS.

Constructing a Class using umFish30.DLL

ccfish30.zip, directory \TipTriC contains an example for constructing a C++ class CFishing for using the umFish30.DLL functions in a manner which is adequate to the language C++ (VC++ version). This is a simple – one file - version for private use. Fore a more professional version look to class CFishFace (Chapter : The class CFishFace).

The class CFishFace

Common

The source for CFishFace contains the following files :

- FishFa30.H : The header file with enums and the class definitions CFishFaceException and CFishFace and some inline functions for property like methods.
  For use with applications this file must be included (#include "FishFace.H").
- csFish30.H : Function declarations for umFish30.DLL cs style.
- FishFa30.CPP : the constructors and method implementations for CFishFace.

The methods which are running for a longer time (Waitxxx, Pause ...) can be canceled by pressing the ESC key.

Most methods are throwing an excepiton for "KeinOpen.methodname" (OpenInterface is missing) and "InterfaceProblem.methodname" (some problem with the interface – may be power off, no connection to the serial port).

CFishFace can be used with console applications as well as with MFC applications. In case of MFC the application form may be sometimes to be 'frozen' because of CFishFace methods are running in a very close loop. In this case applications must give control to the message queque. An other nice solution is to run the CFishFace part of the application in a separate thread (a C++Builder solution for that is described in www.ftcomputing.de/ccthreade.htm).

Using CFishFace

Here are described some small examples in the manner of tips & trics.

The Program Frame

Is a console application in the manner of that before. It is build with the VC++ 6.0 IDE as an empty console project, if it is not copied from ccFish30.ZIP path FiFa30CCP. All the samples are separate routines listed on the beginning of the program. Followed by the main routine with a call for one of them. This call is to be replaced by a call for the interesting routine to test the routine. The source is contained in FiFa30Test.CPP, project directory is FiFa30CPP :

```
#include <windows.h>
#include <iostream.h>
#include "FishFa30.h"

CFishFace ft(true, false, 0);

void SampleRoutine() {
......
}
```

```
void main() {
   cout << "--- FiFaTest started ---" <<  endl;
   cout << "To end press Esc Key" << endl;

   try {
      ft.OpenInterface(2);

      SampleRoutine();

   }
   catch(CFishFaceException& fte) {
      cout << "Fehler " << fte.Nr() << " : " << fte.Text() << endl;
   }

   while(GetAsyncKeyState(VK_ESCAPE) == 0);
   ft.CloseInterface();
}
```

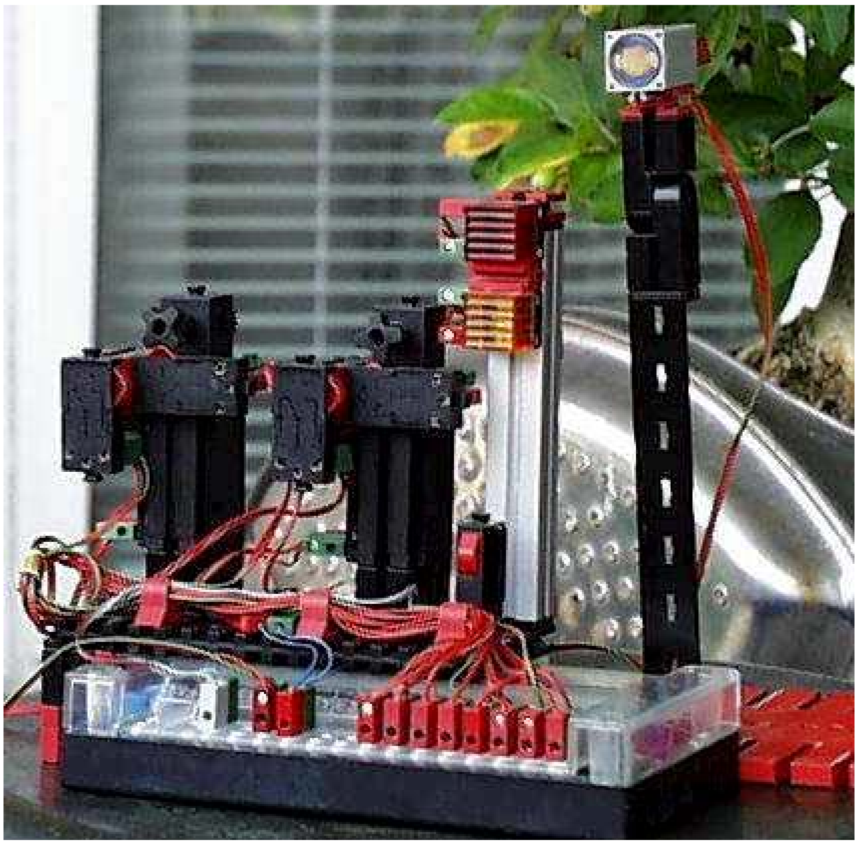
#include windows.h and iostream.h are already known VC++ includes. #include "FishFa30.h" includes the header file of the class CFishFace.

Main begins with a start message and the notice : to end press Esc Key. Next is a try ... catch block with ft.OpenInterface(2); for the connection to COM2 (must be changed for connection to COM1) and the call for the sample. The catch block works with the special CFishFaceException and write an error message on the console. It works for errors thrown by OpenInterface and that from the sample routines.

The main routine ends with waiting for an Esc press and a CloseInterface connection.

