

# Der Fuzzy-C-Code-Generator FALCO

<b>Übersicht</b>	<b>8.3</b>
<b>C-Quellcode-Generierung</b>	<b>8.5</b>
Starten der Code-Generierung	8.5
Laden von FUZ-Dateien	8.6
Einstellungen	8.6
Code-Erzeugung	8.7
Zahlenformate	8.7
Ausgabeverzeichnis und Dateinamen	8.8
Laden und Speichern der Einstellungen	8.9
Darstellung des generierten Codes	8.9
<b>Struktur des generierten Codes</b>	<b>8.10</b>
Datenstruktur eines Fuzzy-Controllers	8.11
Funktionen eines Fuzzy-Controllers	8.13
<b>Verwendung des C-Codes</b>	<b>8.15</b>
Verwendung einer Code-Produktion	8.15
Verwendung mehrerer Code-Produktionen	
unterschiedlicher Zahlenformate	8.18
Verwendung mehrerer Code-Produktionen gleicher	
Zahlenformate mit unterschiedlicher Genauigkeit	8.21
Erstellung von Rahmenprogrammen für beliebige	
Fuzzy-Controller	8.24
Code-Parameter-Dateien	8.24
Erstellen einer Code-Schema-Datei	8.25

Code-Generierung unter Verwendung einer C-Code-Schablone	8.30
--	------

## **Bibliotheken 8.31**

Fuzzy-Bibliotheken	8.31
Numerische Bibliotheken	8.33
Allgemeine Eigenschaften	8.34
Standard Fließpunktzahlen F4, F8 und F10	8.36
2-Byte-Fließpunktzahlen F2	8.36
Festpunktzahlen I2 und I4	8.39

## **Der Schema-Umsetzer in FALCO 8.41**

Funktionsweise	8.41
Variablen und Anweisungen	8.42
Variablen	8.42
Anweisungen	8.42
#WF_code_block	8.43
#WF_code_block_end	8.43
#WF_write_to	8.43
#WF_include	8.43
#WF_define	8.44
#WF_insert_code_block	8.44
#WF_path	8.44
#WF_show_message	8.45
#WF_foreach	8.45
#WF_if	8.46
#WF_expr	8.47

---

---

# Übersicht

Der C-Quellcode-Generator FALCO<sup>1</sup> ermöglicht die Erzeugung von ANSI-C-Code für Fuzzy-Systeme, die mit Hilfe der WinFACT-Fuzzy-Shell FLOP entwickelt wurden. Der C-Quellcode wird auf zwei Dateien aufgeteilt. Eine von ihnen hat die Endung *c* und beinhaltet die Implementierung, die andere ist die zugehörige Headerdatei (Endung *h*); sie definiert die Schnittstelle zur eigenen Anwendung. FALCO zeichnet sich durch folgende Leistungsmerkmale aus:

- Komfortabler Editor für Dateien der Programmiersprache C.
- Parallele Bearbeitung mehrerer Systeme möglich.
- Der generierte C-Code liegt vollständig im Quelltext vor. Er ist kommentiert und gut lesbar gehalten. Durch Aufspaltung in verschiedene Dateien reduziert sich der spätere Aufwand beim Compilieren Ihrer Anwendung.
- Die Datentypen sind vom Anwender frei wählbar (16-Bit-Integer, 32-Bit-Integer, 2-Byte-Float, Float, Double, Long Double). Bei den Ganzzahltypen können Sie das Festpunktzahlenformat, bei dem 2-Byte-Float-Typ die Mantissen- und Exponentenbitbreite definieren.
- Durch Code-Schablonen werden selbst geschriebene Rahmenprogramme für beliebige Fuzzy-Systeme verwendbar.

Die Anwendung FALCO wurde als MDI-Anwendung (*Multiple Document Interface*) entworfen. Dies hat für Sie den Vorteil, gleichzeitig mehrere Code-Produktionen betrachten und vergleichen zu können. Das nachfolgende Bild zeigt das Anwendungsfenster von FALCO unmittelbar nach dem Aufruf des Programms.

---

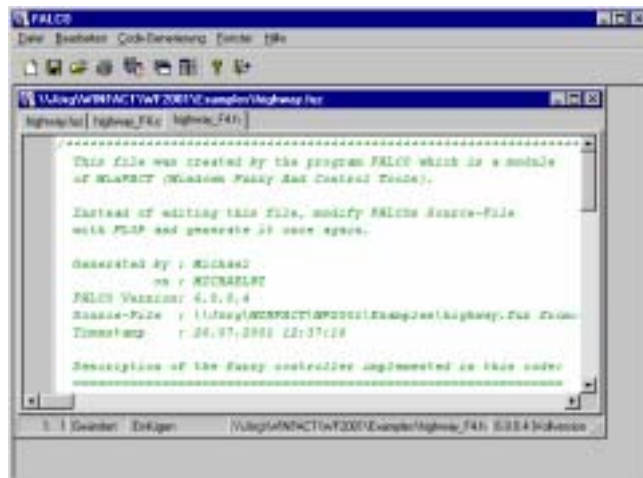
<sup>1</sup> Der Name FALCO steht für "Fuzzy Application C-Code-Generator". Dies finden wir nicht mehr ganz passend, daher erscheint im Info-Dialog "Code-Generierung aus Fuzzy-Controller-Dateien".



*Hauptfenster von FALCO nach dem Aufruf*

Das Fenster enthält neben den Standardkomponenten

- eine horizontale Toolbar unter dem Hauptmenü zum Direktzugriff auf die wichtigsten Funktionen,
- ein oder mehrere Dokumentfenster, die eine FUZ-Datei und die daraus erzeugten Code-Dateien enthalten.



*Dokumentfenster mit erzeugtem Code*

Die Statuszeile am Dokumentfensterrand zeigt die aktuelle Cursorposition, den Zustand des Dokumentes, den Modus (*Einfügen*, *Überschreiben* oder *Nur Lesen*), den vollständigen Namen der Datei und die Version der Software an. Das Dokumentfenster selbst enthält einen Editor, mit dem Sie die erzeugten C-Dateien komfortabel lesen und ändern können. Zum Ändern stehen alle bekannten Funktionen im Menü BEARBEITEN zur Verfügung. Ein Editieren von FUZ-Dateien ist nicht möglich. Die Menüpunkte des BEARBEITEN-Menüs, die Änderungen am Text vornehmen, werden demzufolge passiv geschaltet. Zu den Editierfunktionen gehören:

*Editier-  
funktionen*


- das Rückgängigmachen der letzten Änderung,
- das Wiederherstellen der letzten rückgängig gemachten Aktion,
- das Ausschneiden, Kopieren und Einfügen von Textblöcken in die Zwischenablage und
- das Suchen und eventuelle Ersetzen von Textblöcken.

---


---

## C-Quellcode-Generierung

### Starten der Code-Generierung

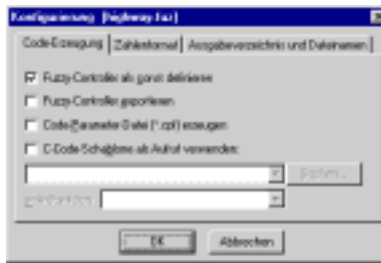
Die Code-Generierung lässt sich nur starten, wenn zuvor eine FUZ-Datei in das aktive Dokumentfenster geladen wurde. Einstellungen für die Generierung beziehen sich auf das aktive Dokumentfenster und können auch ohne eine FUZ-Datei vorgenommen werden. Ist eine FUZ-Datei geladen und sind alle Einstellungen getätigt, kann die Code-Generierung durch den Menüpunkt CODE-GENERIERUNG | CODE GENERIEREN oder die Schaltfläche  angestoßen werden.

## Laden von FUZ-Dateien

Das zu verarbeitende Fuzzy-System muss sich in einer Datei mit der Extension *FUZ* befinden, wie sie von der WinFACT-Fuzzy-Shell FLOP generiert wird. Die Datei kann über die Schaltfläche  oder die Menüfolge DATEI | ÖFFNEN... in das aktive Dokumentfenster geladen werden. Eine eventuell im Dokumentfenster vorhandene Datei wird dabei überschrieben. Falls Code-Produktionen zu dieser Datei erzeugt wurden, werden Sie gefragt, ob diese gespeichert werden sollen. Soll ein neues Dokumentfenster für das Fuzzy-System verwendet werden, so ist dieses zunächst über DATEI | NEU zu erzeugen.

## Einstellungen

Vor der Generierung des C-Codes sind in der Regel einige Einstellungen zu treffen, die unter anderem die Auflösung, d. h. den verwendeten C-Datentyp für die linguistischen Variablen und die Zugehörigkeitswerte betreffen. Sämtliche Einstellungen werden über CODE-GENERATOR | EINSTELLUNGEN... vorgenommen. Die in diesem Dialog getätigte Konfiguration des Code-Generators gilt nur für das aktive Dokumentfenster. Alle anderen bleiben hiervon unbeeinflusst.



*Dialog zur Konfiguration des Code-Generators*

Die Konfigurierung des Code-Generators erstreckt sich über drei Register:

- Code-Erzeugung
- Zahlenformat
- Ausgabeverzeichnis und Dateinamen

## Code-Erzeugung

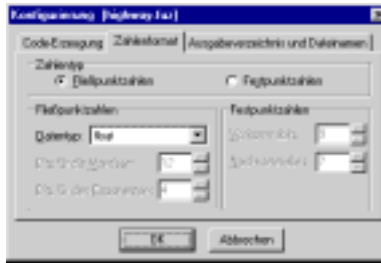
Das Register für die Code-Erzeugung bietet Ihnen die Möglichkeit, den Fuzzy-Controller als konstant zu definieren. Dies bedeutet, dass sämtliche Werte, die den Controller ausmachen, als Konstantendefinition im erzeugten C-Code auftreten. Sie haben im nächsten Kontrollfeld die Möglichkeit festzulegen, ob der Fuzzy-Controller exportiert werden soll. Wenn Sie sich für den Export entscheiden, wird die entsprechende Exportdefinition in die spätere Headerdatei geschrieben. Die weiteren Punkte dieses Registers sollen an dieser Stelle nur kurz umrissen werden. Sie werden später detailliert besprochen.

