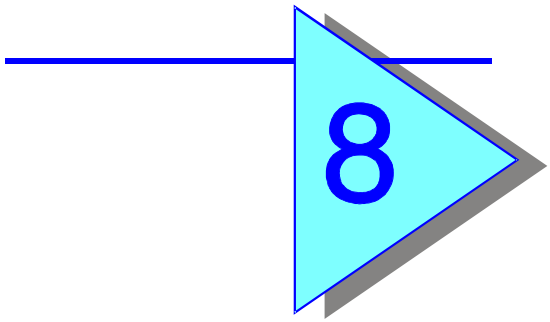


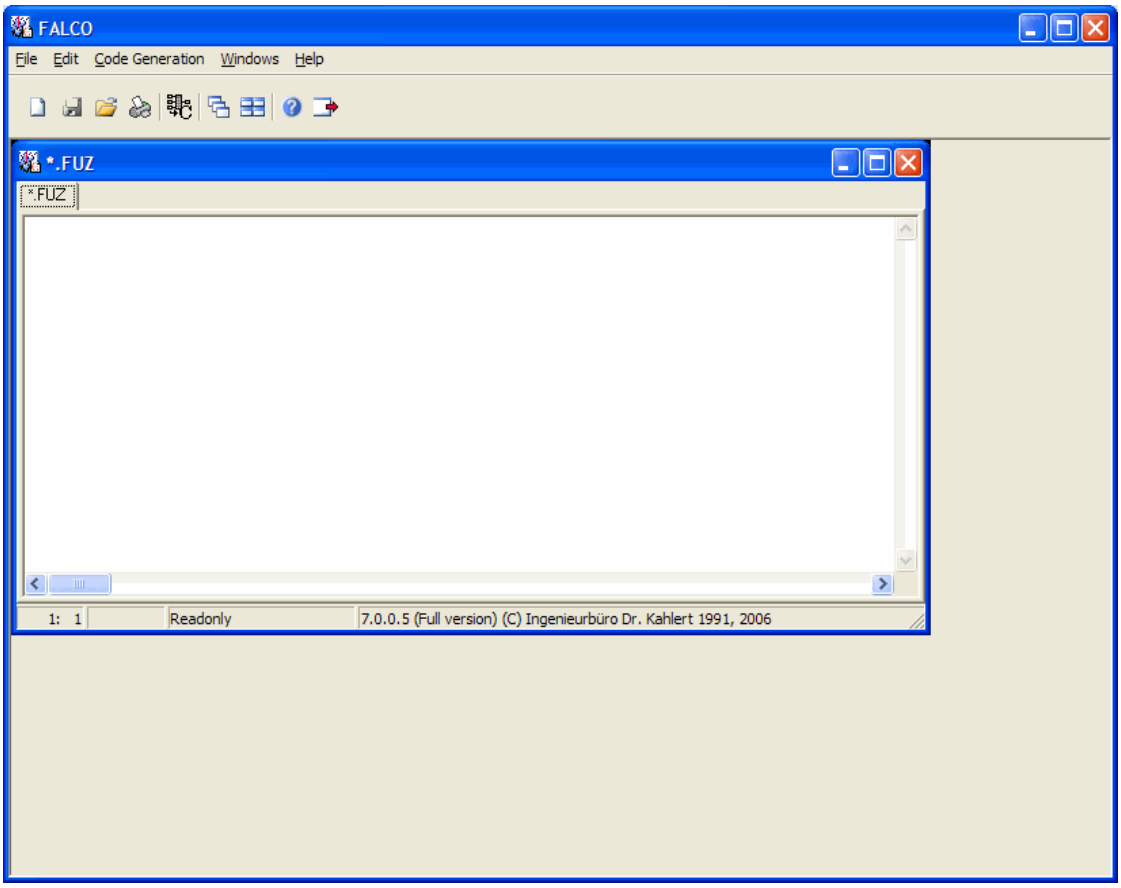
The fuzzy C-code generator FALCO

Overview

The C source code generator FALCO (the name FALCO stands for "Fuzzy Application C code generator) makes the production of ANSI C code for Fuzzy systems possible, which were developed with the help of the WinFACT Fuzzy Shell FLOP. The C source code is divided into two files. One of them has the ending C and contains the implementation, the other one is the associated header file (extension h); it defines the interface for your own application. FALCO is characterised by the following capabilities:

- Comfortable editor for files of the programming language C.
- Parallel processing of several systems possible.
- The generated C code exists completely as the source code. It is commentated and held well readably. By fragmentation into different files the later time of compiling your application is reduced.
- The data types are freely selectable by the user (16 bit integer, 32 bit integer, 2 byte float, float, double, long double). With the integer types you can define fixed point format of numbers. With the 2 byte float type the mantissa and exponent bit width is user defineable.
- By code templates main functions written by yourself can be used for arbitrary Fuzzy systems.

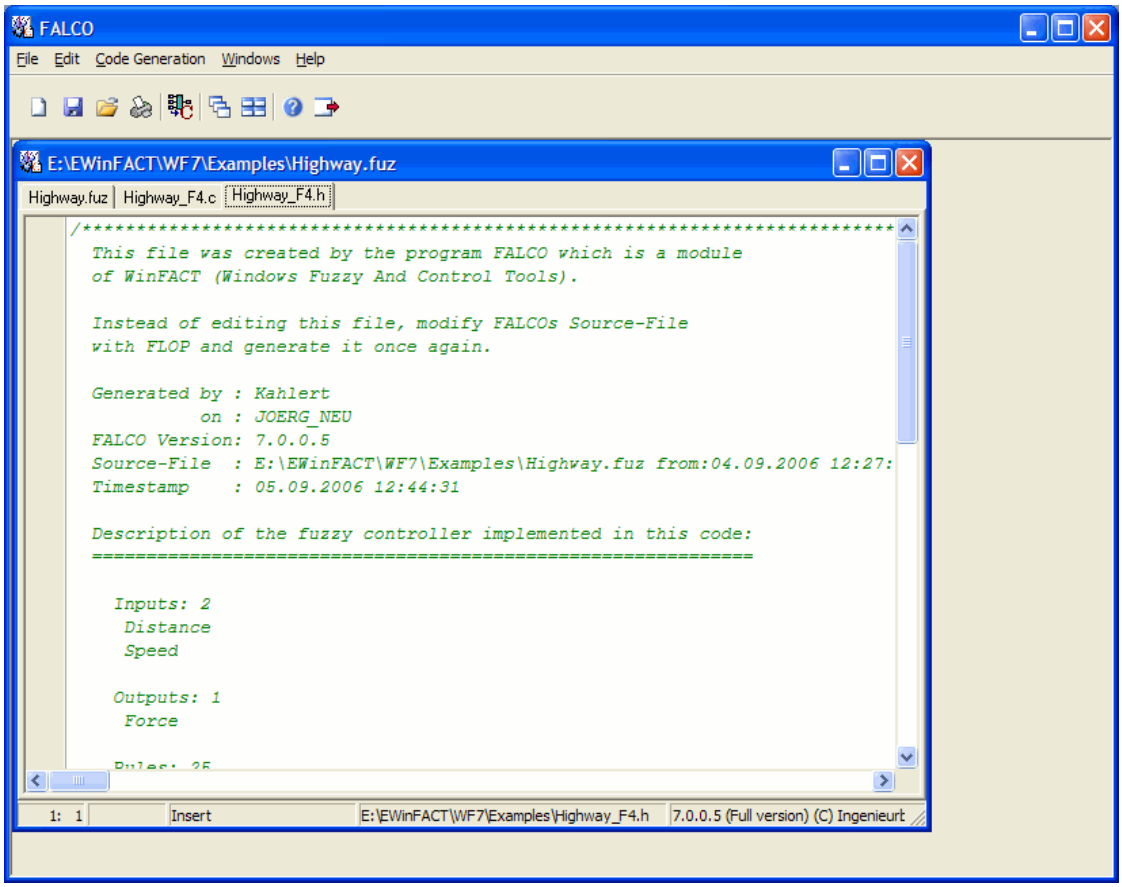
The application FALCO was designed as MDI application (*multiple document interface*). This has the advantage for you to view and validate several code productions at the same time. The following graphic shows the application window of FALCO immediately after start.



FALCOs main window

Besides some standard components the window contains

- a horizontal toolbar below the main menu for direct access to the most important functions,
- one or more document windows which contain a fuzzy file and the generated C code files.



Document window with generated code

The status line at the bottom of the document window shows the current position of the cursor, the state of the document, the mode (*Insert*, *Override*, or *Read only*), the full name of the file and the software version. The document window itself has an editor, which allows comfortable reading and changing of code files. For changing there are all known main functions in the menu EDIT available. Editing FUZ files is not allowed here (you should use the program FLOP). If you select a document window with a FUZ file, the entries of the menu EDIT become disabled. The functions for editing are the following:

Edit operations

- undoing last operation,
- redoing last undone operation,
- cutting, copying and pasting of text and
- searching for and replacing of text.

Generation of C code

Activating the code generation

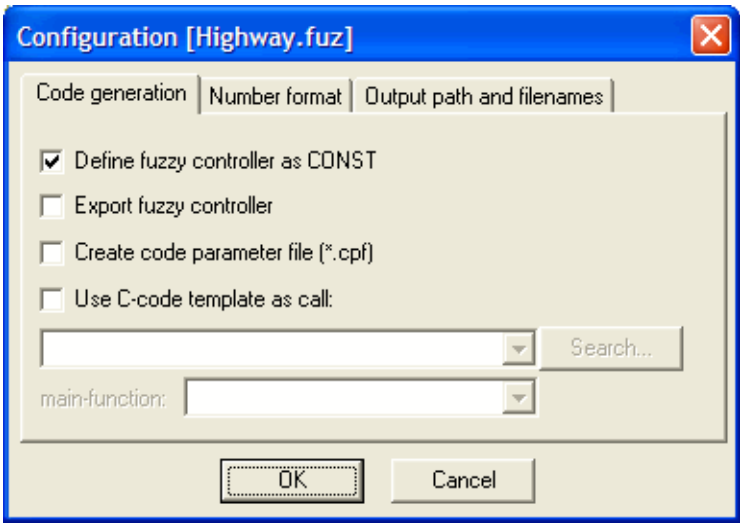
It is only possible to start the code generation, if a FUZ file was loaded in the active document window. The settings for code generation refer to the active document window and can be made without a FUZ file. When settings are made and a FUZ file was loaded, you can start the code generation via the menu item CODE GENERATION | GENERATE CODE or by clicking the  button.

