WinFACT 6

Windows Fuzzy And Control Tools

BENUTZERHANDBUCH

AUTOCODE-GENERATOR

Release 1.0

Ingenieurbüro Dr. Kahlert
Ludwig-Erhard-Str. 45 D-59065 Hamm
Tel. 0 23 81/926 996 Fax 0 23 81/926 997

Inhalt

Installation des AutoCode-Generators

Die Installation des WinFACT-AutoCode-Generators erfolgt komplett dialoggeführt.

> **Hinweis:** Bei der nachfolgenden Beschreibung des Installationsvorgangs wird davon ausgegangen, daß auf dem Zielrechner zuvor ordnungsgemäß das komplette Programmpaket WinFACT installiert wurde!

Das Installationsprogramm meldet sich nach einigen Sekunden mit einem Begrüßungsbildschirm (siehe nachfolgende Bildschirmgrafik).

Begrüßungsbildschirm des AutoCode-Installationsprogramms

Folgen Sie nunmehr einfach den Anweisungen des Installationsprogramms. Sie werden im Laufe des Installationsvorgangs nach dem Zielverzeichnis für den

AutoCode-Generator gefragt; geben Sie in diesem Fall bitte das Programmverzeichnis Ihrer bereits bestehenden WinFACT-Installation an (siehe nachfolgende Bildschirmgrafik).

Angabe des Zielverzeichnisses

Viel Spaß und Erfolg bei der Arbeit mit dem BORIS-AutoCode-Generator! Bei Fragen, Problemen oder Verbesserungsvorschlägen:

Telefon	0 23 81/926-996
Fax	0 23 81/926-997
EMail	support@kahlert.com

Übersicht und Motivation

Der BORIS-AutoCode-Generator gibt dem Anwender ein leistungsfähiges Werkzeug an die Hand, um eine mit BORIS erstellte und simulierte Blockstruktur in ein C-Programm zu überführen. Durch die offene Architektur bietet er dem Anwender optimale Anpassungsfähigkeiten. Sie können

- eigene Routinen dauerhaft implementieren oder mitgelieferte Routinen ändern,
- für jede Hardware, die Sie verwenden, Ihren eigenen Grundstock an C-Routinen dauerhaft hinzubinden. Falls Sie eine Hardware besitzen, die mit unterschiedlichen Erweiterungen ausgeliefert wird, erstellen Sie für jede Form einmalig die Funktionen, die z. B. Ein- und Ausgabe behandeln. Dadurch wird der Produktionszyklus erheblich verkürzt und wesentlich komfortabler.
- durch verschiedene Optionen den Code optimal an verschiedene Compiler bzw. unterschiedliche Hardware anpassen.
- BORIS veranlassen, den Compiler, Linker und Lader im Anschluß an die Code-Generierung automatisch auszuführen. Somit haben sie auf Knopfdruck die komplette Applikation.
- BORIS als graphische Programmiersprache verwenden, indem Sie durch den AutoCode-Generator eine Windows-DLL erzeugen und diese wieder unter BORIS laden. Diese Vorgehensweise empfiehlt sich insbesondere zur Senkung der Berechnungszeit!
- eigene Integrationsverfahren für z. B. steife Systeme implementieren und so die Simulationszeit verkürzen.
- wesentlich höhere Abtastraten beim Einsatz von PC-Einsteckkarten erzielen, da nur noch Berechnungszeit, nicht aber Zeit für Anzeigen und Verwalten von Blöcken benötigt wird.
- nahezu jeden Block der BORIS-Systemblock-Bibliothek verwenden[1].

[1] Welche Blocktypen tatsächlich verfügbar sind, hängt von der erworbenen Version des AutoCode-Generators ab (siehe dazu auch Anhang B).

In diesem Handbuch werden alle Möglichkeiten der Code-Generierung erklärt. Es setzt die Kenntnisse der Programmiersprache C bzw. ANSI-C voraus. Hier bietet der *ANSI C Guide* aus dem IWT-Verlag (Autor: Peter Prinz) eine sehr gute Referenz. Bitte denken Sie ferner daran, daß nahezu jeder C-Compiler seine Eigenheiten hat. So ist zum Beispiel ein Rechts-Shift-Operator (>>) in ANSI-C nicht eindeutig definiert. Es ist compilerabhängig, ob er vorzeichenbehaftet oder nicht vorzeichenbehaftet ausgeführt wird. Wenn Ihr Code also nicht die gewünschten Ergebnisse liefert, mag das auch hieran liegen. Der BORIS-AutoCode-Generator gibt Ihnen jedoch komfortable Möglichkeiten an die Hand, den erzeugten Code auf einfache Weise zu überprüfen.

Bevor Sie Ihre eigenen Applikationen generieren, sollten Sie über die Abläufe bei der Quellcode-Generierung informiert sein. Zunächst aber für den groben Überblick ein kleines Beispiel.

Einführendes Beispiel

Entwerfen Sie folgende Struktur unter BORIS:

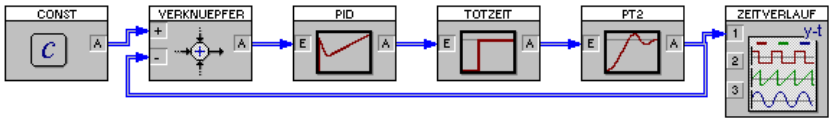

Bild 1. *Beipielstruktur unter BORIS (einfacher Regelkreis)*

Die Parameter der einzelnen Blöcke sollen dabei so belassen werden, wie sie beim Einfügen derselben sind (lediglich im Verknüpfer wurde am zweiten Eingang das Vorzeichen geändert). Stellen Sie nun eine Abtastschrittweite[2] (Simulationsschrittweite) von 0.1 ein. Von diesem Regelkreis wollen wir den PID-Regler in C-Code überführen. Dafür muß dieser entsprechend markiert werden. Dies erreichen Sie, indem Sie nach Betätigen der rechten Maustaste im daraufhin erscheinenden Popup-Menü den Eintrag IN C-CODE-GENERIERUNGSLISTE EINTRAGEN anwählen.

Nach der Eintragung in die C-Code-Generierungsliste erscheint die Titelzeile des Blockes hellgelb, um diesen gegenüber den anderen Blöcken hervorzuheben.

[2] Die unter BORIS eingestellte Simulationsschrittweite entspricht bei einer späteren Echtzeitsimulation auf der Ziel-Hardware immer der Abtastzeit.

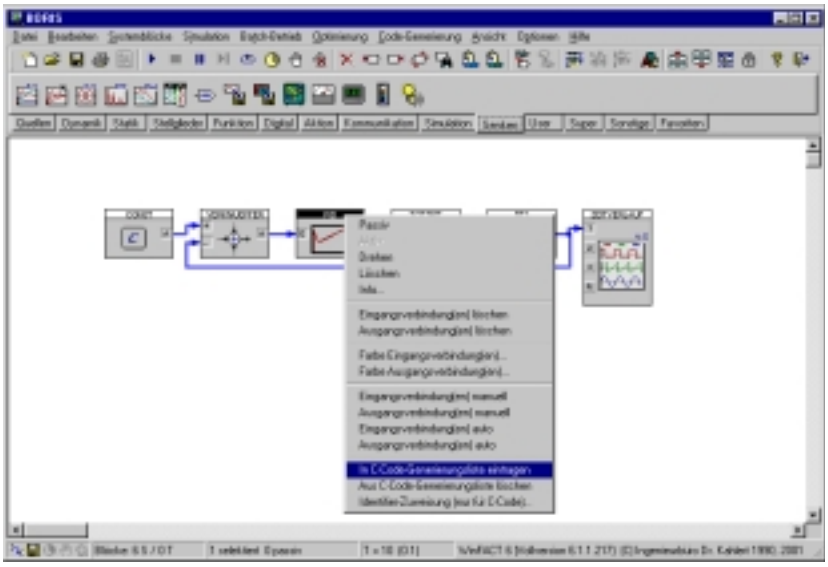

Bild 2. *Markieren und Eintragen des Blockes in die Generierungsliste*

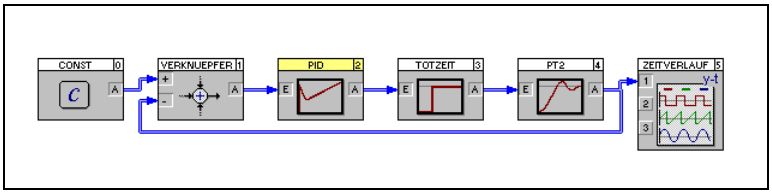

Bild 3. *Für den AutoCode-Generator selektierter Block, der in C-Code überführt werden soll*

Zur Generierung des Blocks in C-Code wählen Sie den Menüeintrag CODE-GENERIERUNG | C-CODE GENERIERUNGS-EINSTELLUNGEN... an. Es erscheint nun ein Dialog, in dem Sie Einstellungen bezüglich des Inhalts des C-Quelltextes vornehmen können. Wählen Sie zunächst die gleichen Einstellungen wie in dem nachfolgend abgebildeten Dialog. Die Erläuterung der einzelnen Optionen erfolgt an späterer Stelle.

Bild 4. *Optionen zur C-Code-Generierung*

Haben Sie in Ihrem Dialog die einzelnen Register auf den abgebildeten Stand gebracht, verlassen Sie ihn durch Betätigen des *OK*-Buttons (Sie können natürlich auch einen anderen Zieldateinamen als TEST.C wählen, sollten aber die

Verweisdatei unverändert lassen). Durch den Menüpunkt CODE-GENERIERUNG | CODE-GENERIEREN wird der Quelltext erzeugt und in die angegebene Datei geschrieben (hier: *C:\temp\test.c*). Dabei wird der Fortschritt in folgendem Dialog angezeigt:

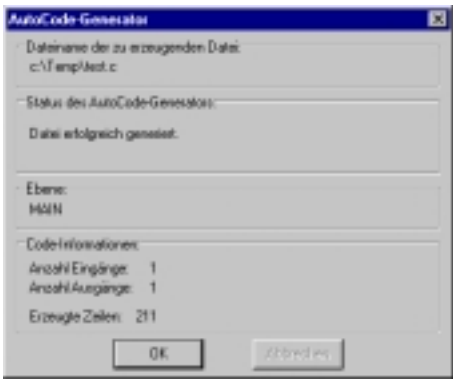

Bild 5. *Anzeige des Fortschritts der C-Code-Generierung*

Laden Sie nun einen Editor und betrachten Sie den C-Code. Dieser sollte folgendermaßen aussehen:

```
1   /************************************************************
    ***************
2     This file was created by the AutoCode-Generator of the
    program BORIS
3     which is a module of WinFACT (Windows Fuzzy And Control
    Tools)
4
5     Instead of editing this file, modify BORIS System-File
6     with BORIS and use the AutoCode-Generator once again.
7
8     Generated by : Michael
9                 on : MICHAELNT
10
11    Boris Version    : 6.1.1.218
12    Source-File      : WAS NOT SAVED YET!
13    Timestamp        : 13.11.2001 14:55:17
14
15  ************************************************************
    **************/
```

```
16  /*******************************************************/
17  /*  all #define directives the c-compiler has to know   */
18  /*******************************************************/
19  #define Rgchk(a) (a)
20  #define MUL(a,b,c) Rgchk((a)*(c))
21  #define DIV(a,b,c) Rgchk((a)/(c))
22  /*******************************************************/
23  /*                 #include directives                 */
24  /*******************************************************/
25  #include "C:\Temp\test.h"
26  #include "regdef.h"
27  #include <stdio.h>
28  #include "dynamic.h"
29  #include <math.h>
30  /*******************************************************/
31  /* declarations of global constants that are needed  */
32  /*******************************************************/
33    const SVartyp One=1;
34    const SVartyp SignalMax=1E32;
35    const SVartyp SignalMin=-1E32;
36    const SVartyp HIGHLEVEL=5;
37    const SVartyp LOWLEVEL=0;
38    const SVartyp THRESHOLD=2.5;
39    const SVartyp PI_DIV_2=1.570796327;
40    const SVartyp PI_MUL_2=6.283185307;
41    const SVartyp PI=3.141592654;
42    const SVartyp EXP1=2.718281828;
43  /*******************************************************/
44  /* declarations of global variables that are needed  */
45  /*******************************************************/
46    MEM_ATTRIBUTE_VAR SVartyp INVDELTAT;
47    MEM_ATTRIBUTE_VAR SVartyp X_VEC[MAX_X_VEC][2];
48    MEM_ATTRIBUTE_VAR SVartyp I_VEC[MAX_I_VEC][2];
49    MEM_ATTRIBUTE_VAR unsigned int TimeStepCounter;
50    MEM_ATTRIBUTE_VAR PIDT1Struct PID_V2;
51  /*******************************************************/
52  /* specific global functions such as integrationmethods */
53  /*******************************************************/
54
55
```

```
56  /*********************************/
57  /* global functions of the blocks */
58  /*********************************/
59
60  void PIDT1_SetParam(PIDT1Struct *p, int id, SVartyp value)
61  {
62    switch (id){
63    case 1 :
64      if (value!=p->Kr){
65        p->Kr=value;
66        p->calcParams=1;
67      }
68      break;
69    case 2 :
70      if ( ((value>0) ? value : 1.0E-10) != p->Tn){
71        p->Tn=(value>0)? value : 1.0E-10;
72        p->calcParams=1;
73      }
74      break;
75    case 3 :
76      if (value!=p->Tv){
77        p->Tv=value;
78        p->calcParams=1;
79      }
80      break;
81    }
82  }
83
84
85  SVartyp PIDT1_GetParam(PIDT1Struct *p, int id)
86  {
87    switch (id){
88    case 1 : return MUL(p->Kr,FracPart,One);
89    case 2 : return MUL(p->Tn,FracPart,One);
90    case 3 : return MUL(p->Tv,FracPart,One);
91    }
92  }
93
94  void PIDT1_CalcInnerParams(PIDT1Struct *p)
95  {
```

```
96    if (!p->calcParams) return;
97    if (p->IOn) p->Ki = p->Kr/p->Tn/INVDELTAT;
98    if (p->DOn) {
99      p->alpha=exp(- p->INVTvz/INVDELTAT);
100     p->Kd = p->Kr * p->Tv * INVDELTAT;
101     p->Kd = p->Kd * (One-p->alpha);
102   }
103   p->calcParams=0;
104 }
105
106
107 MEM_ATTRIBUTE SVartyp PIDT1_init(PIDT1Struct *p, SVartyp *e)
108 {
109   SVartyp zw;
110   p->calcParams=1;
111   PIDT1_CalcInnerParams(p);
112   if (!p->POn) p->yP = 0; else p->yP=p->Kr * *e;
113   if (!p->IOn) p->yI = 0; else p->yI=p->y0;
114   if (!p->DOn) p->yD = 0; else p->yD=p->Kd * *e;
115   if (p->LimOn){
116     zw = p->yP + p->yI;
117     if (zw<p->ymin) zw=p->ymin;
118     if (zw>p->ymax) zw=p->ymax;
119     if (p->AW==1) {
120       if (p->yI<p->ymin) p->yI=p->ymin;
121       if (p->yI>p->ymax) p->yI=p->ymax;
122     }else
123       p->yI = zw - p->yP;
124     zw+=p->yD;
125     if (zw<p->ymin) zw=p->ymin;
126     if (zw>p->ymax) zw=p->ymax;
127   }else
128     zw = p->yP + p->yI + p->yD;
129   p->X1=*e;
130   return zw;
131 }
132
133 MEM_ATTRIBUTE SVartyp PIDT1_fct(PIDT1Struct *p, SVartyp *e)
134 {
135   SVartyp zw;
```

```
136   PIDT1_CalcInnerParams(p);
137   if (!p->POn) p->yP = 0; else p->yP = p->Kr * *e;
138   if (!p->IOn) p->yI = 0; else p->yI+= p->Ki * p->X1;
139   if (!p->DOn) p->yD = 0; else p->yD = p->yD * p->alpha + p-
    >Kd* (*e-p->X1);
140   if (p->LimOn) {
141     zw = p->yP + p->yI;
142     if (zw<p->ymin) zw=p->ymin;
143     if (zw>p->ymax) zw=p->ymax;
144     if (p->AW==1) {
145       if (p->yI<p->ymin) p->yI=p->ymin;
146       if (p->yI>p->ymax) p->yI=p->ymax;
147     } else
148       p->yI = zw - p->yP;
149     zw+=p->yD;
150     if (zw<p->ymin) zw=p->ymin;
151     if (zw>p->ymax) zw=p->ymax;
152   } else
153     zw = p->yP + p->yI + p->yD;
154   p->X1=*e;
155   return zw;
156 }
157
158 /*************************************/
159 /* the controller initialising function */
160 /*************************************/
161 void testinitcontrol(SVartyp PID_2I1,
162                      SVartyp *PID_2O1)
163 {
164
165   {
166     /* initialising the state of the system and used exter-
    nals */
167     unsigned int i;
168     for (i=0; i<MAX_X_VEC; i++){X_VEC[i][0]=0;
    X_VEC[i][1]=0;}
169   }
170   {
171     unsigned int i;
```

```
172     for (i=0; i<MAX_I_VEC; i++){I_VEC[i][0]=0;
    I_VEC[i][1]=0;}
173   }
174   {
175     /* set state of the system from the external inputs */
176     I_VEC[0][0]=PID_2I1;
177   }
178   X_VEC[0][0]=0;
179   PID_V2.POn=1;
180   PID_V2.IOn=1;
181   PID_V2.DOn=1;
182   PID_V2.LimOn=0;
183   PID_V2.AW=1;
184   PID_V2.Kr=1;
185   PID_V2.Tn=1;
186   PID_V2.Tv=1;
187   PID_V2.INVTvz=1000;
188   PID_V2.ymax=1.7E38;
189   PID_V2.ymin=-1.7E38;
190   PID_V2.y0=0;
191
192   /* calls of the used functions in the system */
193   X_VEC[0][0]=PIDT1_init(&PID_V2,&I_VEC[0][0]);
194
195   {
196     /* set all external outputs from the state of the system
    */
197     unsigned int i;
198     for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
199     *PID_2O1=X_VEC[0][0];
200   }
201 }
202
203 /*********************************/
204 /* the controller function itself */
205 /*********************************/
206 void testcontrol(SVartyp PID_2I1,
207                  SVartyp *PID_2O1)
208 {
209
```

```
210   {
211     /* set state of the system from the external inputs */
212     unsigned int i;
213     for(i=0; i<MAX_I_VEC; i++) I_VEC[i][1]=I_VEC[i][0];
214     I_VEC[0][0]=PID_2I1;
215   }
216   X_VEC[0][0]=PIDT1_fct(&PID_V2,&I_VEC[0][0]);
217
218   {
219     /* set all external outputs from the state of the system
    */
220     unsigned int i;
221     for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
222     *PID_2O1=X_VEC[0][0];
223   }
224 }
225 void testfreecontrol(void)
226 {
227 }
228
229
230 int main(void)
231 {
232   unsigned int num;
233   SVartyp PID_2I1=0;
234   SVartyp PID_2O1=0;
235   INVDELTAT=10;
236   TimeStepCounter=0;
237   printf("block-no.:2  1. input: PID=");
238   scanf("%g",&PID_2I1);
239   printf("last simulation step (>1): =");
240   scanf("%d",&num);
241   if(num<=1)num=2;
242   num++;
243   testinitcontrol(PID_2I1,
244                   &PID_2O1);
245   printf("t=%6.3g  output: PID =
    %g\n",TimeStepCounter/INVDELTAT,PID_2O1);
246   for(TimeStepCounter=1; TimeStepCounter<num; TimeStepCoun-
    ter++)
```

```
247     {
248       testcontrol(PID_2I1,
249                   &PID_2O1);
250       printf("t=%6.3g  output: PID =
    %g\n",TimeStepCounter/INVDELTAT,PID_2O1);
251     }
252   testfreecontrol();
253   return 0;
254 }
```

Listing 1. *Demonstration der C-Code-Generierung anhand eines PIDT1-Reglers*

Der C-Code gliedert sich in mehrere Abschnitte, deren Aufbau wir später genau besprechen werden. Der "nackte" Algorithmus des PIDT1-Reglers befindet sich im Abschnitt von Zeile 107 bis Zeile 156. Wie Sie sehen können, gliedert dieser sich wieder in zwei Funktionen, eine mit der Endung *_init* und eine mit der Endung *_fct* . Die erste dient der Initialisierung des Algorithmus sowie der Berechnung der Ausgangsgröße zum Zeitpunkt $t = 0$. Die zweite enthält den Algorithmus selbst und wird für alle Zeitpunkte $t > 0$ aufgerufen. Eine genaue Beschreibung der Funktionen erfolgt an späterer Stelle. Wichtig ist es, hier zu erkennen, daß der PID-Algorithmus im C-Code exakt die gleiche Funktionalität aufweist wie in der BORIS-Simulationsumgebung (z. B. Begrenzung mit Anti-Windup), also nicht einfach als "primitiver" PID-Algorithmus realisiert wird. Dieser Grundgedanke wurde auch bei den anderen Blöcken befolgt. Weiterhin wurde Wert darauf gelegt, daß im nachhinein Parameter ohne große Probleme geändert werden können. Dies gilt auch für die Abtastzeit!