Betrachtet man beispielsweise eine C-Datei, die eine Datenstruktur definiert, so könnten der Typ und der Variablenname als Parameter einer anderen Codesequenz Verwendung finden, in der Sie die Struktur benutzen möchten. Die Codesequenz, die diese Parameter einbindet, besitzt somit für alle Dateien, die diese Parameter definieren, Gültigkeit. Dieser Ansatz wird durch die Code-Parameter-Dateien und die C-Code-Schablonen verfolgt. Soll eine Code-Parameter-Datei außerhalb von FALCO verwendet werden, so kann dies über die Einstellung *Code-Parameter-Datei erzeugen* bewerkstelligt werden.

Durch die Verwendung einer C-Code-Schablone können Sie den Aufruf der erzeugten C-Funktionen bewirken. Dazu muss eine CSH-Datei (Code-Schema-Datei) angegeben werden, die mindestens eine main-Funktion enthält (im Kapitel *Erstellung von Rahmenprogrammen für beliebige Fuzzy-Controller* wird die Verwendung von Code-Schema-Dateien anhand eines Beispiels näher erläutert). Sobald Sie eine CSH-Datei in das entsprechende Textfeld eingetragen haben (dies können Sie auch bequemer über die Schaltfläche *Suchen...* erledigen), erscheinen in der darunter stehenden Auswahlliste die zur Verfügung stehenden main-Funktionen.

## Zahlenformate

Der Code-Generator ist in der Lage, verschiedene Zahlenformate zu verwenden. Das gewählte Zahlenformat bezieht sich dabei auf alle reellwertigen Parameter des Fuzzy-Controllers. Alle anderen Parameter werden durch Ganzzahlen oder Aufzählungen (Enumerationen) abgebildet. Das Register *Zahlenformat* des Konfigurierungsdialoges bietet Ihnen die möglichen Einstellungen an.



Zahlenformate der Code-Generierung

Es sind folgende zwei generelle Zahlenformate möglich:

#### Datentypen

- Fließpunktzahlen
- Festpunktzahlen

Bei den Fließpunktzahlen werden sämtliche nativen C Datentypen (float, double, long double) unterstützt. Zusätzlich wird ein 2-Byte-Fließpunkttyp angeboten, bei dem die Anzahl der Mantissen- und Exponentenbits wählbar ist.

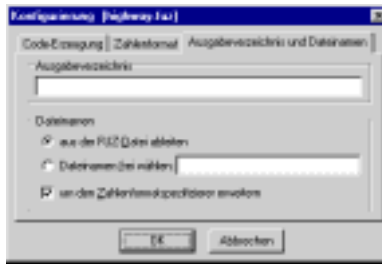
Die Anzahl der Mantissenbits muss zwischen 5 und 14, die der Exponentenbits zwischen 1 und 8 liegen. Nur innerhalb dieser Bereiche lässt sich mit dieser Zahlendarstellung vernünftig rechnen.

Die Festpunktzahlen werden ausschließlich über die Anzahl der Vorkomma- und Nachkommabits definiert. Ist die Summe beider größer als 15, so wird ein 4-Byte-Datentyp, sonst ein 2-Byte-Datentyp verwendet. Im Kapitel *Numerische Bibliotheken* finden Sie eine genaue Beschreibung zu den Festpunktzahlentypen und dem 2-Byte-Fließpunktzahlentyp.

## Ausgabeverzeichnis und Dateinamen

Da zu einer FUZ-Datei, insbesondere wenn mit Code-Schema-Dateien gearbeitet wird, mehrere Quelltext-Dateien generiert werden können, liegt es nahe, diese in ein Verzeichnis abzulegen. Die Wahl des Ausgabeverzeichnisses kann in dem Register *Ausgabeverzeichnis und Dateinamen* vorgenommen werden.





Register zur Konfigurierung von Ausgabeverzeichnis und Dateinamen

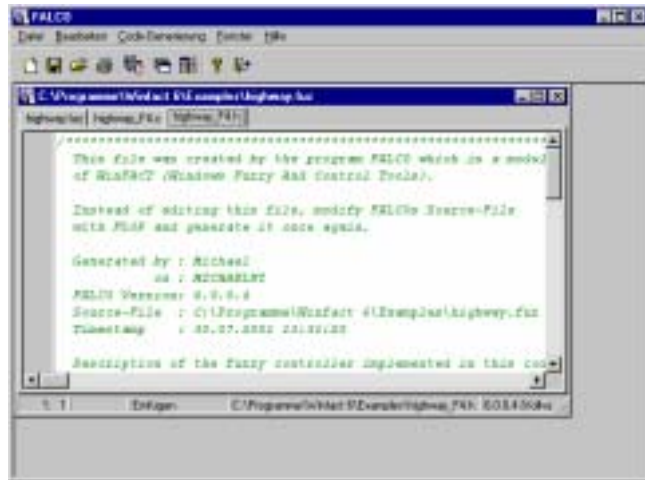
Der Dateiname einer jeden Quellcode-Datei kann aus der FUZ-Datei abgeleitet werden, er kann aber auch von Ihnen frei gewählt werden. Innerhalb des Eingabefeldes für einen frei gewählten Dateinamen darf keine Dateinamenserweiterung (eine Dateinamenserweiterung beginnt mit einem Punkt und endet mit dem Ende der Eingabe) angegeben werden. Soll dem Dateinamen ein Kürzel für das Zahlenformat angehängt werden, so erledigt dies die Auswahl *um den Zahlenformatspezifizierer erweitern*. So kann eine Doppelbezeichnung bei Generierung aus der gleichen FUZ-Datei mit unterschiedlichem Zahlenformat verhindert werden.

## Laden und Speichern der Einstellungen

Alle Einstellungen, die für ein Dokumentfenster getätigt wurden, können über den Menüpunkt DATEI | EINSTELLUNGEN SPEICHERN... gespeichert werden. Die Datei hat die Erweiterung *FCF* (FALCO-Konfigurationsdateien). Mit der Menüfolge DATEI | EINSTELLUNGEN LADEN ... können die gespeicherten Einstellungen für das aktive Dokumentfenster wieder hergestellt werden.

## Darstellung des generierten Codes

Der erzeugte C-Quellcode erscheint in dem gleichen Dokumentfenster, das jedoch nun mehrere Reiter besitzt, über die man die einzelnen Dateien der Code-Produktion erreicht.



*Dokumentfenster nach der Code-Generierung –  
es entsteht für jede Datei ein Reiter*

Im obigen Bild ist die Code-Generierung für die Beispiel-Datei *highway.fuz* ohne Änderung von Einstellungen getätigt worden. Es entstehen zwei Dateien:

1. Eine C-Datei, welche die Struktur des Controllers enthält und
2. eine Header-Datei, die die Schnittstelle zur C-Datei bildet.

Zur Erzeugung eines Programmes, das den generierten Fuzzy-Controller verwendet, sind noch weitere Dateien (Bibliotheken) notwendig, die zum Lieferumfang gehören. Diese enthalten einerseits Funktionen, die benötigt werden, um mit dem eingestellten Zahlenformat rechnen zu können, andererseits werden noch allgemeine Fuzzy-Controller-Funktionen benötigt (s. Kapitel *Verwendung des C-Codes*).

---

## Struktur des generierten Codes

Der von FALCO erzeugte Code besteht aus einer Headerdatei und einer C-Datei. Die Parameter, aus denen sich der Fuzzy-Controller zusammensetzt,

werden in einer Datenstruktur abgelegt. Die Berechnungsvorschrift, die diese Datenstruktur verwendet, ist in den Fuzzy-Bibliotheken, die im Quelltext vorliegen, implementiert. FALCO schreibt Funktionen, die die Funktionsaufrufe der Bibliothek für Sie tätigen.

## Datenstruktur eines Fuzzy-Controllers

Sie benötigen die Kenntnis der Datenstruktur, wenn Sie den Fuzzy-Controller nachträglich ändern möchten.

Die Datenstruktur, aus der sich ein Fuzzy-Controller zusammensetzt, besteht wiederum aus anderen Datenstrukturen. Die nachstehende Liste führt alle Angaben eines Fuzzy-Controllers auf.

- Anzahl der Eingangsvariablen
- Anzahl der Ausgangsvariablen
- Anzahl der Prämissen des Regelwerkes
- Linguistische Eingangsvariablen
- Linguistische Ausgangsvariablen
- Prämissen, Konklusionen und deren Gewichte des Regelwerkes
- Inferenzmechanismus
- Operator für Fuzzy-Und
- Operator für Fuzzy-Oder
- Defuzzifizierungsmethode
- Anzahl der Diskretisierungsschritte für die Schwerpunktmethoden

Alle Daten, die eine Anzahl beschreiben, sind positive Ganzzahlen. Die Angaben für Operatoren, Defuzzifizierungsmethode und Inferenzmechanismus sind Elemente aus Aufzählungen (Enumerationen), damit der C-Code gut lesbar bleibt. Die Deklarationen der Enumerationen stehen in der Datei *fuzzy\_enums.h*, die im Lieferumfang enthalten ist.

Die linguistischen Variablen wurden unterteilt in Eingangs- und Ausgangsvariablen, die unterschiedliche Datenstrukturen beinhalten. Beide bestehen aus einer bestimmten Anzahl Sets, aber nur linguistische Ausgangsvariablen enthalten einen vordefinierten Ausgangswert, der ausgegeben wird, falls der Ausgang durch keine Regel definiert wird und nicht der letzte Ausgangswert aus-

gegeben werden soll. Diese Diskrepanz wird durch die unterschiedlichen Datenstrukturen verdeutlicht.

Die Prämissen und Konklusionen sind eindimensionale Felder, die Ganzzahlen enthalten. Die Zahlen stellen einen Bezug zum jeweiligen Fuzzy-Set her (um 1 verschobener Index zum jeweiligem Fuzzy-Set). Stellt man sich die Regelbasis getrennt als zwei Tabellen vor, von denen die erste die Prämisse und die zweite die Konklusion enthält, so enthalten die Felder einfach nacheinander die Zeilen der jeweiligen Tabellen. Die Spaltennummer der Tabellen ist gleich dem Index der linguistischen Eingangs- bzw. Ausgangsvariablen. Treten in der Prämisse negative Zahlen auf, so bedeuten diese die Negation des entsprechenden Fuzzy-Sets.

Die Gewichtung wird als eindimensionales Feld von reellen Zahlen abgelegt. Man kann sie sich als dritte einspaltige Tabelle vorstellen.