Opening FUZ files

The fuzzy system which is to be processed, must be in a file with the extension *FUZ*, like it is generated by the WinFACT fuzzy shell FLOP at saving. The file can be loaded by clicking the  button or via the menu item FILE | OPEN... into the active document window. A previously loaded file would be overwritten in the document window thereby. If code productions were generated for this file, you are asked whether these should be saved. If you want to use a new document window for the fuzzy system, this must be created by FILE | NEW first.

Settings

Before the generation of C code, usually some settings are to be made, those concern among other things the resolution, i. e. the used C data type for the linguistic variables and membership values. All settings are made via CODE GENERATION | SETTINGS... The configuration of the code generator carried out in this dialog applies only to the active document window. All others remain uninfluenced.



Dialog for configuration of the code generator

The configuration dialog of the code generator contains three registers:

- Code generation
- Number format
- Output path and filenames

Code generation

The register for the code generation offers the possibility to you of defining the fuzzy controller as constant. This means that all values, which constitute the controller, will have constant definition in the produced C code. With the next check box you can specify whether the fuzzy controller is to be exported. If you decide for export, the appropriate export definition will be written into the header file. The further points of this register are discussed in detail later. They are briefly outlined here.

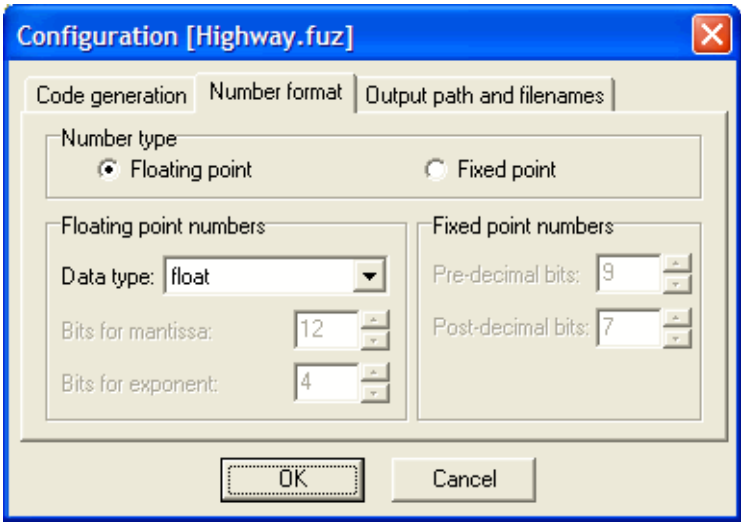
The type and variable identifier of a data structure defined in a C-file for example could be assumed as parameters of another code sequence in which you would like to use the structure. Thus the code sequence, which uses these parameters, has validity for all files, which define these parameters. This basic approach is pursued by the code parameter files and the C code templates. If there is a need for a code parameter file outside FALCO, it can be created by choosing the check box *Create code parameter file*.

By the use of a C code template you can achieve the call of the produced C functions. In addition a CSH file (code sheme file) must be specified, which contains at least one main function (the chapter *Creating programs for arbitrary fuzzy controllers* describes the use of code pattern files in detail with an

example). As soon as you entered a CSH file into the appropriate entry field (this can be done more comfortably by the button *Search...*), the available main functions will appear in the selection box below.

Number formats

The code generator is able to use different formats of numbers. The selected number format will be used by all real valued parameters of the fuzzy controller. All other parameters are represented by integers or enumerations. Via the register *Number format* of the configuration dialog the appropriate format can be set.



Number formats for code generation

The following two basic number formats are possible:

Data types

- Floating point numbers
- Fixed point numbers

Floating point numbers let you choose within all native C data types (*float*, *double*, *long double*). Additionally a two byte floating point type is offered, which has a selectable number of the mantissa and exponent bits.

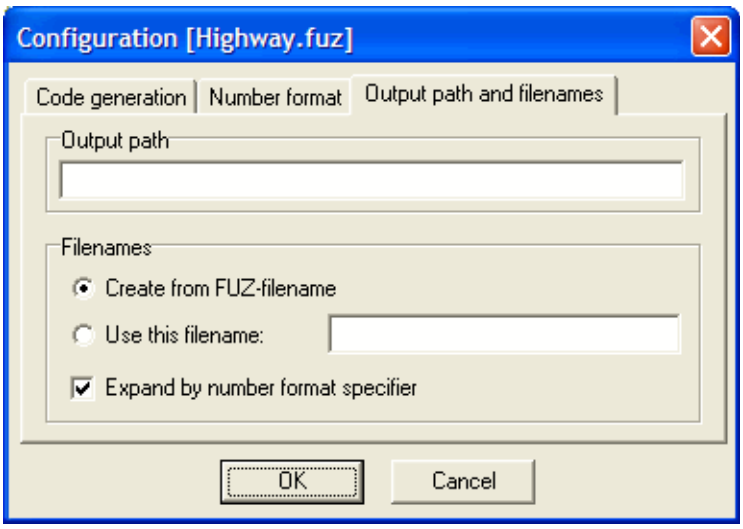
The range for the number of mantissa bits is between 5 and 14, for the number of exponent bits between 1 and 8. Only within these ranges it can be calculated with this number notation reasonably.

The fixed point numbers are defined exclusively by the number of the integer part bits and fractional part bits. If the sum of both is larger than 15, then a four byte data type is used, otherwise a two byte data type is chosen internally. In

the chapter *Numerical Libraries* you can find the specification of the fixed point number types and the two byte floating point type.

Output path and filenames

Because one FUZ file can have more than one generated source code files, espacially when operating with code template files, it is obvious to store these files in **one** directory. You can choose this ouput directory in the register *Output path and filenames*.



Register for configuring path and filenames

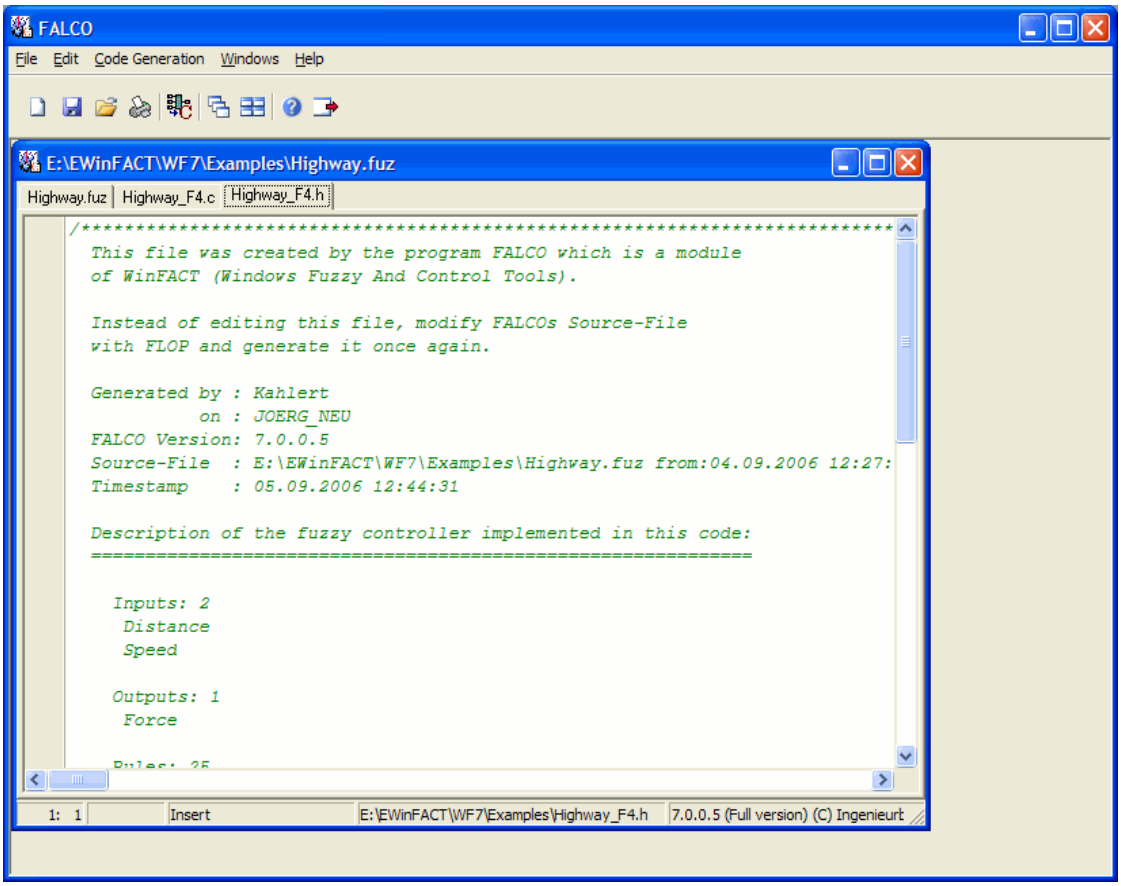
The file name of all source code files can be derived from the FUZ file or it can be predetermined manually. In the entry field the manually set file name may not have an extension (the dot also belongs to the extension). The file name will be enlarged by a token derived from the number format if the check box *Expand by number format specifier* is chosen. So duplicate file names are prohibited when generating code from the same FUZ file with different format of numbers.

Load and save settings

All settings which have been made for a document window, can be saved by the menu item FILE | SAVE SETTINGS... . The created file has the extension FCF (FALCO Configuration Files). By the menu item FILE | LOAD SETTINGS... the settings will be loaded and attached to the active document window.

Representation of the generated code

The generated source code appears in the same document window. The window now contains serveral tabs by which the productions of code can be reached.



*Document window after code generation***Fehler! Verweisquelle konnte nicht gefunden werden.**
every file has its own tab

In the picture above the code generation was carried out for the example file *highway.fuz* without modification of settings. Two files are produced:

1. a C file, which contains the structure of the fuzzy controller and
2. a header file, which forms the interface to the C file.

For usage in programs, several files (libraries) are required additionally, which also belong to the scope of delivery. These cover the functions for calculating with the chosen number format on the one hand, on the other hand some general functions for fuzzy controllers are defined here (chapter *Usage of C code*).