Common Techniques

This techniques are based on the class CFishFace. To test them you can use a simple model like this :



M1 : red lamp
M2 : yellow lamp
M3 : motor with end switch on E5 and impulse switch on E6
M4 : motor with end switch on E7 and impulse swicht on E8
EX : photoresistor
EY : resistor ...
E1 : switch

BlinkingLoop

Lamp on M1 is blinking on sec intervals :

```
void BlinkingLoop() {
  do {
    ft.SetMotor(ftiM1, ftiOn);
    ft.Pause(555);
    ft.SetMotor(ftiM1, ftiOff);
    ft.Pause(444);
  } while(!ft.Finish());
}
```

The parameters can be simple int numbers, but they should be named. This names are from the enums of CFishFace. You can use your own names to : `const int mRed = 1` instead of ftiM1, that is more instructive.

Mostly a program own an all over loop. In this case is that do { ... } while(!ft.Finish()); : The method Finish looks if there is an cancel request. May be the ESC key or optional an E-Input.

AlternateBlinking

Lamps on M1 and M2 are blinking alternating.

Version 1 :

```
void AlternateBlinking1() {
   do {
      ft.SetMotor(ftiM2, ftiOff);
      ft.SetMotor(ftiM1, ftiOn);
      ft.Pause(444);
      ft.SetMotor(ftiM1, ftiOff);
      ft.SetMotor(ftiM2, ftiOn);
      ft.Pause(444);
   } while(!ft.Finish());
}
```

Version 2 (more compact) :

```
void AlternateBlinking2() {
   do {
      ft.SetMotors(0x1);
      ft.Pause(333);
      ft.SetMotors(0x4);
      ft.Pause(333);
   } while(!ft.Finish());
}
```

In this case all M-Outputs are switch with one method : SetMotors. The parameter of SetMotors is a MotorState which contains the states of all M-Outputs (extension module included), each with 2 bits. That means : 00000001 M1 ftiOn and 00000100 M2 ftiOn. All other M-Outputs are off.

Testing an E-Input

If E1 is true --- On --- is written else Off is written :

```
void TestingAnInput() {
   do {
      if(ft.GetInput(ftiE1)) cout << "--- ON ---" << endl;
      else cout << "--- OFF ---" << endl;
      ft.Pause(555);
   } while(!ft.Finish(ftiE5));
}
```

Wait for an E-Input

If E1 is true ---- STARTED ---- is written :

```
void WaitForInput() {
   cout << "For start press E1" << endl;
   ft.WaitForInput(ftiE1);
   cout << "--- STARTED ---" << endl;
}
```

Display all E-Inputs

Continous display off all E-Inputs :

```
void DisplayAllInputs1() {
  do {
    for(int i = 0x80, E = ft.Inputs(); i > 0; i >>= 1) {
      cout << ((E & i) > 0) ? "1" : "0";
    }
    cout << endl;
    ft.Pause(1234);
  } while(!ft.Finish());
}
```

It's a nice very C-like, very short routine, but's a little bit sophisticated :

The for statement is nearly the whole program to transform the max. 16 E-Inputs to a printable bit stream. The index i contains a mask to select on E-Input, starting with E16, E contains all E-Inputs (ft.Inputs(), ft.GetInputs() is possible too, ft.Inputs is the quick one – not interruptable). for ends with i <= 0, each loop the mask is shifted one bit to right to give the next mask.
cout sequences the bit beginning with E16 and cout << endl; (behind the loop) resumes them.
DisplayAllInputs2() is nearly the same, it collects it in a string variable.

Display Analog-Inputs

Continous display of EX / EY :

```
void AnalogDisplay() {
  do{
    cout << "EX : " << ft.GetAnalog(ftiEX) << " EY : " <<
                       ft.GetAnalog(ftiEY) << endl;
    ft.Pause(1111);
  } while(!ft.Finish());
}
```

Notice : If using ft.GetAnalogScan instancing must be done with first parameter = true (AnalogScan). The reason is, that AnalogScan needs a greater PollInterval. In the case of the test frame AnalogScan is true, PollInterval = default.
If analog access is seldom, it is better to use ft.GetAnalogDirect(..). In this case AnalogScan can be false and PollInterval is smaller. While processing ft.GetAnalogDirect the polling is stopped.

Drive a Motor for a fix Time

```
void DriveForTime() {
  ft.SetMotor(ftiM3, ftiLeft);
  ft.Pause(3500);
  ft.SetMotor(ftiM3, ftiOff);
  cout << "Gone for 3.5 sec" << endl;
}
```

Drive to an End Switch

Motor on M3 will run to end switch E5 and than stop :

```
void DriveToEndSwitch1() {
  ft.SetMotor(ftiM3, ftiLeft);
  while(!ft.GetInput(ftiE5));
  ft.SetMotor(ftiM3, ftiOff);
  cout << "Switch E5 true" << endl;
}
```

The better solution is :

```
void DriveToEndSwitch2() {
  ft.SetMotor(ftiM3, ftiLeft);
  ft.WaitForInput(ftiE5);
  ft.SetMotor(ftiM3, ftiOff);
  cout << "Switch E5 true" << endl;
}
```

Motor can be stopped by ESC key.