Beispiel:

Prämisse		Konklusion		Gewichtung
E1	E2	A1	A2	
1	1	1	3	1
-1	2	1	2	0.5
1	3	2	1	0.75
1	4	2	1	0.8

Speicherung der Prämissen: {1, 1, -1, 2, 1, 3, 1, 4}

Speicherung der Konklusionen : {1, 3, 1, 2, 2, 1, 2, 1}

Speicherung der Gewichtungen : {1, 0.5, 0.75, 0.8}

Zur Komplettierung fehlt nur noch die Information, wie die Fuzzy-Sets gespeichert werden. Auch hier wurde ein einfaches Modell gewählt: Ein Fuzzy-Set besteht aus einem Feld an Wertepaaren und der Anzahl der Wertepaare. Jedes Wertepaar enthält dabei eine markante Stelle des Sets.

Zusammenfassend wird hier der den Fuzzy-Controller ausmachende Quellcode der Datei *fuzzy\_F2.h* gezeigt (das Kürzel *F2* ist ein Zahlenformatspezifizierer der für 2-Byte-Fließpunktzahlen steht). Der Typ *NumTypeF2Point\_t* wird in der Bibliothek für das Rechnen mit 2-Byte-Fließpunktzahlen definiert. Die *enum*-Typen werden in *fuzzy\_enums.h* definiert.

```
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;
```

## Funktionen eines Fuzzy-Controllers

Ein von FALCO generierter Fuzzy-Controller besteht aus vier Funktionen. Die Namen der Funktionen setzen sich aus dem Dateinamen der FUZ-Datei, einem

Zahlenformatspezifizierer (sofern dies eingestellt wurde) und dem eigentlichen Funktionsnamen zusammen.

Generiert man aus der Datei *highway.fuz* ohne Änderung von Einstellungen C-Code, so erhält man folgende Funktionsdeklarationen in der erzeugten Header-datei:

```
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);
```

Nachfolgend soll die Bedeutung der Funktionen geschildert werden.

Die Funktion *...SetNumType* dient dem Setzen der Parameter des gewählten Zahlenformates. Hat das Zahlenformat keine Parameter, so ist die Funktionsimplementierung in der generierten C-Datei leer. Lediglich die Festpunktzahlen und 2-Byte-Fließpunktzahlen besitzen weitere Parameter (Anzahl Vorkommabits und Anzahl Nachkommabits bzw. Anzahl der Bits für den Exponenten und Anzahl der Bits für die Mantisse). Bei diesen Formaten ist es zwingend, die Funktion vor den anderen aufzurufen. Es schadet aber nicht, sie generell für alle Zahlenformate vor den anderen Funktionen aufzurufen. In den Fällen, in denen die Implementierung fehlt, wird der Aufruf vom Compiler entfernt.

Die Initialisierungsfunktion (*...\_init*) reserviert Speicher für die Berechnungsfunktion. Sie muss daher **einmalig** vor derselben aufgerufen werden.

Die Berechnungsfunktion (*...\_calc*) ist die einzige Funktion, die mit Argumenten aufgerufen werden muss. Ihr werden die Eingangswerte des Fuzzy-Controllers und die Adressen der Variablen, in die die Ausgangswerte nach der Berechnung zu schreiben sind, übergeben. Die Eingangswerte und Ausgangswerte werden in der Reihenfolge angenommen, in der die linguistischen Eingangs- bzw. Ausgangsvariablen unter FLOP definiert wurden. Die Datentypen der Ein- und Ausgangswerte entsprechen dem bei der Generierung eingestellten Format. Die Datentypen selbst sind in den numerischen Bibliotheken definiert (s. Kapitel *Numerische Bibliotheken*). Die Berechnungsfunktion darf beliebig oft nacheinander aufgerufen werden.

Die Funktion zur Freigabe (*...\_free*) des durch die Initialisierungsfunktion reservierten Speichers darf erst aufgerufen werden, wenn die Berechnungsfunktion nicht mehr aufgerufen wird. Nach Aufruf der Freigabefunktion kann

die Initialisierungsfunktion erneut aufgerufen werden (erfolgt dieser Wechsel zwischen Anfordern und Freigeben von Speicher häufig innerhalb eines Programmablaufes, sollte sichergestellt sein, dass die Speicherverwaltung effizient genug ist, damit umzugehen).

---

---

## Verwendung des C-Codes

Es wird zunächst die einfache Verwendung besprochen. Anschließend soll ein Programm erstellt werden, das die Genauigkeit von mehreren Code-Generierungen vergleichen lässt. Zum Schluss wird auf Besonderheiten bei der Verwendung mehrerer Code-Produktionen mit gleichem Zahlenformat und der Verwendung von Code-Schema-Dateien eingegangen.

## Verwendung einer Code-Produktion

Die Verwendung des generierten Quellcodes innerhalb eines Programmes wird hier an einem Beispiel erläutert. Wir nehmen das Ergebnis der Code-Generierung aus der Beispieldatei *highway.fuz*, die Sie im Unterverzeichnis *Examples* finden. Für dieses Beispiel sollen folgende Einstellungen getroffen werden.

Code-Erzeugung:

*Fuzzy-Controller als const definieren*

Zahlenformat:

*Zahlentyp: Fließpunktzahlen*

*Datentyp: float*

Ausgabeverzeichnis und Dateinamen:

*Ausgabeverzeichnis: c:\temp*

*Dateinamen: aus FUZ-Datei ableiten und  
um den Zahlenformatspezifizierer erweitern*

Der generierte Code besteht aus zwei Dateien mit den Namen *highway\_F4.c* und *highway\_F4.h*. Ein Programm soll nun die Werte für die Eingänge des erzeugten Fuzzy-Controllers von der Tastatur einlesen und das Ergebnis auf dem Bildschirm ausgeben. Es soll dafür lediglich auf ANSI-C-Funktionen zurückgegriffen werden.

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

int main(int argc, char **argv)
{
    NumTypeF4_t e1, e2, a1;
    printf("Bitte geben Sie 2 Werte ein: ");
    scanf("%f %f",&e1, &e2);
    highway_F4_SetNumType();
    highway_F4_init();
    highway_F4_calc(e1, e2, &a1);
    highway_F4_free();
    printf("Ergebnis: %f",a1);
    fflush(stdin);
    getchar();
}
```

Dieses Programm liest über *scanf()* die Eingangswerte von der Tastatur, setzt das Zahlenformat (da hier in Wirklichkeit mit *float* gearbeitet wird, macht die Funktion *highway\_F4\_SetNumType* gar nichts), initialisiert den Fuzzy-Controller, ruft dann *highway\_F4\_calc()* zur Berechnung des Ausgangswertes auf, gibt den Fuzzy-Controller wieder frei und zeigt den Ausgangswert (das Ergebnis) auf dem Bildschirm an.

Der Header *stdio* wird für die Funktionen *printf*, *scanf*, *fflush* und *getchar* benötigt, *highway\_F4* für die Fuzzy-Controller-Funktionen. Der Datentyp *NumTypeF4\_t* wird in *NumType\_F4.h* deklariert und über das Einbinden von *highway\_F4.h* bekannt gemacht. Er entspricht dem Datentyp *float*. Schaut man sich an welche Dateien durch die *#include*-Direktive vom Compiler durchlaufen werden, so kommt man zu folgendem Bild (Standardheader weggelassen!):

```
highway_F4.h
    fuzzy_F4.h
        fuzzy_enums.h
            NumType_F4.h
```

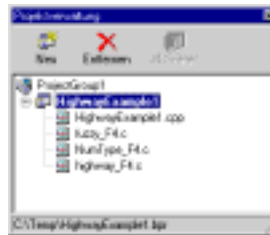
Die Einrückung gibt dabei an, welche Datei welchen Header einbindet. Soll nun dieses recht triviale Programm erstellt werden, so sind dem Compiler diese Headerdateien und die gleichnamigen C-Dateien zugänglich zu machen (zu *fuzzy\_enums.h* gibt es keine C-Datei). Dazu müssen Sie innerhalb ihres Compilers entsprechende Suchverzeichnisse eintragen. Das Verzeichnis für die Fuz-



zy-Headerdateien und C-Dateien liegt im *AutoCode*-Verzeichnis und heißt *fuzzy*, das für die unterschiedlichen numerischen Datentypen liegt ebenfalls im *AutoCode*-Verzeichnis und heißt *NumType*. Diese beiden Unterverzeichnisse müssen dem Compiler als Suchpfade mitgegeben werden, damit die Compilierung problemlos verläuft.

Nach dem Compilierungsvorgang erfolgt das Linken der erzeugten Objektdateien. Dem Linker müssen an dieser Stelle ebenfalls die Pfade zu den benötigten Dateien bekannt gemacht werden, damit alle dateiübergreifenden Referenzen geschlossen werden können. Wie im Einzelnen die Suchpfade von Linker und Compiler einzustellen sind, muss dem Handbuch oder der Hilfe des Compilers entnommen werden.

Als Beispiel wird hier das Projekt unter dem Borland C++ Builder in der Version 3.0 gezeigt.



Das Beispiel im Borland C++ Builder (Version 3.0)

Wie zu sehen ist, besteht das Projekt aus den drei C-Dateien der oben aufgeführten Headerdateien und aus dem Hauptprogramm *HighwayExample1.cpp*, das zu dem obigen Listing lediglich die automatisch von der Entwicklungsumgebung erzeugten Zeilen enthält. Führt man das fertige Programm aus, so erhält man folgende Ausgabe:



Das Beispielprogramm nach der Ausführung

## Verwendung mehrerer Code-Produktionen unterschiedlicher Zahlenformate

Es soll anhand eines Beispiels gezeigt werden, wie mehrere Code-Generierungen einer FUZ-Datei mit verschiedenen Zahlenformaten verglichen werden können. Die Einstellungen zur Code-Generierung werden vom letzten Kapitel übernommen. Die Einstellung des Zahlenformates wird dann geändert.

Folgende Zahlenformate werden verwendet:

1. 2-Byte-Fließpunkt (5-Bit-Exponent, 11-Bit-Mantisse),
2. 2-Byte-Festpunkt (9-Bit-Vorkomma, 7-Bit-Nachkomma) und
3. Fließpunktzahlen vom Typ *double*

Zur Code-Generierung können Sie drei Dokumentfenster über DATEI | NEU erstellen und darin jeweils die Datei *highway.fuz* laden. Tätigen Sie nun die Einstellungen für ein Dokumentfenster und speichern Sie diese. Anschließend laden Sie die gespeicherten Einstellungen für die anderen Dokumentfenster und ändern dort nur noch die Einstellung des Zahlenformates. Achten Sie unbedingt darauf, dass sich Aktionen (Einstellen, Laden und Speichern), die Einstellungen betreffen, immer auf das aktive Dokumentfenster beziehen!

Wenn alles richtig gemacht wurde, erhalten Sie folgende Dateien:

*highway\_F8.h* und *highway\_F8.c*

*highway\_F2.h* und *highway\_F2.c*

*highway\_I2.h* und *highway\_I2.c*

Natürlich müssen Sie nicht in der hier beschriebenen Weise vorgehen, Sie können auch ein Dokumentfenster verwenden und die Dateien Schritt für Schritt erzeugen (dreimalig: Einstellungen ändern, Code generieren, Dateien speichern).

Nun werden diese Dateien in das Hauptprogramm eingebunden:

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

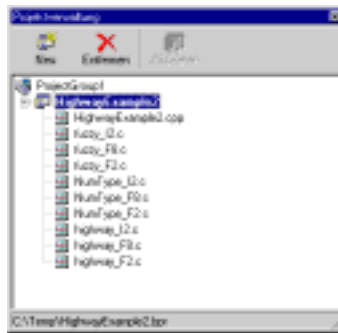
int main(int argc, char **argv)
{
    NumTypeF8_t f8e1, f8e2, f8a1;
    NumTypeF2_t f2e1, f2e2, f2a1;
    NumTypeI2_t i2e1, i2e2, i2a1;
    NumeratorF2_t f2n;
    DenominatorF2_t f2d;
    NumeratorI2_t i2n;
    DenominatorI2_t i2d;
    double rf2, ri2;

    printf("Bitte geben Sie 2 Werte ein: ");
    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("Ergebnis(F8 F2 I2): %lf %lf %lf", f8a1, rf2, ri2);
    fflush(stdin);
    getchar();
}
```