Structure of the generated code

The code generated by FALCO consists of a header file and a C file. The parameters, which build the data of the fuzzy controller, are stored in a C code structure. The function using this structure, is defined in the fuzzy libraries, which are delivered fully in source code. FALCO writes functions, in which the library functions are called.

Data structure of a fuzzy controller

You need the knowledge of the data structure, if you want to change the fuzzy controller later.

The main structure, which contains the whole fuzzy controller parameters, consists itself of other data structures. The following list shows all items of a fuzzy controller.

- Number of input variables
- Number of ouput variables
- Number of premises of the rule base
- Linguistic input variables
- Linguistic output variables
- Premise, conclusion and weighting of the rule base
- Inference mechanism
- Operator for fuzzy-AND
- Operator for fuzzy-OR
- Defuzzification method
- Number of integration steps for defuzzification methods which have to solve an integral like center of gravity

All data, which represents a number of, are positive integer values. The declarations of operators, method of defuzzification and inference mechanism are

done by enumerations, to get well readably C code. The enumeration types are defined in *fuzzy_enums.h*, which is enclosed in the scope of delivery.

The linguistic variables are divided into input and output variables, to assign different types of data structures to them. Both consist of a number of sets, but only the structure of the output variables contains a predefined output value. This value is used in case that no rule defines the output and the output is not set to be unchanged. Just to take this discrepance into account, two different data structures are implemented.

The premises and conclusions are unidimensional arrays, which contain integers. The values index the fuzzy sets (the index is shifted by one). If you imagine the rule base as two tables, one containing the premises and the other containing the conclusions, the values of the arrays represent the values of the rows of the respective table successively. The column number of the tables is identical to the index of the linguistic input resp. output variables. Negative indices in the array of the premise indicate a negatived fuzzy set.

The weighting is a unidimensional array of real numbers. You can realize this as a third table with only one column.

Example:

Premise	
E1	E2
1	1
-1	2
1	3
1	4

Conclusion	
A1	A2
1	3
1	2
2	1
2	1

Weighting
1
0.5
0.75
0.8

Saving of premises: {1, 1, -1, 2, 1, 3, 1, 4}

Saving of conclusions : {1, 3, 1, 2, 2, 1, 2, 1}

Saving of weightings : {1, 0.5, 0.75, 0.8}

For completion only the information on how the fuzzy sets are stored is missing. Here was also defined a simple data type: A fuzzy set consists of an array of pair of values and their number. Every pair of values describes a striking point of the fuzzy set.

Altogether the whole data for a fuzzy controller in source code is stored as it is shown in the following listing of the file *fuzzy_F2.h* (the token *F2* is a number format specifier standing for 2 byte floating point numbers). The type *Num-*

TypeF2_t is defined in the library for calculating with 2 byte floating point numbers. The *enum* types are defined in *fuzzy_enums.h*.

```
typedef unsigned char NumOfVal_t;

typedef struct{
  NumOfVal_t               n;
  const NumTypeF2Point_t*  p;
}FuzzySetF2_t;


typedef struct{
  NumOfVal_t            n;
  const FuzzySetF2_t*   fs;
  NumTypeF2_t           defaultvalue;
  char                  defaultactive;
}LinguisticOutputVariableF2_t;

typedef struct{
  NumOfVal_t           n;
  const FuzzySetF2_t*  fs;
}LinguisticInputVariableF2_t;

typedef struct{
  NumOfVal_t nI;
  NumOfVal_t nO;
  NumOfVal_t nR;
  const LinguisticInputVariableF2_t* iL;
  const LinguisticOutputVariableF2_t* oL;
  const char* pre;
  const char* con;
  const NumTypeF2_t* w;
  enum Inference_t inf;
  enum Defuzzy_t method;
  unsigned char steps;
  enum FuzzyAnd_t AndOp;
  enum FuzzyOr_t OrOp;
}FuzzyControllerF2_t;
```

Functions of a fuzzy controller

A fuzzy controller generated by FALCO constists of four functions. The names of the functions are composed by the name of the FUZ file, the number format specifier (if this was selected) and the inherent function name.

The following function declarations are generated without modification of the settings from the file *highway.fuz* in the header file:

```
void highway_F4_SetNumType(void);
void highway_F4_init(void);
void highway_F4_calc(
       const NumTypeF4_t i0,
       const NumTypeF4_t i1,
       NumTypeF4_t *o0);
void highway_F4_free(void);
```

Successivly the meaning of these functions is described.

The function ...*SetNumType* ensures the correct parameters of the chosen number format. If the number format has no parameters, the implementation of this function in the generated C file is empty. Only the fixed point numbers and 2 byte floating point numbers have parameters (number of precomma bits and number of postcomma bits resp. number of the bits for the exponent and number of the bits for the mantissa). With these formats it is necessary to call the function before the others. However, it is not harming to call it generally for all numbers formats before the other functions. In the cases in which the implementing is absent the call is removed by the compiler.

The initialization function (... *_init*) allocates memory for the calculation function. Therefore, it must be called **once** before calling this function.

The calculation function (... *_calc*) is the one and only function which must be called with arguments. It needs the input values of the fuzzy controller and the addresses of the variables in which the output values are to be stored after the computation. The input values and output values have to be given in the same order in which the linguistic input variables and output variables were defined within FLOP. The data types correspond to the number format you selected when generating code. The data types themselves are defined in the numeric libraries (s. Chapter *Numerical libraries*). The calculation function may be called arbitrarily often one after another.

The function to release (... *_free*) the reserved memory (reserved by the initialization function) may be called when the calculation function is not needed any

more. After the call of the release function the initialization function can be called again (if allocation and release of memory takes place frequently within the program execution, it should be guaranteed that the memory management is efficient enough to handle it).

Usage of the C code

First of all the simple usage is discussed. Afterwards a program will be created, which compares the accuracy of several code generations. Finally we'll have a look at peculiarities which occur by the usage of several code productions with the same number format and the usage of code template files.

Usage of a code production

The usage of the generated source code within a program is explained with an example here. We use the result of the code generation from the example file *highway.fuz*, which you find in the subdirectory *Examples*. The following settings should be made to this example.

Code generation:

Define fuzzy controller as const

Number Format:

Number Type: *Floating point*

Data type: *float*

Output path and filenames:

Output path: *c:\temp*

Filenames: *Create from FUZ-filename* and
Expand by number format specifier

The generated code consists of two files with the names *highway_F4.c* and *highway_F4.h*. A program is to read the values of the inputs of the produced

fuzzy controller from the keyboard and show the result on the screen. Only ANSI-C functions should be used.

```
#include <stdio.h>
#include "highway_F4.h"

int main(int argc, char **argv)
{
  NumTypeF4_t e1, e2, a1;
  printf("Please enter two values: ");
  scanf("%f %f",&e1, &e2);
  highway_F4_SetNumType();
  highway_F4_init();
  highway_F4_calc(e1, e2, &a1);
  highway_F4_free();
  printf("Result: %f",a1);
  fflush(stdin);
  getchar();
}
```

This program reads the input values from the keyboard by *scanf()*, sets the number format (because the internal number format is *float*, the function *highway_F4_SetNumType* does nothing at all), initializes the fuzzy controller, then applies *highway_F4_calc()* for the calculation of the output value, releases the fuzzy controller again and shows the result on the screen.

The header *stdio* is demanded for the functions *printf*, *scanf*, *fflush* and *getchar*, *highway_F4* for the fuzzy controller functions. The data type *NumTypeF4_t* is declared in *NumType_F4.h* and is announced by including *highway_F4.h*. It corresponds to the data type float. If you write down the files, which are included by the compiler, in the order it runs through the source code, you will get the following list (standard header left out!):

highway_F4.h

fuzzy_F4.h

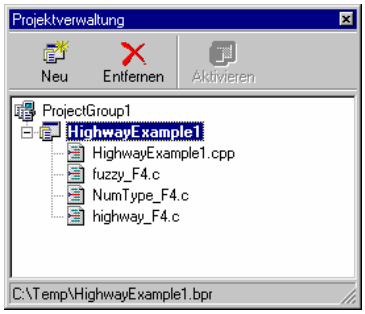
fuzzy_enums.h

NumType_F4.h

The indentation indicates which file integrates which header. Should this rather trivial program be created now, you have to ensure that these header files and the C files of the same name can be accessed by the compiler (fuzzy_enums.h has no corresponging C file). Hence you must enter appropriate searching paths within your compiler. The directory for the fuzzy header files and C files is in the *autocode* directory and is called *fuzzy*. For the different numeric data types there is also a subdirectory in the *autocode* directory which is called *NumType*. These both subdirectories must be passed to the compiler as searching paths, so that the compiling process runs without problems.

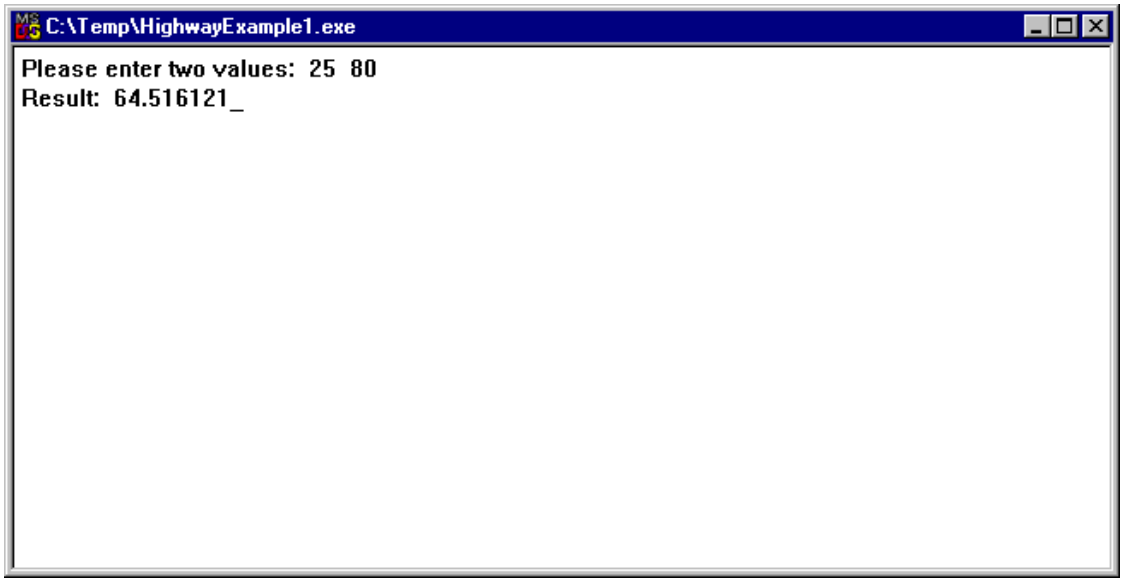
After the compiling process the produced object files have to be linked. The linker also must know the searching paths to get access to the required files, so that all external references can be found. For details of setting the searching paths of the linker and compiler please refer to the manual or the help of the compiler.