Drive for a fix Number of Steps

WaitForChange

Motor on M3 with impulse switch on E6 will run 12 steps :

```
void DriveForStepsC() {
  ft.SetMotor(ftiM3, ftiLeft);
  ft.WaitForChange(ftiE6, 12);
  ft.SetMotor(ftiM3, ftiOff);
}
```

WaitForPositionDown

Motor on M3 drives from actual position 12 to destination position 0,
impulse count with E6 in direction of 0 (end switch = E5) :

```
void DriveForStepsD() {
  int ActPosition = 12;
  ft.SetMotor(ftiM3, ftiLeft);
  ActPosition = ft.WaitForPositionDown(ftiE6,
                ActPosition, 0, ftiE5);
  ft.SetMotor(ftiM3, ftiOff);
  cout << "ActPosition : " << ActPosition << endl;
}
```

The actual position after Motor is stopped is returned in ActPosition (may be one step more or less) . The method WaitForPositionDown can be stopped, if reaching E5 before the count position 0 is reached.

WaitForPositionUP

Motor on M3 drives from actual position 12 to destination position 24, impulse counting in direction leving the end switch :

```
void DriveForStepsU() {
  int ActPosition = 12;
  ft.SetMotor(ftiM3, ftiRight);
  ActPosition = ft.WaitForPositionUp(ftiE6, ActPosition, 24);
  ft.SetMotor(ftiM3, ftiOff);
  cout << "ActPosition : " << ActPosition << endl;
}
```

The actual after the method is noted in ActPostion.

WaitForMotors

Motor on M3 runs for 12 impulses on E2 with reduced speed to left :

```
void DriveForStepsW() {
  ft.SetMotor(ftiM3, ftiLeft, ftiHalf, 12);
  ft.WaitForMotors(0, ftiM3);
  cout << "ActPosition += 12" << endl;
}
```

The program waits for reaching the destination. If ft.WaitForMotors is dropped, the programm does every thing else and Motor will stop too, if destination is reached.

Two motors (M3 – full speed 121 impulse to left and M4 half speed 64 impulses to right. Impulse count on E6 / E8) simultaneously with contious display of the actual position :

```
void DriveForStepsW34() {
  ft.SetMotor(ftiM3, ftiLeft,  ftiFull, 121);
  ft.SetMotor(ftiM4, ftiRight, ftiHalf, 64);
  do {
    cout << "Position M3 - M4 : " << ft.GetCounter(ftiE6) << " - "
      << ft.GetCounter(ftiE8) << endl;
  } while(ft.WaitForMotors(300, ftiM3, ftiM4) == ftiTime);
    cout << "Position M3 - M4 : " << ft.GetCounter(ftiE6) << " - "
      << ft.GetCounter(ftiE8) << "   --- Final ---" << endl;
}
```

ft.WaitForMotors controls the position of M3 / M4 and return every 0.3 secs until ftiEnd or ftiESC. The loop is to display the actual position (ft.GetCounter(..);). If finished the final position is displayed. Notice a value 0 for waiting time means endless waiting (see sample before).

Light Barriers

Wait for broken Light Barrier

Light barrier with lamp on M1 an phototransistor on E1 :

```
void WaitForLightBarrierBroken() {
  const int mLight = 1, ePhoto = 1;
  ft.SetMotor(mLight, ftiOn);
  ft.Pause(555);
  ft.WaitForInput(ePhoto, false);
  cout << "LightBarrier M1 - E1 is broken" << endl;
}
```

Lamp is switched on, 0.5 waiting for 'warming up' the phototransistor, than waiting for a broken barrier.

Wait for Enter a Light Barrier

Barrier with M1 and E1, feeder motor M3 :

```
void WaitForLightBarrierEnter() {
   const int mLight = 1, mFeeder = 3, ePhoto = 1;
   ft.SetMotor(mLight, ftiOn);
   ft.Pause(555);
   ft.SetMotor(mFeeder, ftiLeft);
   ft.WaitForLow(ePhoto);
   ft.SetMotor(mFeeder, ftiOff);
   cout << "LightBarrier M1 - E1 is entered" << endl;
}
```

Light barrier is clear when program starts, feeder (with an parcel) runs until the parcel breaks the barrier.

Wait for Leaving a Light Barrier

Barrier with M1 and E1, feeder motor M3 :

```
void WaitForLightBarrierLeave() {
   const int mLight = 1, mFeeder = 3, ePhoto = 1;
   ft.SetMotor(mLight, ftiOn);
   ft.Pause(555);
   ft.SetMotor(mFeeder, ftiLeft);
   ft.WaitForHigh(ePhoto);
   ft.SetMotor(mFeeder, ftiOff);
   cout << "LightBarrier M1 - E1 is free" << endl;
}
```

Light barrier is broken when program starts, feeder (with a parcel) runs until the parcel is clear off the barrier.

Switching all M-Outputs at once

SetMotors can switch all M-Outputs with one stroke. Therefore the parameter MotorStatus must have the right values, 2 bit for each M-Outputs, right beginning with M1 : 00 00 00 00 mean all M-Outputs are off (with extension module additional four 00's). 01 means turn left, 10 turn right.10 01 00 00 for example M4 right, M3 left others off.