Es gelten hier die gleichen Bedingungen bezüglich der Suchpfade zum Compilieren wie beim *Verwenden einer Code-Produktion*. Neu an diesem Quelltext sind die Funktionsaufrufe, die mit *op* beginnen, und die Numerator- und De-

nominator datentypen; sie werden in den Bibliotheken *NumType\_F2* bzw. *NumType\_I2* definiert. Die *opXXFromFraction*-Funktionen wandeln einen durch Zähler und Nenner gegebenen Bruch in das Zahlenformat XX. Die *opXXToFraction*-Funktionen wandeln die im Zahlenformat XX als erstes Argument übergebene Zahl in einen Bruch, dessen Zähler und Nenner in die letzten beiden Argumente abgelegt werden. Jede Bibliothek, die ein Zahlenformat definiert, das nicht ein Standardzahlenformat (float, double, long double) ist, definiert auch diese beiden Funktionen zur Umwandlung von Brüchen.

Das Projekt im Borland C++ Builder (Version 3.0) sieht folgendermaßen aus:



Die Projektverwaltung des Borland C++ Builders (Version 3.0)

Wie dem obigen Bild zu entnehmen ist, sind entsprechende Dateien für die Fuzzy-Controller-Funktionen der einzelnen numerischen Datentypen (*fuzzy\_XX.c*) und die Dateien für das Rechnen mit diesen Datentypen (*NumType\_XX.c*) in das Projekt einzubinden.

Wird nun das Projekt erstellt und ausgeführt, so ergibt der Programmablauf folgende Ausgabe:



*Konsolenfenster nach der Ausgabe des Programmes*

Der Vergleich der Genauigkeit der einzelnen Code-Produktionen für unterschiedliche numerische Datentypen lässt sich nun an dem Ergebnis durchführen. Das Ergebnis der 8-Byte-Fließpunktzahlen kann dabei als das hinreichend genaue angesehen werden. Vergleicht man nun die anderen beiden Ergebnisse mit diesem, so ist offensichtlich, dass der Fuzzy-Controller mit Festpunktzahlenarithmetik eine geringere Genauigkeit besitzt als mit 2-Byte-Fließpunktzahlenarithmetik.

## **Verwendung mehrerer Code-Produktionen gleicher Zahlenformate mit unterschiedlicher Genauigkeit**

Die Zahlenformate, die eine Einstellung der Genauigkeit erlauben, sind die 2-Byte-Fließpunktzahlen und die Festpunktzahlen. Die Genauigkeit der Fließpunktzahlen wird über die Anzahl der Bits in der Mantisse bestimmt, die der Festpunktzahlen durch die Anzahl der Bits des Nachkommateils. Dieser Abschnitt gilt nur für die oben genannten Zahlenformate!

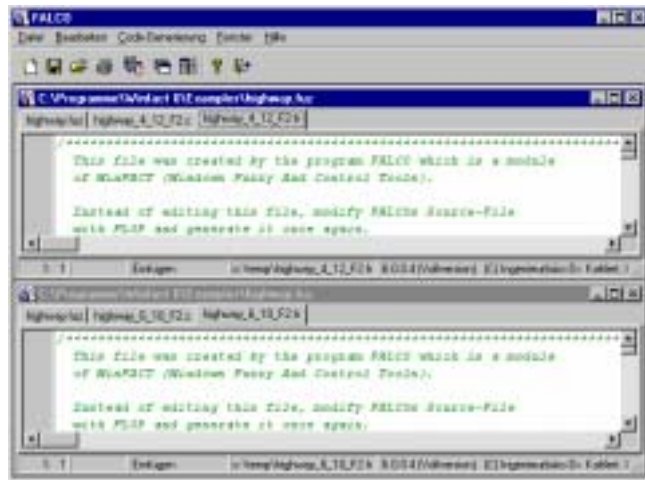
Wieder ausgehend von der Datei *highway.fuz* sollen zwei Code-Produktionen erzeugt werden mit dem Zahlenformat 2-Byte-Fließpunkt, das eine Mal mit 4 Bits für den Exponenten und 12 Bits für die Mantisse, das andere Mal mit 6 Bits für den Exponenten und 10 Bits für die Mantisse. Da beide Code-Produktionen 2-Byte-Fließpunktzahlen verwenden, hilft in diesem Fall die Erweiterung des Dateinamens um den Zahlenformatspezifizierer nicht weiter. Verwenden Sie daher zusätzlich die Möglichkeit, den Dateinamen frei zu wählen.

Hier werden als Dateinamen

*highway\_4\_12* und

*highway\_6\_10*

gewählt. Nach Generierung des Codes unter Verwendung zweier Dokumentenfenster sieht FALCO wie folgt aus:



*Erzeugung zweier unterschiedlicher Code-Produktionen aus der gleichen FUZ-Datei*

Es sollen nun diese Code-Produktionen mit einer mit dem Fließpunktzahlentyp *double* generierten verglichen werden (letzterer wurde bereits im vorherigen Abschnitt erzeugt). Das Programmlisting ist dem aus dem Abschnitt *Verwendung mehrerer Code-Produktionen unterschiedlicher Zahlenformate* sehr ähnlich.

```
#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("Bitte geben Sie 2 Werte ein: ");
    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;

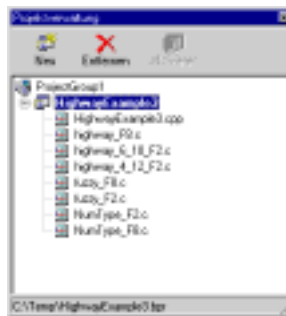
highway_6_10_F2_SetNumType();
highway_6_10_F2_init();
f2e1 = opF2FromFraction(f8e1*100,100);
f2e2 = opF2FromFraction(f8e2*100,100);
highway_6_10_F2_calc(f2e1, f2e2, &f2a1);
highway_6_10_F2_free();
opF2ToFraction(f2a1, &f2n, &f2d);
r_6_10_f2=(double)f2n/f2d;

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

```

Im obigen Listing wurde der Ablauf der einzelnen Berechnungen separiert. Der Grund ist der folgende: Die Bibliothek für das 2-Byte-Fließpunktzahlenformat kann nur mit einer Angabe für Mantissen- und Exponentenbits rechnen. Würden die *SetNumType*-Funktionen direkt hintereinander wie im vorherigen Abschnitt aufgerufen, so würde nur der letzte Aufruf Wirkung zeigen.

Die Darstellung der Projektverwaltung zeigt, welche Dateien zu diesem Beispiel gehören:



*Projektverwaltung des Borland C++ Builders (Version 3.0)*

Auf die Ausgabe des Programmes als Grafik soll hier verzichtet werden. Folgende Zeilen verdeutlichen die Programmausführung.

Bitte geben Sie 2 Werte ein: 25 80

Ergebnis (F8 4\_12 6\_10): 64.516129 64.531250 64.375000

## Erstellung von Rahmenprogrammen für beliebige Fuzzy-Controller

Nach den bisherigen Ausführungen ist es Ihnen möglich, eine durch FALCO erzeugte Code-Produktion manuell in ein Programm einzubinden. In diesem Abschnitt wird erläutert, wie ein Programm für beliebige Fuzzy-Controller verwendet werden kann. Dazu sind die Kenntnisse über den Aufbau von Code-Schema-Dateien von Vorteil (s. Kapitel *Der Schema-Umsetzer in FALCO*). Allerdings sind die Anweisungen so verständlich, dass das entsprechende Kapitel auch erst später gelesen werden kann.

Das Programm soll alle Eingangswerte eines Fuzzy-Controllers von der Tastatur lesen und die Ausgangswerte auf dem Bildschirm ausgeben. Dies soll durch den Aufruf einer Funktion geschehen. Die main-Funktion ruft also einfach nur diese C-Funktion auf. Sie ist folgendermaßen in einer ebenfalls zu erzeugenden Headerdatei *KeybInFuzzyMonOut.h* zu deklarieren:

```
void KeybInFuzzyMonOut(void);
```

Die Implementation soll in die Datei *KeybInFuzzyMonOut.c* geschrieben werden.

