# C/C++ Corner

ftComputing : Programme für die fischertechnik-Interfaces und -konstruktionskästen



*ftComputing.de*

Home
Back
HelloFish
About Threads
umFish30 Notes

Sitemap
Index
Links
Impressum

Mail

## General

This page will give a short overview were C/C++ items are to be found (preferred english pages, but german too) and will give in addition some explanations to selected items.

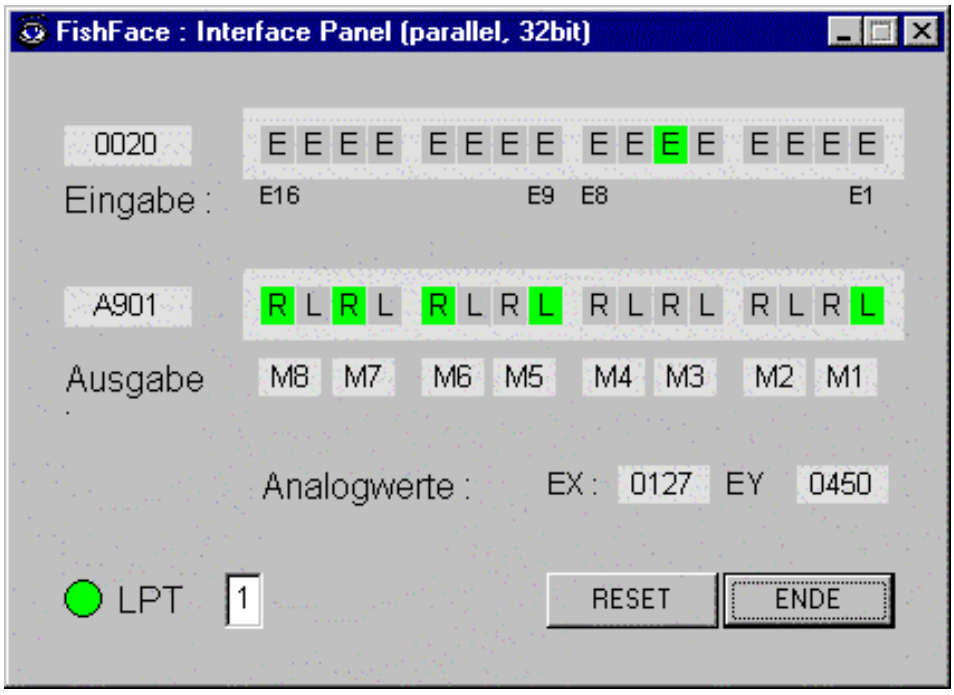
## Programming

Programming is based on umFish30.DLL (programmed in VC++ 6.0, contained in zip-file umFish30.ZIP). The ZIP contains Declarations, the class TFishFace and examples for VC++ Console- and C++Builder Windows programming. An english documentation for umFish30.ZIP (not special for C/C++) you will find in umFish30NotesE.PDF. English "Notes and Overview to umFish30 for C/C++" (PDF format) are here. A ZIP-file containing the "Notes" and the sources for the discussed examples is ccFish30.ZIP.

- **umFish30VC.H / umFish30.LIB** static declarations for direct access to the umFish30 functions.
  Note : umFish30.LIB has VC++ 6.0 specific format, therefore it only can be used with VC++ 6.0.
- **umFish30Load.H** dynamic loading of the umFish30 functions. Can be used with VC++ (all versions) and C++Builder.
- **FishFa30.H/CPP** an C++ class using umFish30.DLL / umFish30Load with some additional, more sophisticated functions and an exception handling. For use with the C++Builder.

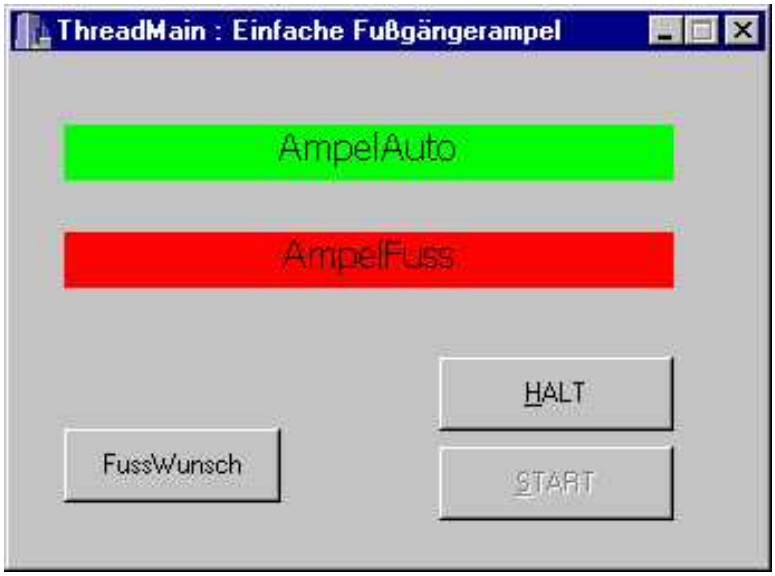
A HelloFish program using umFish30 with cs style functions is here.