Traffic Lights

The phases are Green – Yellow – Red – RedYellow. The lamps are on M1 green, M2 yellow, M3 red. The constants needed : mGreen = 00 00 00 01 (0x1), mYellow = 00 00 01 00 (0x4) and mRed = 00 01 00 00 (0x10) :

```
void TrafficLights() {
   const int mGreen = 0x1, mYellow = 0x4, mRed = 0x10;
   while (!ft.Finish()) {
      ft.SetMotors(mGreen);
      ft.Pause(1000);
      ft.SetMotors(mYellow);
      ft.Pause(250);
      ft.SetMotors(mRed);
      ft.Pause(1000);
      ft.SetMotors(mRed + mYellow);
      ft.Pause(250);
   } }
```

List controlled Traffic Lights

Using a constant interval, it is possible to control the lamps by a list of switch values :

```
void TrafficLightsList() {
   const int mGreen = 0x1, mYellow = 0x4, mRed = 0x10;
   int Phase[] = {mGreen,  mGreen, mGreen, mGreen,
            mYellow, mRed,   mRed,   mRed,   mRed,   mRed + mYellow};
   while(!ft.Finish()) {
      for(int i = 0; i <= 9; i++) {
         ft.SetMotors(Phase[i]);
         ft.Pause(250);
      }
   }
}
```

In this case the interval is 0.25 secs. This prcedure is useful by mor complex applications.

Running Lights

If you have connected 4 lamps on the interface (all M1 .. M4) you can try a nice running light :

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

Phase is running index and MotorStatus. Only one lamp is switched on at the same time. To do that a 01 bit combination is shift through the MotorStatus, beginning with 01.

Reference

umFish30 – cs Type Functions

This version uses internally control blocks. The OpenInterface returns a handle to the actual one (CloseInterface frees it). This handle is used by all other function as a parameter instead of the control block. In cs style the number of functions is larger because of there is no direct possibility to access the control block.

The cs style is ought to be used with programming languages which can't operate with static structures. But it can be used with all programming languages, if it seems nicer to use it. A mix of both styles is not possible.

Used Variable Types

AnalogNr	Number of an analog input (EX = 0, EY = 1)
CounterNr	Number of an impulse counter (1-8(16))
Direction	Revolving direction of a motor (00 = ftiOff, 01 = ftiLeft, 10 = ftiRight) Left is the direction clicking an "L" on the interface panel.
ICount	Position count, counted in number of impulses (true/false or false/true), started on the actual position. Value without sign, direction is noted separate.
iHandle	Handle for the internal control block
InputNr	Number of a digital input (1 – 8(16))
InputStatus	State of all ditigal inputs. 1 bit for each input, start with 0 for E1 (1 = true, 0 = false).
LampNr	Number of a "half" output (1-8(16))
Mode	Operating mode of an output (0 : normal, 1 : RobMode)
ModeStatus	State of the operating modes of all outputs, 4 bit for each output, starting with 0 – 3 for M1. Values 0000 normal – 0001 RobMode.
MotorNr	Number of an output (1-4(8))
MotorStatus	Nominal state of all outputs. 2 bit for each output, started with 0 – 1 for M1 (00 = ftiOff, 01 = ftiLeft, 10 = ftiRight).
OnOff	Means true/false, but values (1-0)
PortNr	Number of the choosen port (0-11 : LPT, COM1-8, LPT1-3)
Speed	Speed running an output (0-15(ftiFull))
SpeedStatus	Speed of all outputs. 4 bit for each output, starting with 0-3 for M1, values 0000 – 1111 (ftiFull).
Value	32bit value

All variables have the type DWORD (32bit, unsigned).

Functions

iHandle	**csOpenInterface**(int PortNr, int AnalogScan, int Slave, int PollInterface);	Setting of parameters, setting up the connection to the interface. Returns a handle (or ftiFehler), used by all other functions to identify the internal control block. AnalogScan = 0 without, = 1 Scan of the analog inputs, Slave = 0 ohne, = 1 mit Slave Modul, PollInterval = 0 OpenInterface will use an default value, > 0 value to be used.
iHandle	**csOpenInterfaceEx**(int PortNr, int AnalogScan, int Slave, int PollInterface, int LPTAnalog, int LPTDelay);	like csOpenInterface but with additional parameters operating the parallel interface directly with LPT1-3. LPTAnalog = 0 OpenInterface will use an default value, > 0 value to be used, LPTDelay = 0 OpenInterface will use an default value, > 0 value to be used.
int	**csCloseInterface**(int iHandle);	Close the connection to the interface.
int	**csVersion**();	Read the umFish30.DLL version.
int	**csGetAnalogScan**(int iHandle);	Read Scanning of the analog inputs is true (0 = false, 1 = true)
int	**csGetLPTAnalog**(int iHandle);	Read analog scaling
int	**csGetLPTDelay**(int iHandle);	Read output delay.
OnOff	**csGetSlave**(int iHandle);	Read slave is available
int	**csGetPollInterval**(int iHandle);	Read PollInterval (msecs).
OnOff	**csGetInput**(int iHandle, int InputNr);	Read the value of a digital input.
int	**csGetInputs**(int iHandle);	Read all digital input values.
int	**csGetAnalog**(int iHandle, int AnalogNr);	Read the value of a analog input.
int	**csGetAnalogDirect**(int iHandle, int AnalogNr);	Read direct a analog EX/EY, polling is stopped for that time.
int	**csSetMotor**(int iHandle, int MotorNr, int Direction);	Write an output.
int	**csSetMotorEx**(int iHandle, int MotorNr, int Direction, int Speed);	Write an output, speed is included.
int	**csGetMotors**(int iHandle);	Read all outputs.
Mode	**csGetModeStatus**(int iHandle, int MotorNr);	Read the mode of an output.
void	**csSetModeStatus**(int iHandle, int MotorNr, int Mode).	Write the mode of an output