As an example, the project is shown within the Borland C++ Builder Version 3.0.



The example within the Borland C++ Builder (Version 3.0)

As you can see, the project consists of three C files, the header files specified at the top and of the main program *HighwayExample1.cpp*, which contains only the above listing and some lines automatically produced by the development environment. When program execution is done, you will get the following ouput:



The example after execution

Usage of several code productions with different number formats

On the basis of an example it will be shown how several code generations of a FUZ file can be compared to different number formats. The settings to the code generation are adopted from the last chapter. Then the setting of the number format is changed.

The following number formats are applied:

1. 2 byte floating point (5 bit exponent, 11 bit mantissa),
2. 2 byte fixed point (9 integer part bits, 7 fraction part bits) and
3. floating point numbers with the data type *double*

For code generation you can create three document windows via FILE | NEW and load the file *highway.fuz* in each of them. Modify the settings for a document window and save them. Afterwards you load the saved settings for the other document windows and change only the number format. Pay attention at all costs to the fact that actions (Configure, Load and Save), which concern the settings, always belong to the active document window!

If everything was done properly, you will get the following files:

highway_F8.h and *highway_F8.c*

highway_F2.h and *highway_F2.c*

highway_I2.h and *highway_I2.c*

Of course it is not necessary to do this in the way described here, you can also use a document window and produce the files step by step (three times: change configuration, generate code, save files).

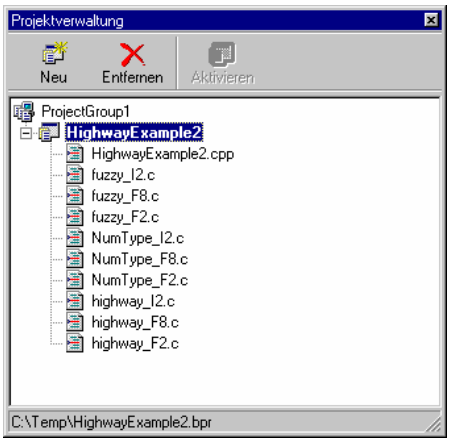
Now these files are integrated into the main program:

```
#include <stdio.h>
#include "highway_F8.h"
#include "highway_F2.h"
#include "highway_I2.h"

int main(int argc, char **argv)
{
  NumTypeF8_t f8e1, f8e2, f8a1;
  NumTypeF2_t f2e1, f2e2, f2a1;
  NumTypeI2_t i2e1, i2e2, i2a1;
  NumeratorF2_t f2n;
  DenominatorF2_t f2d;
  NumeratorI2_t i2n;
```

```
  DenominatorI2_t i2d;
  double rf2, ri2;

  printf("Please enter two values: ");
  scanf("%lf %lf",&f8e1, &f8e2);
  highway_F8_SetNumType();
  highway_F2_SetNumType();
  highway_I2_SetNumType();
  highway_F8_init();
  highway_F2_init();
  highway_I2_init();
  f2e1 = opF2FromFraction(f8e1*100,100);
  f2e2 = opF2FromFraction(f8e2*100,100);
  i2e1 = opI2FromFraction(f8e1*100,100);
  i2e2 = opI2FromFraction(f8e2*100,100);
  highway_F8_calc(f8e1, f8e2, &f8a1);
  highway_F2_calc(f2e1, f2e2, &f2a1);
  highway_I2_calc(i2e1, i2e2, &i2a1);
  highway_F8_free();
  highway_F2_free();
  highway_I2_free();
  opF2ToFraction(f2a1, &f2n, &f2d);
  opI2ToFraction(i2a1, &i2n, &i2d);
  rf2=(double)f2n/f2d;
  ri2=(double)i2n/i2d;
  printf("Result(F8 F2 I2): %lf %lf %lf", f8a1, rf2, ri2);
  fflush(stdin);
  getchar();
}
```

Here the same conditions, considering the searching paths for compiling and linking, exist as described in *Usage of a code production*. New in this source code are the functional calls which begin with *op*, and the *Numerator* and *Denominator* types; they are defined in the libraries *NumType_F2* resp. *NumType_I2*. The *opXXFromFraction* functions calculate a fraction given by the numerator and denominator in the number format *XX*. The other direction (calculating a value to a fraction) can be done by *opXXToFraction*, which assumes the first argument as the corresponding value and the last both arguments as the addresses where to store the numerator and denominator values. Every library which defines a number format that is not a standard numbers format (*float*, *double*, *long double*), also defines these both functions for the transformation of fractions.

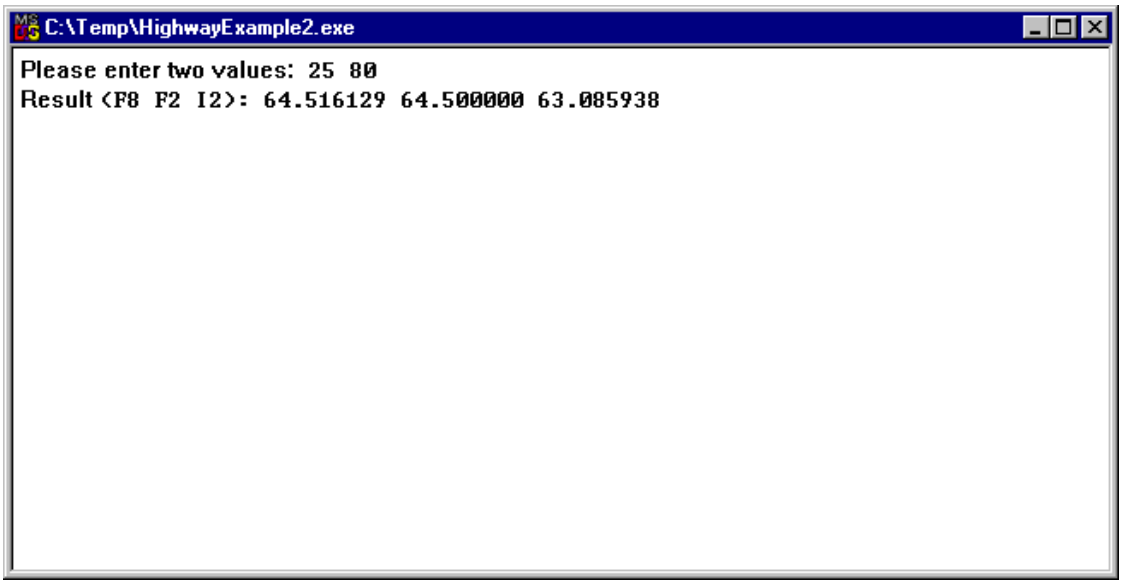
The project in the Borland C++ Builder (Version 3.0) looks like the following:



Project management of Borland C++ Builder (Version 3.0)

As shown in the above picture, appropriate files for the fuzzy controller's functions of the separate numeric data types (*fuzzy_XX.c*) and the files for computing with these data types (*NumType_XX.c*) are to be integrated into the project.

If the project is built and executed, the program issues the following:



Terminal window after program execution

The comparison of the accuracy of separate code productions for different numeric data types can be performed with the result now. Besides, the result of the 8 byte floating point number can be regarded as the sufficiently exact one. If one compares now the other both results to this one, it is obvious that the fuzzy controller with fixed point number arithmetic has a lower exactness than with 2 byte floating point number arithmetic.

Usage of several code productions with the same number format but different precision

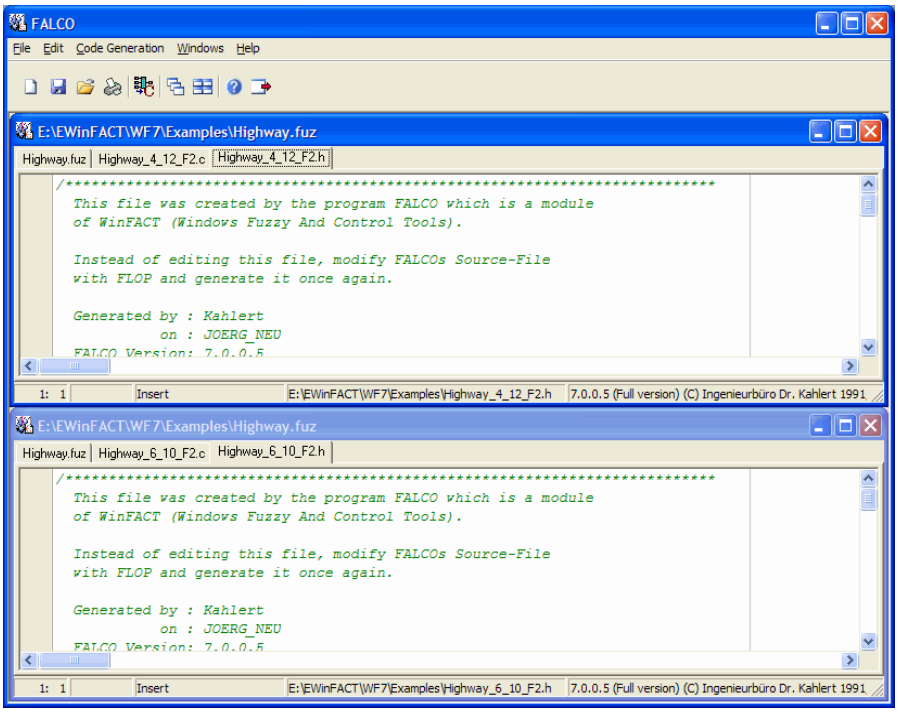
The number formats which allow a configuration of the accuracy are the 2 byte floating point numbers and the fixed point numbers. The exactness of the floating point numbers is determined by the number of the bits in the mantissa, those of the fixed point numbers by the number of the bits of the fractional part. This section refers only to the number formats mentioned at the top!