## InterfacePanel




The InterfacePanel is an separate tool for controlling the Interface. Especially it can be used for testing the model connections and to give the model a proper standing after a program abort. The InterfacePanel should be the first program after installation to control the interface to work fine. The InterfacePanel comes with umFish30.ZIP.

## Model Programs




**AmpelThread** : Operating traffic lights (pedestrians) on an Intelligent Interface with extension module. Operating the car and the pedestrian lights in separate threads. C++Builder4 Win Program.

**Stamping machine** : Operationg the model Stamping Machine with Feeder (51 663) C++Builder4 Win Programm.


**RobStamping** : Feeding the Stamping Machine 51 663 with an Industry Robot. Feeding runs simoultaneously in its own thread. C++Builder Win Program.

The sources are contained in cb4Model.ZIP.

## Details / Notes

- Notes for programming with Threads
- Notes to umFish30.DLL

Last Update : 10.08.2004

# HelloFish

ftComputing : Programme für die fischertechnik-Interfaces und -konstruktionskästen





## HelloFish for umFish30 cs style

The following HelloFish version is designed for use with VC++ 6.0

### Setting up the VC++ Project

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

Some more details for a new VC++ 6.0 console project :

- New Workspace : Console
- Be sure umFish30.DLL can be accessed to be situated in directory \Debug or \WinNT\System32
- Add to the project : umFishVC.h, umFish30.lib, umFish30VC.cpp
- Compile (F7)

### Connect the Interface

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

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

## The Source

The interesting parts of umFish30VC.cpp

```
#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 <iostream.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.

Last Update : 10.08.2004

# About Threads

ftComputing : Programme für die fischertechnik-Interfaces und -konstruktionskästen





## General

umFish30.DLL separates the access to the interface resources and the access of the application. Therefore umFish30 can be used with threads. Separation is done by placing the interface accesses in a special poll thread. The poll thread is controlled by the MultiMediaTimer. The poll thread accesses interface in fix intervals and places the belonging values in a special (internal) control block (the ftiDCB). The application reads and writes this ftiDCB asynchronously.

This technique anables different threads to access the interface (if it is needed : with extension module) without problems. The original access to the unique resource only happens once by the one poll thread. The access to single E-Inputs / M-Outputs can be done without additional work. Writing to all M-Outputs with one function (SetMotor**s**) requieres an masking to save the M-Outputs of other threads.

The following examples refer to C++Builder programs with thread classes inherited from VCL class TThread. Working directly with Win-API functions is possible in the same manner, there are some additional possibilties (look to Robot-Plant).

## Program RobStamp

The Programs **AmpelThread** and **RobStamp** are constructed with the same schema. AmpelThread is very much easier to handle (there is robot to crash against the stamp) and should be the first program to try. RobStamp is more interesting, therefor it is discussed here. The actual program AmpelThread uses the Intelligent Interface with extension module connected to COM1, RobStamp uses COM1 and COM2 (corresponding to the exsisting models).

| | |
|---|---|
| frmThreadMain | Der main thread containing the GUI and the general control |
| ftR | Instance of the class TFishFace for robot control |

| | |
|---|---|
| ftS | Instance of the class TFishFace stamp control |
| umFish30.DLL | At the background : the Poll Thread |
| RobReady | Event : Robot has place one piece<br>Standard Security, autoReset, not signaled |
| StanzeReady | Event : Stamp can process the next piece<br>Standard Security, autoReset, not signaled |
| RobThread | Thread for robot control |
| StanzThread | Thread for stamp control |

## TfrmThreadMain::cmdActionClick

- **Create ftR and ftS instances** :
  ```
  ftR = new TFishFace(false, false, 0);
  ftS = new TFishFace(false, false, 0);
  ```
  no AnalogScan, no Slave, with default PollInterval
- **Connection to the interfaces** :
  ```
  ftR->OpenInterface("COM2"), false);
  ftS->OPenInterface("COM1", false);
  ```
  no interrupt by Application.ProcessMessages();
  using a try - catch construct to detect open errors. Some more are useful, but you always will hear it, if it crashes.
- **Create the threads for Rob/Stamp**
  ```
  RobThread = new TRobThread(true);
  StanzThread = new TStanzThread(true);
  ```
  without starting them
- **Thread end functions**
  ```
  RobThread->OnTerminate = RobEnde;
  StanzThread->OnTerminate = StanzEnde;
  ```
- **Starting the threads**
  ```
  RobThread->Suspended = false;
  StanzThread->Suspended = false;
  ```

## TRobThread::Execute

Working method of the threads.

- **Drive to Home Position**
  ```
  ftR->SetMotor(mSaule, ftiLinks, 15, 999);
  ftR->SetMotor(mArmV, ftiLinks, 15, 999);
  ftR->SetMotor(mArmH, ftiLinks, 15, 999);
  ftR->SetMotor(mGreifer, ftiLinks, 15, 999);
  ftR->WaitForMotors(0, mSaule, mArmV, mArmH,
  mGreifer);
  ```
  The four motors of the robot are started simultanously with full speed (15) to run for 999 impulses or until reaching the end switch. The real maximum of impulses may be 200, that means end switch will come first in any case. WaitForMotors is waiting for all end switches are true.
- **Processing loop**