## Code-Parameter-Dateien

Betrachtet man eine Code-Parameter-Datei, so erkennt man leicht, welchen Parametern eine zu schreibende C-Funktion aufgrund des generierten C-Codes unterliegt. Im nachfolgenden Listing wird die Code-Parameter-Datei, die bei Generierung der Datei *highway.fuz* für Fließpunktzahlenarithmetik (Datentyp *float*) erzeugt wurde, dargestellt.

```
@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 der Code-Parameter-Datei werden Variablen für den Code-Schema-Umsetzer definiert, die durch @ eingeschlossen sind. Auf die Bedeutung der Variablen wird nicht im Einzelnen eingegangen, da sie durch die Namen schon gegeben ist (weitere Information folgt implizit im nächsten Abschnitt). Sie beinhalten Listen (die hier nur ein Element groß sind) und haben in den Code-Schema-Dateien folgende Funktionalität:

- `@WF_num_inputs@` wird ersetzt durch die rechts vom Gleichheitszeichen stehende Liste (hier nur die '2').
- `@WF_num_inputs[i]@` wird ersetzt durch das *i*-te Element der rechts vom Gleichheitszeichen befindlichen Liste (ist hier *i*>0, gibt es einen Fehler, da die Liste nur ein Element enthält und der Index bei Null beginnt; ist *i*=0, so ist wieder die '2' gemeint; für den obigen Fall gilt also `@WF_num_inputs@=@WF_num_inputs[0]@`)

Die indizierte Form ist konkreter und zu bevorzugen, auch wenn sie für diesen Fall identisch mit der nicht indizierten ist. Die Variablen `@WF_par_type@` und `@WF_par_var@` sind nur dann gefüllt, wenn der Fuzzy-Controller bei der Code-Generierung exportiert wurde. `@WF_num_format@` wird in dem Abschnitt zu `#WF_expr` (im Kapitel *Der Schema-Umsetzer in FALCO*) erläutert.

## Erstellen einer Code-Schema-Datei

Das Erstellen eines Gerüsts einer Code-Schema-Datei kann schrittweise in verbaler Form erfolgen. So wird der inhaltliche Ablauf beschrieben und kann im zweiten Schritt direkt in das Code-Schema, das aus Variablen und Anweisungen für den Code-Schema-Umsetzer und reinem C-Code besteht, transferiert werden. Bei der verbalen Formulierung helfen sinngemäße Übersetzungen der Anweisungen für den Schema-Umsetzer, um zu verdeutlichen, welche Anweisungen geeignet erscheinen.

Nachfolgend wird das oben stehende Beispiel wieder aufgegriffen. Es werden abwechselnd die verbale Beschreibung und deren Umsetzung in ein Code-Schema aufgeführt:

1. Als erstes soll in die Headerdatei *KeybInFuzzyMonOut.h* geschrieben werden. Es ist lediglich die Funktionsdeklaration zu schreiben. Falls die

Headerdatei durch ein größeres Projekt eventuell mehrfach eingebunden wird, kann dies durch die *#ifndef*-C-Präprozessordirektive vermieden werden. Weiterhin muss durch die *extern "C"*-Direktive spezifiziert werden, falls ein C++ tauglicher Compiler eingesetzt wird, dass die Funktion nach den C-Bindungskonventionen einzubinden ist.

```
#WF_write_to(KeybInFuzzyMonOut.h)

#ifndef KeybInFuzzyMonOut
#define KeybInFuzzyMonOut KeybInFuzzyMonOut

#ifdef __cplusplus
extern "C" {
#endif

    void KeybInFuzzyMonOut(void);

#ifdef __cplusplus
}
#endif

#endif
```

2. Die Headerdatei ist damit fertiggestellt und wir schreiben von nun an die Funktionsimplementierung in die Datei *KeybInFuzzyMonOut.c*, die diese Headerdatei und eine Headerdatei, die von FALCO generiert wurde, einbindet. Zusätzlich wird noch die dem ANSI-C-Standard zugehörige Headerdatei *stdio.h* für die Ein- und Ausgabefunktionen benötigt.

```
#WF_write_to(KeybInFuzzyMonOut.c)

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

void KeybInFuzzyMonOut(void)
{
```

3. Für jeden Eingang und für jeden Ausgang sind Variablen anzulegen, die Eingangswerte bzw. Ausgangswerte speichern. Ihr Typ ist der eingestellte numerische Datentyp.

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

4. Falls 2-Byte-Fließpunkt oder Festpunktzahlen verwendet werden, müssen Variablen zur Umrechnung von und zu Brüchen angelegt werden. Ebenso werden Variablen zur Speicherung des reellen Wertes eines Bruches benötigt.

```
#WF_if (@WF_nf@ ~~ F2,I2,I4)
```

```
Numerator@WF_nf@_t    n;
Denominator@WF_nf@_t  d;
double                reell;
#WF_endif
```

##### 5. Setzen des Zahlenformates (Aufruf der *SetNumType*-Funktion)

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

##### 6. Für jede Eingangsvariable ist ein Wert von der Tastatur zu lesen und, falls 2-Byte-Fließpunkt- oder Festpunktzahlen verwendet werden, muss jeder eingegebene Wert in das Zahlenformat umgerechnet werden.

```
for (i=0; i<@WF_num_inputs[0]@; i++){
    printf("Bitte geben Sie den %d. Wert ein:",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
}
```

Anmerkung:

Die Schleifenvariable *i* muss bei der Zusammensetzung der Code-Schemafragmente noch definiert werden. Außerdem ist die Variable *reell* nur vorhanden wenn 2-Byte-Fließpunkt- oder Festpunktzahlen verwendet werden; dies muss ebenfalls geändert werden.

##### 7. Aufruf der Initialisierungsfunktion

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

##### 8. Aufruf der Berechnungsfunktion mit allen Eingangswerten und allen Adressen der Ausgangsvariablen.

```
@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. Aufruf der Freigabefunktion

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

10. Alle Ausgangsvariablen sollen auf dem Bildschirm ausgegeben werden. Dabei muss, falls 2-Byte-Fließpunkt- oder Festpunktzahlen verwendet werden, der berechnete Wert in das Zahlenformat umgerechnet werden.

```
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. Ausgang = %lf\n", i+1, reell);
}
```

11. Abschließend soll auf einen Tastendruck gewartet werden und die unter Punkt 2 geöffnete Klammer muss wieder geschlossen werden.

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

Unter Berücksichtigung der Anmerkung unter Punkt 6 und Hinzufügen der `#WF_code_block`-Anweisungen erhält man folgendes Code-Schema (zur besseren Abgrenzung der durch dieses Schema erzeugten Dateien wurden die entsprechenden Zeilen hervorgehoben):

```
#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("Bitte geben Sie den %d. Wert ein:",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. Ausgang = %lf\n", i+1, reell);
}
fflush(stdin);
getchar();
}
#WF_code_block_end

```

Wichtig ist, dass die Anweisung zur Eröffnung eines Code-Blockes als letztes mit dem Argument *MAIN* abgeschlossen wird. Nur dann wird diese Funktion in FALCO überhaupt als main-Funktion einstellbar.



---

---

# Bibliotheken

Der generierte Quellcode enthält nicht alle Informationen, die zur Funktion des Fuzzy-Controllers erforderlich sind. Die Numerik und die generelle Funktion eines Fuzzy-Controllers sind bewusst ausgelagert. Dies erhöht die Übersichtlichkeit und beschleunigt eine wiederholte Compilierung bei größeren Projekten, da nicht mehr alles übersetzt werden muss. Die Auslagerung der Numerik wurde für jeden numerischen Typ einzeln durchgeführt. Für die generellen Fuzzy-Bibliotheken existiert eine Code-Schema-Datei, mit der sich die einzelnen *fuzzy\_XX.h*- und *fuzzy\_XX.c*-Dateien erzeugen lassen (das XX im Namen der Dateien steht für den Zahlenformatspezifizierer).

## Fuzzy-Bibliotheken

Die Fuzzy-Bibliotheken sind, wie oben schon erwähnt, allgemein als Code-Schema verfasst worden. Die Datei *CreateFuzzyLibraries.CSH* kann dazu benutzt werden, die Fuzzy-Bibliotheken für alle numerischen Datentypen zu erstellen. Es ist lediglich erforderlich, eine FUZ-Datei in ein Dokumentfenster zu laden und als Einstellung für die Code-Generierung

- die C-Code-Schema-Datei zu verwenden,
- als main-Funktion *Create Fuzzy Libraries* einzustellen und
- das Ausgabeverzeichnis anzupassen.

Der Vorteil dieser Vorgehensweise ist, dass man Änderungen nur an einer Datei zu tätigen braucht, das Code-Schema anwendet und alle Fuzzy-Bibliotheken mit dieser Änderung versehen hat. Sollen nur bestimmte Bibliotheken (z. B. die für Festpunktzahlen) die Änderungen erfahren, so lässt sich dies mit Hilfe der *#WF\_if*-Anweisung des Code-Schema-Umsetzers lösen.

Eine Fuzzy-Bibliothek definiert sämtliche Operationen, die zur Berechnung eines Fuzzy-Controllers notwendig sind. Dazu gehören folgende grundlegende Funktionen, auf die nicht näher eingegangen wird:

- Bestimmung des Zugehörigkeitswertes
- die Operation Fuzzy-Und
- Berechnung der Inferenz

- Bestimmung eines Ausgangswertes durch verschiedene Defuzzifizierungsmethoden

Weiterhin werden als Schnittstelle durch die Headerdatei folgende Funktionen herausgeführt (die *XX* sind zu ersetzen durch den jeweiligen Zahlenformatspezifizierer):

- Eine Initialisierungsfunktion:

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

- Eine Berechnungsfunktion:

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

- Eine Freigabefunktion:

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

Die Deklarationen der Schnittstellenfunktionen dürfen niemals verändert werden, da diese in genau dieser Form von FALCO erwartet werden. Die Implementierung hingegen dürfen Sie ändern. In der Headerdatei werden auch die Datentypen der Argumente definiert. Von den Datentypen wurde *FuzzyControllerXX\_t* schon im Abschnitt *Datenstruktur eines Fuzzy-Controllers* beschrieben. Diese Datenstruktur wird von FALCO ebenfalls exakt in dieser Form erwartet. Änderungen an ihr sind nicht zulässig!