Again based on the file *highway.fuz,* two code productions should be produced with the number format 2 byte floating point, one time with 4 bits for the exponent and 12 bits for the mantissa, the other time with 6 bits for the exponent and 10 bits for the mantissa. Because both code productions use 2 byte floating point numbers, the extension of the file names by the number format specifier does not ensure unique file names in this case. Therefore additionally use the possibility to choose the file name freely. Here the file names

highway_4_12 and

highway_6_10

are chosen. After the code generation in two document windows FALCO looks like it is shown in the following picture:



Creation of two different code productions from the same FUZ file

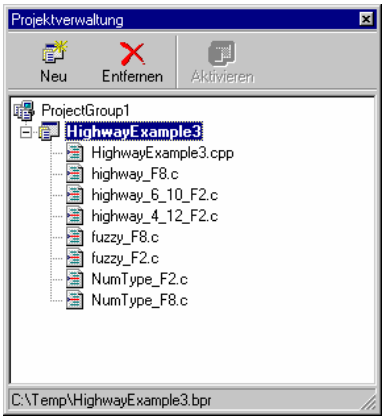
Now these code productions should be compared to one generated with the floating point data type *double* (it was already produced in the previous section). The source code of the main function is very similar to that in the section *Usage of several code productions with different number formats*.

```
#include <stdio.h>
#include "highway_F8.h"
#include "highway_4_12_F2.h"
#include "highway_6_10_F2.h"

int main(int argc, char **argv)
{
  NumTypeF8_t f8e1, f8e2, f8a1;
  NumTypeF2_t f2e1, f2e2, f2a1;
  NumeratorF2_t f2n;
  DenominatorF2_t f2d;
  double r_6_10_f2, r_4_12_f2;

  printf("Please enter two values: ");
  scanf("%lf %lf",&f8e1, &f8e2);
  highway_F8_SetNumType();
  highway_F8_init();
  highway_F8_calc(f8e1, f8e2, &f8a1);
  highway_F8_free();

  highway_4_12_F2_SetNumType();
  highway_4_12_F2_init();
  f2e1 = opF2FromFraction(f8e1*100,100);
  f2e2 = opF2FromFraction(f8e2*100,100);
  highway_4_12_F2_calc(f2e1, f2e2, &f2a1);
  highway_4_12_F2_free();
  opF2ToFraction(f2a1, &f2n, &f2d);
  r_4_12_f2=(double)f2n/f2d;

  highway_6_10_F2_SetNumType();
  highway_6_10_F2_init();
  f2e1 = opF2FromFraction(f8e1*100,100);
  f2e2 = opF2FromFraction(f8e2*100,100);
  highway_6_10_F2_calc(f2e1, f2e2, &f2a1);
  highway_6_10_F2_free();
  opF2ToFraction(f2a1, &f2n, &f2d);
  r_6_10_f2=(double)f2n/f2d;

  printf("Result (F8 4_12 6_10): %lf %lf %lf", f8a1, r_4_12_f2,
r_6_10_f2);
  fflush(stdin);
  getchar();
}
```

In the above listing the sequence of the different calculations was separated. The reason is the following: the 2 byte floating point library can handle only one information for the number of mantissa and exponent bits at a time. If the

SetNumType functions were called directly one after another like it is done in the previous section, only the last call would show effect.

The representation of the project administration shows, which files belong to this example:



Project management of the Borland C++ Builder (Version 3.0)

The screen shot of the ouput of the program is not shown here. The following lines make the program execution clear.

```
Please enter two values: 25 80
Result (F8 4_12 6_10): 64.516129 64.531250 64.375000
```

Creating programs for arbitrary fuzzy controllers

After the previous chapters it should be possible for you to integrate a code production produced by FALCO manually into a program. In this section, it is explained how a program can be applied for arbitrary fuzzy controllers. In addition the knowledge about the construction of code template files is an advantage (s. Chapter *FALCO's template transscriptor*). But the instructions are so clear, that the appropriate chapter can also be read later.

The program is to read all input values of a Fuzzy controller from the keyboard and ouput the results of the fuzzy controller on the screen. This should happen by the call of a function. So the main function simply calls this C function. It is to be declared in a header file *KeybInFuzzyMonOut.h* which has also to be produced:

void KeybInFuzzyMonOut(void);

The implementation should be written into the file *KeybInFuzzyMonOut.c*.

Code parameter files

Looking at a code parameter file you can easily identify what parameters are needed for creating the function template. The following listing represents the code parameter file, which was produced when generating the file *highway.fuz* for floating point arithmetic (data type *float*).

```
@WF_num_inputs@=2
@WF_num_outputs@=1
@WF_numtype@=NumTypeF4_t
@WF_out_file@=highway_F4
@WF_call_set_numtype@=highway_F4_SetNumType
@WF_call_init@=highway_F4_init
@WF_call_calc@=highway_F4_calc
@WF_call_free@=highway_F4_free
@WF_nf@=F4
@WF_par_type@=
@WF_par_var@=
@WF_num_format@=F=4 M=11 E=4 V=9 N=7 Xn=0x nX=
```

In the code parameter file variables, surrounded by *@*, are defined for the code template transformer. The meaning of the variables is not reflected here in particular, because it is almost given by the names (further information follows implicitly in the next section). Variables are lists (here the lists count only one element) and have the following functional character in the code template files:

- *@WF_num_inputs@* is replaced by the list on the right side of the equals sign (in this case only the '2').
- *@WF_num_inputs[i]@* is replaced with the *i*-th element of the zero based list on the right side of the equals sign (assuming *i*>0, would cause an error in this example, because the list contains only one element and is zero based; for *i*=0 the result would be '2' and is identical to the above case; when the list contains only one element *WF_num_inputs@=@WF_num_inputs[0]@* is valid)

The indicated form is more specific and is to prefer, even if it is not needed because of the case of equality. The variables *@WF_par_type@* and *@WF_par_var@* are assigned only if the fuzzy controller should be exported, which is a setting of the code generation. *@WF_num_format@* is explained in the section to *#WF_expr* (s. chapter *FALCO's template transscriptor*).

Creating a code template file

The creation of a code template file can take place gradually in verbal form. So the substance is described in human language and transferred in the second step directly to the code template which consists of variables and instructions for the template transscriptor and pure C code. The verbal formulation should be as close as possible to the corresponding instructions of the template transscriptor to make clear which instructions are suitable (this of cause can be done even better if you know the instructions, but in this case, it is a good method for learning them).

In the following we will pick up the example from the previous chapter. The verbal description and its conversion into the template are given alternately:

1. First of all in the header file *KeybInFuzzyMonOut.h* should be written. Only the functional declaration is to be written. If the header file is integrated into a bigger project several times, we can ensure that it is only included once by the *#ifndef* C preprocessor directive. Furthermore the *extern "C"* directive must be specified, if a C++ compilers is used and we have to pay attention on linking conventions.

```
#WF_write_to(KeybInFuzzyMonOut.h)

#ifndef KeybInFuzzyMonOut
#define KeybInFuzzyMonOut KeybInFuzzyMonOut


#ifdef __cplusplus
extern "C"{
#endif

  void KeybInFuzzyMonOut(void);

#ifdef __cplusplus
}
#endif

#endif
```

2. The header file is completed with that and from now on we write the functional implementing in the file *KeybInFuzzyMonOut.c*, which includes this header file and a header file which was generated by FALCO. In addition, the ANSI C standard header file *stdio.h* is still demanded for input/output handling.

```
#WF_write_to(KeybInFuzzyMonOut.c)

#include "KeybInFuzzyMonOut.h"
#include "@WF_out_file[0]@.h"
#include <stdio.h>
```



```
void KeybInFuzzyMonOut(void)
{
```

3. For every input value and for every output value we have to declare variables, which store the corresponding values. Their type is the selected numerical data type.

```
@WF_numtype@ e[@WF_num_inputs[0]@];
@WF_numtype@ a[@WF_num_outputs[0]@];
```

4. If 2 byte floating point or fixed point numbers are used, variables for conversion between fractions and these types have to be defined. Also a variable is needed for storage of the real value of a fraction.

```
#WF_if (@WF_nf@ ~~ F2,I2,I4)
  Numerator@WF_nf@_t    n;
  Denominator@WF_nf@_t  d;
  double                reell;
#WF_endif
```

5. Setting the parameter of the number format (calling of *SetNumType*-function)

```
@WF_call_set_numtype[0]@();
```

6. For each input variable a value has to be read form the keyboard and if 2 byte floating point or fixed point numbers are used, every given value has to be converted into the appropriate format.

```
for (i=0; i<@WF_num_inputs[0]@; i++){
   printf("Please enter the %d. value:",i+1);
   scanf("%lf", &reell);
   #WF_if (@WF_nf[0]@ ~~ F2,I2,I4)
   e[i] = op@WF_nf[0]@FromFraction(reell*100,100);
   #WF_else
   e[i] = reell;
   #WF_endif
}
```

Note:

The value *i* for the loop has to be defined before putting the fragments of the template together. Furthermore the variable *real* is only defined when using 2 byte floating point or fixed point numbers. This also has to be changed.