```
do {} while(!Terminated)
```

runs until the thread geht from outside the thread an end request.

- **Inside the processing loop**
  frmThreadMain->lblStatus->Caption = "...."
  Anzeige des aktuellen Status
  ```
  ftR->SetMotor(...
  ftR->SetMotor(...
  ft->WaitForMotors(0, ...);
  ```
  Processing the single steps : fetch the piece, go to deposit place on the feeder.
- **Wait for StanzeReady**
  ```
  while(!StanzeReady-WaitFor(100) == wrSignaled);
  ```
  Deposit the piece on the feeder after signal
  then signal RobReady.
  ```
  RobReady->SetEvent();
  ```
  Back to the store without a stop.
- If receiving a **Terminate-Request**
  Drive to a resting positition.

## TStanzThread::Execute

Working method of the threads, similar to TRobThread.

- **Drive to Home Position**
- **Processing loop**
- **Inside the processing loop**
  At the beginning Wait for **RobReady** and there after signal **StanzeReady**
- **Continue the processing loop**
- After getting a **Terminate request**
  Clear the lamps.

## TfrmThreadMain::cmdEndeClick

An artificial ending the stamp processing :

- **Terminate request to robot**
  ```
  RobThread-Terminate();
  StanzeReady-SetEvent();
  RobThread-WaitFor();
  ```
  The terminate request is not noticed by the thread, because the thread is waiting for StanzeReady. That is done by setting a special StanzeReady. After that the RobThread will end the thread normaly.
- **Terminate request to Stamp**
  analog robot
  remember : one piece is requested for
  ```
  ftS->WaitForLow(ePhotoV);
  ftS->Pause(1234);
  ```
- Cleaning

## Notes

**Using threads** : umFish30.DLL see above. VCL (Windows GUI) mostly, sometimes it is better to use Synchronize(.....); or the use of special components (e.g. TThreadListe ...).

**Global Variables** : frmThreadMain, ftR, ftS are some and very nice to use and therefore nothing for OOP purists. frmThreadMain comes from VCL, but ftR, ftS can be an parameter for the thread constructor.

Last Update : 10.08.2004

# umFish30 Notes

ftComputing : Programme für die fischertechnik-Interfaces und -konstruktionskästen





## General

umFish30.DLL is intended to support a greater number of programming languages with functions to operate the fischertechnik interfaces. The operating system ist Windows 32bit.

There are to different interface for nearly the same functions the um style with some few functions and the access to an control block (ftiDCB) for the properties. The cs style interface has a greater number of functions an only an handle to an internal ftiDCB which can't access only via a function. This is used by languages, which can't access structures or have a special garbage collection which can't accept a fix address for an control block (like C # - CSharp).

To recognize (nearly) all changes on the E-Inputs, they are polled in a sperate thread under control of the MultiMediaTimer CALLBACK routine. The application will read the values of the E-Inputs (indirectly) from the ftiDCB. This serves in addition to the request of an refresh of the M-Output at least every 0.3 sec.

The CALLBACK routine has some more functions for counting all changes of the E-Inputs, controlling the on times off an M-Output (PWM - speed control) and the control

of an special impulse counter to stop an M-Output after reaching the requested number of impulses, same is done if the special end switch is true.

The sources for umFish30.DLL are located in umFish30.ZIP.

# Structure

About names : Only function with the prefix um or cs are external functions.

## Access to the Interface

### Connection to the Interface

The connection to the interface is done with the function **umOpenInterface**. The function itself goes more in details beside of interface and operating system :

- umOpenInterface
  - OpenInterfaceCOM : Intelligent Interface
  - OpenInterfaceLPT : Universal (parallel) Interface (direct)
    - OpenInterfaceRT : access via driver

umOpenInterface makes the default settings for ftiDCB.

umCloseInterface closes the connection to the interface (CloseInterfaceRT if accessed via driver).

Ther is an control block ftiSave which contains the data of the last opened ftiDCB. With unload of umFish30.DLL by the operating system in DllMain a "forgotten" umCloseInterface of the application can be done here.

## Read of the E-Inputs - Refresh of the M-Outputs

GetInputs is the central internal routine to do this job :

- GetInputs
  - GetInputsCOM1 : Intelligent Interface only
  - GetInputsCOM2 : with extension module
  - GetInputsRT : via driver
  - GetInputsLPT : direct access

GetInputs first transfers ftiDCB.OutputStatus and than reads the values of all E-Inputs to ftiDCB.InputStatus.

## Read of the Analog-Inputs

GetAnalog is the central internal routine to do this job :

- GetAnalog
  - GetAnalogCOM1 : Intelligent Interface only
  - GetAnalogCOM2 : with extension module
  - GetAnalogRT : via driver
  - GetAnalogLPT : direct access

GetAnalog fills ftiDCB.Analogs[] and Intelligent Interface only, the ftiDCB.InputStatus.

## Polling the Interface

The olling of the interface is controlled by the MultiMediaTimer, it is done with the CALLBACK-Routine PollInterface. At the beginning of the routine the parameter DWORD DCB is converted in a better usable form :