Die Gründe dafür, die Algorithmen für sämtliche Blöcke auf diese Weise zu implementieren, liegen auf der Hand:

1. Sie sollen die Möglichkeit haben, schnell Anpassungen durchzuführen.
2. Die auf der späteren Zielhardware minimal mögliche Abtastzeit ist vorab nur schwer oder gar nicht ermittelbar. Wird diese variabel gestaltet, kann ein Hardwaresystem die minimale Abtastzeit selber einstellen (diese Möglichkeit erzwingt die Bedingung, daß der erzeugte Code grundsätzlich terministisch ist!)
3. Dadurch, daß Sie Windows-DLLs erzeugen können, diese aber wiederum unter BORIS simulieren können, werden Sie gleiche Resultate erwarten. Dieses kann nur bei gleichen Algorithmen gewährleistet werden.

Arbeiten mit dem Handbuch

Die nachfolgende Liste gibt Ihnen einen kurzen Überblick, was Sie in welchem Kapitel finden und wann Sie ein Kapitel lesen sollten.

Kapitel	Wichtig...
Ablaufphasen beim Generieren von C-Code	für **grundlegendes Verständnis** der Ablaufphasen beim Generieren von C-Code
Strukturanalyse	**immer**; erklärt Ein-/Ausgangsdefinitionen
Definitionsphase	für **Eigenentwicklungen** und **Verständnis** des C-Codes.
Funktionssynthese	für **Eigenentwicklungen** und **Verständnis** des C-Codes.
Schreiben der Systemfunktionen	für das **Verständnis** des C-Codes.
Schreiben der Rahmen-Funktion	für **Eigenentwicklungen**, **Debugging** und **Generierung von DLLs**
Aufbau der I/O-Beschreibungsdatei	für **Eigenentwicklungen**
Funktionen für C-Code-Ein-/Ausgänge	für **Eigenentwicklungen**
C-Code-Ein-/Ausgangsblöcke	für **direkte Hardwareanbindung** innerhalb der BORIS-Oberfläche
Einstellungen für die Code-Generierung	**immer**
Parameter der C-Code-Generierung	**immer**
Wahl der I/O-Beschreibungsdatei	für **Eigenentwicklungen**

Kommandozeile nach der Quellcode-Generierung	für höheren **Komfort**
Einfache Beispiele zur C-Code-Generierung	für das **Verständnis** der C-Code-Generierung sowie zur **Fehleranalyse**
Integrationsverfahren	für das **Verständnis** des C-Codes dynamischer Systemblöcke sowie **Eigenentwicklungen**
Festpunktzahlen und deren Arithmetik	für **Eigenentwicklungen** und **Fehleranalyse**
Anpassung von Blöcken an Benutzerwünsche	für **Eigenentwicklungen**
Benutzerdefinierte Systemblöcke (User-DLLs)	für die C-Code-Generierung benutzerdefinierter Systemblöcke (**Eigenentwicklungen**)
Anbindung der Ziel-Hardware	für **Eigenentwicklungen**
Anbindung von PC-Einsteckkarten	für **Eigenentwicklungen**
Anbindung an Micro-controller-Systeme	für **Eigenentwicklungen**
Anpassung von Ausgangsblöcken	für **Eigenentwicklungen**

Das Handbuch ist so aufgebaut, daß zunächst einmal grundlegende Abläufe erläutert werden. Hierbei werden Begriffsdefinitionen getroffen, auf die später zurückgegriffen wird. Diese "Schicht" ist für den Anwender nicht von der Oberfläche her sichtbar. Sie muß daher rein theoretisch abgehandelt werden.

Anschließend befassen wir uns mit der unter BORIS "faßbaren" Seite. In diesem Teil des Handbuches werden die Dialoge, die Einstellungen unter BORIS bezüglich der Code-Generierung erlauben, erläutert. Dadurch, daß Begriffe aus vorangegangenen Kapiteln immer wieder erwähnt werden, werden diese hier vertieft und für Sie zur Selbstverständlichkeit.

Wenn Sie die ersten beiden Teile überstanden haben, werden wir praxisgerechter. Wir wollen dann anhand von Beispielen einige Quelltexte analysieren. Wir werden den Quelltext selber jedoch unangetastet lassen und vergleichen nur die Ergebnisse des C-Codes mit denen unter BORIS. Anwendereingriffe sind hier noch nicht erforderlich, da alle C-Code-Beispiele als Quelldateien im

Lieferumfang enthalten sind. Weiterhin werden hier die Debugging-Möglichkeiten gezeigt.

Zum Schluß werden wir anhand von Beispielen die volle Leistungsfähigkeit der Quell-Code-Generierung ausschöpfen. Wir werden benutzerderfinierte Blöcke in Anlehnung an die BORIS-User-DLL-Schnittstelle schreiben sowie die Hardwareanbindung auf Basis von Microcontrollern und PC-Einsteckkarten implementieren.

Ablaufphasen beim Generieren von C-Code

Übersicht

Bei der C-Code-Generierung werden von BORIS verschiedene Aktionen vorgenommen, um den entsprechenden Quelltext zu erzeugen[3]. Diese Aktionen lassen sich in die folgenden fünf Phasen unterteilen, die in entsprechender Reihenfolge durchlaufen werden:

1. Analyse der zu generierenden Struktur (Strukturanalyse)
2. Schreiben aller #define-, #include-Compiler-Direktiven und globaler Variablen (Definitionsphase)
3. Schreiben der benötigten C-Funktionen für die Systemblöcke (Funktionssynthese)
4. Schreiben der Funktionen, die die unter 3. erzeugten Funktionen aufrufen und somit das System darstellen (Schreiben der Systemfunktionen)
5. Schreiben der gewünschten Rahmen-Funktion (Schreiben der Rahmen-Funktion). Dies ist in den meisten Fällen die `main`-Funktion.

[3] Wir setzen an dieser Stelle voraus, daß der Anwender sämtliche erforderlichen Einstellungen getätigt hat (was immer diese auch seien mögen, interessiert uns an dieser Stelle noch nicht!).

In genau diesem Schema finden Sie später Ihren C-Code wieder. Dies bedeutet nichts anderes, als daß in dem Quelltext zunächst die `#define` und `#include`-Anweisungen stehen, dann die globalen Variablen, gefolgt von den Systemblockfunktionen und den Systemfunktionen. Zu guter Letzt steht im Quelltext die `main`-Funktion, die wiederum die Systemfunktionen aufruft (dies kann aber auch eine Task sein, die durch eine weitere Funktion in das gesamte Echtzeitbetriebssystem eingeplant wird.)

Als Resultat der Quellcode-Generierung können also die Systemfunktionen angesehen werden. Sie enthalten in ihren Aufrufparametern die Ein- und Ausgänge des Systems.

Strukturanalyse

Für diese Phase müssen Sie die zu generierende Struktur durch Selektion der betreffenden Blöcke festlegen. Gleichzeitig legen Sie damit fest, welche Ein- bzw. Ausgänge bei der späteren Code-Generierung definiert werden. Dabei gelten für die Eingänge folgende Konventionen:

- Alle offenen Eingänge markierter Blöcke sind Eingangsparameter des C-Codes.
- Alle Eingänge markierter Blöcke, die ihre Werte durch nicht markierte Blöcke erhalten, sind Eingangsparameter des C-Codes.
- Ausgabeblöcke wie Zeitverlauf, Oszilloskop usw. erzeugen keinen Eingang.
- Quellen und Senken, Multiplexer und Demultiplexer können nie eigene Eingänge bilden, sondern veranlassen die markierten Blöcke an ihren Ausgängen, diese zu erzeugen.

Beispiele (selektierte Blöcke jeweils grau hinterlegt):

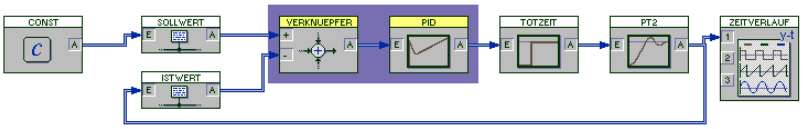

Bild 6. *Zwei Eingangsparameter, da der Verknüpfer keinen markierten Block als Eingangsblock hat*

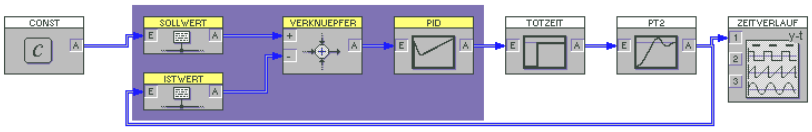

Bild 7. *Wie oben, jedoch werden die Eingangsparameter der erzeugten C-Funktion entsprechend der Label-Blöcke benannt*

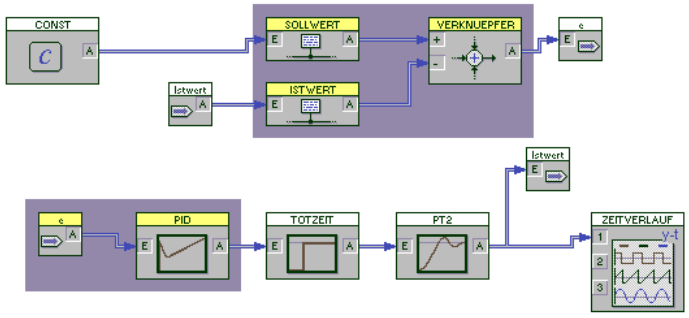

Bild 8. *Drei Eingänge, da hier die Senke der oberen Teilstruktur nicht markiert wurde*

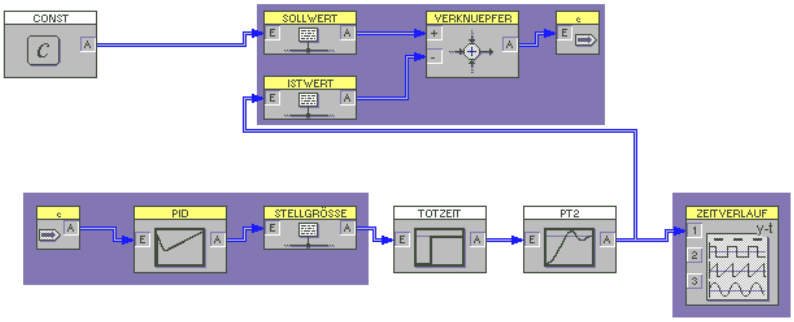

Bild 9. *Zwei Eingangsparameter, da der Zeitverlauf ein Ausgabeblock ist*

Die in Bild 9 dargestellte Struktur hat den Vorteil, daß die Stellgröße in der Simulation dargestellt wird, der Zeitverlaufsblock jedoch keinen weiteren Eingang im C-Code erzeugt. Die Struktur muß also nicht erst verändert werden,

wenn Signale aus dem zu C-Code zu generierenden Teilsystem angezeigt werden sollen. Es sind lediglich die Ausgabeblöcke zu markieren.

In Analogie zur Definition, welche Eingänge im C-Code als Eingangsparameter erstellt werden, läßt sich über die Ausgangsparameter schlußfolgern:

1. Alle offenen Ausgänge markierter Blöcke sind Ausgangsparameter des C-Codes.
2. Alle Ausgänge markierter Blöcke, die ihre Werte durch nicht markierte Blöcke erhalten, sind Ausgangsparameter des C-Codes.
3. Quellen und Senken, Multiplexer und Demultiplexer können nie eigene Ausgänge bilden, sondern veranlassen die markierten Blöcke, diese an ihren Eingängen zu erzeugen.

Zur Hilfe werden nach der Code-Generierung die Titelzeilen der Blöcke, die Ein-/Ausgänge erzeugen, an der entsprechenden Seite grün eingefärbt.

Aber nicht nur die Anzahl der Ein- und Ausgangsparameter wird in dieser ersten Phase festgesetzt, sondern auch die Anzahl der inneren Zustandsvariablen.

Hinweis: Ein-/Ausgänge, die als Parameter des C-Codes anzusehen sind, werden in Zukunft als *externe Ein-/Ausgänge* benannt, während alle anderen *interne Ein-/Ausgänge* bzw. einfach nur *Ein-/Ausgänge* heißen sollen.

Als letzter Schritt der Strukturanalyse wird die Berechnungsreihenfolge der einzelnen Blöcke festgelegt. Sie wird in der dritten Phase, dem Schreiben der Systemfunktionen, benötigt.

Definitionsphase

In diesem ersten Teil der Quellcode-Generierung werden grundlegende Definitionsdirektiven, Includedirektiven und globale Variablen spezifiziert. Dies geschieht in Abhängigkeit von den Einstellungen, die Sie für den C-Code getätigt haben. Die folgende Liste zeigt Ihnen, wann welche `#define`-Direktiven (symb. Konstanten bzw. Makros) eingebaut werden und welche Bedeutung diese haben.

Makro bzw. symb. Konstante	Bedeutung
`MEM_ATTRIBUTE`	Symbolische Konstante für einen Speicherklassifizierer (z. B. static, extern o. a.) für Funktionen. Sie wird in der Headerdatei REGDEF.H definiert.
`MEM_ATTRIBUTE_VAR`	Symbolische Konstante für einen Speicherklassifizierer (z. B. static, extern o. a.) für globale Variablen. Sie wird in der Headerdatei REGDEF.H definiert.
`IntPart`	Symbolische Konstante, die die Anzahl der Vorkommabits wiedergibt.
`FracPart`	Symbolische Konstante, die die Anzahl der Nachkommabits wiedergibt.
`One`	Symbolische Konstante, die den Signalwert 1 wiedergibt.
`SignalMax`	Symbolische Konstante, die den maximal zulässigen Signalwert wiedergibt.
`SignalMin`	Symbolische Konstante, die den minimal zulässigen Signalwert wiedergibt.
`FracMask`	Symbolische Konstante, die zur Maskierung der Nachkommabits verwendet wird.
`IntMask`	Symbolische Konstante, die zur Maskierung der Vorkommabits verwendet wird.
`MUL(a,b,c)`	Makro zur Berechnung des Produktes zweier Signale bzw. eines Parameters mit einem Signal, nicht jedoch zweier Parameter (dazu später mehr)!
`DIV(a,b,c)`	Makro zur Berechnung des Quotienten aus zwei Signalen bzw. aus einem Parameter und einem Signal, nicht jedoch aus zwei Parametern (dazu später mehr)!
`HIGHLEVEL`	Symbolische Konstante, die den in BORIS verwendeten logischen High-Pegel wiedergibt.

`LOWLEVEL`	Symbolische Konstante, die den in BORIS verwendeten logischen Low-Pegel wiedergibt.
`THRESHOLD`	Symbolische Konstante, die den in BORIS verwendeten logischen Umschaltpegel zwischen Low und High wiedergibt.
`PI_DIV_2`	Symbolische Konstante $\pi/2$
`PI_MUL_2`	Symbolische Konstante 2π
`PI`	Symbolische Konstante π
`EXP1`	Symbolische Konstante *e*
`FLOATINGPOINT`	Wird für `#ifdef`-Direktiven verwendet, um zwischen Festpunkt- und Fließpunktzahlen zu unterscheiden.
`SVartyp`	Symbolische Konstante, die den Variablentyp für Signale enthält.
`CastVartyp`	Symbolische Konstante, die den Variablentyp für Castoperationen enthält.
`MAX_X_VEC`	Symbolische Konstante, die die Anzahl der Zustandsgrößen des Systems wiedergibt (entspricht der Anzahl der Ausgänge aller selektierten Blöcke).
`MAX_I_VEC`	Symbolische Konstante, die die Anzahl der Eingangsgrößen des Systems definiert (entspricht der Anzahl aller externen Eingänge).
`SystemIntegration`	Wird für `#ifdef`-Direktiven verwendet, um Deklarationen vom Integrationsverfahren abhängig machen zu können.
`IntMethod`	Symbolische Konstante, die auf den Namen der entsprechenden Integrationsmethode gesetzt wird (alle dynamischen Blöcke bis auf den Block für Differentialgleichungssysteme verwenden diese).
`DGLIntMethod`	Symbolische Konstante, die auf die entsprechende Integrationsmethode für Differentialgleichungssysteme gesetzt wird.

`MAX_DYNAMIC_ORDER`	Symbolische Konstante, die die höchste zulässige Ordnung bei dynamischen Gliedern angibt.
`MAX_DGLSYS_ORDER`	Symbolische Konstante, die die höchste zulässige Ordnung bei Differentialgleichungssystemen angibt.
`MAX_INPUT`	Symbolische Konstante, die die maximale Anzahl der Eingänge enhält.

Tabelle 1. *Erläuterung der verwendeten* `#define`*-Direktiven*

Die `#include`-Anweisungen bedürfen keiner Erklärung. Sie sollen jedoch der Vollständigkeit halber aufgelistet werden.