Die Datenstruktur *FCMemXX\_t* dient der Speicherung innerer Zustände. Ihre Felder werden durch die Initialisierungsfunktion allokiert und in der Freigabefunktion freigegeben. Hier die Deklaration der Datenstruktur:

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

Nach einem Aufruf der Berechnungsfunktion enthält sie folgende Informationen:

- Ob das *n*-te Set der *m*-ten linguistischen Eingangsvariablen durch den scharfen Eingangswert berührt wurde:

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



- Welcher Zugehörigkeitswert für den scharfen Eingangswert bezüglich des  $n$ -ten Sets der  $m$ -ten Eingangsvariablen berechnet wurde:

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

- Ob das  $n$ -te Set der  $m$ -ten linguistischen Ausgangsvariablen durch eine aktive Regel erfasst wurde:

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

- Zu welchem Grad das  $n$ -te Set der  $m$ -ten linguistischen Ausgangsvariablen durch eine aktive Regel erfüllt wurde:

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

- Welcher scharfe Wert für die  $m$ -te linguistische Ausgangsvariable ermittelt wurde:

```
res[m]
```

FALCO legt in dem generiertem Code eine Variable mit dem Typ dieser Datenstruktur an. Soll auf diese Variable zugegriffen werden, so muss der Fuzzy-Controller exportiert werden (s. Kapitel *Einstellungen*). Nur dann legt FALCO die Variable ohne den Speicherklassenspezifizierer *static* an (sie wird dann vom Compiler als *extern* betrachtet, nur dass die Bindungsart nicht explizit angegeben ist). Mit Hilfe einer Code-Schema-Sequenz kann die Variable in die zugehörige Headerdatei geschrieben werden, was den Zugriff bewerkstelligt.

```
#WF_code_block(Zugriff auf FCMem)
#WF_write_to(@WF_out_file[0]@.h)

#ifdef @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
```

## Numerische Bibliotheken

Die numerischen Bibliotheken definieren die mathematischen Grundoperationen (+, -, \*, /) für einen bestimmten numerischen Datentyp.

Für das Verständnis eines generierten Codes ist das Verstehen der Bibliotheken normalerweise nicht erforderlich! Es kann jedoch sinnvoll sein, eine spezielle Implementierung der Numerik für eine bestimmte Zielhardware vorzunehmen, um deren Fähigkeiten besser auszuschöpfen.

## Allgemeine Eigenschaften

Es ist in ANSI-C nicht möglich, Operatoren abhängig vom Variablentyp zu definieren. Will man aber von der Numerik unabhängigen C-Code schreiben, so lässt sich dies nur durch Funktionen bewerkstelligen, die diese grundlegenden Operationen nachbilden. Bei den Fließpunktzahlentypen (*float*, *double* und *long double*) sind diese Funktionen lediglich als Makros ausgeführt, die die ursprüngliche Operation enthalten. Bei den Festpunktzahlen und den nicht standardisierten Fließpunktzahlen sind die Funktionen echte Implementierungen.

Die Namensgebung der numerischen Bibliotheken ist eindeutig, damit ein Mischen verschiedener Typen innerhalb einer Anwendung möglich wird. Die Eindeutigkeit wird durch einen Zahlenformatspezifizierer erzielt.

Die nachfolgende Tabelle stellt den Zusammenhang dar.

Zahlenformat	Spezifizierer	Bibliotheksnamen
2-Byte-Festpunktzahlen	I2	NumType_I2.c NumType_I2.h
4-Byte-Festpunktzahlen	I4	NumType_I4.c NumType_I4.h
2-Byte-Fließpunktzahlen	F2	NumType_F2.c NumType_F2.h
4-Byte-Fließpunktzahlen	F4	NumType_F4.c NumType_F4.h
8-Byte-Fließpunktzahlen	F8	NumType_F8.c NumType_F8.h
10-Byte-Fließpunktzahlen	F10	NumType_F10.c NumType_F10.h

Jede Bibliothek definiert einen Datentyp *NumTypeXX\_t*, wobei das *XX* durch den Spezifizierer zu ersetzen ist. Dieser Datentyp wird von einem grundlegendem Datentyp abgeleitet (s. u.). Mit diesem Typ können alle Berechnungen durchgeführt werden. Des weiteren steht ein Datentyp für Wertepaare zur Verfügung (*NumTypeXXPoint\_t*), der aus zwei Feldern *x* und *y* des Datentyps *NumTypeXX\_t* besteht.

Bei dem Zahlenformat F4 (4-Byte-Fließpunktzahlen) sehen die Deklarationen folgendermaßen aus:

```
typedef float NumTypeF4_t;
```

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

Der grundlegende Datentyp der Bibliothek für 4-Byte-Fließpunktzahlen ist *float*.

Die Bibliotheken definieren Konstanten vom Typ *NumTypeXX\_t* für die maximalen und minimalen darstellbaren Zahlen sowie für den Wert 1:

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

Ebenso definiert jede Bibliothek ihre eigenen Funktionen (teilweise auch Makros; s. o.) zur Implementierung von binären und unären Operatoren. Alle Namen der Funktionen, die Operatoren darstellen, fangen mit *op* gefolgt von dem Zahlenformatspezifizierer an. Sie enden auf den Namen der Funktion, die sie beinhalten (*Mul*, *Div*, *Add*, ...).

In der nachstehenden Tabelle werden Funktionen aufgelistet, die in allen Bibliotheken enthalten sind. Die verwendeten Variablenbezeichner *a*, *b* und *c* sind dabei immer vom Datentyp *NumTypeXX\_t*.

Operatorfunktion	Bedeutung
$a = \text{opXXAdd}(b, c)$	$a = b + c$
$a = \text{opXXSub}(b, c)$	$a = b - c$
$a = \text{opXXMul}(b, c)$	$a = b * c$
$a = \text{opXXDiv}(b, c)$	$a = b / c$
$a = \text{opXXNeg}(b)$	$a = -b$
$a = \text{opXXInv}(b)$	$a = 1 / b$
$i = \text{opXXG}(b, c)$	$i = b > c$
$i = \text{opXXGE}(b, c)$	$i = b \geq c$
$i = \text{opXXL}(b, c)$	$i = b < c$
$i = \text{opXXLE}(b, c)$	$i = b \leq c$
$i = \text{opXXE}(b, c)$	$i = b == c$
$i = \text{opXXNE}(b, c)$	$i = b != c$

<code>a = opXXCast(d)</code>	<code>a = d</code> im Zahlenformat <code>XX</code> Der Variablentyp von <code>d</code> muss dabei dem grundlegenden Datentyp des verwendeten Zahlenformates entsprechen.
------------------------------	---

Die Operatorfunktionen erlauben es, in Verbindung mit dem Code-Schema-Umsetzer C-Code zu schreiben, der vom Zahlenformat unabhängig ist (s. Kapitel *Fuzzy-Bibliotheken*). Dieses wird allerdings erkauft durch einen gewissen Verlust an Lesbarkeit, den man jedoch ein wenig reduzieren kann.

Es soll kurz anhand eines arithmetischen Ausdrucks gezeigt werden, wie eine derartige Umsetzung aussieht. Der Ausdruck

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

hat mit den obigen Operatorfunktionen für 2-Byte-Fließpunktzahlen folgendes Aussehen:

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

Die Lesbarkeit dieser Ausdrücke kann durch Einrückung, wie es in Programmiersprachen allgemein üblich ist, erhöht werden.

## Standard Fließpunktzahlen F4, F8 und F10

Die für die Formate F4, F8 und F10 verwendeten Fließpunktzahlen entsprechen den Variablentypen *float*, *double* und *long double* (dies sind auch die entsprechenden grundlegende Datentypen für *opFXCast*). Alle Operatorfunktionen sind als Makros ausgeführt. Die Division durch Null hat als Ergebnis die maximal darstellbare positive bzw. negative Zahl (je nach Vorzeichen des Dividenden) des entsprechenden Formates.

## 2-Byte-Fließpunktzahlen F2

Die 2-Byte-Fließpunktzahlen besitzen einstellbare Anzahlen an Exponenten- und Mantissenbits. Daher müssen diese Einstellungen der Bibliothek mitgeteilt werden, bevor mit diesem Zahlentyp gerechnet werden kann. Dies erfolgt durch den Aufruf der Funktion *SetNumTypeF2Format*, deren Deklaration hier zu sehen ist:

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

Berechnungen, die vor dem Aufruf erfolgen, sind unbestimmt. Erfolgt ein weiterer Aufruf mit einer anderen Anzahl Exponenten- oder Mantissenbits, so dürfen die Ergebnisse der vorausgehenden Berechnungen nicht ohne weiteres verwendet werden.

Beispiel:

```
NumTypeF2_t a, b, c;

SetNumTypeF2Format(11, 4);
/*ein wenig rechnen*/
a=F2One;
b=opF2Add(F2One, F2One);

SetNumTypeF2Format(10, 5);
/*ab hier werden a und b falsch interpretiert*/
c=opF2Sub(b, a); /*c ist hier nicht = F2One !!!!*/

SetNumTypeF2Format(11, 4);
/*ab hier stimmt die Interpretation von a und b wieder*/
c=opF2Sub(b, a); /*c = F2One !!!!*/
```

Um zu erläutern, was im Einzelnen in diesem Beispiel passiert, ist zunächst zu klären, wie Werte im Format F2 gespeichert bzw. interpretiert werden.

Die Speicherung der Zahlenwerte erfolgt in dem grundlegenden Datentyp *short* in Anlehnung an das IEEE-Format. Dies bedeutet, dass die Darstellung der Gleitpunktzahl  $x$  auf einer Zerlegung in ein Vorzeichen  $v$ , eine Mantisse  $m$  und einen Exponenten  $e$  zur Basis 2 beruht:

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

Da der Darstellungsbereich von  $m$  und  $e$  endlich ist, tritt ein entsprechender Fehler bei der Abbildung auf; die Zahl  $x$  wird also nicht genau repräsentiert. Die Genauigkeit ist durch die Anzahl der Bits in der Mantisse, der Wertebereich durch die Anzahl der Bits im Exponenten bestimmt. Werden 4 Bits für den Exponenten und 11 Bits für die Mantisse eingestellt, so wird eine Zahl folgendermaßen innerhalb der 2 Byte abgebildet:

Bitposition:	15	14	11	10	0
	v	eeee	mmm	mmmm	mmmm

Dabei hat  $v$  für negative Zahlen den Wert 1, sonst 0. Die Mantisse wird so erzeugt, dass ihr Wert zwischen 1 und 2 liegt. Bei der Speicherung wird die 1 vor dem Komma nicht mit abgelegt. Dem Exponenten wird eine Verschiebung (Bias) addiert, so dass dieser nur positiv und von Null verschieden ist. Die

Ausnahme dieser Regel bildet die Zahl Null, bei der  $v$ ,  $e$  und  $m$  zu Null gesetzt werden.