7. Call of the initialization function

```
@WF_call_init[0]@();
```

8. Call of the calculating function with all input values and all addresses of the output values.

```
@WF_call_calc[0]@(
  #WF_foreach($var, 0..@WF_num_inputs[0]@-1){
    e[$var],
  }
  #WF_foreach($var, 0..@WF_num_outputs[0]@-1){
    #WF_if($WF_LAST_1$)
    &a[$var]
    #WF_else
    &a[$var],
    #WF_endif
  });
```

9. Call of the function for releasing allocated memory

```
@WF_call_free[0]@();
```

10. All output values have to be displayed on the screen. For 2 byte floating point and fixed point, the values have to be converted into real values before.

```
for (i=0; i<@WF_num_outputs[0]@; i++){
   #WF_if (@WF_nf[0]@ ~~ F2,I2,I4)
   op@WF_nf[0]@ToFraction(a[i], &n, &d);
   reell = (double)n/d;
   #WF_else
   reell = a[i];
   #WF_endif
   printf("%d. Output = %lf\n", i+1, reell);
}
```

11. Last but not least we should wait for a keystroke to get a look at the outputs and have to close the function with the right brace (in 2 we used the left one).

```
  fflush(stdin);
  getchar();
}
```

We obtain the following code template, considering the note in item 6 and adding the *#WF_code_block* instructions (for a better survey of the two files generated by this template the corresponding lines are highlighted):

```
#WF_code_block(Tastatureingabe & Bildschirmausgabe, MAIN)

#WF_write_to(KeybInFuzzyMonOut.h)

#ifndef KeybInFuzzyMonOut
#define KeybInFuzzyMonOut KeybInFuzzyMonOut

#ifdef __cplusplus
extern "C"{
#endif

void KeybInFuzzyMonOut(void);
```



```
#ifdef __cplusplus
}
#endif

#endif

#WF_write_to(KeybInFuzzyMonOut.c)

#include "KeybInFuzzyMonOut.h"
#include "@WF_out_file[0]@.h"
#include <stdio.h>

void KeybInFuzzyMonOut(void)
{
  @WF_numtype@ e[@WF_num_inputs[0]@];
  @WF_numtype@ a[@WF_num_outputs[0]@];
  double       reell;
  int          i;

  #WF_if (@WF_nf@ ~~ F2,I2,I4)
  Numerator@WF_nf@_t    n;
  Denominator@WF_nf@_t  d;
  #WF_endif

  @WF_call_set_numtype[0]@();

  for (i=0; i<@WF_num_inputs[0]@; i++){
     printf("Please enter the %d. value:",i+1);
     scanf("%lf", &reell);
     #WF_if (@WF_nf[0]@ ~~ F2,I2,I4)
     e[i] = op@WF_nf[0]@FromFraction(reell*100,100);
     #WF_else
     e[i] = reell;
     #WF_endif
  }

  @WF_call_init[0]@();

  @WF_call_calc[0]@(
    #WF_foreach($var, 0..@WF_num_inputs[0]@-1){
      e[$var],
    }
    #WF_foreach($var, 0..@WF_num_outputs[0]@-1){
      #WF_if($WF_LAST_1$)
      &a[$var]
      #WF_else
      &a[$var],
      #WF_endif
    });

  @WF_call_free[0]@();

  for (i=0; i<@WF_num_outputs[0]@; i++){
```

```
      #WF_if (@WF_nf[0]@ ~~ F2,I2,I4)
      op@WF_nf[0]@ToFraction(a[i], &n, &d);
      reell = (double)n/d;
      #WF_else
      reell = a[i];
      #WF_endif
      printf("%d. Output = %lf\n", i+1, reell);
   }
   fflush(stdin);
   getchar();
}
#WF_code_block_end
```

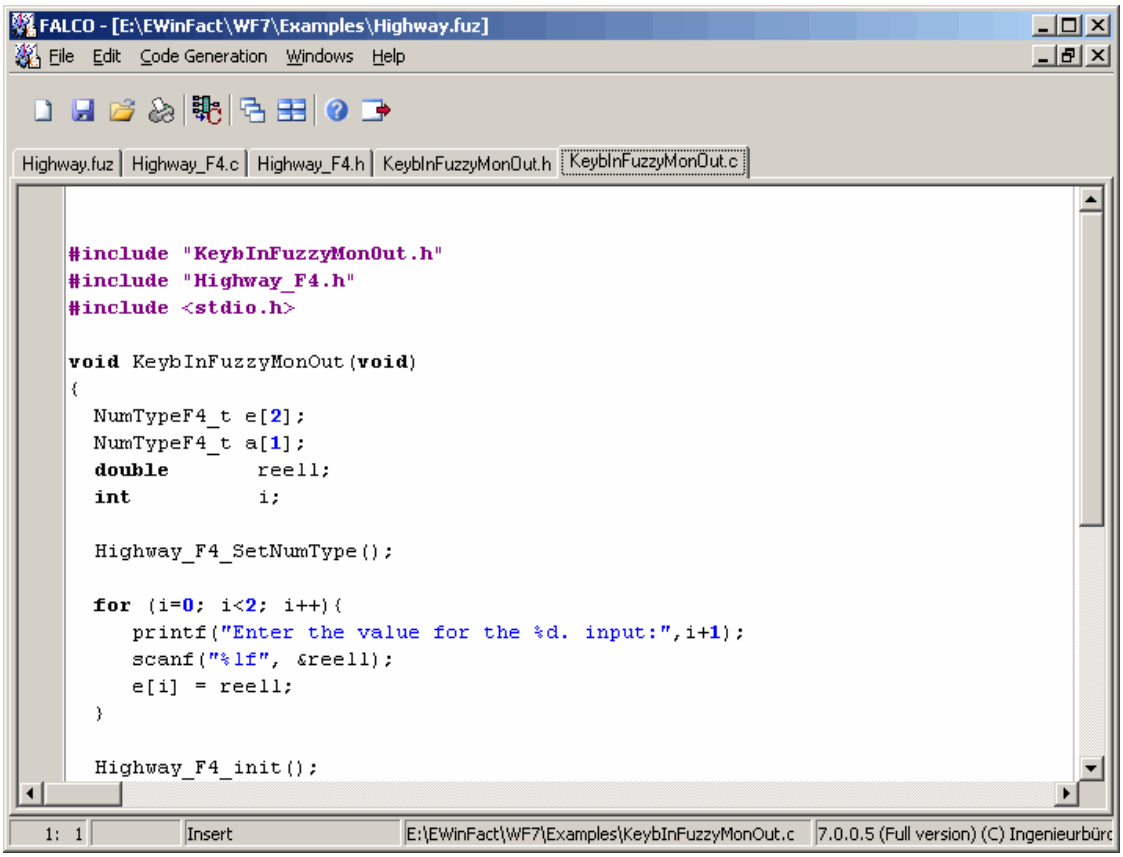
It is important that the instruction for starting a code block has *MAIN* as the last argument. Only then FALCO detects this function as a main function and you can select it in the configuration dialog.

Code generation using a code template file

The code generation with a code template file is only carried out, if the according settings are chosen:

- *Use c code template as call* has to be selected,
- the file, which contains the template described before, has to be entered and
- the main function also has to be selected.

After code generation with the above settings, a document window gets two more tabs which have the names of the new files:



FALCO after code generation with the use of the code template from the previous chapter

The information how to build the programm was described in the chapters before, so there is no need to repeat it here.

Libraries

The generated source code does not contain all information needed for the computation of a fuzzy controller. The numerical and the general functions of a fuzzy controller are conciously separated. This increases clarity and speeds up a repeating compilation in greater projects, because the libraries need to be compiled only once. The separation for the numerics is done for every single numerical type. A code template exists for the general fuzzy libraries, which

will create the files *fuzzy_XX.h* and *fuzzy_XX.c* when applied (*XX* denotes the number format specifier).

Fuzzy libraries

The fuzzy libraries were, as mentioned above, generally composed as C code template. The file *CreateFuzzyLibraries.CSH* can be used to create fuzzy libraries for all numerical types. It is only required to load a FUZ file in a document window and do the following configuration for code generation:

- select *Use C code template as call*,
- select the main function: *Create Fuzzy Libraries*
- specify the *Output path*

The advantage of this strategy is, that changes have to be made in only one file and occur in all fuzzy libraries after applying the template. To change specific libraries (e. g. the fixed point ones), the *#WF_if* instruction of the code template transscriptor can be used.

A fuzzy library defines all operations, which are needed for a computation of a fuzzy controller. These are the following fundamental functions, which are not discussed here:

- calculation of a membership value
- the operation fuzzy-AND
- calculation of the inference
- computation of an output value via different defuzzification methods

The following functions are declared as interface in the header file (*XX* denotes the number format specifier):

- A function for allocating resources and initialization:

```
void FCXX_init(const FuzzyControllerXX_t *fc,
               FCMemXX_t *v);
```

- A function for computation:

```
void FCXX_calc(const FuzzyControllerXX_t *fc,
               FCMemXX_t *v,
               NumTypeXX_t *e,
               NumTypeXX_t *a);
```

- A function for releasing resources:


```
void FCXX_free(const FuzzyControllerXX_t *fc,
               FCMemXX_t *v);
```

The declarations of the interface functions may never be changed, because these are expected by FALCO in just this form. However, you may change the implementation. In the header file the data types of the arguments are also defined. *FuzzyControllerXX_t* was described by the data types already in the section *Data structure of a Fuzzy controller*. This data structure is expected from FALCO also accurately in this form. Changes in it are not allowed!

The data structure *FCMemXX_t* serves for the storage of inner states. Their fields are allocated by the initialization function and released in the free function. Here is the declaration of the data structure:

```
typedef struct{
    Bool_t       **ihit;
    Bool_t       **ohit;
    NumTypeXX_t  **imv;
    NumTypeXX_t  **omv;
    NumTypeXX_t  *res;
 }FCMemXX_t;
```

After a call of the function for computation, it contains the information:

- if the *n*-th set of the *m*-th linguistic input variable was hit by a crisp input value:

```
ihit[m][n]
```

- which membership value was calculated for the *n*-th set of the *m*-th input variable:

```
imv[m][n]
```

- if th *n*-th set of the *m*-th linguistic ouput is to be defined by a active rule:

```
ohit[m][n]
```

- the match of degree for the *n*-th set of the *m*-th linguistic output variable:

```
omv[m][n]
```

- the crisp value for the *m*-th linguistic output variable:

```
res[m]
```

FALCO creates a variable with the type of this data structure in the generated code. Should this variable be accessed, the fuzzy controller must be exported (s. chapter *Settings*). Only then FALCO declares this variable without storage-class-specifier *static* (it is considered by the compiler as external, only that the kind of linkage is not indicated explicitly). With the help of a code template sequence the variable can be written in the associated header file what manages the access.

```
#WF_code_block(Access to FCMem)
#WF_write_to(@WF_out_file[0]@.h)

#ifndef @WF_out_file[0]@_FCMem
#define @WF_out_file[0]@_FCMem @WF_out_file[0]@_FCMem

extern FCMem$XX$_t @WF_out_file[0]@_FCMem;

#endif

#WF_code_block_end
```

Numerical Libraries

The numerical libraries define the basic mathematic operations (+,-, *,/) for a certain numeric data type.