Name der einzubindenen Datei	**Herkunft**	**Grund des Einbindens**
regdef.h	WinFACT	Enthält Typdeklarationen und Definitionen, die in anderen Typen verwendet werden. Wird immer eingefügt.
actions.h	WinFACT	Enthält Typdeklarationen, die für die Aktionsblöcke verwendet werden
actuator.h	WinFACT	Enthält Typdeklarationen, die für die Stellglieder verwendet werden
digitals.h	WinFACT	Enthält Typdeklarationen, die für die digitalen Blöcke verwendet werden
dynamic.h	WinFACT	Enthält Typdeklarationen, die für die dynamischen Blöcke verwendet werden
function.h	WinFACT	Enthält Typdeklarationen, die für die Funktionsblöcke verwendet werden
fuzzy.h	WinFACT	Enthält Typdeklarationen, die für den FC-Block verwendet werden
Inputs.h	WinFACT	Enthält Typdeklarationen, die für die Eingangsblöcke verwendet werden
outputs.h	WinFACT	Enthält Typdeklarationen, die für die Ausgangsblöcke verwendet werden

statics.h	WinFACT	Enthält Typdeklarationen, die für die statischen Blöcke verwendet werden
tcpipdde.h	WinFACT	Enthält Typdeklarationen, die für die Kommunikationsblöcke verwendet werden
parameter.h	WinFACT	Enthält Typdeklarationen, die für die Parameter-Modifizierer- und Parameter-Wert-Blöcke verwendet werden
signal.h	Compiler-Hersteller	Wird hinzugefügt, wenn die Generierungsparameter *Bereichsüberprüfung* und/oder *Verwenden von Signalen* aktiv sind
limits.h	Compiler-Hersteller	Wird hinzugefügt, wenn Festpunktarithmetik ohne Shift-Operation verwendet wird und die Signalbitbreite>16 ist
stdio.h	Compiler-Hersteller	Wird hinzugefügt, wenn als Rahmen-Funktion (main-Funktion) die *Standard Ein-/Ausgabe* generiert werden soll
limits.h	Compiler-Hersteller	Verwenden von *Shiftoperationen* bei Festpunktzahlen, die größer als 16 Bit sind
math.h	Compiler-Hersteller	*Fließpunktzahlen* in Verbindung mit Blöcken, die Funktionen beinhalten, sollen verwendet werden.

Tabelle 2. *Erläuterung der verwendeten* `#include`*-Direktiven*

Im Gegensatz zu den obigen Tabellen über die Compiler-Anweisungen gestaltet sich die Erzeugung der globalen Variablen etwas schwieriger. Zunächst müssen Variablen für die inneren Zustände des Systems definiert werden. Dies sind immer dieselben Variablen:

Globale Variablen

```
1   MEM_ATTRIBUTE_VAR SVartyp INVDELTAT;
2   MEM_ATTRIBUTE_VAR SVartyp X_VEC[MAX_X_VEC][2];
3   MEM_ATTRIBUTE_VAR SVartyp I_VEC[MAX_I_VEC][2];
4   MEM_ATTRIBUTE_VAR unsigned int TimeStepCounter;
```

Listing 2. *Definition globaler Variablen, die immer vorhanden sind*

Wie zu sehen, hängen die Größen der Vektoren vom definierten System ab. Die Vektoren sind zweidimensional, da ein Systemzustand häufig von dem vorherigen abhängt. Dies ist der Fall, sobald Sie dynamische Blöcke in einem

System verwenden. Die Variable *X_VEC* enthält die inneren Zustände des Systems (Zustandsvektor), die Variable *I_VEC* die externen Eingangsgrößen des Systems (Eingangsvektor). Beide Variablen werden nur dann erzeugt, wenn ein entsprechender Vektor notwendig ist (dies ist aber fast immer der Fall).

`TimeStepCounter` wird nur erzeugt, wenn Sie bei der C-Code-Generierung als Rahmen-Funktion *Standard-Ein-/Ausgabe* oder *Ausgabe in SIM-Dateien* angewählt haben. Dabei dient diese Variable als Schrittzähler.

Anschließend müssen Variablen für die Parameter der Blöcke angelegt werden. Hierbei spielen die mitgelieferten Headerdateien eine wesentliche Rolle. Sie enthalten alle Typdefinitionen der Blöcke in Abhängigkeit davon, ob ein Quelltext für Festpunktzahlen oder Fließpunktzahlen generiert werden soll. Damit die ganze Sache übersichtlich bleibt, wird in diesen Dateien zu jedem Blocktyp, der unter BORIS existiert und auch zu C generiert werden kann, mindestens eine Struktur, die alle Parameter dieses Blocks aufnimmt, deklariert. Zu der eindeutigen Deklaration der PIDT1-Parameter in unserem Beispiel gehören also folgende Programmzeilen:

```
1   #define FLOATINGPOINT
2   #define SVartyp float
3   #define RVartyp float

4   #include <regdef.h>
5   #include "dynamic.h"

6   static PIDT1Struct 'Name der Variablen';
```

Listing 3. *Programmzeilen zur eindeutigen Deklaration des Parametersatzes eines PIDT1-Blocks*

Namen von Variablen

Der Name der Variablen steht auch für die Lesbarkeit des Quelltextes. Also mußte hier ein Kompromiß zwischen eindeutiger und aus Sicht des Anwenders gut lesbarer Namensvergabe gefunden werden. Der endgültige Name der Variablen setzt sich somit aus mehreren Komponenten zusammen:

Variablenname = modifizierter Blockname + _V + *eindeutige Nummer*

Modifizierter Blockname

Die *eindeutige Nummer* entspricht der Reihenfolge, in der die Blöcke innerhalb von BORIS eingefügt wurden. Um nicht irgendwelche ungültigen Variablennamen zu erzeugen, kann nicht direkt der Blockname eingesetzt werden, sondern es muß ein modifizierter, auf unzulässige Zeichen untersuchter Blockname verwendet werden. Die unzulässigen Zeichen werden entsprechend der nachfolgenden Tabelle ersetzt.

ursprünglich	neu
ä	ae
Ä	Ae
ü	ue
Ü	Ue
ö	oe
Ö	Oe
ß	ss
\	_bk_
/	_div_
#	_cr_
*	_mul_
+	_pl_
~	_tilde_
-	_mi_
.	_pkt_
,	_co_
:	_col_
;	_sem_
?	_quest_
§	_phr_
$	_phr_
"	_da_
!	_not_
&	_and_
|	_or_

(	_b_
)	_e_
[	_b2_
]	_e2_
{	_bl_
}	_el_
=	_eq_
^	_pt_
>	_gr_
<	_le_
°	_deg_
@	at
%	_perc_
`	_grav_
'	_apost_
²	_pt2_
³	_pt3_
µ	_mue_
Leerzeichen	_
Namen, die mit einer Ziffer beginnen	Name wird durch führenden Unterstrich korrigiert.

Tabelle 3. *Ersetzungszeichen bei Generierung von Variablennamen*

Beispiel:

Sie haben einen Block, einen Differenzierer, mit dem Namen '*de/dt*' definiert. In der gewandelten Form, also im generierten Quelltext, finden Sie die Parameter dieses Blocks unter dem Variablennamen '*de_div_dt*' mit dem eindeutigen Anhängsel '*_V0*', *wenn dieser Block der zuerst eingefügte ist. Der endgültig Name lautet demnach:* 'de_div_dt_V0'**.**

Funktionssynthese

Die Funktionssynthese beinhaltet das Schreiben aller benötigten Funktionen. Dazu entnimmt fast jeder zu generierende Block Informationen darüber, wo er seine C-Funktion unter welchem Namen findet, einer Verweisdatei.

Verweisdatei:
In der Verweisdatei steht, *wo* BORIS den Funktionsquellcode eines Blocks *unter welchem Namen* suchen muß. Verweisdateien haben die Dateiendung REF. Standardmäßig greift der AutoCode-Generator auf die mitgelieferte Verweisdatei WO_WAS.REF zurück (siehe Anhang A).

Ist die Funktion ausfindig gemacht, wird sie an das augenblickliche Ende des zu generierenden Quelltextes angehängt. Dieses Verfahren läßt sich zusammengefaßt in folgende Schritte unterteilen:

1. Einlesen der eingestellten Verweisdatei
2. Ermitteln der für den gerade zu erzeugenden Block notwendigen Verweise (Datei, in welcher der Funktionscode steht, und Name des Funktionscodes)
3. Einlesen der entsprechenden Datei, auf die unter 2 verwiesen wurde
4. Suchen des Funktionsquellcodes durch den Namen desselben
5. Übertragen des Funktionsquellcodes in die zu generierende C-Datei, bis eine Endemarke gefunden wurde.

Der fünfte Schritt gliedert sich noch feiner, da innerhalb des Funktionsquellcodes Anweisungen stehen dürfen, die von BORIS interpretiert und bearbeitet werden. Wir werden weiter unten auf diese eingehen (s. Kapitel *Funktionscodeabschnitte*).

Aufbau von Verweisdateien

Eine Zeile in einer Verweisdatei hat folgenden Aufbau:

```
/*Blockname*/    Quelldatei    /*Funktionscodename*/
```

Die Zeile kann folgendermaßen interpretiert werden: Der zu /*Blockname*/ gehörige C-Quelltext befindet sich in der Datei *Quelldatei* unter dem darin befindlichen Funktionscodeabschnitt /*Funktionscodename*/.

Die Zeichenfolgen '/' und '*/'*, die in C einen Kommentar eingrenzen, sind in der obigen Form vorgeschrieben. Sie müssen also beim ersten und dritten Zeileneintrag, dürfen aber nicht beim zweiten verwendet werden. Bei der Suche nach einem Funktionscode wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Der erste Eintrag, /*Blockname*/, darf nicht verändert werden! Sollte dies dennoch passieren, findet BORIS den entsprechenden Blocktyp in Ihrer Verweisdatei nicht mehr, was zu einer Fehlermeldung führt.

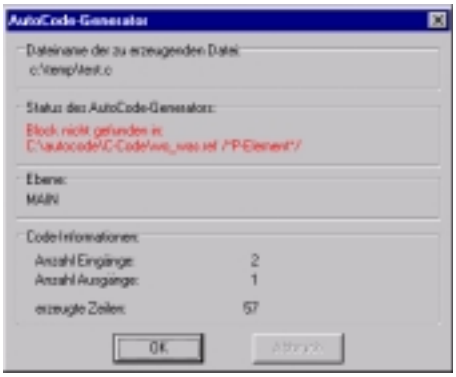

Bild 10. *Fehlermeldung bei Änderung des /*Blocknamens*/ für das P-Glied*

Die Fehlermeldung in Bild 10 zeigt die Reaktion von BORIS, wenn der */*Blockname*/* Eintrag für das P-Glied geändert wurde. In dieser Fehlermeldung sehen Sie auch, unter welchem */*Blocknamen*/* BORIS das P-Glied in der angegebenen Datei (hier: *c:\autocode\C-Code\wo_was.ref*) sucht. Abschließend sollen zwei Beispiele dieses Verfahren nochmals darlegen.

Beispiel 1:

Das P-Glied habe folgenden Eintrag in der Verweisdatei:

*/*P-element*/ dynamic.c /*P-Glied*/*

BORIS wird hierdurch angewiesen, das P-Glied in der Datei *dynamic.c* unter dem Eintrag */*P-Glied*/* zu suchen. Die Datei *dynamic.c* wird hier in allen zugänglichen Verzeichnissen, also zuerst in dem Verzeichnis, in dem auch die Zieldatei des BORIS-C-Code-Generators geschrieben wird und anschließend in den Verzeichnissen, die unter *Optionen / Anpassen...* als Suchverzeichnisse eingestellt wurden, gesucht.

Beispiel 2:

Das I-Glied habe folgenden Eintrag in der Verweisdatei:

*/*Integrator (Euler)*/ 80C166\dynamic.c /*I-Funktionscode*/*

Bei diesem Beispiel würde der Funktionscode, der durch /*I-Funktionscode*/ spezifiziert ist, in der Datei *dynamic.c* im Verzeichnis *80C166* (welches wiederum unter dem aktuellen Verzeichnis steht) gesucht, als nächstes würde die Datei in den Suchverzeichnissen von BORIS gesucht werden.

Die Tabellen im *Anhang A* enthalten den Inhalt der kompletten mitgelieferten Verweisdatei WO_WAS.REF. In dieser wurde */*Funktionscodename*/* meist gleich */*Blockname*/* gewählt.

Funktionscodeabschnitte

Nahezu jeder Blocktyp unter BORIS hat seinen eigenen, durch einen Eintrag in der Verweisdatei referenzierten Funktionscode. Er kann mit Hilfe der Einträge *Quelldatei* und */*Funktionscodename*/* genau lokalisiert werden. Der Codeabschnitt innerhalb der Quelldatei beginnt immer mit dem Eintrag */*Funktionscodename*/* und endet immer mit /*END*/, wobei auch hier die Groß-/Kleinschreibung außer acht gelassen wird. Zwischen diesen beiden Abschnittsmarkierungen steht der Quelltext, der eventuell auch Anweisungen für BORIS enthält.

Das nachfolgende Listing zeigt Ihnen, wie der Funktionscode des P-Gliedes aussieht. Die Funktionen *P_SetParam* und *P_GetParam* werden lediglich von den Blöcken Parameter-Modifizierer und Parameter-Wert verwendet. Die Bedeutung dieser Funktionen dürfte aufgrund der Namensgebung klar sein.

```
1   /*P-Element*/
2   void P_SetParam(PStruct *p, int id, SVartyp value)
3   {
4     p->K = value;
5   }
6
```

```
7   SVartyp P_GetParam(PStruct *p, int id)
8   {
9     return MUL(p->K,FracPart,One);
10  }
11
12  MEM_ATTRIBUTE SVartyp P_fct(PStruct *p, SVartyp e)
13  {
14   return MUL(p->K,FracPart,e);
15  }
16  /*END*/
```

Listing 4. *Funktionscode eines P-Gliedes*

In diesem Listing sind noch keine Anweisungen für BORIS enthalten; anders beim nächsten:

```
1   /*Integrator (Euler)*/
2   void I_SetParam(IStruct *p, int id, SVartyp value)
3   {
4     switch (id){
5     case 1:
6   #ifdef FLOATINGPOINT
7       p->Tn=(value>0)?value: 1.0E-10;
8   #else
9       p->Tn=(value>0)? value: 1;
10  #endif
11      break;
12    case 2:
13      p->y0=value;
14      break;
15    case 3:
16      p->ymin=value;
17      break;
18    case 4:
19      p->ymax=value;
20    }
21  }
22
23  SVartyp I_GetParam(IStruct *p, int id)
24  {
25    switch (id){
```

```
26    case 1: return p->Tn;
27    case 2: return p->y0;
28    case 3: return p->ymin;
29    case 4: return p->ymax;
30    }
31  }
32
33  MEM_ATTRIBUTE SVartyp I_init(IStruct *p,SVartyp *e)
34  {
35    p->y1=p->y0;
36    return p->y0;
37  }
38
39  MEM_ATTRIBUTE SVartyp I_fct(IStruct *p, SVartyp *e)
40  {
41    p->y1=Rgchk(p->y1+DIV(DIV(*e,FracPart,p-
    >Tn),FracPart,#InsertINVDELTAT));
42    if ( p->y1<p->ymin ) p->y1=p->ymin;
43    if ( p->y1>p->ymax ) p->y1=p->ymax;
44    return p->y1;
45  }
46  /*END*/
```

Listing 5. *Funktionscode eines I-Gliedes beim Euler-Integrationsverfahren*

Wie leicht zu sehen ist, ist hier eine Funktionen mehr (*I_init*) im Funktionscodeabschnitt enthalten, da beim Integrierer für die Initialisierung eine separate Behandlung erforderlich ist. Dies ist bei sehr vielen Blocktypen der Fall.

Die in Funktionscodeabschnitten enthaltenen Funktionsköpfe dürfen auf gar keinen Fall geändert werden. Geschieht dies dennoch, ruft der generierte C-Code Funktionen auf, die nicht mehr definiert sind. Dieser Fehler kann nur noch durch den Anwender behoben werden. Im *Anhang B* finden Sie deshalb eine Tabelle, welche Blöcke welche Funktionen erwarten.

In Listing 5 haben wir die erste Anweisung in Zeile 41. #InsertINVDELTAT veranlaßt den AutoCode-Generator, hier die Variable für die Abtastfrequenz einzusetzen. Eine Tabelle soll nun Aufschluß über die möglichen Anweisungen geben. Sie beginnen alle mit dem Doppelkreuz (#) und werden nicht auf Groß-

/Kleinschreibung geprüft. Alle Anweisungen sind komplett in eine Zeile zu schreiben!

Die in Tabelle 3 dargestellten Anweisungen können nur innerhalb der Funktionscodeabschnitte angewendet werden. Außerhalb dieser Abschnitte werden sie nicht berücksichtigt.

Anweisung für den Quellcode-Generator	**Wirkung dieser Anweisung**
#InsertINVDELTAT	Anweisungszeichenkette; wird ersetzt durch den Variablennamen der Abtastfrequenz. Diese Anweisung kann überall im Funktionscode stehen.
#Message /*Mitteilung */	Öffnet (unmittelbar) ein Fenster mit dem Mitteilungstext und einem Ok-Button
#InsertBlock /*Blockname*/	Weist den Quellcode-Generator darauf hin, daß an dieser Stelle ein Funktionscode, der in der Verweisdatei unter dem Namen */*Blockname*/* zu suchen und anschließend zu dereferenzieren ist, eingefügt werden soll (der Funktionscodeabschnitt wird nur dann eingefügt, wenn dieser bis zu diesem Augenblick der Code-Generierung noch nicht eingebunden wurde).
#InsertGlobalBlock /**Blockname*/	Veranlaßt an dieser Stelle gleiches wie #InsertBlock, ist aber bei der Generierung des Quellcodes zeitaufwendiger (vgl. *Aufbau der I/O-Beschreibungsdatei*)
#include "*Dateiname*" oder #include <*Dateiname*>	Bleibt unverändert, wird jedoch an den Anfang des generierten C-Codes geschrieben, um die Lesbarkeit zu erhöhen. Sollte es vorkommen, daß ein Dateiname mehrmals in Verbindung mit dieser Anweisung auftaucht, so wird, falls die Bindungsart (< > bzw. " ") identisch ist, dieser nur einmal eingebunden.

#define ...	Diese Zeile wird in den Definitionsabschnitt des erzeugten C-Codes übernommen. Da dies nur auf eine Zeile zutrifft, müssen Definitionen in einer Zeile stehen. Es wird nur die zuerst aufgetretene Definition eines *Namens* übernommen, alle weiteren werden ignoriert.