int	**csSetMotors**(int iHandle, int MotorStatus); Write all outputs
int	**csSetMotorsEx**(int iHandle, int MotorStatus, int SpeedStatus) Write all outputs, speed ist included. See also Notes for RobFunctions.
int	**csSetLamp**(int iHandle, int LampNr, int OnOff); Write a "half" output.
int	**csRobMotor**(int iHandle, int MotorNr, int Direction, int Speed, int ICount); Write an output if in RobMode
int	**csRobMotors**(int iHandle, int MotorStatus, int SpeedStatus, int ModeStatus); Write all outputs, mode can be choosen. The Counterss belonging to the outputs are to be set before this function.
int	**csGetCounter**(int iHandle, int CounterNr); Read an impulse counter.
void	**csSetCounter**(int iHandle, int CounterNr, int Value); Write an impulse counter.
void	**csClearCounters**(int iHandle); Clear all impulse counters.

class CFishFace

Frequently used Variables

The variable are mostly of the type int. Here is given a short description of those variable use in the CFishFace reference following. The enum values are use to describe the value range of the variables. The enum name is noted in brackets.

All parameters are value parameters.

AnalogNr	Number of an Analog-Input (Nr) EX = 0, EY = 1
Counter	Value of an Counter (int)
Direction	Revolving direction of a motor (Dir) ftiOff = 0, ftiOff = 1, ftiLeft = 1, ftiRight = 2
InputNr	Number of an E-Input (Nr) ftiE1 = 1 ... ftiE16 = 16
InputStatus	Actual state of all E-Inputs, one bit for each Input, beginning with 0 (0 = E1, 1 = E2 .... 15 = E16)
LampNr	Number of a 'half' M-Output (int) values 1 - 16
ModeStatus	Actual running mode of all M-Outputs.4 bit for each Output. Beginning with 0-3 for M1, value 0000 = normal, 0001 = RobMode
MotorNr	Number of a M-Output (Nr) ftiM1 = 1 ... ftiM8 = 8
MotorStatus	Actual state of all M-Outputs. 2 bit for an Output. Beginning with 0-1 for M1(00 = ftiOff, 01 = ftiLeft, 10 = ftiRight).
mSek	Time in milliseconds
NrOfChanges	Number of impulses (int)
OnOff	Switching Off/On an Output ftiOff = false, ftiOn = true
PortNr	Number of a Port (Port) LPT = 0, COM1 = 1 ... COM8 = 8, LPT1 = 9, LPT2 = 10
Position	Position given in number of impulses from the end switch (int)
Speed	Speed to run an M-Output (Speed) ftiOff = 0, 1 – 15 (ftiFull)
SpeedStatus	Actual speeds of all M-Outputs. 4 bit for each M-Output. Values 0000 – 0111(ftiFull).
TermInputNr	Number of an E-Input to cancel a method. (Nr) ftiE1 = 1 ... ftiE16 = 16
Value	Common int value

Enums

Used for symbolic name for method parameters.

Dir	Revolving direction of M-Outputs ...
Nr	Input / Output names
Speed	Speed names
Wait	Return values method WaitForMotors
Port	Portname

Alternativly numeric values (e.g. if storing than in tables) or own numeric constants can be used.

Constructor

FishFace()
Default values for AnalogScan = false, Slave = false,
PollInterval = 0, LPTAnalog = 0, LPTDelay = 0

FishFace(bool AnalogScan, bool Slave, int PollInterval)
PollInterval = 0 OpenInterface will choose a value, >0 this value is used.
LPTAnalog = 0, LPTDelay = 0

FishFace(bool AnalogScan, bool Slave, int PollInterval,
int LPTAnalog, int LPTDelay)
PollInterval = 0 OpenInterface will choose a value, >0 this value is used.
LPTAnalog, LPTDelay : handled like PollInterval

AnalogScan : To scan the analog-Inputs EX/EY set to true

Slave : With Slave/Extension Module set to true.

PollInterval : time distance in milliseconds in which the interfaces is polled. Value = 0 means to be set default value with OpenInterface. The actual value can be read (after OpenInterface) with the method PollInterval(). It can be modified with caution for a new instance. NOTICE : too small values can be a reason for 'hanging up' the whole system.

LPTAnalog : Scale factor for analog values (LPT1 / 2 only on Win9x systems)

LPTDelay : Output delay (LPT1/2 only on Win9x systems, values 50 – 1000).

Property like Methods

bool	**AnalogScan** (get) Scanning of the analog inputs (default = false)
int	**AnalogsEX** (get) Read EX value
int	**AnalgosEY** (get) Read EY value
int	**Inputs** (get) Read all E-Inputs
int	**LPTAnalog**(get) Read analog scaling (LPT only)
int	**LPTDelay**(get) Read output delay (LPT only)
bool	**NotHalt** Cancel request(default = false).
int	**Outputs** Read/write all M-Outputs (MotorStatus)
int	**PollInterval** (get) Interval (millisecs) for polling the interface
bool	**Slave** (get) Extension module connected (default = false)
string	**Version** (get) Version of the class

get : value only can be read, but not changed.