The understanding of the libraries is usually not required for the understanding of a generated code! However, it can be meaningful to carry out a special implementing of the numeric for a certain target hardware, to exhaust around capabilities better.

General properties

It is not possible in ANSI C to define operators dependent on variable type. If one wants to write C code independent from the numeric, this can be managed only by functions which recreate these fundamental operations. With the floating point data types (*float*, *double* and *long double*) these functions are implemented only as macros which contain the original operation. With the fixed point numbers and the not standardized floating point numbers the functions are real implementations.

The choice of name of the numeric libraries is unambiguous, so that mixing different types within an application will be possible. The clarity is achieved by a number format specifier.

The following table represents the context.

Number format	**Specifier**	**Library name**
2-Byte-Fixed point	I2	NumType_I2.c NumType_I2.h
4-Byte- Fixed point	I4	NumType_I4.c NumType_I4.h

2-Byte-Floating point	F2	NumType_F2.c NumType_F2.h
4-Byte- Floating point	F4	NumType_F4.c NumType_F4.h
8-Byte- Floating point	F8	NumType_F8.c NumType_F8.h
10-Byte- Floating point	F10	NumType_F10.c NumType_F10.h

Every library defines a data type *NumTypeXX_t*, whereat *XX* is to be replaced with the specifier. This data type is derived from a fundamental data type (s. below). With this type all calculations can be executed. Besides a data type is available for a pair or values (*NumTypeXXPoint_t*) which consists of two fields *x* and *y* of the data type *NumTypeXX_t*.

The declarations for the number format F4 (4 byte floating point) looks as follows:

```
typedef float NumTypeF4_t;

typedef struct{
  NumTypeF4_t x, y;
}NumTypeF4Point_t;
```

The fundamental data type of the library for 4 byte floating point numbers is *float*.

The libraries define constants of the type *NumTypeXX_t* for the maximal and minimal representable numbers as well as for the value 1:

- *XXMax*
- *XXMin*
- *XXOne*

As well every library defines their own functions (partly also macros; s. above) for the implementation of binary and unary operators. All names of the functions which represent operators start with *op* followed by the number format specifier. They end with the name of the function which they implement (*Mul, Div, Add*, ...).

In the following table functions are listed which are contained in all libraries. Besides, the applied identifier a, b and c have always the data type *NumTypeXX_t*.

Operator function	Meaning
a = opXXAdd(b, c)	a = b + c
a = opXXSub(b, c)	a = b - c
a = opXXMul(b, c)	a = b * c
a = opXXDiv(b, c)	a = b / c
a = opXXNeg(b)	a = -b
a = opXXInv(b)	a = 1 / b
i = opXXG(b, c)	i = b > c
i = opXXGE(b, c)	i = b >= c
i = opXXL(b, c)	i = b < c
i = opXXLE(b, c)	i = b <= c
i = opXXE(b, c)	i = b == c
i = opXXNE(b, c)	i = b != c
a = opXXCast(d)	a = d in the number format XX The type of the variable *d* has to be of the fundamental type of the used number format.

The operator functions allow to write C code in connection with the code template transscriptor, which is independent of the number format (s. chapter *Fuzzy libraries*). However, this is obtained by a certain loss of legibility which can be reduced a little.

It should be shown with an arithmetic expression how such a conversion looks. The expression

$$a = b + c \cdot d - a$$

with the above operator functions for 2 byte floating point numbers has the following appearance:

```
a = opF2Add(
            b,
            opF2Sub(
              opF2Mul(c, d),
              a)
            );
```


The readability of these expressions can be increased by indentation, like it is usually done in source codes.

Standard floating point numbers F4, F8 and F10

The data types for the formats F4, F8 and F10 correspond to the floating point types *float*, *double* and *long double* (these are also the appropriate fundamental data types for *opFXCast*). All operator functions are implemented as macros. If a division by zero occurs, the result will be the maximum representable positive or negative number (according to sign of the dividend) of the appropriate format.

2 byte floating point F2

The 2 byte floating point numbers have an adjustable number of exponent and mantissa bits. Therefore, the library must be informed of these settings, before it can be calculated with this number type. This is made by the call of the function *SetNumTypeF2Format* the declaration of which is to be seen here:

```
void SetNumTypeF2Format(const unsigned char anF2ManBit,
                        const unsigned char anF2ExpBit);
```

The calculations which are executed before the call of this function are indefinite. If another call of this function with another number of exponent or mantissa bits takes place, the results of the previous calculations may not be applied just like that.

Example:

```
NumTypeF2_t a, b, c;

SetNumTypeF2Format(11, 4);
/*calculate a little bit*/
a=F2One;
b=opF2Add(F2One, F2One);

SetNumTypeF2Format(10, 5);
/*from this point a and b are interpreted wrong !*/
c=opF2Sub(b, a);  /*c ist hier nicht = F2One !!!!*/

SetNumTypeF2Format(11, 4);
/*from this point a and b are interpreted correct !*/
c=opF2Sub(b, a);  /*c = F2One !!!!*/
```

To explain what happens in particular in this example, it is first to be described how values are stored resp. interpreted in the format F2.

The numerical values are stored in the fundamental data type short in dependence on the IEEE format. This means that the representation of the floating

point *x* is based on a decomposition of a sign *v*, a mantissa *m* and an exponent *e* to the basis 2:

$$x = v \cdot m \cdot 2^e$$

Because the number of bits of *m* and *e* is finite, an appropriate inaccuracy appears with the representation; the number *x* is not represented exactly. The exactness is given by the number of bits in the mantissa, the co-domain is determined by the number of the bits in the exponent. If 4 bits are configured for the exponent and 11 bits for the mantissa, a number is presented as follows within 2 bytes:

Bit position:

15	14 … 11	10 … 0
v	eeee	mmm mmmm mmmm

v has for negative numbers the value 1, otherwise 0. The mantissa is produced in such a way that their value is between 1 and 2. The 1 before the comma is not stored to save space. To the exponent a shift (Bias) is added, so that this is positive and different from zero in all cases. The exception of this rule is the number zero, here *v*, *e* and *m are* set to zero.

As an example the numbers from -5 to +5 with an 11 bit wide mantissa and a 4 bit wide exponent are represented here. The Bias within this representation is 8. The Term 1 in the sum of the mantissa *m* corresponds to the not stored 1 mentioned at the top. It is inapplicable for the value 0.

Value	**Mapping**	**Description**
-5	0xD200	v=1 e=10-8 m=1+1/4
-4	0xD000	v=1 e=10-8 m=1
-3	0xCC00	v=1 e=9-8 m=1+1/2
-2	0xC800	v=1 e=9-8 m=1
-1	0xC000	v=1 e=8-8 m=1
0	0x0000	v=0 e=0 m=0
1	0x4000	v=0 e=8-8 m=1
2	0x4800	v=0 e=9-8 m=1
3	0x4C00	v=0 e=9-8 m=1+1/2
4	0x5000	v=0 e=9-8 m=1
5	0x5200	v=0 e=10-8 m=1+1/4

If one looks at the example above, the value for the constant *F2One* is mapped to 0x4000. Before the second call of *SetNumTypeF2Format* the variables contain the following:

- a = 0x4000 corresponds to the value 1
- b = 0x4800 corresponds to the value 2

After changing the format, the interpretation of the mapping would result to:

- a = 0x4000 still corresponds to the value 1
- b = 0x4800 now corresponds to the value 4

As a result *c* gets the value 3 (0x4600), which is interpreted after the last call of *SetNumTypeF2Format* to 1.75.

Simply avoid a mixture of different F2 numbers!

In the library *NumType_F2.c* there are further functions implemented, which allow a conversion of a number into a fraction and vice versa. The data types used within these functions are *NumeratorF2_t* and *DenominatorF2_t*, they are defined in the header file of the library. The data type *DenominatorF2_t* can only contain positive integers! The functions are declared as follows:

```
NumTypeF2_t opF2FromFraction(NumeratorF2_t    numerator,
                             DenominatorF2_t  denominator);

void opF2ToFraction(NumTypeF2_t     f1,
                    NumeratorF2_t   *numerator,
                    DenominatorF2_t *denominator);
```

Example:

The number 3.125 should be converted into the *F2* format with 11 mantissa bits and 4 bits for the exponent:

```
SetNumTypeF2Format(11,4);
F2Result = opF2FromFraction((short)(3.125*1000), 1000);
```

After execution *F2Result* contains the sedecimal value 0x4C80. If *F2Result* should be converted back into a real number, the small code sequence below can be used (assuming that the number of bits for exponent and mantissa wasn't changed):

```
NumeratorF2_t    zaehler;
DenominatorF2_t  nenner;
double           reell;

opF2ToFraction(F2Result, &zaehler, &nenner);
reell = (double)zaehler/nenner;
```

The conversion into fractions ensures, that the value of the denominator is different to zero. When converting a fraction into a real number, the denominators inequality to zero is **not** checked!

An overflow caused by an operation, is treated by giving back the maximal represantable number with the correct sign.

Fixed point numbers I2 and I4

The fixed point numbers have the settings:

- number of the precomma bits and
- number of the postcomma bits.

Before it can be calculated with fixed point numbers, the library must be informed about these settings via the call of the function *SetNumTypeI2Format* or *SetNumTypeI4Format*.

A number is stored in the fundamental data type *short* or *long int*. The number to be mapped is simply multiplied by the value 2^{nn}, whereat *nn* is the number of the postcomma bits. The following table shows of the numbers -2 to +2 in steps of 0.5:

Value	Mapping
-2	-256
-1.5	-192
-1	-128
-0.5	-64
0	0
0.5	64
1	128
1.5	192
2	256

In the fixed point number libraries *NumType_I2.c* and *NumType_I4.c* functions are also implemented for the conversion in and from a fraction. The declarations for it have been made in dependence on the numbers format *F2*. In these libraries also exist own data types for numerator and denominator values. By

the data type *DenominatorI2_t* or *DenominatorI4_t* it is guaranteed that denominator values can contain only positive integers.