Tabelle 4. *Anweisungen für Funktionscodeabschnitte*

Schreiben der Systemfunktionen

Bei der C-Code-Generierung werden in der Regel drei Funktionen geschrieben. Die erste soll *Init-Systemfunktion*, die zweite nur *Systemfunktion* und die dritte *Freigabe-Systemfunktion* genannt werden. Sie stellen das System durch Aufrufe der einzelnen Blockfunktionen dar. Wie der Name der ersten schon andeutet, wird diese zur Initialisierung des Systems eingesetzt. Die Funktionsköpfe der ersten beiden sind bis auf den Namen identisch. Die dritte hat keine Parameter und wird nach der *Systemfunktion* aufgerufen (also wenn die Berechnung des Systems endet), um Freigaben (Speicherfreigabe, Schließen von Dateien etc.) zu tätigen.

```
void Dateinameinitcontrol(
            SVartyp 1. EingangsParametername,
            SVartyp 2. EingangsParametername,
             ...,
            SVartyp * 1. AusgangsParametername,
            SVartyp * 2. AusgangsParametername,
             ...)

void Dateinamecontrol    (
            SVartyp 1. EingangsParametername,
            SVartyp 2. EingangsParametername,
             ...,
            SVartyp * 1. AusgangsParametername,
            SVartyp * 2. AusgangsParametername,
             ...)
void Dateinamefreecontrol    (void)
```

Listing 6. *Funktionsköpfe der Systemfunktionen*

Das im Funktionskopf kursiv geschriebene Prefix *Dateiname* wird durch den Dateinamen der zu generierenden C-Datei ersetzt. Die Namensgebung der einzelnen Parameter der Funktionen erfolgt in ähnlicher Weise wie bei der Namensvergabe der einzelnen globalen Variablen (siehe *Definitionsphase*).

EingangsParametername = *modifizierter Blockname* + _ + *eindeutige Nummer*+ I + *Eingangsnummer des Blocks*

AusgangsParametername = *modifizierter Blockname* + _ + *eindeutige Nummer* + O + *Ausgangsnummer des Blocks*

Neu an diesem "Namenbildungsgesetz" sind die Zeichenketten *Eingangsnummer des Blocks* bzw. *Ausgangsnummer des Blocks*. Dieser Zusatz wird benötigt, um bei einem Block mit mehreren externen Eingängen bzw. Ausgängen eindeutige Variablennamen zu erzeugen.

Beispiel:

Nachfolgende Struktur soll zu C-Quellcode generiert werden. Wir wollen dabei die Variablennamen für sämtliche Ein-/Ausgangsparameter für den Fall ermitteln, daß die Blöcke in der Reihenfolge von links nach rechts eingefügt wurden.

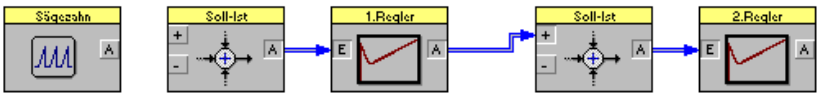

Bild 11. *Beispiel für die Namensvergabe von Ein-/Ausgangsparametern bei Systemfunktionen*

Die folgende Tabelle gibt die Namen der Ein-/Ausgangsparameter wieder.

Beschreibung des Ein-/Ausgangs	**Name des Ein-/Ausgangs**
Erster Soll - Ist -Block (pos. Eingang)	Soll__mi__Ist_1I1
Erster Soll - Ist -Block (neg. Eingang)	Soll__mi__Ist_1I2
Zweiter Soll - Ist - Block (neg. Eingang)	Soll__mi__Ist_3I2
Sägezahn Ausgang	Saegezahn_0O1
2.Regler Ausgang	_2_pkt_Regler_4O1

Tabelle 5. *Namensgebung der Systemfunktionsparameter für obiges Beispiel*

Es liegt also wesentlich an den Blocknamen, wie gut lesbar die im C-Quelltext verwendeten Namen sind!

Nach dem Funktionskopf folgt der Rumpf, das Implementationsstück der Funktion. Hier unterscheiden sich die beiden Funktionen voneinander. In der Init-Systemfunktion wird zunächst der innere Systemzustand initialisiert (d. h. der Zustandsvektor (*X_VEC*) wird auf Null bzw. auf den in dem verantwortlichen Block enthaltenen Anfangszustand gesetzt und der Eingangsvektor (*I_VEC*) wird entsprechend der Eingangsparameter gesetzt).

Anschließend wird jede Blockfunktion in der durch die Systemanalyse festgelegten Berechnungsreihenfolge eingefügt. Dabei werden die entsprechenden Parameterlisten aufgefüllt. Zum Schluß werden die Ausgangsgrößen gesetzt und der alte Systemzustand mit dem neuen überschrieben.

Hier als Beispiel noch einmal der C-Code der beiden Systemfunktionen aus dem obigen Beipiel:

```
1   /*************************************/
2   /* the controller initialising function */
3   /*************************************/
4   void testinitcontrol(SVartyp Soll_mi_Ist_1I1,
5                        SVartyp Soll_mi_Ist_1I2,
6                        SVartyp Soll_mi_Ist_2I2,
7                        SVartyp *Saegezahn_0O1,
8                        SVartyp *2_pkt_Regler_4O1)
9   {
10
11    {
12      /* initialising the state of the system and used exter-
    nals */
13      unsigned int i;
14      for (i=0; i<MAX_X_VEC; i++){X_VEC[i][0]=0;
    X_VEC[i][1]=0;}
15    }
16    {
17      unsigned int i;
18      for (i=0; i<MAX_I_VEC; i++){I_VEC[i][0]=0;
    I_VEC[i][1]=0;}
19    }
20    {
21      /* set state of the system from the external inputs */
```

```
22      I_VEC[0][0]=Soll_mi_Ist_1I1;
23      I_VEC[1][0]=Soll_mi_Ist_1I2;
24      I_VEC[2][0]=Soll_mi_Ist_2I2;
25    }
26    Saegezahn_V0.ZeroStart=0;
27    Saegezahn_V0.Amplitude=1;
28    Saegezahn_V0.Offset=0;
29    Saegezahn_V0.DelayTime=0;
30    Saegezahn_V0.Counter=0;
31    Saegezahn_V0.TWidth=1;
32    Saegezahn_V0.TPeriod=1;
33    Saegezahn_V0.TRise=1;
34    Saegezahn_V0.TFall=0;
35    X_VEC[3][0]=0;
36    1_pkt_Regler_V3.POn=1;
37    1_pkt_Regler_V3.IOn=1;
38    1_pkt_Regler_V3.DOn=1;
39    1_pkt_Regler_V3.LimOn=0;
40    1_pkt_Regler_V3.AW=1;
41    1_pkt_Regler_V3.Kr=1;
42    1_pkt_Regler_V3.Tn=1;
43    1_pkt_Regler_V3.Tv=1;
44    1_pkt_Regler_V3.INVTvz=1000;
45    1_pkt_Regler_V3.ymax=1.7E38;
46    1_pkt_Regler_V3.ymin=-1.7E38;
47    1_pkt_Regler_V3.y0=0;
48    X_VEC[4][0]=0;
49    2_pkt_Regler_V4.POn=1;
50    2_pkt_Regler_V4.IOn=1;
51    2_pkt_Regler_V4.DOn=1;
52    2_pkt_Regler_V4.LimOn=0;
53    2_pkt_Regler_V4.AW=1;
54    2_pkt_Regler_V4.Kr=1;
55    2_pkt_Regler_V4.Tn=1;
56    2_pkt_Regler_V4.Tv=1;
57    2_pkt_Regler_V4.INVTvz=1000;
58    2_pkt_Regler_V4.ymax=1.7E38;
59    2_pkt_Regler_V4.ymin=-1.7E38;
60    2_pkt_Regler_V4.y0=0;
61
```

```
62    /* calls of the used functions in the system */
63    X_VEC[0][0]=PulseGenerator(&Saegezahn_V0);
64    X_VEC[1][0]=+I_VEC[0][0]
65                -I_VEC[1][0];
66    X_VEC[3][0]=PIDT1_init(&1_pkt_Regler_V3,&X_VEC[1][0]);
67    X_VEC[2][0]=+X_VEC[3][0]
68                -I_VEC[2][0];
69    X_VEC[4][0]=PIDT1_init(&2_pkt_Regler_V4,&X_VEC[2][0]);
70
71    {
72      /* set all external outputs from the state of the system
    */
73      unsigned int i;
74      for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
75      *Saegezahn_0O1=X_VEC[0][0];
76      *2_pkt_Regler_4O1=X_VEC[4][0];
77    }
78  }
79
80  /*********************************/
81  /* the controller function itself */
82  /*********************************/
83  void testcontrol(SVartyp Soll_mi_Ist_1I1,
84                   SVartyp Soll_mi_Ist_1I2,
85                   SVartyp Soll_mi_Ist_2I2,
86                   SVartyp *Saegezahn_0O1,
87                   SVartyp *2_pkt_Regler_4O1)
88  {
89
90    {
91      /* set state of the system from the external inputs */
92      unsigned int i;
93      for(i=0; i<MAX_I_VEC; i++) I_VEC[i][1]=I_VEC[i][0];
94      I_VEC[0][0]=Soll_mi_Ist_1I1;
95      I_VEC[1][0]=Soll_mi_Ist_1I2;
96      I_VEC[2][0]=Soll_mi_Ist_2I2;
97    }
98    X_VEC[0][0]=PulseGenerator(&Saegezahn_V0);
99    X_VEC[1][0]=+I_VEC[0][0]
100               +I_VEC[1][0];
```

```
101   X_VEC[3][0]=PIDT1_fct(&1_pkt_Regler_V3,&X_VEC[1][0]);
102   X_VEC[2][0]=+X_VEC[3][0]
103               -I_VEC[2][0];
104   X_VEC[4][0]=PIDT1_fct(&2_pkt_Regler_V4,&X_VEC[2][0]);
105
106   {
107     /* set all external outputs from the state of the system
    */
108     unsigned int i;
109     for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
110     *Saegezahn_0O1=X_VEC[0][0];
111     *2_pkt_Regler_4O1=X_VEC[4][0];
112   }
113 }
114 void testfreecontrol(void)
115 {
116 }
```

Listing 7. *Systemfunktionen des Beipiels aus Bild 11*

Wie Sie sicher bemerkt haben, werden die beiden Verknüpfer nicht durch Funktionsaufrufe, sondern durch direktes Einfügen der Operation (hier Summation) dargestellt. Demzufolge hat der Block 'Verknüpfer' auch keinen Eintrag in der Verweisdatei. Die Freigabe-Systemfunktion ist in diesem Beispiel leer, da keiner der verwendeten Blöcke angeforderten Speicher freigeben oder geöffnete Dateien schließen muß. Nun bleibt noch der Aufruf dieser Funktion durch die Rahmen-Funktion zu erklären.

Schreiben der Rahmen-Funktion

Sie können unterschiedliche Typen von Rahmen-Funktionen im Parameterdialog (siehe *Einstellungen für die Code-Generierung*) einstellen. Wir wollen hier zunächst die ersten drei Typen durchsprechen. Den vierten Typus werden wir in einem eigenständigen Kapitel behandeln.

Standard-Ein-/Ausgabe

Bei dieser Rahmen-Funktion, die eine `main`-Funktion darstellt, werden vom Anwender alle externen Eingangsgrößen über den Standard-Eingabekanal (beim PC ist dies die Tastatur) erfragt. Anschließend wird der Anwender zur

Eingabe der Anzahl der Simulationsschritte aufgefordert. Sind diese Eingaben abgeschlossen, so wird mit diesen der Simulationslauf durchgeführt. Dabei werden alle externen Ausgänge für jeden einzelnen Simulationsschritt ausgegeben.

Die `main`-Funktion läßt sich demnach in drei Teile gliedern:

1. Eingabe aller für die Simulation notwendigen Daten
2. Aufruf der Init-Systemfunktion mit anschließender Ausgabe der externen Größen
3. Durchführen der Simulationsschleife, die auch die Ausgabe der externen Größen enthält.

Abschließend wollen wir an dieser Stelle noch die `main`-Funktion aus unserem letzten Beispiel vorstellen:

```
1   int main(void)
2   {
3     unsigned int num;
4     SVartyp Soll_mi_Ist_1I1=0;
5     SVartyp Soll_mi_Ist_1I2=0;
6     SVartyp Soll_mi_Ist_2I2=0;
7     SVartyp Saegezahn_0O1=0;
8     SVartyp 2_pkt_Regler_4O1=0;
9     INVDELTAT=10;
10    TimeStepCounter=0;
11    printf("block-no.:1  1. input: Soll_mi_Ist=");
12    scanf("%g",&Soll_mi_Ist_1I1);
13    printf("block-no.:1  2. input: Soll_mi_Ist=");
14    scanf("%g",&Soll_mi_Ist_1I2);
15    printf("block-no.:2  2. input: Soll_mi_Ist=");
16    scanf("%g",&Soll_mi_Ist_2I2);
17    printf("last simulation step (>1): =");
18    scanf("%d",&num);
19    if(num<=1)num=2;
20    num++;
21    testinitcontrol(Soll_mi_Ist_1I1,
22                    Soll_mi_Ist_1I2,
23                    Soll_mi_Ist_2I2,
24                    &Saegezahn_0O1,
25                    &2_pkt_Regler_4O1);
```

```
26   printf("t=%6.3g  output: Sägezahn =
   %g\n",TimeStepCounter/INVDELTAT,Saegezahn_0O1);
27   printf("t=%6.3g  output: 2.Regler =
   %g\n",TimeStepCounter/INVDELTAT,2_pkt_Regler_4O1);
28   for(TimeStepCounter=1; TimeStepCounter<num; TimeStepCoun-
   ter++)
29     {
30       testcontrol(Soll_mi_Ist_1I1,
31                   Soll_mi_Ist_1I2,
32                   Soll_mi_Ist_2I2,
33                   &Saegezahn_0O1,
34                   &2_pkt_Regler_4O1);
35       printf("t=%6.3g  output: Sägezahn =
   %g\n",TimeStepCounter/INVDELTAT,Saegezahn_0O1);
36       printf("t=%6.3g  output: 2.Regler =
   %g\n",TimeStepCounter/INVDELTAT,2_pkt_Regler_4O1);
37     }
38   testfreecontrol();
39   return 0;
40 }
```

Listing 8. *main - Funktion bei der Einstellung* Standard-Ein-/Ausgabe

Bei der Ausführung des kompletten Programms werden Sie bemerken, daß dreimal die Eingabe von 'Soll__mi__Ist' erfragt wird. Dies liegt daran, daß der Block 'Soll-Ist' unter BORIS zweimal existiert. Der eine liefert den ersten und zweiten, der andere nur den zweiten Eingang als externen Eingang. Zur genaueren Unterscheidung an dieser Stelle sollte man sich also eine eindeutige Namensgebung der Blöcke innerhalb der Simulationsumgebung einfallen lassen oder die für diesen Zweck vorgesehenen Label-Blöcke verwenden!

Ausgabe in SIM-Dateien

Diese Art der Rahmen-Funktion (`main`-Funktion) erfragt vom Anwender alle externen Eingänge der entworfenen Struktur (über die Standard-Eingabe; meist also über die Tastatur). Abschließend muß noch die Anzahl der Simulationsschritte vorgegeben werden. Während der nun erfolgenden Simulation werden alle errechneten externen Ausgangsgrößen in jeweils eine SIM-Datei[4] geschrieben. Die Namen der SIM-Dateien entsprechen dem Namen der zu erzeu-

[4] Zur Erläuterung des Aufbaus von SIM-Dateien kann das WinFACT-Benutzerhandbuch herangezogen werden.

genden C-Quellcodedatei, wobei eine fortlaufende Nummer an den Dateinamen angehängt wird. Heißt Ihre Zieldatei z. B. REGLER.C und die generierte Struktur enthielte zwei externe Ausgänge, so erhielten die beim Simulationslauf erzeugten SIM-Dateien die Namen REGLER0.SIM und REGLER1.SIM. Um zu wissen, was in welcher SIM-Datei enthalten ist, brauchen Sie nur den Kommentarteil derselben zu inspizieren (geschieht beim Öffnen einer SIM-Datei unter WinFACT automatisch). In diesem steht der Name des Blocks, dessen Ausgang Sie betrachten möchten, in der gleichen Notation, wie Sie ihn unter BORIS finden.

Zum Abschluß wieder zurück zu unserem Beispiel. Das nachfolgende Listing zeigt das Aussehen der erzeugten `main`-Funktion bei der Einstellung *Ausgabe in SIM-Dateien.*

```
1   int main(void)
2   {
3     unsigned int num;
4     FILE *fp[2];
5     SVartyp Soll_mi_Ist_1I1=0;
6     SVartyp Soll_mi_Ist_1I2=0;
7     SVartyp Soll_mi_Ist_2I2=0;
8     SVartyp Saegezahn_0O1=0;
9     SVartyp 2_pkt_Regler_4O1=0;
10    INVDELTAT=10;
11    TimeStepCounter=0;
12    printf("block-no.:1  1. input: Soll_mi_Ist=");
13    scanf("%g",&Soll_mi_Ist_1I1);
14    printf("block-no.:1  2. input: Soll_mi_Ist=");
15    scanf("%g",&Soll_mi_Ist_1I2);
16    printf("block-no.:2  2. input: Soll_mi_Ist=");
17    scanf("%g",&Soll_mi_Ist_2I2);
18    printf("last simulation step (>1): =");
19    scanf("%d",&num);
20    if(num<=1)num=2;
21    num++;
22    testinitcontrol(Soll_mi_Ist_1I1,
23                    Soll_mi_Ist_1I2,
24                    Soll_mi_Ist_2I2,
25                    &Saegezahn_0O1,
26                    &2_pkt_Regler_4O1);
27    fp[0]=fopen("C:\\Temp\\test0.SIM","wt");
```

```
28   if (fp[0]==NULL)
29     {
30       fprintf(stderr,"fopen(C:\\Temp\\test0.SIM) failed!");
   return 1;
31     }
32   fprintf(fp[0],"! Sägezahn\n");
33   fprintf(fp[0],"%g
   %g\n",TimeStepCounter*(float)1/INVDELTAT,Saegezahn_0O1);
34   fclose(fp[0]);
35   fp[1]=fopen("C:\\Temp\\test1.SIM","wt");
36   if (fp[1]==NULL)
37     {
38       fprintf(stderr,"fopen(C:\\Temp\\test1.SIM) failed!");
   return 1;
39     }
40   fprintf(fp[1],"! 2.Regler\n");
41   fprintf(fp[1],"%g
   %g\n",TimeStepCounter*(float)1/INVDELTAT,2_pkt_Regler_4O1);
42   fclose(fp[1]);
43   for(TimeStepCounter=1; TimeStepCounter<num; TimeStepCoun-
   ter++)
44     {
45       testcontrol(Soll_mi_Ist_1I1,
46                   Soll_mi_Ist_1I2,
47                   Soll_mi_Ist_2I2,
48                   &Saegezahn_0O1,
49                   &2_pkt_Regler_4O1);
50       fp[0]=fopen("C:\\Temp\\test0.SIM","at");
51       fprintf(fp[0],"%g
   %g\n",TimeStepCounter*(float)1/INVDELTAT,Saegezahn_0O1);
52       fclose(fp[0]);
53       fp[1]=fopen("C:\\Temp\\test1.SIM","at");
54       fprintf(fp[1],"%g
   %g\n",TimeStepCounter*(float)1/INVDELTAT,2_pkt_Regler_4O1);
55       fclose(fp[1]);
56     }
57   testfreecontrol();
58   return 0;
59 }
```

Listing 9. *main-Funktion bei der Einstellung* Ausgabe in SIM-Datei

In den Zeilen 32 und 40 wird der Kommentar geschrieben, der für die Identifizierung der Daten in der SIM-Datei herangezogen werden kann.