Als Beispiel werden hier die Zahlen von -5 bis +5 mit einer 11 Bit breiten Mantisse und einem 4 Bit breiten Exponenten dargestellt. Der Bias bei dieser Darstellung ist 8. Der Term 1 in der Summe der Mantisse  $m$  entspricht der oben erwähnten, nicht mit abgespeicherten 1. Bei dem Wert 0 entfällt diese.

Zahlenwert	Abbildung	Erläuterung der Abbildung
-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	$v=0$ $e=10-8$ $m=1+1/4$

Betrachtet man nun das obige Beispiel, so ergibt sich für die Konstante *F2One* der Wert 0x4000. Vor dem zweiten *SetNumTypeF2Format*-Aufruf haben die Variablen folgende Inhalte:

- $a = 0x4000$  entspricht dem Wert 1
- $b = 0x4800$  entspricht dem Wert 2

Anschließend wird das Format gewechselt, wodurch sich die Interpretation der Werte ändert:

- $a = 0x4000$  entspricht nach wie vor dem Wert 1
- $b = 0x4800$  entspricht nun dem Wert 4

Demzufolge erhält  $c$  den Wert 3 (0x4600), der nach dem letzten Aufruf von *SetNumTypeF2Format* zu 1.75 interpretiert wird.

Vermeiden Sie also einfach ein Gemisch aus verschiedenen F2-Zahlen!

In der Bibliothek *NumType\_F2.c* sind weitere Funktionen implementiert, die eine Konvertierung in einen und aus einem Bruch erlauben. Die in diesen Funktionen verwendeten Datentypen *NumeratorF2\_t* und *DenominatorF2\_t* werden in der Headerdatei dieser Bibliothek definiert. Der Datentyp *DenominatorF2\_t* kann nur positive Ganzzahlen enthalten! Die Funktionen sind wie folgt deklariert:

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

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

Beispiel:

Die Zahl 3.125 soll in das Format F2 mit 11 Bit für die Mantisse und 4 Bit für den Exponenten gewandelt werden:

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

*F2Result* enthält nach der Ausführung den hexadezimalen Wert 0x4C80. Soll nun *F2Result* zurück in eine reelle Zahl gewandelt werden, so erledigt dies folgende Codesequenz (unter der Annahme, dass das Format noch eingestellt ist):

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

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

Bei der Wandlung in einen Bruch ist sichergestellt, dass der Wert des Nenners immer von Null verschieden ist. Bei der Wandlung eines Bruches in einen Zahlenwert wird **nicht** geprüft, ob der Nenner ungleich Null ist!

Tritt durch eine Operation ein Zahlenüberlauf ein, so wird das Ergebnis auf die maximal darstellbare positive oder negative Zahl beschränkt.

## Festpunktzahlen I2 und I4

Die Festpunktzahlen besitzen die Einstellungen

- Anzahl der Vorkommabits und
- Anzahl der Nachkommabits.

Bevor mit den Festpunktzahlen gerechnet werden kann, müssen diese Einstellungen der Bibliothek durch den Aufruf *SetNumTypeI2Format* bzw. *SetNumTypeI4Format* bekannt gemacht werden.

Die Speicherung einer Zahl erfolgt in dem grundlegenden Datentyp *short* bzw. *long int*. Dabei wird die abzubildende Zahl einfach mit dem Wert  $2^m$ , wobei *m* die Anzahl der Nachkommabits ist, multipliziert. Folgende Tabelle zeigt die Darstellungen der Zahlen -2 bis 2 in 0.5-er Schritten an:

Zahlenwert	Abbildung
-2	-256
-1.5	-192
-1	-128
-0.5	-64
0	0
0.5	64
1	128
1.5	192
2	256

In den Festpunktzahlenbibliotheken *NumType\_I2.c* und *NumType\_I4.c* sind ebenfalls Funktionen zur Konvertierung in einen und aus einem Bruch implementiert. Die Funktionsdeklarationen dafür sind in Anlehnung an das Zahlenformat F2 getroffen worden. Auch in diesen Bibliotheken existieren eigene Datentypen für Zähler- und Nennerwerte. Durch den Datentyp *DenominatorI2\_t* bzw. *DenominatorI4\_t* ist sichergestellt, dass Nennerwerte nur positive Ganzzahlen enthalten können.

Als Beispiel soll wieder die Zahl 3.125 in das Format I2 mit 9 Vorkommabits und 7 Nachkommabits gewandelt werden:

```
NumTypeI2_t I2Result;

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

*I2Result* enthält nach der Ausführung die Zahl 400 ( $400 = 2^7 \cdot 3.125$ ). Soll *I2Result* zurück in einen Bruch gewandelt werden, so erledigt dies der Aufruf *opI2ToFraction*:

```
NumeratorI2_t    zaehler;
```



```
DenominatorI2_t  nenner;  
  
opI2ToFraction(I2Result, &zaehler, &nenner);
```

---

---

## Der Schema-Umsetzer in FALCO

Der Schema-Umsetzer dient dazu, Ihren Programmieraufwand zu minimieren. Sie erstellen eine Datei in dem hier spezifizierten Schema, der Umsetzer macht daraus entsprechende Quellcode-Dateien, die Sie für Ihre Projekte benötigen. Das Prinzip ist dabei sehr einfach. Der Schema-Umsetzer liest Ihre Schema-Datei und führt die darin enthaltenen Anweisungen aus. Dabei werden Variablen durch ihren Wert ersetzt. So wird es möglich, ein Schema für beliebige Code-Generierungen einzusetzen.

### Funktionsweise

Die Funktionsweise des Schema-Umsetzers ist recht einfach. Er bekommt die Inhalte einer CPF-Datei (Code-Parameter-Datei) und einer CSH-Datei (Code-Schema-Datei) als Eingabedaten und beginnt die CSH-Datei ab dem Eintrittspunkt (s. u.) zu interpretieren.

Das Code-Schema der CSH-Datei enthält Anweisungen, die vom Umsetzer ausgeführt werden und Variablen, deren Werte aus der Code-Parameterdatei entnommen werden. Die im Code-Schema auftretenden Variablen werden einfach durch ihre Werte ersetzt.

Damit ein Codestück für beliebige Fuzzy-Controller passt, sind die Anzahl der Ein-/Ausgangparameter, der verwendete numerische Typ, die Namen der Funktionen usw. als Variablen in dieses Codestück zu schreiben. In dem Abschnitt *Code-Parameter-Dateien* finden Sie ein Beispiel einer CPF-Datei, in der sämtliche Variablen aufgeführt sind. Es ist nicht notwendig, die Einstellung *Code-Parameter-Datei erzeugen* gewählt zu haben, um mit dem Umsetzer zu arbeiten.

Die angegebene CSH-Datei ist in einzelne Codefragmente oder auch Code-Blöcke gegliedert, von denen mindestens einer als Eintrittspunkt für den Um-

setzer ausgezeichnet ist (s. Anweisung *#WF\_code\_block*). Der Schema-Umsetzer liest die CSH-Datei, führt die enthaltenen Anweisungen aus und schreibt das Resultat in eine Liste namens *main.c*. Diese kann anschließend als Datei unter diesem Namen gespeichert werden. Durch die Anweisung *#WF\_write\_to* (s. u.) kann das Schreiben in eine Liste mit anderem Namen (Dateinamen) bewirkt werden. Erfolgt die Anweisung direkt am Eintrittspunkt der CSH-Datei, so entsteht erst gar keine *main.c*-Liste.

## Variablen und Anweisungen

Der Schema-Umsetzer kennt verschiedene Anweisungen und Variablen, die bei der Interpretation der CSH-Datei auftreten dürfen. Er unterscheidet zwischen lokalen und globalen Variablen.

### Variablen

Der Umsetzer unterscheidet zwischen lokalen und globalen Variablen. Die globalen Variablen entstammen immer aus einer CPF-Datei. Lokale Variablen können nur durch die Anweisungen *#WF\_foreach* und *#WF\_code\_block* erzeugt werden. Weitere Hinweise entnehmen Sie den Abschnitten zu diesen Anweisungen.

Die Bedeutung der meisten globalen Variablen geht aus ihrem Namen hervor. Sie beginnen alle mit *@WF\_* und enden immer mit dem Zeichen *@*, um sie besser vom übrigen Code zu trennen. In dem Abschnitt *Code-Parameter-Dateien* finden Sie ein Beispiel einer CPF-Datei, in der sämtliche globalen Variablen aufgeführt sind.

### Anweisungen

Anweisungen beginnen immer mit dem Zeichen *#* gefolgt von den beiden Großbuchstaben *WF* und einem durch einen Unterstrich eingeleiteten, klein geschriebenen Anweisungsnamen. Argumente einer Anweisung werden generell in runde Klammern gesetzt.

Bei der Beschreibung einer Anweisung werden optionale Argumente in eckige Klammern gesetzt. Soll aus mehreren Möglichkeiten eine ausgewählt werden, so werden die einzelnen Möglichkeiten durch *|* voneinander getrennt und in geschweifte Klammern gesetzt. Die Bedeutung von *{1 | 2 | 3}* ist also *entweder 1 oder 2 oder 3*, nicht aber alle Zahlen und auch nicht keine der Zahlen!

Manche Anweisungen können Listen enthalten. Damit sind einfach durch Kommata getrennte Zeichenketten gemeint (welche die Elemente der Liste sind). Zusätzlich gibt es ganzzahlige Bereichsdefinitionen. Dies sind zwei Zahlen, die durch **zwei** Punkte voneinander getrennt sind (z. B.: -2..2). Bereichsdefinitionen dürfen auch als Elemente einer Liste auftauchen (z. B.: -2..2, 4..10, a, b, c). Die beiden Zahlenangaben bei Bereichsdefinitionen dürfen auch algebraische Ausdrücke sein (z. B.: 5+2..-3+@WF\_num\_inputs[0]@)

### #WF\_code\_block

Syntax:

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

Definiert den Anfang des Code-Blockes *Name*. Der Code-Block wird zum Eintrittspunkt für den Schema-Umsetzer verfügbar gemacht, wenn *main* statt der Liste gewählt wird. Der Code-Block kann immer durch *#WF\_insert\_code\_block* an eine andere Stelle innerhalb des Schemas eingefügt werden. Steht statt *main* die geklammerte Liste (*\$a*, *\$b*, ...), so werden bei der Interpretation des Code-Blockes die Elemente der Liste als lokale Variablen angelegt (es kann zweckmäßig sein, die lokalen Variablen auch auf *\$* enden zu lassen, siehe *#WF\_foreach*). Diese besitzen ihre Gültigkeit nur innerhalb dieses Code-Blockes. Die Werte der Variablen werden bei der Verwendung des Blockes durch *#WF\_insert\_code\_block* definiert. Ein Code-Block muss immer mit *#WF\_code\_block\_end* abgeschlossen sein!

### #WF\_code\_block\_end

Syntax:

```
#WF_code_block_end
```

Definiert das Ende des aktuellen Code-Blockes

### #WF\_write\_to

Syntax:

```
#WF_write_to(Dateiname)
```

Die Ausgabe wird solange in *Dateiname* umgeleitet, bis diese Anweisung mit anderem Dateinamen auftaucht oder der Code-Block verlassen wird. Zu Beginn des Schema-Umsetzers wird in die Datei *main.c* geschrieben, bis diese Anweisung vom Umsetzer gelesen wird.

### #WF\_include

Syntax:

```
#WF_include Zeichenkette
```

Es wird die Präprozessor-Direktive *#include Zeichenkette* erstellt und an das Ende der Include-Liste der aktuellen Ausgabedatei gestellt, falls dieser Eintrag nicht schon in der Liste vorkommt.

## #WF\_define

Syntax:

```
#WF_define a b
```

Es wird eine *#define*-Direktive erstellt und an das Ende der Define-Liste der aktuellen Ausgabedatei gestellt. Diese Anweisung darf den Backslash zur Maskierung des Zeilenendes verwenden (die Maskierung bedeutet eine Aufhebung; es wird so getan, als ob alles in einer Zeile stünde):

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

## #WF\_insert\_code\_block

Syntax:

```
#WF_insert_code_block(Name[(Liste von Argumenten)]
[ , { Dateiname{.REF|.CSH} | HERE } [ ,EVER] ] )
```

Fügt einen Code-Block in das Schema ein. An den Code-Block werden die *Argumente* übergeben und in den lokalen Variablen, die in der *#WF\_code\_block*-Anweisung auftreten, der Reihe nach abgelegt. Der Code-Block wird in der aktuellen Datei gesucht, wenn kein *Dateiname* oder *HERE* angegeben ist. Hat der *Dateiname* die Erweiterung *REF*, so wird der angegebene *Name* in einer Referenzdatei gesucht. In diesem Fall wird mit der entsprechenden Referenz weitergearbeitet. Ist die Dateinamenserweiterung *CSH*, so wird der Block in der entsprechenden CSH-Datei gesucht. Die angegebene Datei wird dabei im aktuellen und allen durch *#WF\_path* angegebenen Verzeichnissen gesucht.

Der Bezeichner *HERE* bedeutet, dass der Code-Abschnitt aus dieser Schemadatei genommen werden soll. Wird *EVER* angegeben, so wird das Einfügen dieses Code-Abschnittes auch dann ausgeführt, wenn dieser schon in die aktuelle Ausgabedatei eingefügt wurde. Standardmäßig kann jeder Code-Block nur einmal in eine Ausgabedatei geschrieben werden.

## #WF\_path

Syntax:

```
#WF_path(Verzeichnis1[; Verzeichnis2;...])
```

Stellt die Suchverzeichnisse für die Anweisung `#WF_insert_code_block` (s. o.) ein.

### #WF\_show\_message

Syntax:

```
#WF_show_message(Mitteilungstext, Fenstertitel)
```

Sobald der Schema-Umsetzer auf diese Anweisung stößt, wird in einem modalen Dialogfenster mit entsprechendem *Fenstertitel* der *Mitteilungstext* ausgegeben. Die Mitteilung muss durch Betätigung der Schaltfläche *OK* bestätigt werden!

### #WF\_foreach

Syntax:

```
#WF_foreach ($Variable[$], Liste ){  
  ...Code-Schemazeilen  
}
```

Diese Anweisung leitet eine Schleife ein. Es ist zweckmäßig, die Schleifenvariable mit einem `$`-Zeichen enden zu lassen, wenn die Code-Schemazeilen *\$Variable* mitten in einem Bezeichner enthalten.

Beispiel:

Folgende Ausgabe soll erzeugt werden: "x\_t y\_t z\_t ". Das folgende Codefragment sieht zunächst korrekt aus.

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

Leider funktioniert es aber nicht, da der Umsetzer vermutet, dass es sich bei *\$var\_t* um eine eigenständige lokale Variable handelt. Durch das abschließende `$` wird Klarheit geschaffen:

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

Die Schleifenvariable nimmt bei jedem Durchlauf den nächsten Wert der *Liste* an. Ist das Element der Liste eine ganzzahlige Bereichsdefinition, so wird der definierte Bereich vollständig durchlaufen (d. h. *\$Variable\$* nimmt also auch jeden einzelnen Wert dieser Bereichsdefinition an). Bei Bereichsdefinitionen *a..b* wird immer mit dem Wert von *a* begonnen und in Richtung von *b* in Schritten von 1 gegangen. Dabei können *a* und *b* algebraische Ausdrücke sein.

Innerhalb der nachfolgenden Code-Schemazeilen wird die Schleifenvariable ersetzt durch den Wert des aktuellen Schrittes.

Beispiel:

Die Mächtigkeit der `#WF_foreach`-Anweisung demonstriert folgende Code-Sequenz:

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

Sie erzeugt folgende Ausgabe:

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

Die `#WF_foreach`-Anweisung definiert eine lokale Variable mit dem Namen `$WF_LAST_n$`. Dabei steht  $n$  für die  $n$ -te in diesem Code-Block geschachtelte Schleife. Begonnen wird mit  $n=1$ . Das obige Beispiel erzeugt also für die äußere Schleife eine Variable `$WF_LAST_1$` und für die innere eine Variable mit dem Namen `$WF_LAST_2$`. Die Werte der Variablen sind 1, wenn der letzte Schleifendurchlauf der Schleife, die diese Variable angelegt hat, erfolgt, sonst 0. Die Variablen existieren nur innerhalb der Schleifen. Sie sind extrem nützlich für Verzweigungen.

## #WF\_if

Syntax:

```
#WF_if (einfacher logischer Ausdruck)
Code-Schemazeilen A
[#WF_else
Code-Schemazeilen B]
#WF_endif
```

Die `#WF_if`-Anweisung realisiert eine Verzweigung. Die *Code-Schemazeilen A* werden nur dann interpretiert, wenn der *einfache logische Ausdruck* als Resultat wahr liefert. Evaluiert er zu falsch, so werden die *Code-Schemazeilen B*, falls es einen `#WF_else`-Zweig gibt, gelesen und interpretiert.

Ein *einfacher logischer Ausdruck* hat nur **einen** binären Operator, der die linke und rechte Seite, die arithmetische Ausdrücke sein dürfen, voneinander trennt. Folgende Tabellen führen alle binären Operatoren auf:

Operatoren für algebraische Ausdrücke

Operator	Bedeutung
<code>==</code>	gleich
<code>!=</code>	ungleich
<code>&gt;=</code>	größer gleich
<code>&lt;=</code>	kleiner gleich
<code>&gt;</code>	größer
<code>&lt;</code>	kleiner

Zusätzlich ist es möglich zu prüfen, ob eine Variable ungleich Null ist, indem einfach nur die Variable angegeben wird.

Operatoren für Zeichenketten

Operator	Bedeutung
~~	die rechte Seite darf in diesem Fall eine Liste sein. Wenn sich die linke Seite in der Liste befindet, ist die Bedingung wahr.
!~	negierte Version von ~~

Zusätzlich ist es möglich zu prüfen, ob eine Variable keine Zeichen enthält, indem einfach nur die Variable angegeben wird.

## #WF\_expr

Syntax:

```
#WF_expr(Zahlenformatbeschreibung, algebraischer Ausdruck)
```

Die Anweisung berechnet das Ergebnis des algebraischen Ausdruckes und konvertiert dieses in das angegebene Zahlenformat. Das Zahlenformat wird dabei durch folgende Angaben beschrieben.

F= <i>n</i>	Es sollen Fließpunktzahlen mit <i>n</i> Bytes verwendet werden
M= <i>n</i>	Ist nur von Bedeutung, wenn F=2 gesetzt wird. <i>n</i> gibt dann die Anzahl Mantissenbits an.
E= <i>n</i>	Ist nur von Bedeutung, wenn F=2 gesetzt wird. <i>n</i> gibt dann die Anzahl der Exponentenbits an.
V= <i>n</i>	<i>n</i> entspricht der Anzahl der Vorkommenbits bei Festpunktzahlen
N= <i>n</i>	<i>n</i> entspricht der Anzahl der Nachkommenbits bei Festpunktzahlen
Xn=Z	Bei F2-Zahlen wird Z vor dem hexadezimalen Wert der Zahl ausgegeben. (Für C sollte also Xn=0x gesetzt sein).
nX= Z	Bei F2-Zahlen wird Z nach dem hexadezimalen Wert der Zahl ausgegeben (Für C sollte also nX nicht gesetzt sein: nX=).

Die globale Variable `@WF_num_format@` enthält die vollständige Angabe zur Konvertierung in das bei der Code-Generierung eingestellte Zahlenformat.

Format	Benötigte Angaben
F10	F=10
F8	F=8
F4	F=4
F2	F=2 M= <i>Mantissenbits</i> E= <i>Exponentenbits</i> Xn= <i>0x</i> nX=
I4 und I2	V=Vorkommabits und N=Nachkommabits

Beispiel:

Der Wert 3.125 soll in das Zahlenformat I4 mit 13 Vorkomma- und 11 Nachkommabits konvertiert werden:

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