```
ftiDCB *d = (ftiDCB*)DCB;
```

## Testing of the E- and Analog-Inputs, Refresh M-Outputs

save of the InputStatus : `StatusAlt = d->InputStatus;`

Set StatusNeu

- Intelligent Interface :
  - `GetInputs(*d);` if `AnalogScan = 0`
  - changing `GetAnalog(*d, 0/1);` if `AnalogScan = true`
- Universal (parallel) Interface :
  - `GetInputs(*d);`
  - chhanging `GetAnalog(*d, 0/1);` if `AnalogScan = true`

Determining StatusDelta : `StatusDelta = StatusAlt ^ StatusNeu;`

Set ftiDCB : `d->InputStatus = StatusNeu;`

## Loop for each M-Output

The following operations will be executes in a loop for all available M-Outputs (Indices : `iMot, iEnd, iImpuls`)

## ImpulseCounter : Couting of the Impulses on the E-Inputs

Done if NormalMode (MotMode, separate for each M-Output)

```
if(StatusDelta & EMaske[.] > 0) d-Counters[.]++;
```

## Position : ImpulsCounter-Control

Done if RobMode (ModMode, separate for each M-Output)

```
if(StatusDelta & EMaske[.] > 0) d-
Counters[.]--;
```

Braking if Counter < 6 : `d->SpeedStatus = ...`

## Speed : Controling the On Time of the M-Outputs

Done with Normal- and RobMode, if M-Ausgang is on (SollDirection >0) and Speed < Full and > off.

`d->OutputStatus &= MAus[.];` M-Output off
`SpeedValue = ...;` according the OnOffTab[.] M-Output on / off.
`d->OutputStatus |= AktDirection;` new OutputStatus

## Application : Access to the Controlblock ftiDCB

The fucntion with the prefix um or cs can be used by the application alternatively. In many cases they only copy values from or to the ftiDCB. Function which access single E-Inputs or M-Outputs mask the corresponding ftiDCB.field. She are comfort functions. But not easy to handle e.g. in the case of Speed-Control. In this case the common routine SetMotorAll is used.

A special thing is the function GetAnalogDirect. Here the polling is stopped uring reading the Analog value. Reason : Accessing the EX/EY lasts more time then accessing the other resources. Otherwise the poll interval must be increased.

The um style functions need a parameter ftiDCB, to be situated in the application.

The cs style function need handle to the internal ftiDCB as parameter. Internally they access the corresponding um styl functions or access directly the ftiDCB.

csOpenInterface(Ex) is called without handle. It returns a handle to be used with the following functions. As parameter some ftiDCB values are in use.

# Details

## MultiMediaTimer

The MultiMediaTimer calls at fixed time intervals CALLBACK routine PollInterface. He is started in umOpenInterface :

```
Interface.FID = timeSetEvent(
   Interface.PollInterval,  // Time
interval
   0,                       // with older
systems :
                            // max value
for that system
   PollInterface,           // Address of
the CALLBACK routine
   DWORD(&Interface),       // address of
the ftiDCB as DWORD
   TIME_PERIODIC);          // Periodic
call
```

The CALLBACK-Routine uses the following parameters :

```
void CALLBACK PollInterface(UINT
wTimerID, UINT msg, DWORD DCB, DWORD
dw1, DWORD DW2)
```

The most interesting parameter is DWORD DCB, which is copied to an ftiDCB pointer on the beginning of the routine.

```
ftiDCB *d = (ftiDCB+)DCB;
```

umCloseInterface cancels the polling :

```
if(Interface.FID != 0)
timeKillEvent(Interface.FID);
```

Because of the CALLBACK routine runs in its own thread, a reliable cancel of the MultiMediaTimer is important (see also umCloseInterface).

Last Update : 10.08.2004