Windows-DLL

Mit dieser Einstellung veranlassen Sie den AutoCode-Generator, eine C-Datei zu schreiben, deren Compilat eine Windows-DLL darstellt. Diese können Sie wieder mit Hilfe der BORIS-User-DLL-Schnittstelle[5] in die Simulation einbinden. Dieses Verfahren eignet sich besonders gut für die numerische Optimierung von Regler-Strukturen auf Basis von Evolutionsstrategien[6], da es ganz erheblich die Dauer der Optimierung verkürzt. Es kann aber auch zur übersichtlicheren Darstellung komplexerer Strukturen oder einfach aus Geschwindigkeitsgründen angewendet werden. Dabei ist jedoch eine Einschränkung genereller Art zu beachten:

Achtung: Eine mittels des AutoCode-Generators erzeugte USER-DLL darf nicht *mehrmals* in einer BORIS-Systemstruktur auftreten. Wird dieses nicht beachtet, werden innere Zustandsgrößen des ersten User-DLL-Blocks durch den Aufruf der zweiten bzw. nachfolgenden DLL zerschrieben. Dadurch werden Ihre Simulationsergebnisse unbrauchbar! Sie können die gleiche DLL nur dann mehrmals innerhalb einer Simulationsstruktur verwenden, wenn Sie die DLL unter einem anderen Namen laden. Dazu müssen Sie die gleiche DLL mehrmals mit unterschiedlichem Namen auf Ihren Datenträger kopiert haben.

Beim Schreiben eines Quelltextes für eine USER-DLL werden sechs Funktionscodeabschnitte geschrieben. Ihre Parameter */*Blockname*/* heißen */*Windows-DLL Part 1*/* bis */*Windows-DLL Part 6*/*. Diese Teile werden der Reihe nach geschrieben, wobei zwischen den einzelnen Teilen die entsprechenden Abschnitte des auch sonst erzeugten C-Codes eingefügt werden. Das Generieren einer Windows-DLL erfolgt demnach in den folgenden Schritten:

1. Schreiben von /*Windows-DLL Part 1*/
2. Durchführen aller obigen Ablaufphasen bis die Systemfunktionen geschrieben worden sind.

---

[5] Die User-DLL-Schnittstelle ist ein Zusatzmodul für das WinFACT-Modul BORIS und muß, falls sie nicht schon in Ihren Softwarepaket enthalten ist, zusätzlich erworben werden.

[6] Die numerische Optimierung von Reglerstrukturen kann nur erfolgen, wenn Sie das Zusatzmodul für diese Optimierung erworben haben.

3. Schreiben von /*Windows-DLL Part 2*/
4. Einfügen der Definitionen für die Namensgebung der Ein- und Ausgangsgrößen
5. Schreiben von /*Windows-DLL Part 3*/
6. Einfügen des Aufrufs der Init-Systemfunktion
7. Schreiben von /*Windows-DLL Part 4*/
8. Einfügen des Aufrufs der Systemfunktion
9. Schreiben von /*Windows-DLL Part 5*/
10. Einfügen des Aufrufs der Freigabe-Systemfunktion
11. Schreiben von /*Windows-DLL Part 6*/

An dieser Reihenfolge sehen Sie den grundsätzlichen Aufbau des C-Quelltextes, so daß sich ein Listing an dieser Stelle erübrigt.

Benutzerdefinierte Rahmen-Funktionen

Der AutoCode-Generator unterstützt das Hinzufügen benutzerdefinierter Funktionen, die die Systemfunktion aufrufen. Die Funktionen können als Task eines Echtzeitbetriebssystems oder auch einfach als `main`-Funktion gestaltet sein; sie müssen in einer bestimmten Datei abgelegt sein, die wieder verschiedene Funktionscodeabschnitte enthalten darf. Diese Datei soll von nun an *I/O-Beschreibungsdatei* (die Funktionscodeabschnitte dieser Datei werden überwiegend zur Implementierung von hardwarenahen Ein-/Ausgabefunktionen geschrieben) genannt werden. Die Erläuterung dieser Datei finden Sie im nachfolgenden Kapitel.

Aufbau der I/O-Beschreibungsdatei

I/O-Beschreibungs-datei

Die I/O-Beschreibungsdatei ist ein wesentlicher Bestandteil der offenen Architektur des AutoCode-Generators unter BORIS. Sie bietet Ihnen die Möglichkeit, direkt aus der BORIS-Oberfläche Ihre Hardwarekanäle (z. B. Analog-ein- und -ausgänge) einzubinden. Dazu dient der Block *C-Code-Funktion* im *Systemblöcke*-Menü von BORIS. In diesem Block lassen sich die Funktionen, die Sie in der I/O-Beschreibungsdatei geschrieben haben, einbinden und mit den übrigen Systemblöcken "verdrahten". Ebenso haben Sie mit Hilfe dieser Dateiform die Möglichkeit, eigene Rahmen-Funktionen beim Generieren von C-Code hinzubinden zu lassen. I/O-Beschreibungsdateien tragen in der Regel wie C-Quelldateien die Extension C.

Funktionen für den Block *C-Code-Funktion*

Der grundlegende Aufbau der I/O-Beschreibungsdatei ähnelt sehr dem Inhalt der *Quelldateien*, die in der Verweisdatei angegeben wurden (vgl. *Funktionssynthese*). Auch in der I/O-Beschreibungsdatei werden Funktionscodeabschnitte spezifiziert. Dabei ist ein Abschnitt im Hinblick auf Funktionen für die C-Code-Funktionsblöcke durch die in Listing 10 aufgeführten Merkmale gekennzeichnet.

```
/*Abschnittsname*/
#usedAddresses: PortA;
#Inputs: Anzahl
#InsertBlock /*Abschnittsname aus dieser  I/O-
Beschreibungsdatei*/
Funktionskopf
Funktionsrumpf
/end*/
```

Listing 10. *Funktionscodeabschnitt innerhalb einer I/O-Beschreibungsdatei*

An dem Abschnittsnamen müssen Sie Ihre Funktion in den BORIS *C-Code-Ein-* bzw. *Ausgangsblöcken* wiedererkennen. Die in Zeile 3 stehende Anweisung '`#Inputs:`' kann, um einen Ausgangskanal auf der Hardware zu spezi-

fizieren, auch '`#Outputs:`' lauten. Unter BORIS hätte der C-Code-Funktionsblock bei Anwahl dieser Funktion entsprechend viele Eingänge. Beide beschreiben die Anzahl der Ein- bzw. Ausgänge, mit denen der Block unter BORIS versehen werden soll.

Verriegelung von Kanälen

Die Anweisung '`#usedAddresses:`' ermöglicht die gegenseitige Verriegelung von Funktionsabschnitten, die auf der Zielhardware gleiche Kanäle ansprechen. So können Sie in der Simulationsumgebung einem Ausgangskanal nicht mehrfach einen Wert zuweisen. Zur Verriegelung können beliebige Namen für die verwendeten Adressen (in obigem Listing z. B. *PortA*) vergeben werden. Die Namen müssen durch Kommata voneinander getrennt sein, ein Semikolon beendet die Liste, muß aber nicht vorhanden sein. Die Funktionsweise der Verriegelung ist denkbar einfach: Funktionen, die einen oder mehrere gleiche Adress-Namen in der Anweisung '`#usedAddresses:`' enthalten, werden gegeneinander verriegelt. Sie sind somit nur einmalig unter der BORIS-Simulationsumgebung verfügbar.

Beispiel:

Nehmen wir an, Sie haben zwei 8-Bit-D/A-Wandler an den 16 Bit breiten Port 2 Ihrer Hardware angeschlossen, einen an die höherwertigen acht Bit, einen an die niederwertigen acht. Die Funktionen seien jedoch so geschrieben, daß sie immer die kompletten 16 Bit in diese Portadresse schreiben. Würden Sie beide D/A-Wandler innerhalb eines Systems verwenden, überschriebe der erste den Wert des zweiten, da ja grundsätzlich die kompletten 16 Bit geschrieben würden. Dieser Mißstand kann nun dadurch behoben werden, daß Sie es erst gar nicht zu dieser Möglichkeit kommen lassen. Es soll also immer nur eine dieser beiden C-Code-Ausgangsfunktionen ansprechbar sein. Demnach müssen Sie dafür sorgen, daß in der Namensliste bei '`#usedAddresses:`' der gleiche Name auftaucht. Im untenstehenden Listing ist es 'Port 2'.

```
1   /*Analog Out (high byte von Port2)*/
2   #Outputs: 1
3   #usedAddresses: Port 2
4   ....
5   /*end*/

6   /*Analog Out (low byte von Port2)*/
7   #Outputs: 1
8   #usedAddresses: Port 2
9   ....
10  /*end*/
```

Listing 11. *Beispiel gegenseitig verriegelter Funktionen*

Werden statt der 16 Bit nur die jeweils benötigten acht Bit in den Funktionen beeinflußt (die übrigen acht bleiben unverändert), dann könnten die Funktionen folgendermaßen aussehen:

```
1   /*Analog Out (high byte von Port2)*/
2   #Outputs: 1
3   #usedAddresses: H_Port 2
4   ....
5   /*end*/

6   /*Analog Out (low byte von Port2)*/
7   #Outputs: 1
8   #usedAddresses: L_Port 2
9   ....
10  /*end*/
```

Listing 12. *Beispiel nicht gegenseitig verriegelter Funktionen, die aber die Verriegelungstechnik nutzen, indem sie Einträge bei* `#usedAddresses` *aufweisen.*

Hätten Sie nun eine dritte Funktion, die den 16 Bit breiten Port 2 komplett als Digitalausgang verwenden würde, müßten Sie diese gegen die obigen beiden verriegeln:

```
11  /*Digital Out (Port2)*/
12  #Outputs: 1
13  #usedAddresses: H_Port 2, L_Port 2
14  ....
15  /*end*/
```

Listing 13. *Mögliche dritte Funktion, die auf den gleichen Ausgang zugreift*

Kommen wir nun zu der Ihnen schon aus dem Kapitel *Funktionssynthese* bekannten `#InsertBlock`-Anweisung. Diese weist den Quellcode-Generator von BORIS an, an dieser Stelle einen Funktionscodeabschnitt einzufügen (sofern dieser bis zu diesem Zeitpunkt noch nicht eingefügt wurde). Hierbei wird aber **nicht** auf die Verweisdatei zugegriffen, sondern, da der Abschnitt aus einer I/O-Beschreibungsdatei stammt, wird auch nur in dieser gesucht! Soll auf einen Funktionscodeabschnitt mit Hilfe der Verweisdatei zugegriffen werden, so ist die Anweisung `#InsertGlobalBlock` zu verwenden.

Die Anweisungen `#Message` und `#InsertINVDELTAT` bleiben uneingeschränkt verwendbar.

Die nachfolgende Tabelle verschafft nochmal einen schnelleren Überblick über die in einer I/O-Beschreibungsdatei innerhalb der Funktionscodeabschnitte verwendbaren Anweisungen für die Quellcode-Generierung. Alle Anweisungen müssen komplett in einer Zeile stehen!

Anweisung für den Quellcode-Generator	**Wirkung dieser Anweisung**
#InsertINVDELTAT	Anweisungszeichenkette; wird ersetzt durch den Variablennamen der Abtastfrequenz. Diese Anweisung kann überall im Funktionscode stehen.
#Message /**Mitteilung*/	Öffnet (unmittelbar) ein Fenster mit dem Mitteilungstext und einem Ok-Button
#InsertBlock /**Blockname*/	Weist den Quellcode-Generator darauf hin, daß an dieser Stelle ein Funktionscode unter dem Namen /**Blockname*/ eingefügt werden soll. Das Einfügen wird nur durchgeführt, wenn dieser Abschnitt noch nicht eingefügt wurde. **Es wird davon ausgegangen, daß der zugehörige Funktionscode in dieser I/O-Beschreibungsdatei steht!**
#InsertGlobalBlock /**Blockname*/	Arbeitet wie #InsertBlock. Jedoch wird hierbei davon ausgegangen, daß der zugehörige Funktionscodeabschnitt in der Verweisdatei eingetragen ist.

#include "*Dateiname*" oder #include <*Dateiname*>	Bleibt unverändert, wird jedoch an den Anfang des generierten C-Codes geschrieben, um die Lesbarkeit zu erhöhen. Sollte ein Dateiname mehrmals in Verbindung mit dieser Anweisung auftreten, so wird, falls die Bindungsart (< > bzw. " ") identisch ist, dieser nur einmal eingebunden.
#define	Diese Zeile wird in den Definitionsabschnitt des erzeugten C-Codes übernommen. Da dies nur auf eine Zeile zutrifft, müssen Definitionen in einer Zeile stehen. Treten mehrere Definitionen des gleichen Namens auf, wird nur die zuerst aufgetretene Definition übernommen, alle weiteren werden ignoriert.
#Inputs: *n*	Gibt die Anzahl der Eingänge für diesen Funktionsabschnitt an
#Outputs: *n*	Gibt die Anzahl der Ausgänge für diesen Funktionsabschnitt an
#Bitmap: *Dateiname.bmp*	Bei Auswahl des Funktionsabschnittes im Block *C-Code-Funktion* wird die Bitmap als Bild in diesem Block angezeigt. Beim Drucken und bei der Strukturübersicht wird die Datei *Dateiname*_P.*bmp* gesucht.
#usedAddresses: *N1, N2, N3*	Gibt BORIS Auskunft über gegeneinander zu verriegelnde Funktionen. Funktionen, die bei dieser Anweisung ein oder mehrere gleiche Listenelemente enthalten, werden in der Simulationsumgebung nur einmalig verfügbar gemacht.

Tabelle 6. *Auflistung der innerhalb der Funktionsabschnitte für C-Code-Ein-/Ausgangsblöcke verwendbaren Anweisungen für BORIS bzw. den AutoCode-Generator*

Der Funktionskopf einer selbstdefinierten Ein- bzw. Ausgangsfunktion muß als Parameterliste immer genau so viele Argumente enthalten wie bei `#Inputs`

und #Outputs zusammen angegeben wurden. Die Typen der Eingangsparameter der Funktion dürfen 'SVartyp *' oder 'SVartyp' (s. Kapitel *Definitionsphase*) sein; die der Ausgangsparameter müssen als Referenz 'SVartyp *' definiert sein. Der Name der Funktion selbst ist beliebig, muß aber den Notationen Ihres C-Compilers genügen. BORIS interpretiert immer den Namen der ersten Funktion innerhalb eines Funktionsabschnittes als aufzurufende Funktion; falls diese Funktion weitere selbst geschriebene benötigt, müssen diese durch die #InsertBlock-Anweisung hinzugebunden werden!

Die aus dem obigen Beispiel verwendeten Funktionen, ergänzt um eine Eingangsfunktion und eine Funktion, die sowohl einen Eingang als auch einen Ausgang hat, könnten mit Funktionskopf wie folgt aussehen:

```
1   /*3 Analog In (0,1,2)*/
2   #Inputs: 3
3   #usedAddresses:
4   #InsertBlock /*Multiplexer Eingangskanal*/
5   void AD1_2_3(SVartyp *ad0,SVartyp *ad1,SVartyp *ad2)
6   { *ad0=Multiplexer(0);
7     *ad1=Multiplexer(1);
8     *ad2=Multiplexer(2);
9   }
10  /*end*/
11
12  /*Analog Out (high byte von Port2)*/
13  #Outputs: 1
14  #usedAddresses: H_Port 2
15  void DAPort2(SVartyp e1)
16  {...}
17  /*end*/
18
19  /*Analog Out (low byte von Port2)*/
20  #Outputs: 1
21  #usedAddresses: L_Port 2
22  void DAPort2(SVartyp e1)
23  {....}
24  /*end*/
25
26  /*Digital Out (Port2)*/
27  #Outputs: 2
```

```
28  #usedAddresses: H_Port 2, L_Port 2;
29  void Digitalout_on_Port2(SVartyp e1,SVartyp e2)
30  {....}
31  /*end*/
32  /*AD (mux-select)*/
33  #Inputs: 1
34  #Outputs: 1
35  #usedAdresses:
36  #InsertBlock /*Multiplexer Eingangskanal*/
37  void AD_select(SVartyp ch, Svartyp *i1)
38  {
39   #ifdef FLOATINGPOINT
40    *i1=Multiplexer((int)ch);  /* (int)-Cast verursacht
                                   wahrscheinlich die
                                   Warnung: 'Conversion may
                                   loose signifcant bits' wir
                                   wollen aber genau dies ! */
41   #else
42    *i1=Multiplexer(ch/One);   /* One ist bei
                                   Festpunktzahlen > 1 ! */
43   #endif
44  }
```

Listing 14. *Zusammenhang zwischen Funktionsköpfen und Angabe der Anzahl der Ein- bzw. Ausgänge der Funktion*

Eigene Rahmen-Funktionen

Um vom Anwender geschriebene Funktionen, die die Systemfunktionen aufrufen (z. B. eine `main`-Funktion), dauerhaft optional einzubinden, wurden weitere Anweisungen für den AutoCode-Generator realisiert.

1. `#MainFunction` spezifiziert die nach dieser Anweisung stehende Funktion als main-Funktion; diese Anweisung ist sinnvoll, wenn die in diesem Funktionscodeabschnitt stehende Funktion anders als *main* lautet (z.B.: *void Task1(void)*).
2. `#InsertInitControlCall` fügt den Init-Systemfunktionsaufruf ein.
3. `#InsertControlCall` fügt den Systemfunktionsaufruf ein.

4. `#InsertFreeControlCall` fügt den Freigabe-Systemfunktionsaufruf ein.

Allein ihre Namen dürften die Bedeutung dieser Anweisungen klären. Die innerhalb eines `main`-Funktionsabschnittes verwendbaren Anweisungen sind in der folgenden Tabelle aufgeführt.

Anweisung für den Quellcode-Generator	**Wirkung dieser Anweisung**
#InsertINVDELTAT	s. o.
#Message /*Mitteilung */	s. o.
#InsertBlock /*Blockname*/	s. o.
#include "*Dateiname*" oder #include <*Dateiname*>	s. o.
#define ...	s. o.
#MainFunction	Die nächste nach dieser Anweisung stehende Funktion wird als main-Funktion interpretiert.
#InsertInitControlCall	Diese Anweisung wird durch den Init-Systemaufruf ersetzt. Sie darf an beliebiger Stelle innerhalb des Funktionstextes, also auch in den von dem Rahmen-Funktionscodeabschnitt durch die Anweisung #InsertBlock eingefügten Funktionscodeabschnitten stehen.
#InsertControlCall	Diese Anweisung arbeitet gleichermaßen wie #InsertInitControlCall, jedoch mit dem Unterschied, daß hier die normale Systemfunktion statt der Init-Systemfunktion eingefügt wird.
#InsertFreeControlCall	Diese Anweisung arbeitet gleichermaßen wie #InsertInitControlCall, jedoch mit dem Unterschied, daß hier die Freigabe-Systemfunktion statt der Init-Systemfunktion eingefügt wird.

Tabelle 7. *Anweisungen bezüglich benutzerdefinierter main-Funktionen*

Um diese trockene Auflistung von Anweisungen für den AutoCode-Generator abzurunden, nachfolgend wieder ein kurzes Beispiel.