Methods

ClearCounter

Clear (0) an input counter.

ft.**ClearCounter**(InputNr)

See also : ClearCounters, GetCounter, SetCounter

ClearCounters

Clear (0) all Counters

ft.**ClearCounters**()

See also : ClearCounter, GetCounter, SetCounter

ClearMotors

Switch off all M-Outputs

ft.**ClearMotors**()

Exception : InterfaceProblem, KeinOpen

See also : SetMotor, SetMotors, SetLamp Outputs

CloseInterface

Close the connection to the interface

ft.**CloseInterface**()

See also : OpenInterface

Finish

Cancel request (NotHalt, Escape, E-Input(optional))

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

Exception : InterfaceProblem, KeinOpen.

See also : GetInput, GetInputs, Inputs

Example :

```
do {
   cout << "running" << endl;
  ft.Pause(2345);
} while (!ft.Finish(ftiE1));
```

The do .. while loop will run as long until ended with `ft.NotHalt(true);` ESC key is pressed or E1 becomes true. The loop will run once in each case.

Alternative :

```
while (ft.Finish(ftiE1) == false) {
   cout << "running" << endl;
  ft.Pause(2345);
}
cout << "--- FINIS ---" << endl;
```

This loop will be skipped if an cancel request comes with starting the loop. An while(!ft.Finish(ftiE1) is possible too.

GetAnalog

Reading an internal analog value (EX / EY, no access to the interface).
Parameter AnalogScan of the constructor must be true.

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

Exception : InterfaceProblem, KeinOpen

See also : GetAnalogDirect, AnalogsEX, AnalogsEY, AnalogScan,

Example :
```
cout << " EX : " << ft.GetAnalog(ftiEX) << endl;
```

GetAnalogDirect

Direct reading EX / EY from the interface. Therefore the polling is halted. Makes sense, if there are only few accesses to EX/EY. In this case AnalogScan can be false.

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

Exception : InterfaceProblem, KeinOpen

See also : GetAnalog, AnalogsEX, AnalogsEY, AnalogScan

Example :
```
cout << " EX : " << ft.GetAnalogDirect(ftiEX) << endl;
```

GetCounter

Read the value auf counter InputNr

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

See also : SetCounter, ClearCounter, ClearCounters

Example
```
cout << "Counter für E2 : " << ft.GetCounter(ftiE2) endl;
```

The actual counter value corresponding to E2 is displayed.

GetInput

Read the value of InputNr.

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

Exception : InterfaceProblem, KeinOpen.

See also : GetInputs, Inputs, Finish, WaitForInput

Example :
```
if (ft.GetInput(ftiE1)) {
  ...
}
else {
  ...
}
```

If E-Input E1 (switch, phototransistor, reedrelais) is true, the first block is executed. With `!ft.GetInput(ftiE1)` the else path will be executed. Possible too is i`f (ft.GetInput(ftiE1) == false)` { ... } or `if(!ft.GetInput(ftiE1)) {...}`

GetInputs

Read all E-Inputs

IntputStatus = ft.**GetInputs**()

Exception : InterfaceProblem, KeinOpen.

See also : GetInputs, Inputs, Finish, WaitForInputs

Example :
```
int E13;
  E13 = ft.GetInputs();
  if ((E13 & (0x1 + 0x4)) > 0) cout << "TRUE" << endl;
```
cout is executed if the expression is true (E1 and E3 must be true)
Alternative :
```
  if ((E13 & 0x1) > 0 || (E13 & 0x4) > 0) cout << "TRUE" << endl;
```

OpenInterface

Setting of CFishFace properties, connection to the interface

ft.**OpenInterface**(PortNr)

Exception : InterfaceProblem

See also : CloseInterface

Example :
```
try {
  ft.OpenInterface(ftiCOM2)
......
}
catch(FishFaceException eft) {
  cout << eft.Text() << endl;
}
ft.CloseInterface();
```
Connection to the interface situated on COM2. In case off Error the text '`InterfaceProblem.Open`' is displayed on console.

Pause

Stop the program execution for mSek millisecs

ft.**Pause**(mSek)

Exception : InterfaceProblem, KeinOpen; Can be canceled

See also : WaitForTime

Example :
```
ft.SetMotor(ftiM1, ftiLeft);
ft.Pause(1000);
ft.SetMotor(ftiM1, ftiOff);
```
Motor on M-Output M1 is running for 1 sec.

SetCounter

Set the counter for the noted E-Input.

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

See also : GetCounter, ClearCounter, ClearCounters

SetLamp

Set a 'half' M-Output. To connect a lamp or a magnet ... to a contact of a M-Output and ground.

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

Exception : InterfaceProblem, KeinOpen

See also : SetMotors, SetMotors, ClearMotors

Example :
```
const int lGreen = 1, lYellow = 2, lRed = 3;

  ft.SetLamp(lGreen, ftiOn);
  ft.Pause(2000);
  ft.SetLamp(lGreen, ftiOff);
  ft.SetLamp(lYellow, ftiOn);
```

The green lamp on M1 (in front) and ground is switch on for 2 secs and than the yellow on M1 back.

SetMotor

Set one M-Output (motor).

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

Exception : InterfaceProblem, KeinOpen;

See also : SetMotors, ClearMotors, SetLamp, Outputs.