As an example the number 3.125 should be converted into the format *I2* with 9 precomma bits and 7 postcomma bits:

```
NumTypeI2_t I2Result;

SetNumTypeF2Format(9,7);
I2Result = opI2FromFraction((short)(3.125*1000), 1000);
```

After the execution, *I2Result* contains the number 400 ($400 = 2^7 \cdot 3.125$). Should *I2Result* be changed back into a fraction, it can be made by the call *opI2ToFraction*:

```
NumeratorI2_t    zaehler;
DenominatorI2_t  nenner;

opI2ToFraction(I2Result, &zaehler, &nenner);
```

FALCO's template transscriptor

The template transscriptor minimizes your time and effort of programming. You create a file in the template specified here, the transscriptor creates the appropriate source code files which you need for your projects. The principle is very simple. The template transscriptor reads your template file and executes the instructions contained in it. Variables are replaced by their value. So it will be possible to use one template for arbitrary code generations.

Functionality

The functionality of the template transscriptor is rather simple. He receives the contents of a CPF file (code parameter file) and a CSH file (code scheme/template file) as input data and interprets the CSH file from the entry point (s. below).

The template of the CSH file contains instructions which are executed by the transscriptor and variables, the values of which are taken from the code parameter file. The variables in the template are simply replaced by their values.

If a piece of code should be applied for arbitrary fuzzy controllers, the number of the input and output parameters, the numeric type, the names of the functions etc. have to be written as variables. In the section *Code parameter files*, you find an example of a CPF file in which all variables are specified. It is not necessary, to choose the setting *Create code parameter file* to work with the transscriptor.

The given CSH file is divided into separate code fragments, also named code blocks, whereat at least one is marked as an entry point for the transscriptor (s. Instruction *#WF_code_block*). The template transscriptor reads the CSH file, executes the contained instructions and writes the result in an internal list with the name *main.c*. Afterwards this can be saved as a file under this name. By the instruction *#WF_write_to* (s. below) the writing can be redirected in a list with another name (file name). If the instruction is the first after the entry point of the CSH file, the *main.c* will not be created.

Variables and instructions

The template transscriptor knows different instructions and variables which may occur during the interpretation of the CSH file. He distinguishes between local and global variables.

Variables

The transscriptor distinguishes between local and global variables. The global variables are always those which are defined in a CPF file. Local variables can be produced only by the instructions *#WF_foreach* and *#WF_code_block*. For further information on these instructions consult the appropriate sections (s. below).

The meaning of most global variables can be concluded from their names. They all begin with *@WF_* and end with *@* to separate them from the rest of the code. In the section *Code parameter files* you find an example of a CPF file, in which all global variables are explained.

Instructions

Instructions always begin with the mark # followed from two capital letters *WF_* and an instruction name written in small letters. Arguments of an instruction are generally put in parentheses.

Within the following description of instructions optional arguments are put in brackets. To select one from several possibilities, these possibilities are separated by |. The set of all possibilities is given in braces. The meaning of *{1 | 2 | 3}* is *either 1 or 2 or 3*, but not all numbers and also not none of the numbers!

Some instructions can contain lists. These are character strings, which are separated by comma. In addition, there are integer range definitions. Given by two numbers with **two** dots inbetween (e.g.: -2..2). Range definitions may also be elements of a list (e. g.:-2. 2, 4. 10, a, b, c). Both integer values of a range definition may be algebraic expressions (e. g.: 5+2 .-3 + @WF_num_inputs [0]@)

#WF_code_block

Syntax:

```
#WF_code_block(Name [{($a,$b,...) | ,main } ] )
```

Defines the beginning of the code block *name*. The code block is made available as an entry point for the template transscriptor, if the *main* is chosen instead of the list. A code block can always be inserted by *#WF_insert_code_block* at another place within the template. If the parenthised list *($a, $b...)* is used instead of the *main*, the elements of the list become local variables with the interpretation of the code block (it can be appropriate to let the local variables also end with $, see *#WF_foreach*). These are valid only within this code block. The values of the variables are defined by the use of the block by *#WF_insert_code_block*. A code block must always be closed with *#WF_code_block_end*!

#WF_code_block_end

Syntax:

```
#WF_code_block_end
```

Defines the end of the current code block.

#WF_write_to

Syntax:

```
#WF_write_to(filename)
```

The output stream is redirected in an internal list with the name *filename* until this instruction with another *file name* occurs or the code block is left. At the start of the template transscriptor it is written in the file *main.c*, until this instruction is parsed by the transscriptor.

#WF_include

Syntax:

```
#WF_include character string
```

The preprocessor-directive *#include character string* is constructed and is appended to an internal include list of the current output file, if the *character string* is not already found in the list.

#WF_define

Syntax:

```
#WF_define a b
```

A define directive is created and appended to the define list of the current output file. This instruction may use the backslash to mask the end of line (this means the next line is concatenated with the previous from the view of the transscriptor):

```
#WF_define abs3D(a,b,c) \
  (sqrt (   (a)*(a) +\
            (b)*(b)+\
            (c)*(c)\
        )\
   )
```

#WF_insert_code_block

Syntax:

```
#WF_insert_code_block(Name[(List of arguments)]
          [, { Filename{.REF|.CSH} | HERE }  [,EVER] ] )
```

Inserts a code block in the template. The parameters will be transfered in order to the local variables of the code block; the order is defined by the *#WF_code_block* instruction. The code block is searched in the current file if no *filename* or *HERE* is indicated. If the file name has the extension *.REF*, the indicated name is searched in a reference file. In this case the work goes on with the appropriate reference. If extension is *.CSH*, the block is searched in the appropriate CSH file. The given file is searched in current and all directories set by the *#WF_path* instruction.

The build in identifier *HERE* means that the code section should be taken from this template file. If *EVER* is indicated, the insertion of this code section is

implemented also if this was already inserted in the current output file. Normally every code block can be written only once in an output file.

#WF_path

Syntax:

```
#WF_path(Directory1[; Directory2;....])
```

Sets the searching directories for the instruction *#WF_insert_code_block* (s. above).

#WF_show_message

Syntax:

```
#WF_show_message(Message, Window title)
```

As long as the template transscriptor parses this instruction, the *message* is displayed in a modal dialog window with appropriate *window title*. It must be confirmed by the button OK!

#WF_foreach

Syntax:

```
#WF_foreach ($Variable[$], List ){
...further lines of the code template
}
```

This instruction is used for creation of loops. It is appropriate to end the loop variable with *$*, if the *further lines of the code template* contain *$variable* in the middle of an identifier.

Example:

The following output should be produced: " *x_t y_t z_t* ". The following code fragment looks correct first.

```
#WF_foreach ($var, x, y, z){$var_t }
```

However, unfortunately, it does not work, because the transscriptor supposes that $var_t is a local variable. By ending with *$* it is clarified:

```
#WF_foreach ($var$, x, y, z){$var$_t }
```

The loop variable is set to the next value of the list with every iteration. If the element of the list is an integer range definition, the defined range will passed through entirely (i.e. *$variable$* is also set to every separate value of this range definition). In range definitions *a..b* iteration is always started with the value *a* and it ends with the value *b* in steps of 1. *a* and *b* can be algebraic expressions.

Within the following template lines the loop variable is replaced with the value of the current step.

Example:

The following code sequence demonstrates the strength of the *#WF_foreach* instruction:

```
#WF_foreach($i$, a,b,c){
  /* $i$ :*/#WF_foreach($j$, 0..2){$i$$j$=}0;
}
```

It creates the following output:

```
/* a: */ a0=a1=a2=0;
/* b: */ b0=b1=b2=0;
/* c: */ c0=c1=c2=0;
```

The *#WF_foreach* instruction implicitly defines a local variable with the name *WF_LAST_n*. *n* stands for *n*-th nested loop in this code block. It is started with *n*=1. The above example produces a variable *WF_LAST_1* for the outer loop and for the inner one a variable with the name *WF_LAST_2*. The values of the variables equal to 1 if the last iteration of the loop, which created this variable, is reached, otherwise 0. The variables exist only within the loops. They are extremely useful for branching.

#WF_if

Syntax:

```
#WF_if (simple logical expression)
  Code template lines A
[#WF_else
  Code template lines B]
#WF_endif
```

The *#WF_if* instruction realizes a branch. The *code template lines A* are only interpreted if the *simple logical expression* has true as a result. If the result of the *simple logical expression* is false and assumed that *#WF_else* exists, the *code template lines B* are read and interpreted. A simple logical expression has only **one** binary operator which separates the left and right side, which may be arithmetic expressions. The following tables specify all binary operators:

Operators for algebraic expressions

Operator	**Meaning**
==	equal
!=	unequal
>=	greater or equal
<=	less or equal

>	greater
<	less

In addition, it is possible to examine whether a variable is not equal to zero, in this case simply the variable is to be indicated.

Operators for character strings

Operator	Meaning
~~	the right side is allowed to be a list. If the left string is in that list the condition is met (true).
!~	negated version of ~~

In addition, it is possible to examine whether a variable contains no letters, in this case also simply the variable is to be indicated.

#WF_expr

Syntax:

```
#WF_expr(number format description, algebraic expression)
```

The instruction calculates the result of the *algebraic expression* and converts it into the indicated number format. The number format is given by the following statements:

F=*n*	Uses floating point numbers with *n* bytes
M=*n*	If F=2 the number of mantissa bits is to be set to *n*, otherwise it will be ignored.
E=*n*	If F=2 the number of exponent bits is to be set to *n*, otherwise it will be ignored.
V=*n*	If F=*n* is not indicated, *n* is the number of precomma bits for fixed point numbers.
N=*n*	If F=*n* is not indicated, *n* is the number of postcomma bits for fixed point numbers.
Xn=*Z*	If F=2, the representation for a sedecimal number is defined. *Z* is to be issued before the hexadecimal value (has to be set to *Xn=0x* in ANSI-C).
nX= *Z*	If F=2, the representation for a sedecimal numbers is defined. *Z* is to be issued after the hexadecimal value (is not to be set in ANSI-C !).

The global variable *@WF_num_format@* contains the complete information to convert a number into the selected number format for code generation.

Format	**Required statements**
F10	F=10
F8	F=8
F4	F=4
F2	F=2 M=*Mantissa bits* E=*Exponent bits* Xn=*0x* nX=
I4 and I2	V=number of bits for the integer part N=number of bits for the fraction part

Example:

The value 3.125 should be converted into the number format I4 with 13 pre-comma and 11 postcomma bits:

```
#WF_expr(V=13 N=11, 3.135)
```