Beispiel:

Das nachfolgende Listing demonstriert die beiden neuen Anweisungen durch eine `main`-Funktion, die einen Polling-Betrieb eines Systems zeigt.

```
1   /*Polling Hauptfunktion*/
2   #include <stdlib> /* wegen atexit() */
3   int main()
4   {
5     atexit(#InsertFreeControlCall);
6     #InsertInitControlCall;
7     while(1) #InsertControlCall;
8   }
9   /*end*/
```

Listing 15. *Beispiel der Verwendung von #Insert(Init)ControlCall*

Wie in dem Listing zu sehen ist, müssen die Code-Generierungsanweisungen wie bei den Anweisungen unter C syntaktisch korrekt abgeschlossen werden (Semikola am Ende der Anweisungen!).

Beispiel:

Aus dem nachfolgenden Listing wird ersichtlich, daß vor einer `main`-Funktion andere in der `main`-Funktion benötigte Funktionen spezifiziert werden dürfen (bei den Funktionen für C-Code-Ein-/Ausgänge war dies unzulässig!). Im folgenden Listing ist es die Funktion `Graphinit`().

```
1   /*realtime graphics*/
2   #include <stdlib.h>  /* wegen atexit() */
3   /************************************/
4   /*    main for graphical outputs    */
5   /************************************/
6
7   void Graphinit()
8   { int gdriver = DETECT, gmode, errorcode;
9     /* initialize graphics mode */
10    initgraph(&gdriver, &gmode, "");
11
12    /* read result of initialization */
13    errorcode = graphresult();
```

```
14
15    if (errorcode != grOk){
16      printf("Graphics error:%s\n",grapherrormsg(errorcode));
17      exit(1);             /* return with error code */
18    }
19  }
20
21  MEM_ATTRIBUTE int main(void)
22  { Graphinit();
23    atexit(#InsertFreeControlCall);
24    #InsertInitControlCall;
25    while(1) #InsertControlCall;
26    closegraph();
27  }
28  /*end*/
```

Listing 16. *Funktionscodeabschnitt für eine main-Funktion, die eine vor ihr im gleichen Abschnitt deklarierte Funktion aufruft (unzulässig bei Funktionen für C-Code-Ein-/Ausgänge)*

C-Code-Funktionsblöcke

Durch die C-Code-Funktionsblöcke haben Sie die Möglichkeit, direkt aus der BORIS-Umgebung die Schnittstellen beliebiger Hardware für die C-Code-Generierung einzubinden (nicht aber zu simulieren!).

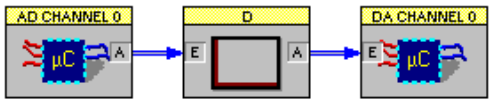

Bild 12. *C-Code-Block speist über einen Differenzierer einen weiteren C-Code-Funktionsblock*

In Bild 12 wurde ein C-Code-Eingangsblock über einen Differenzierer direkt mit einem C-Code-Ausgangsblock verschaltet. Aus den Blocknamen kann man schließen, daß aus Sicht der späteren Hardware der A/D-Wandler Kanal 0 differenziert wird und auf dem D/A-Wandler Kanal 0 ausgegeben wird.

Dazu muß die entsprechende I/O-Beschreibungsdatei, die den Quelltext zum Ansprechen der Schnittstellen in Funktionscodeabschnitten enthält, eingestellt sein (s. *Wahl der I/O-Beschreibungsdatei*). Innerhalb der Dialoge der C-Code-

Funktionsblöcke können Sie den gewünschten Funktionscodeabschnitt einstellen:

Bild 13. *Dialog eines C-Code-Funktions-Blockes*

In dem Listenfeld stehen nur die Funktionscodeabschnitte, die nicht verriegelt sind. Wird ein Eintrag aus dem Listenfeld gewählt, wird der Name des Funktionscodeabschnitts direkt als Blockname verwendet. So haben Sie stets die Übersicht, welche Funktion in welchem C-Code-Ein-/Ausgangsblock enthalten ist.

Ist keine I/O-Beschreibungsdatei gewählt, so werden die C-Code-Ein-/Ausgangsblöcke auf den einzigen Eintrag, der immer in der Auswahlliste vorhanden ist, *Keine Funktion*, eingestellt. Sobald Sie nun eine Beschreibungsdatei einstellen, füllen sich auch automatisch die Auswahllisten dieser Blöcke.

Bei Änderung der I/O-Beschreibungsdatei versuchen die C-Code-Ein-/Ausgangsblöcke, ihre Funktion beizubehalten. Dies bedeutet, sie durchsuchen der Reihe nach die Datei nach ihrem Funktionscodeabschnitt. Wird er gefunden, so bleibt dieser Block unverändert. Wird der entsprechende Abschnitt nicht gefunden, so wird der Block auf *Keine Funktion* gestellt. Er enthält aber dennoch alle zugänglichen Funktionscodeabschnitte, die noch nicht verriegelt wurden, in seinem Listenfeld.

Simulation mit C-Code-Ein-/Ausgangs-blöcken

C-Code-Funktionsblöcke geben bei der Simulation immer den Wert Null aus und ignorieren ihren Eingangswert; sie verhalten sich wie passive Blöcke. Will man sowohl in der Simulation als auch in dem generierten C-Code frei definierte Funktionalität haben, so muß der User-DLL-Block verwendet werden. Dieser wird in einem eigenen Kapitel behandelt.

Beispiele zur Verwendung der C-Code-Funktionsblöcke werden an späterer Stelle im Kapitel *Anbindung an Ziel-Hardware* aufgezeigt.

Einstellungen zur Code-Generierung

Damit der zu generierende C-Code möglichst gut an die Hardware oder auch an die Eigenheiten des Compilers angepaßt werden kann, bietet BORIS verschiedene Optionen. Wir wollen uns zunächst mit den Parametern bezüglich der Generierung selbst beschäftigen. Anschließend werden wir uns mit der Auswahl einer I/O-Beschreibungsdatei, mit der direkten Ausführung einer Kommandozeile nach der Generierung und dem Speichern und Laden sämtlicher Einstellungen befassen. Zur Abrundung des Ganzen werden wir endlich einige kleine Beispiele, die Sie direkt nachvollziehen können, durchgehen.

Parameter der C-Code-Generierung

Nach der langen Durststrecke des vorangehenden Kapitels nun zu den ersten Einstellungen, die Sie bei der Generierung von C-Code tätigen können. Durch Auswahl des Menüpunktes CODE-GENERIEUNG | CODE-GENERIERUNGS-EINSTELLUNGEN... gelangen Sie in den nachfolgenden Dialog. Dieser bietet vor der eigentlichen Code-Generierung eine ganze Palette von Optionen, die den erzeugten C-Code beeinflussen. Wesentlich dabei sind vor allem die Abstimmung des Zahleformates auf die spätere Ziel-Hardware, sowie die Wahl der vom AutoCode-Generator automatisch zum C-Code hinzugefügten Rahmen-Funktion. Hier besteht neben einigen vordefinierten Funktionstypen auch die Möglichkeit, benutzereigene Funktionen aus einer I/O-Beschreibungsdatei einzubinden.

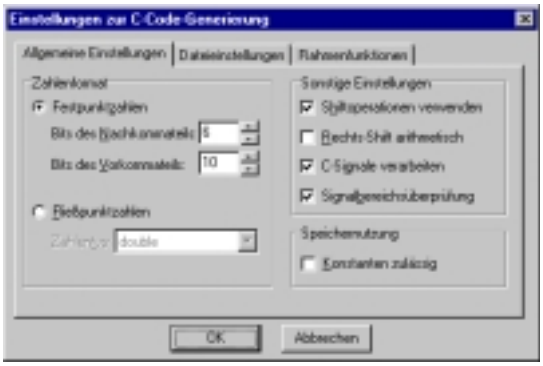

Bild 14. *Optionen des AutoCode-Generators, Palette 1*

Zahlenformat

Die hier möglichen Einstellungen sind ein wesentliches Merkmal für die Genauigkeit und die Geschwindigkeit des erzeugten C-Codes.

Zunächst wird die Unterscheidung getroffen, ob der erzeugte Code mit Fließ- oder Festpunktzahlen arbeiten soll. Bei Festpunktzahlen müssen Sie die Anzahl der Bits für Nach- und Vorkommastellen angeben. Die Anzahl der Vorkommastellen muß hierbei mindestens genau so groß sein wie die der Nachkommastellen. Je mehr Nachkommastellen Sie angeben, desto genauer werden die Ergebnisse des C-Codes sein. Beide Angaben dürfen den Wert 15 nicht überschreiten!
Bei Fließkommazahlen können Sie zusätzlich noch den zu verwendenden Datentyp einstellen (Vorsicht: Einige Compiler können nur *float*; diverse andere Compiler definieren *long double* als *double.* Im Zweifelsfall testen Sie dieses. Die ANSI C Headerdatei *float.h* Ihrers Compilers sollte Aufschluß hierüber geben!).

Ob der Code bei Fließ- oder bei Festpunktzahlen schneller arbeitet, hängt im wesentlichen von der Zielhardware und dem was der Compiler aus dem C-Code macht ab.

Sonstige Einstellungen

Bei diesen Einstellungen handelt es sich um hardware- bzw. compilerspezifische Einstellungen. Wenn Ihre Hardware z. B. eine 16-Bit-Schiebeoperation schneller durchführen kann als eine 16-Bit-Multiplikation bzw. 16-Bit-Division, sollten Sie die *Shiftoperationen verwenden*. Wird jedoch bei Ihrem Compiler das Rechtsschieben durch einen logischen anstatt eines arithmetischen Schiebebefehls dargestellt, so dürfen Sie auf keinen Fall *Rechts-Shift arithmetisch* auswählen. Dieses würde zu Fehlern bei den Ergebnissen führen, sobald diese Operation an einem negativen Signal oder Parameter durchgeführt würde.

Der nächste Punkt, *Signalnummern verarbeiten*, sollte gewählt werden, falls Sie das unter ANSI-C definierte Signal (*SIGFPE*), das die Meldungen *Floating point exception*, *Divison by zero* etc. auslöst, abfangen und in einer eigenen Funktion verarbeiten möchten (s. u.).

Letztlich bleibt noch der Auswahlpunkt *Signalbereichsüberprüfung*. Die Auswahl dieses Feldes bewirkt, daß an kritischen Stellen eine Überprüfung des Signalbereiches durchgeführt wird. Dadurch wird Ihr Code allerdings langsamer und umfangreicher. Wenn Sie sicher gehen können, daß das Signal in der von Ihnen gewählten Auflösung darstellbar bleibt (verantwortlich ist hierfür die Anzahl der Vorkommabits), sollte dieser Punkt nicht ausgewählt sein.

Speichernutzung

Die Speichernutzung kann so organisiert werden, daß möglichst viele Konstanten gebildet werden (ROM-fähig). Dies ist besonders für kleine Microcontroller mit wenig RAM interessant. Diese Option ist zur Zeit nur für die speicherintensiven Blöcke umgesetzt worden (Fuzzy-Controller, benutzerdefinierte Kennlinie und benutzerdefiniertes Kennfeld).

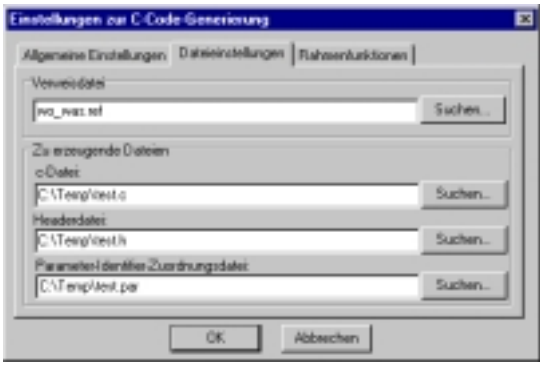

Bild 15. *Optionen des AutoCode-Generators, Palette 2*

Verweisdatei Innerhalb dieses Gruppenfeldes muß die Verweisdatei (s. o.) eingestellt werden.

Zu erzeugende Dateien Die zu erzeugenden Dateien werden hier eingetragen. Erzeugt werden dabei immer eine Headerdatei, eine C-Datei und eine Datei, die Parameterzuordnungen enthält.

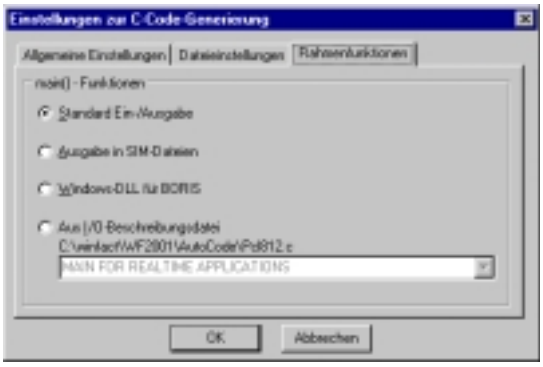

Bild 16. *Optionen des AutoCode-Generators, Palette 3*

main()-Funktionen

Wie schon erwähnt, lassen sich verschiedene main-Funktionen erzeugen (s. *Schreiben der main-Funktion*).

Bei *Standard-Ein-/Ausgabe* werden Ein- und Ausgaben zur Standard-Ein-/Ausgabe geleitet. Diese Option ist sinnvoll, wenn Sie Einzelschritte nachvollziehen oder Ihren Code debuggen möchten.

Ausgabe in SIM-Dateien leitet alle Ausgaben in SIM-Dateien um. Die Dateinamen erhalten eine fortlaufende Numerierung nach den ersten vier Zeichen des Namens der C-Quelldatei. Zur Feststellung, welcher Ausgang in welcher SIM-Datei steht, kann die Datei-Info zu Rate gezogen werden.

Die Auswahl *Windows-DLL (LibMain)* erzeugt beim Generieren des Quellcodes einen Quelltext, dessen Compilat einer Windows-DLL entspricht, die über die BORIS-User-DLL-Schnittstelle wieder eingebunden werden kann.

Wenn Sie den Punkt *Aus I/O-Beschreibungsdatei* anwählen, wird das darunter befindliche Auswahlfeld aktiviert. Wählen Sie hier nun einen entsprechenden Eintrag aus. Die Einträge entsprechen den Namen der Funktionscodeabschnitte in der I/O-Beschreibungsdatei (s. Abschnitt *Eigene Rahmen-Funktionen*).

Ist der Punkt *Signalnummern verarbeiten* angewählt, so fügt BORIS dem C-Code eine entsprechende Behandlungsroutine hinzu, die bei obigen Fehlern angesprungen wird. Der eingefügte Quelltext befindet sich in der Datei *common.c* innerhalb des Funktionscodeabschnittes */*All signal handlers*/* (vgl. Anhang A).

```
1   MEM_ATTRIBUTE void SIGFPEfkt(int sig)
2   {  if (sig!=SIGFPE) exit(1);
3      /*ignore further floating point exceptions*/
4      signal(SIGFPE,SIG_IGN);
```

```
5      /*Insert exception handling at this position*/
6      printf("FLOATING POINT EXCEPTION");
7      signal(SIGFPE,SIG_DFL); /*restore default handling*/
8   }
```

Listing 17. *Funktion zum Abfangen von Ausnahmefehlern*

Der Kommentar *'Insert exception handling at this position'* soll Sie darauf hinweisen, daß an dieser Stelle Ihre Ausnahme-Routine angesprungen werden sollte. Dazu muß in der `main`-Funktion die obige Funktion als signalverarbeitende Funktion eingerichtet werden. Dafür wird der Abschnitt */*All signals that are used*/* an entsprechender Stelle innerhalb der main-Funktion eingebunden. Er steht ebenfalls in der Quelldatei *common.c.*

```
signal(SIGFPE,SIGFPEfkt);
```

Listing 18. *Funktionsaufrufe zur Einrichtung von Signal-Handlingfunktionen (hier wird SIGFPEfkt eingerichtet)*

Signal-Handling-funktionen

In diesem Abschnitt wird keine Funktion spezifiziert, sondern es werden lediglich die im vorangehenden Listing stehenden Funktionen als Signalbearbeitungsfunktionen eingerichtet. Damit wird festgelegt, welche Funktion bei welchem Signal auftritt.

Ob Ihr Compiler die in ANSI-C festgelegten Signale unterstützt, entnehmen Sie bitte dem Handbuch des Compilers.

Die Erklärung der vierten Einstellung (*Aus I/O-Beschreibungsdatei*) erfolgte an früherer Stelle im Kapitel *Aufbau der I/O-Beschreibungsdatei*.

Wahl der I/O-Beschreibungsdatei

Die aktuelle I/O-Beschreibungsdatei kann über CODE-GENERIERUNG | I/O-BESCHREIBUNGSDATEI WÄHLEN... gewählt werden. Nach Auswahl dieses Menüpunktes erscheint folgender Dialog.

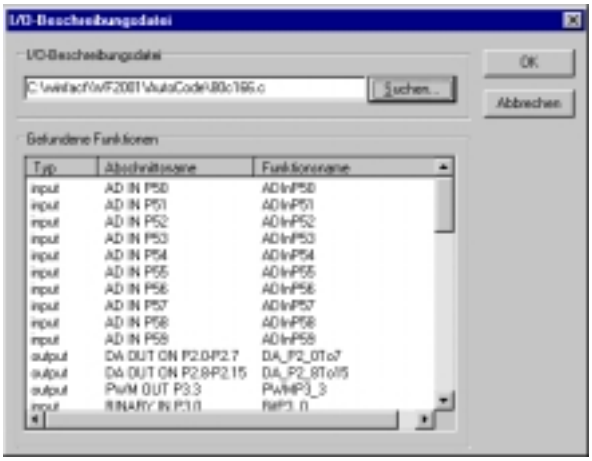

Bild 17. *Dialog zur Wahl der I/O-Beschreibungsdatei*

In diesem Dialog können Sie die gewünschte I/O-Beschreibungsdatei einstellen, indem Sie sie über *Suchen* auswählen. Die I/O-Beschreibungsdateien sollten die Endung *C* besitzen. Der Dialog zeigt Ihnen in der Liste an, welche Funktionscodeabschnitte die eingestellte Datei enthält, wie diese lauten, welchen Typ diese haben und wie die zugehörige C-Funktion dieses Abschnittes lautet.

Laden einer Parameter-Identifier-Spezifikation

Eine Parameter-Identifier-Spezifikation kann über CODE-GENERIERUNG | PARAMETERIDENTIFIER-SPEZIFIKATION LADEN... geladen werden.

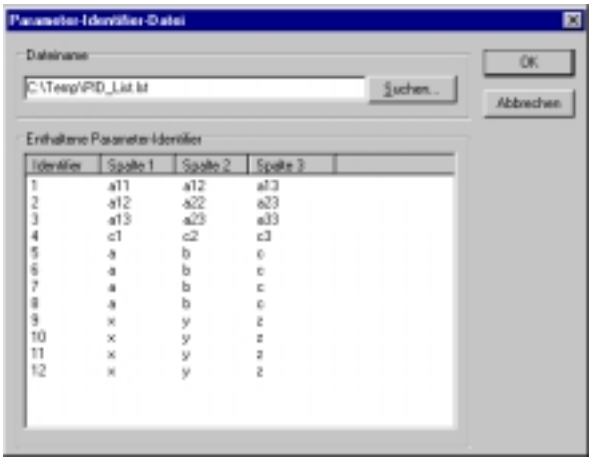

Bild 18. *Laden einer Parameter-Identifier-Datei*

Die Verwendung von Parameter-Identifiern wird eingehend im Kapitel *Parameter-Identifier* besprochen.