Example 1
```
ft.SetMotor(ftiM1, ftiRight, ftiFull);
ft.Pause(1000);
ft.SetMotor(ftiM1, ftiLeft, ftiHalf);
ft.Pause(1000);
ft.SetMotor(ftiM1, ftiOff);
```

Motor on M1 is switched on for 1 sec, right revolving, full speed and than for 1 sec left revolving, half speed.

Example 2
```
ft.SetMotor(ftiM1, ftiLeft, 12, 123);
```

Motor on M-Output M1 runs for 123 impulses (counted on E2) with speed 12. Motor is stopped, if E1 is true before reaching the 123 impulses. The program doesn't wait for ready. See also example WaitForMotors.

SetMotors

Set the state of all M-Outputs. SpeedStatus default = 15 (ftiFull), ModeStatus = 0 (normal).

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

Exception : InterfaceProblem, KeinOpen;

See also : ClearMotors, SetMotors, SetLamp, Outputs

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

The M-Output M1 is set to ftiLeft and that of M4 to ftiRight. All other M-Outputs are ftiOff. After 1 sec running, all M-Outputs are stopped.

WaitForChange

Wait for NrOfChanges impulses on InputNr or TermInputNr = true

The counter of InputNr is used for counting impulses.

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

Exception : InterfaceProblem, KeinOpen; Can be canceled.

See also : WaitForPositionDown, WaitForPositionUp, WaitForInput, WaitForLow, WaitForHigh.

Example

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

M-Output M1 is started with ftiLeft and runs for 123 impulses on E2 or E1 = true, M1 is then turned off.

WaitForHigh

Wait for a false/true changing for InputNr

ft.**WaitForHigh**(InputNr)

Exception : InterfaceProblem, KeinOpen; Can be canceled

See also : WaitForLow, WaitForChange, WaitForInput.

Example

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

A light barrier with lamp on M1 and phototransistor on E-Input E1 is switched on. A Feeder with Motor on M2 is started, than waiting until a parcel on the feeder has leaved the light barrier (light barrier will be true (closed)). After this feeder motor is switched off. The light barrier must be false when starting this sequence.

WaitForInput

Wait for InputNr becomes OnOff. (default = true).

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

Exception : InterfaceProblem, KeinOpen; Can be canceled

See also : WaitForChange, WaitForLow, WaitForHigh.

Example :

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

Motor on M-Output M1 is started, than waiting for E-Input E1 becomes true. Motor is than switched off : Runnig to an end position.

WaitForLow

Wait for a true/false changing for InputNr

ft.**WaitForLow**(InputNr)

Exception : InterfaceProblem, KeinOpen; Can be canceled

See also : WaitForChange, WaitForInput, WaitForHigh.

Example :
```
ft.SetMotor(ftiM1, ftiOn);
ft.SetMotor(ftiM2, ftiLeft);
ft.WaitForLow(ftiE1);
ft.SetMotor(ftiM2, ftiOff);
```
A light barrier with lamp on M1 and phototransistor on E-Input E1 is switched on. A Feeder with Motor on M2 is started, than waiting until a parcel on the feeder will enter the light barrier (light barrier will be false (opened)). After this feeder motor is switched off. The light barrier must be true when starting this sequence.

WaitForMotors

Wait for a MotorReady event or for timeout of Time.

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

Time (int) : time in millisecs. Time = 0 endless waiting : all motors of the list are off.

MotorNr(Nr) : List of M-Outputs in any order to be waiting for stop.

WaitWert(Wait) : reason why the method has ended
ftiEnde : all requested M-Outputs are ftiOff
ftiTime : the requested waiting time is off
ftiNotHalt : NotHalt = true, all requested motors are stopped.
ftiESC : ESC key was pressed, all requested motors are stopped.

Exception : InterfaceProblem, KeinOpen; Can be canceled.

See also : SetMotor

Example :
```
ft.SetMotor(ftiM4, ftiLeft, Speed.Half, 50);
ft.SetMotor(ftiM3, ftiRight, Speed.Full, 40);
do {
  cout << ft.GetCounter(ftiE6) << " - " <<
               ft.GetCounter(ftiE8) << endl;
} while (ft.WaitForMotors(300, 4, 3) == ftiTime);
cout << ft.GetCounter(ftiE6) << " - " <<
             ft.GetCounter(ftiE8) << endl;
```
Motor on M-Output M4 starts with half speed, left for 50 impulses, that on M3 with full speed, right for 40 impulses. The do while loop waits for reaching the position (ft.WaitForMotors). Every 0.3 secs the actual postion is display within the loop (300 ... ftiTiime). If position is reached (<> ftiTime), the job is done, motors have stopped already. The final position is displayed.
Notice : The loop can be broken by NotHalt or ESC key, that is not controlled.

WaitForPositionDown

Wait for reaching the (destination)Position, by decrement the (actual)Counter:

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

Exception : InterfaceProblem, KeinOpen; Can be canceled

See also : WaitForPositionUp, WaitForChange

Example :
```
int Zaehler = 12;
  ft.SetMotor(ftiM1, ftiLeft);
  ft.WaitForPositionDown(ftiE2, ref Zaehler, 0, ftiE1);
  ft.SetMotor(ftiM1, ftiOff);
  cout << "Counter : " << Zaehler << endl;
```

The actual position is 12 (Zaehler), motor on M1 is started left. WaitForPositonDown is waiting for reaching position 0, motor is stopped than. Same is done, if E1 becomes true.