Kommandozeile nach der Quellcode-Generierung

BORIS bietet Ihnen die Möglichkeit, nach der Generierung des Quelltextes eine Kommandozeile, die beispielsweise den Compiler oder eine Batch-Datei aufruft, auszuführen. Unter dem Menüeintrag CODE-GENERIERUNG | KOMMANDOZEILE EINGEBEN können Sie im folgenden Dialog die gewünschte Kommandozeile eingeben:

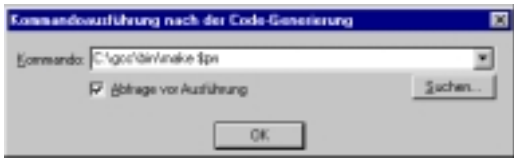

Bild 19. *Eingabe einer Kommandozeile*

In diesem Dialog werden die letzten zehn unterschiedlichen Einträge gespeichert. Die Verwendung der unter BORIS zulässigen Parameter *$pn* und *$n*, die bei Ausführung durch den Namen der zu generierenden Quelltextdatei mit (*$pn*) und ohne (*$n*) Pfadangabe (beide jedoch ohne Dateinamenextension) ersetzt werden, erlauben die Ausnutzung der gleichen Kommandozeile mit verschiedenen Quelltextdateien. Bei einigen wenigen Anwendungen kann es

vorteilhaft sein, die Abtastfrequenz oder die Abtastschrittweite zu übergeben. Dies kann durch die Parameter *$INVDeltaT* und *$DeltaT* (Groß-/Kleinschreibung wird nicht beachtet) geschehen. Eine Kommandozeile, die eine Batch-Datei (besser gesagt ein Shell-Script) ausführt, könnte also wie folgt beschrieben werden:

c:\GNU\WinFACT\make.bat $pn $deltat

Soll die Kommandozeile nur manchmal ausgeführt werden, so wählen Sie *Abfrage vor Ausführung* aus. Der Quellcode-Generator fragt nach der Generierung des Codes, ob die Kommandozeile nun ausgeführt werden soll. Hätten Sie im Dialog die obige Kommandozeile eingegeben, so würde die ausgewertete Kommandozeile und somit auch die Abfrage wie folgt aussehen:

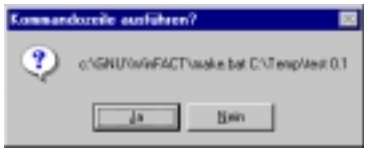

Bild 20. *Abfrage vor Ausführung der Kommandozeile*

Bei dieser Frage werden die Parameter *$pn* und *$deltat* entsprechend ersetzt, so daß Sie die Zeile so sehen können, wie sie letztendlich ausgeführt wird.

Enthält die Kommandozeile keinen Eintrag, so erfolgt auch keine Ausführung derselben und der obige Abfragedialog wird nicht angezeigt.

Konfigurierung speichern und laden

Sämtliche Einstellungen können gespeichert und wieder geladen werden (*Konfiguration laden...* bzw. *Konfiguration speichern...* unter dem Menüpunkt *Code-Generierung*). Beim Speichern werden

- die Einstellungen für die Code-Generierung,
- der Name der I/O-Beschreibungsdatei,
- der Name der Parameter-Identifier-Spezifikationsdatei und
- die letzten zehn Kommandozeilen

gespeichert. Beim Laden werden genau diese Informationen wieder hergestellt. Öffnen Sie jedoch eine BORIS-Systemdatei (Endung BSY), die C-Code Eingangs- und/oder C-Code-Ausgangsblöcke enthält, so wird, damit das System

unverändert bleibt, auch die entsprechende I/O-Beschreibungsdatei geladen. Kann diese nicht gefunden werden, so werden die Blöcke auf *Keine Funktion* gesetzt (s. *C-Code-Funktionsblöcke*). Nach Einstellen der richtigen I/O-Beschreibungsdatei ist alles wieder beim Alten.

Einfache Beispiele zur C-Code-Generierung

Die Ergebnisse der nachfolgenden Beispiele wurden durch einen gängigen ANSI C-Compiler für PCs (Borland C++) erstellt.

Signalgenerator

Dieses Beispiel erläutert Schritt für Schritt die C-Code-Generierung von BORIS. Dabei wird vom Generieren über das Compilieren, Linken und Ausführen der entstandenen Anwendung bis zum Vergleich der Simulationsergebnisse in der BORIS-Simulationsumgebung alles durchgegangen.

Erläuterung: Aus der nachfolgenden Struktur wird nur der Generator zu C-Code generiert. Der Generator soll dabei als Puls-Generator arbeiten und das untenstehende Signal (s. Bild 21) erzeugen. Die Abtastschrittweite ist auf 0.1 eingestellt und die Simulationsdauer umfaßt 100 Abtastschritte. Die Simulation endet somit zu dem Zeitpunkt $t = 10$.

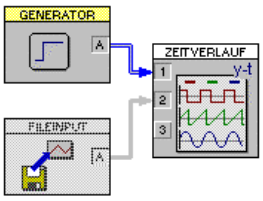

Bild 21. *Signalgenerator mit Blöcken, die den Vergleich mit der später erzeugten SIM-Datei ermöglichen*

Die Struktur nach Bild 21 liefert folgenden Signalverlauf:

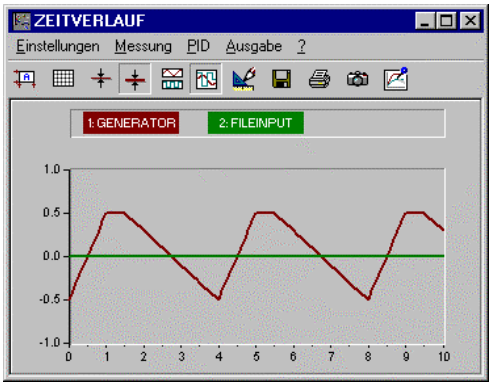

Bild 22. *Signalverlauf des Generators. Da der FILEINPUT-Block passiv ist, ist sein Ausgangssignal hier null*

Fließpunktzahlen

Hinweis: Nachfolgendes Beispiel befindet sich in den mitgelieferten Dateien BEISP_1.BSY bzw. BEISP_1.C im Unterverzeichnis *AutoCode\Examples*.

Es soll zunächst der C-Code des Generators für Fließpunktzahlen erzeugt werden. Dazu wird **nur** der Generator markiert (der FILEINPUT-Block muß, falls er keinen korrekten Dateinamen enthält, passiv gesetzt werden). Es sollen folgende Einstellungen (unter *Code*-GENERIERUNG | CODE-GENERIERUNGS-EINSTELLUNGEN...) getätigt werden:

Parameter:	**Wert:**
Zahlenformat	Fließpunktzahlen (float)
Rahmen-Funktion	Ausgabe in SIM-Dateien
Sonstige Parameter	-

Starten Sie nun die Code-Generierung durch den Menüeintrag CODE-GENERIERUNG | CODE-GENERIEREN. Compilieren Sie den C-Code mit einem C-Compiler für den PC, führen Sie die entstandene Anwendung aus und simulieren Sie bis zum 100-sten Simulationsschritt.

Bild 23. *Abfrage nach der Anzahl der Simulationsschritte unter DOS*

Tragen Sie nun die vom C-Code erzeugte SIM-Datei in den FILEINPUT-Block ein und starten Sie die Simulation unter BORIS (der FILEINPUT-Block muß nun wieder aktiv sein!). Die Simulation sollte Ihnen das in Bild 24 dargestellte Ergebnis liefern.

Führt man nun einen konkreten Vergleich durch, indem man die beiden zu vergleichenden Größen voneinander subtrahiert, so wird man feststellen, daß geringfügige Abweichungen vorhanden sind. Dieser Effekt liegt darin begründet, daß in BORIS mit 80-Bit-Auflösung gerechnet wird, während die Berechnungen des C-Codes in 32 Bit durchgeführt werden. Der Fehler ist aber vernachlässigbar klein (Bild 25).

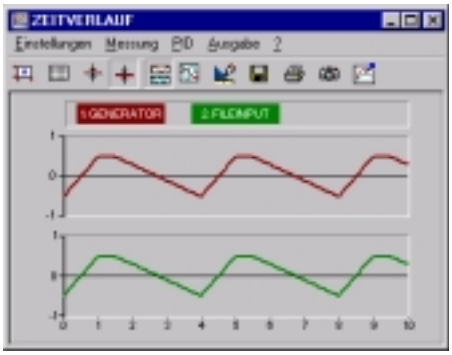

Bild 24. *Ergebnis des Simulationslaufes unter BORIS nach dem Simulationslauf des generierten C-Codes*

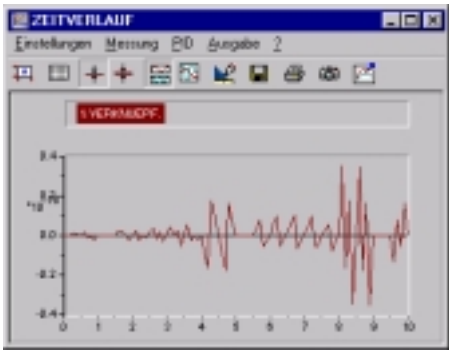

Bild 25. *Fehler bei mit Fließpunktarithmetik (32-Bit) arbeitendem C-Code gegenüber der BORIS-Umgebung*

Festpunktzahlen

Bei der Generierung desselben Systems unter Verwendung von Festkommazahlen tritt ein wesentlich größerer Fehler auf. Tätigen Sie folgende Einstellungen zur Erzeugung des C-Codes:

1. Fall

Parameter:	**Wert:**
Zahlenformat	15 Nachkommabits und 15 Vorkommabits
Rahmen-Funktion	Ausgabe in SIM-Dateien
Sonstige Parameter	Shiftoperation verwenden

Nachfolgende Grafik zeigt das Ergebnis.

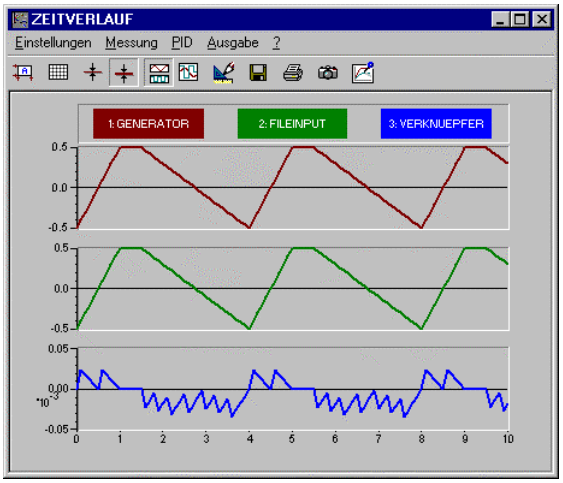

Bild 26. *Vergleich bei 15 Nach- und 15 Vorkommabits*

2. Fall

Parameter:	**Wert:**
Zahlenformat	8 Nachkommabits und 8 Vorkommabits
Rahmen-Funktion	Ausgabe in SIM-Dateien
Sonstige Parameter	Shiftoperation verwenden, Rechts-Shift arithmetisch (compilerabhängig)

Nachfolgende Grafik zeigt das Ergebnis.

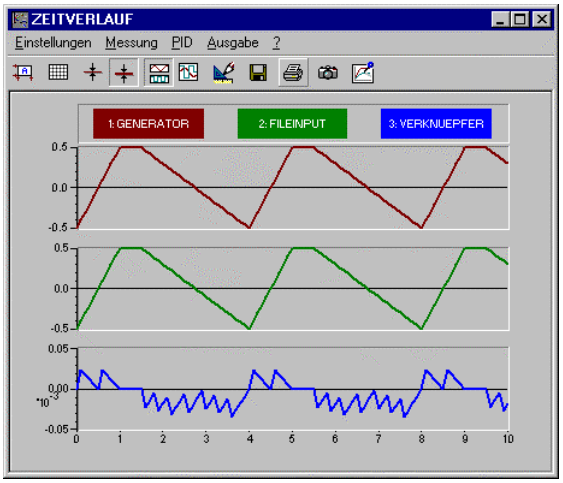

Bild 27. *Vergleich bei 8 Nach- und 8 Vorkommabits*

Als Endergebnis kann man festhalten, daß der C-Code dieses Beispiels in den Grenzen des Bitfehlers

$$bf = \frac{1}{2^l} \qquad (bf\text{: Bitfehler}, \quad l\text{: Anzahl der Nachkommabits})$$

betrachtet gleiche Ergebnisse erzielt. Problematisch wird es, wenn sich derartige Fehler (zu stark) aufsummieren. Darauf werden wir im nächsten Beispiel näher eingehen.

Einfacher Integrierer

Zur Übung soll das nachfolgend dargestellte System aufgebaut werden.

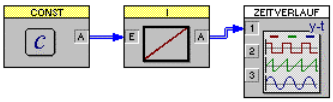

Bild 28. *Einfache Integration eines konstanten Werts von 1.*

Der konstante Wert 1 soll über die Simulationszeit von 0 bis 1 integriert werden. Die Abtastschrittweite des Systems beträgt 0.1. Auch hier wird der direkte Vergleich des Resultats des C-Codes mit dem aus der BORIS-Simulation her-

angezogen, um Aussagen über die Genauigkeit des C-Codes zu treffen. **Lassen Sie zunächst alle Parameter der Blöcke unverändert**.

Fließpunktzahlen

Zunächst wollen wir unser obiges System wieder bei Fließpunktzahlen untersuchen. Dazu sind folgende Einstellungen für die C-Code-Generierung erforderlich.

Parameter:	**Wert:**
Zahlenformat	Fließpunktzahlen (float)
Rahmen-Funktion	Standard-Ein-/Ausgabe
Sonstige Parameter	-

Erzeugen Sie nun die ausführbare Datei aus dem C-Code und starten diese Anwendung. Bei der Abfrage des letzten Simulationsschrittes (*last simulation step (>1)*) geben Sie 10 ein. Somit wird bis zum Zeitpunkt $t = 1$ simuliert. Das nachfolgende Bild zeigt das Resultat des Simulationslaufes.

```
last simulation step (>1): =10
t=     0  output: I = 0
t=   0.1  output: I = 0.1
t=   0.2  output: I = 0.2
t=   0.3  output: I = 0.3
t=   0.4  output: I = 0.4
t=   0.5  output: I = 0.5
t=   0.6  output: I = 0.6
t=   0.7  output: I = 0.7
t=   0.8  output: I = 0.8
t=   0.9  output: I = 0.9
t=     1  output: I = 1
```

Bild 29. *Simulationsergebnis des C-Codes*

Dieses Ergebnis entspricht unseren Erwartungen. Damit es aber nun etwas interessanter wird, generieren wir den C-Code desselben Systems mit Festpunktzahlen.

Festpunktzahlen

Für Untersuchungen des obigen Systems bei Festpunktzahlen belassen Sie das System aus Bild 28 immer noch im "Urzustand". Generieren Sie den C-Code mit folgenden Einstellungen:

Parameter:	**Wert:**
Zahlenformat	7 Nach-, 9 Vorkommabits
Rahmen-Funktion	Standard-Ein-/Ausgabe
Sonstige Parameter	Shiftoperation verwenden Rechts-Shift arithmetisch (compilerabhänging!)

Starten Sie die C-Code-Generierung, so wirft der AutoCode-Generator zunächst einmal eine Fehlermeldung aus:

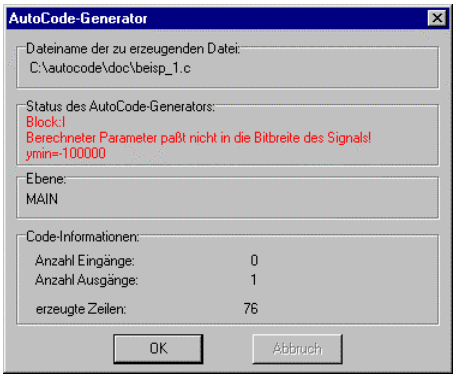

Bild 30. *Fehlermeldung des AutoCode-Generators*

Diese Meldung beruht darauf, daß sich die untere bzw. obere Grenze (y_{min} bzw. y_{max}) des Integrierers, die voreingestellt bei ±100000 liegt, nicht im augenblicklichen Zahlenformat darstellen lassen. Zusätzlich zu diesem Hinweis wird der entsprechende Block mit Namen *I* selektiert, während alle anderen nicht selektiert werden. So lassen sich schnell und gezielt bei dem gewählten Zahlenformat nicht darstellbare Parameter korrigieren (s. *Anhang C*).

Setzen wir den Wert *ymin*, der die untere Grenze des Integrierers angibt, auf -10 und - damit nicht noch eine Fehlermeldung auftritt - *ymax*, die obere Grenze des Integrierers, auf +10. Danach kann der C-Code ohne Probleme generiert werden. Die nachfolgende Ausgabe, die bei Ausführung des Compilats ensteht, verdeutlicht das Ergebnis:

```
last simulation step (>1): =10
t=0.00000  output: I =0.000000
t=0.10000  output: I =0.093750
t=0.20000  output: I =0.187500
t=0.30000  output: I =0.281250
t=0.40000  output: I =0.375000
t=0.50000  output: I =0.468750
t=0.60000  output: I =0.562500
t=0.70000  output: I =0.656250
t=0.80000  output: I =0.750000
t=0.90000  output: I =0.843750
t=1.00000  output: I =0.937500
```

Bild 31. *Ergebnis bei einem Zahlenformat von 7 Nach- und 9 Vorkommabits*

Deutlich ist hier die Aufsummation des Fehlers zu sehen. Der Quantisierungsfehler (=Bitfehler) ist hier

$$bf \le \frac{1}{2^7} = 0.0078125 .$$

Nimmt man den exakten Wert zum Zeitpunkt t = 0.1 und subtrahiert den aus Bild 31 zum Zeitpunkt t = 0.1, so erhält man den hier innerhalb der Berechnung auftretenden Quantisierungsfehler:

$$bf = 0.1 - 0.09375 = 0.00625$$

Wir wollen den entstandenen Fehler analytisch nachvollziehen. Der Integrierer arbeitet nach der Beziehung

$$y_k = y_{k-1} + \frac{\Delta T}{Ti} \cdot x_{k-1}$$

mit

k	Nummer des Abtastschrittes
x	Eingangssignal
y	Ausgangssignal
ΔT	Abtastschrittweite
Ti	Integrierzeit

Dabei wird der Quotient $\Delta T/Ti$ nur innerhalb des eingestellten Zahlenformats berechnet. Für unser Beispiel ist $Ti = 1, \Delta T = 0.1$, der Bitfehler bf beträgt bei sieben Nachkommastellen $bf = 2^{-8} = 1/128$. Statt mit dem exakten Wert von 0.1 wird der Quotient $\Delta T / Ti$ im C-Code also dargestellt durch den Wert

$$\left(\frac{\Delta T}{Ti}\right)^{*} = \text{int}\left(\frac{\Delta T}{Ti \cdot bf}\right) bf = \text{int}(0.1 \cdot 128)\frac{1}{128} = \frac{12}{128} = 0.09375.$$

Somit erhält man bei einer konstanten Eingangsgröße von $x_k = 1,\ k = 0,1,\ldots$ für den vom C-Code ermittelten Ausgangswert zum Zeitpunkt $t = 1$

$$y(t=1) = y_{10} = \sum_{k=1}^{10}\left(\frac{\Delta T}{Ti}\right)^{*} x_k = 10 \cdot 1 \cdot \left(\frac{\Delta T}{Ti}\right)^{*} = 0.9375.$$

Dies ist gerade der in Bild 31 ermittelte Wert.