WaitForPositionUp

Wait for reaching the (destination)Position, by increment the (actual)Counter.

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

Exception : InterfaceProblem, KeinOpen; Can be canceled

See also : WaitForPositionDown, WaitForChange

Example :
```
int Zaehler = 0;
  ft.SetMotor(ftiM1, ftiRight);
  ft.WaitForPositionUp(ftiE2, Zaehler, 24);
  ft.SetMotor(ftiM1, ftiOff);
  cout << "Counter : " << Zaehler << endl;
```

The actual position is 0 (Zaehler), motor on M1 is started right. WaitForPositionUp ist waiting for reaching position 24, motor is stopped than.

WaitForTime

Stop the program execution for mSek millisecs

ft.**WaitForTime**(mSek)

Same as Pause

Exception : InterfaceProblem, KeinOpen; Can be canceled.

See also : Pause

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

Loop do ... while switches first M-Output (lamp) M1 on and all others off (00 01), wait for 555 millisecs, M2 (lamp) switched on (all others off, 01 00) and waiting for 555 millisecs. Result is a alternating blinker. Loop ends with pressing the ESC key.

Overview

Notes to the Counters

An essential element of determining the position are the counters. There is a counter for each E-input (attention : E1 in some languages is 0 in others is 1). The counters will notify (and count) each change of the state of an input (e.g. opening or closing a switch).

The counter are part of the control block and can read from it. In cs style there are special functions. The counter are used internally be some functions (e.g. SetMotor with Counter parameter and most of the Wait methods. umFish30.DLL uses the counters only with um/csRobMotor(s)).

Notes to the Speed Control

The speed control is based on a cyclic switch on/off of the M-outputs (PWM). For that reason internally there is a list of switch commands. The speed is determined by the parameter Speed (SetMotor) and SpeedStatus (SetMotors). The speed control is located in a separate thread of umFish30.DLL which controls the motors in this manner until they are switched off by SetMotor(s).

Notes to the Rob Functions

The Rob functions are running in a special operating mode, the RobMode. In this mode the involved counters are decreased. Reaching the value 0, the motor belonging to that counter is switched off. On the last 6 impulses they will operate with half speed to have a more exact positioning. Sometime it may happen one more impulse is counted. It can determined by read the counter. Values > 0 signal a plus position. They actual position should be corrected.

Operating of a motor in RobMode uses a fix concept of wiring the motors. Each motor is associated with an end switch and an impulse switch :

Motor	**End Switch**	**Impulse Switch**
1	1	2
2	3	4
3	5	6
4	7	8
5	9	10
6	11	12
7	13	14
8	15	16

The motors turn "left". That means they run to the end switch if operated in direction ftiLeft. Motors are switched off, if they are reaching the end switch before the counter is zero.

The motors can be operated with umRobMotor/csRobMotor/SetMotor (a single motor). The parameter lCount/Counter noted the way to go in items of impulses (a true/false or false/true signal on the appropriate impulse switch). The impulse switches are decreased during polling. They can be accessed via ftiDCB.Counter or the function csGetCounter. Note Counter set by the application will be changed in this turn.

The motors can be operated all together with one function : umRobMotors / csSetMotors / SetMotors. Therefore the parameters must be prepared in the following manner :

MotorStatus : each motor 2bit, starting with M1 : bit 0 and 1.
00 : off, 01 left, 10 right.
SpeedStatus : each motor 4bit, starting with M1 : bit 0-3,
0000 off, 1000 half speed, 11111 full.
ModeStatus : each motor 4 bit, starting with M1 : bit 0-3,
0000 Normal-Mode, 0001 Rob-Mode, others free for further use.
(may be stepper motors).

Example : csRobMotors(ft, 0x9, 0xF6, 0x11);
0x means Hexa, binary : 1001 | 11110110 | 10001 -> M2 = right, speed 15, Rob-Mode, M1 = lelft, speed 6 RobMode. Other motors are off.

Before operating the motors, the counters are to be set for each involved motor.

Direction = 0 or the appropriate bit value in MotorStatus overrides the speed parameter, motor is stopped.

The motors are running simultaneously (up to 8 motors). They can be switched one after the other by umRobMotor/csRobMotor. They will started with the next polling cycle and run asynchronous (that means independent of the actions of the application) until they have reached the mentioned positon. Than they are switched off during the normal polling.

To observe, the motors reaching their position and to synchronized the application a WaitForMotors can be used. The FishFace classes own such a method. umFish30.DLL offers none.

Structure of C/C++ Corner

Beginning on sitemape.htm

- ccppe.htm : english version of ccpp.htm : overview
- ccthreade.htm : - , - using umFish.DLL in threads.
- ccfishe.htm : -,- Details to umFish30.DLL
- ccfirste.htm : HelloFish - a little console example
- umFish30NotesE.PDF : Shortreference for some supported languages
- ccFish30e.PDF : this document
- ccFish30.ZIP : ccFish30e.PDF and the sources (projects) for it.

Sample projects

All sample projects are console projects for VC++ 6.0

HelloFish

HelloFish with the cs style interface to umFish30.DLL

TipsTriCS

Some console samples for the cs style interface to umFish30.DLL

TipTriC

CFishing : A short introduction for constructing a simple class based on umFish30.DLL

FiFa30CCP

The complex, more professional, class CFishFace which capsulate the umFish30.DLL functions and adds some useful more complex methods.