Freidefinierte periodische Signalerzeugung

Nachfolgendes Beispiel befindet sich in den Dateien BEISP_2.BSY bzw. BEISP_2.C. **Achtung**: Dieses Beispiel ist mit der Lite-Edition des AutoCode-Generators **nicht** compilierbar!

Die in Bild 32 dargestellte Struktur dient zur Erzeugung eines periodischen Signals. Die selektierten Blöcke sollen in C-Code überführt werden.

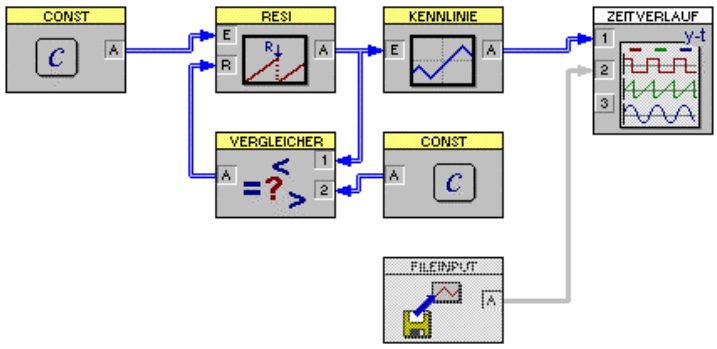

Bild 32. *Freidefinierte periodische Signalerzeugung*

Fließpunktzahlen

Stellen Sie zur Erzeugung des C-Codes folgende Parameter ein:

Parameter:	**Wert:**
Zahlenformat	Fließpunktzahlen (float)
Rahmen-Funktion	Ausgabe in SIM-Dateien
Sonstige Parameter	-

Ist der C-Code generiert, so muß aus diesem zunächst, falls Sie diesen Schritt nicht mit Hilfe der Kommandozeile automatisiert haben, eine ausführbare Datei mit Hilfe Ihres Compilers und Linkers produziert werden. Nach dem Starten lassen Sie die Simulation bis zum 4000-sten Abtastschritt durchführen.

Ergebnis:

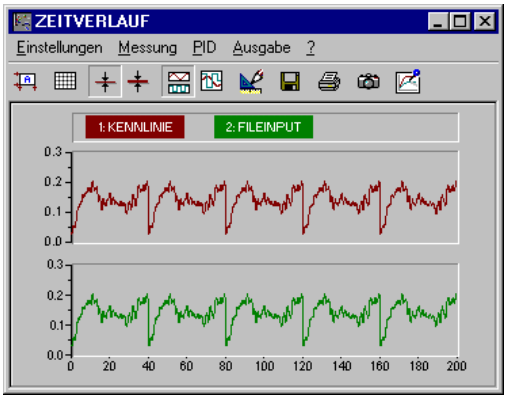

Bild 33. *C-Code und BORIS-Simulation im Vergleich*

Wie unschwer zu erkennen ist, ist das Signal des C-Codes mit dem unter BORIS erzeugten identisch.

Im weiteren Verlauf wird davon ausgegangen, daß Sie den Ablauf zum Vergleich von Ergebnissen des C-Codes mit den von BORIS gelieferten beherrschen. Ebenso wird vorausgesetzt, daß die Anzahl der durchzuführenden Abtastschritte im C-Code mindestens gleich der Anzahl der Abtastschritte unter BORIS ist. Desweiteren wird nicht mehr erwähnt, daß Sie den FILEINPUT-Block auf die richtige, durch den Ablauf des C-Codes erzeugte SIM-Datei, die für den momentanen Vergleich relevant ist, einzustellen haben.

Festpunktzahlen

Wird das obige System mit Festpunktzahlen berechnet, werden Sie feststellen, daß dort das Signal umso "gestreckter" aussieht, je weniger Nachkommabits Sie der C-Code-Generierung zur Verfügung stellen. Wählen Sie dazu folgende Parameter:

Parameter:	**Wert:**
Zahlenformat	7 Nachkommabits, 9 Vorkommabits
Rahmen-Funktion	Ausgabe in SIM-Dateien
Sonstige Parameter	Shiftoperation verwenden Rechts-Shift arithmetisch (compilerabhängig)

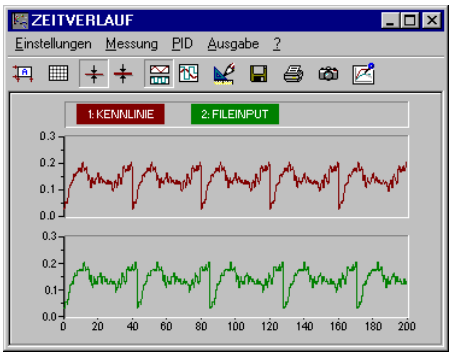

Bild 34. *Streckung des vom C-Code generierten Signals gegenüber dem tatsächlichen Verlauf*

Die in Bild 34 zu erkennende Streckung des Signals hat als Ursache die Aufsummation des Quantisierungsfehlers im rücksetzbaren I-Glied, das hier als Zeitgeber für die Kennlinie anzusehen ist.

Genau betrachtet sind in diesem System bei Festpunktzahlen zwei Fehlerarten festzustellen:

1. Ein Quantisierungsfehler, der bei der statischen Kennlinie auftritt (dieser Fehler tritt bei allen statischen Gliedern auf).

2. Ein durch den Quantisierungsfehler im dynamischen rücksetzbaren Integrierer verursachter Zeitfehler (dieser Fehler tritt bei allen dynamischen Gliedern auf).

Beide Fehler können nur durch die Erhöhung der Anzahl der Nachkommabits minimiert werden (Anzahl Vorkommabits muß mindestens genauso groß sein!).

Integrationsverfahren

Darstellung dynamischer Glieder

Alle dynamischen Glieder werden in BORIS intern in Form eines Systems von *n* Differentialgleichungen 1. Ordnung der Gestalt

$$\begin{aligned} \dot{\underline{x}} &= \underline{f}(\underline{x}, \underline{u}) \\ y &= g(\underline{x}, \underline{u}) \end{aligned} \qquad \underline{x}(t=0) = \underline{x}_0$$

dargestellt (*Zustandsraumdarstellung*). Dabei ist *n* die Ordnung des Systems, $\underline{u}$ der Vektor mit den Eingangsgrößen, $\underline{x}$ der Zustandsvektor und y die Ausgangsgröße. Mit Ausnahme des DGLSYS-Blocks sind alle dynamischen Blökke linear und besitzen nur eine Eingangsgröße. In diesem Fall lautet das Dgl.-System speziell:

$$\begin{aligned} \dot{\underline{x}} &= \underline{A}\underline{x} + \underline{b}u \\ y &= \underline{c}^{\mathrm{T}}\,\underline{x} + d\,u \end{aligned} \qquad \underline{x}(t=0) = \underline{x}_0$$

mit

$\underline{A}$	Systemmatrix
$\underline{b}$	Eingangsvektor

$\underline{c}^T$	transponierter Ausgangsvektor
d	Durchgriff

In der Regel liegen die linearen dynamischen Glieder zunächst als gebrochen rationale Übertragungsfunktion in der Form

$$G(s) = \frac{p_0 + p_1 s + p_2 s^2 + \cdots + p_n s^n}{q_0 + q_1 s + q_2 s^2 + \ldots + q_n s^n} \qquad \left| q_n = 1 \right.$$

vor. Diese Übertragungsfunktion kann überführt werden in ein äquivalentes Zustandsraummodell in Steuerungsnormalform gemäß

$$\dot{\underline{x}} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -q_0 & -q_1 & -q_2 & \cdots & -q_{n-1} \end{bmatrix} \underline{x} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} u$$

$$y = (p_0 - p_n q_0, p_1 - p_n q_1, \ldots, p_{n-1} - p_n q_{n-1}) + p_n u$$

Für die interne Darstellung dieser Glieder brauchen also nur die letzte Zeile der Systemmatrix sowie der Ausgangsvektor $\underline{c}$ und der Durchgriff d abgespeichert zu werden. Dynamische Blöcke, die Export-Parameter aufweisen, erzeugen diese Darstellung innerhalb des C-Codes.

Integration nach Euler

Beim Integrationsverfahren *Euler* wird das Dgl.-System durch Rechteckintegration angenähert. Die zugehörige Iterationsvorschrift lautet allgemein

$$\underline{x}_{k+1} = \underline{x}_k + \Delta T \, \underline{f}(\underline{x}_k, u_k)$$
$$y_{k+1} = g(\underline{x}_k, u_k)$$

wobei der Index k den k-ten Iterationsschritt - beginnend zum Zeitpunkt $k = 0$ - bezeichnet und ΔT die Simulationsschrittweite (Abtastzeit). Für die linearen Glieder ergibt sich daraus die spezielle Vorschrift

$$\underline{x}_{k+1} = \underline{x}_k + \Delta T \left(\underline{\underline{A}} \underline{x}_k + \underline{b} u_k \right)$$
$$y_{k+1} = \underline{c}^{\mathrm{T}} \underline{x}_{k+1} + d u_{k+1}$$

Integration nach Runge-Kutta

Beim Runge-Kutta-Verfahren 4. Ordnung werden zur Bestimmung des Systemzustands zum Zeitpunkt t_{k+1} zunächst vier Stützwerte $\underline{k}_1 \ldots \underline{k}_4$ berechnet. Die Berechnungsvorschriften dafür lauten beim Originalverfahren

$$\underline{k}_1 = \underline{f}(\underline{x}_k, \underline{u}_k)$$
$$\underline{k}_2 = \underline{f}(\underline{x}_k + \frac{\Delta T}{2}\underline{k}_1, \underline{u}_{k+1/2})$$
$$\underline{k}_3 = \underline{f}(\underline{x}_k + \frac{\Delta T}{2}\underline{k}_2, \underline{u}_{k+1/2})$$
$$\underline{k}_4 = \underline{f}(\underline{x}_k + \Delta T\underline{k}_3, \underline{u}_{k+1})$$

Darin sind $\underline{u}_{k+1/2}$ bzw. $\underline{u}_{k+1}$ die Eingangsgrößen des Systems zu den Zeitpunkten $t_{k+1/2} = t_k + \Delta T / 2$ bzw. $t_{k+1} = t_k + \Delta T$. Da diese jedoch zum Zeitpunkt t_k noch nicht bekannt sind, werden sie durch Extrapolation aus den Werten zu den Zeitpunkten t_k und t_{k-1} gemäß

$$\underline{u}_{k+1/2} \approx \underline{u}_k + \frac{1}{2}\left(\underline{u}_k - \underline{u}_{k-1}\right)$$
$$\underline{u}_{k+1} \approx \underline{u}_k + \left(\underline{u}_k - \underline{u}_{k-1}\right)$$

ermittelt.

Aus den auf diese Weise berechneten Stützwerten ergibt sich dann der neue Zustandsvektor zu

$$\underline{x}_{k+1} = \underline{x}_k + \frac{\Delta T}{6}\left(\underline{k}_1 + 2\underline{k}_2 + 2\underline{k}_3 + \underline{k}_4\right).$$

Die Berechnung der aktuellen Ausgangsgröße erfolgt dann wie gehabt.

Für die linearen Glieder ergibt sich die zugehörige Iterationsvorschrift wiederum, indem die allgemeinen Funktionsterme durch die linearen Terme gemäß

$$\underline{f}(\underline{x}, \underline{u}) \rightarrow \underline{\underline{A}}\,\underline{x} + \underline{b}u$$
$$g(\underline{x}, \underline{u}) \rightarrow \underline{c}^{\mathrm{T}}\,\underline{x} + du$$

ersetzt werden.

Integration nach dem Matrizenexponentialverfahren

Da das in BORIS realisierte Matrizenexponentialverfahren bezüglich Speicherplatzbedarf und Ausführungsgeschwindigkeit sehr ungünstige Eigenschaften aufweist, ist eine Übertragung auf die meisten Typen von Ziel-Hardware nicht sinnvoll. Daher wird im C-Code bei Anwahl des Matrizenexponentialverfahrens innerhalb der BORIS-Simulationsoberfläche stattdessen das Euler-Integrationsverfahren *mit einem balancierten Zustandsraumsystem* realisiert.

Balanciertes System: Ein balanciertes System ist eine durch Ähnlichkeitstransformation des Zustandsraummodells gewonnene Systemdarstellung, die möglichst gleiche Spalten- und Zeilensummen der Systemmatrix aufweist. Die inneren Zustandsgrößen ändern sich dabei, das Ein-/Ausgangsverhalten des Systems bleibt jedoch gleich.

Dieses Verfahren besitzt gegenüber dem auf der Steuerungsnormalform basierenden Euler-Verfahren günstigere numerische Eigenschaften bei Systemen mit stark unterschiedliche Koeffizienten in der Übertragungsfunktion. Der Speicherplatzbedarf ist jedoch größer, da für dieses Verfahren die komplette *A-Matrix*, der *b-Vektor*, der transponierte *c-Vektor* und der Durchgriff *d* als Parameter des dynamischen Blocks übergeben werden müssen. Dynamische Blöcke, die Export-Paramerter aufweisen, verzichten weitestgehend auf eine Balancierung des Systems.

Eigene Integrationsverfahren

Eigene Integrationsverfahren können nur dann dazugebunden werden, wenn Sie einen Verweis auf ein Verfahren in der Verweisdatei ändern. Sie sollten immer nur das Verfahren für die Einstellung *Matrizenexponential* verändern, da dieses die einzige Einstellung ist, bei der im C-Code die kompletten Systemmatrizen bzw. -vektoren verwendet werden.

Zur Implementierung eines eigenen Integrationsverfahrens ändern Sie einfach die *Quelldatei*-Angabe in dem entsprechenden Verweis in der eingestellten Verweisdatei:

vorher:

*/*IntegrationMethod*/* ***dynamic.c*** */*Balanced system integration*/*

nachher:

*/*IntegrationMethod*/* ***I_method.c*** */*Balanced system integration*/*

Anschließend kopieren Sie den kompletten Funktionscodeabschnitt */*Balanced system integration*/* aus der Datei *dynamic.c* in die neue Datei *I_method.c*. Sie können nun den Inhalt der Funktion beliebig ändern. **Ändern Sie auf keinen Fall den Funktionskopf!** Dies dürfte erstens nicht erforderlich sein und zweitens kommt es dadurch, daß die Aufrufe innerhalb der Funktionen der dynamischen Glieder nicht mehr stimmen, zu einer Unmenge an Fehlern. Zur Sicherheit ist dieser Teil hier noch einmal dargestellt worden:

```
1   MEM_ATTRIBUTE SVartyp IntegrationMethod (
2               const int n,
3               const SVartyp *const A,
4               const SVartyp *const B,
5               const SVartyp *const CT,
6               const SVartyp *const d,
7               const SVartyp *const e,
8               SVartyp *const X)
```

Darin ist:

n	Die Ordnung des Systems (Anzahl der Elemente der Vektoren*)*
A	Ein Zeiger auf eine balancierte Systemmatrix
B	Ein Zeiger auf den Eingangsvektor
CT	Ein Zeiger auf den transponierten Ausgangsvektor
d	Ein Zeiger auf den Durchgriff
e	Ein Zeiger auf den Eingangswert
X	Ein Zeiger auf den veränderbaren Zustandsvektor

Statt die Quelldatei zu ändern, können Sie natürlich auch den Namen des Funktionscodeabschnittes ändern. Dazu müssen Sie in der Datei *dynamic.c* einen entsprechenden Funktionscodeabschnitt einfügen, der obige Funktion enthält.

Festpunktzahlen und ihre Arithmetik

In diesem Kapitel wollen wir auf die Besonderheiten der Festpunktarithmetik eingehen. Sobald Sie einen kleinen Mikrocontroller mit dem von BORIS generierten C-Code "füttern" möchten, ist es sinnvoll, Festpunktzahlen einzusetzen. Diese sind meist genau genug und benötigen erheblich weniger Rechenzeit, falls Ihre Ziel-Hardware keinen Fließkommaprozessor oder einen anderen "CPU-Entlaster" besitzt.

Zahlendarstellung

Signale und die nicht diskreten Parameter eines Blockes erhalten alle den Typ `SVartyp.` Dieser Typ wird bei der C-Quellcode-Generierung entsprechend der von Ihnen eingestellten Auflösung für Signalparameter bestimmt. Ist die Auflösung (Vorkommastellen + Nachkommastellen) größer als 16 Bit, so wird der Datentyp `long` verwendet, sonst der Typ `int`. Zur Darstellung von Werten dieses Typs sei auf das folgende Beispiel verwiesen.

Beispiel:

Das Signal mit dem Wert x=8.52 soll dargestellt werden. Eingestellt worden sind bei der C-Code-Generierung 6 Nachkommabits und 10 Vorkommabits.

Die zugehörige Zahl X im Binärformat sieht wie folgt aus:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wert:	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1
FracMask	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
IntMask	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
One	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Die Berechnung ist trivial: $X = \text{round}\left(8.52 \cdot 2^6\right)$

Die Anzahl der Nachkommastellen wird im Quelltext durch `FracPart,` die Anzahl der Vorkommastellen durch `IntPart` ausgedrückt. Somit kann die Zahl wie folgt zurückgerechnet werden:

$$x \approx \frac{X}{2^{\text{FracPart}}}$$

Dabei kann die Division in C gut durch den Shiftoperator ausgedrückt werden.

Zu guter Letzt noch die Darstellung einer negativen Zahl wie z. B. x = -3.152. Diese soll nun bei 8 Vor- und 8 Nachkommabits dargestellt werden (auch als Hexadezimalzahl). Man erhält

$$X = \text{round}\left(-3.152 \cdot 2^{8}\right) = -807 = 0\text{xfcd9}\,.$$

Addition (Subtraktion) von Festpunktzahlen

Die Addition zweier Signale (*a*,*b*) ist denkbar einfach und kann mit Hilfe des binären + Operators im C-Code geschehen.

```
c = a+b;
```

Multiplikation von Festpunktzahlen

Für die Multiplikation zweier Festpunktzahlen gibt es eine vorgefertigte Funktion. Diese ist folgendermaßen deklariert:

```
SVartyp MUL(SVartyp a, unsigned char fa, SVartyp b);
```

Die erste Festpunktzahl wird bei dieser Funktion mit der Anzahl der Nachkommastellen *fa* angegeben. Sollen zwei Signale miteinander multipliziert werden, so kann dies mit Hilfe der Konstanten *FracPart* als *fa* geschehen:

```
c = MUL(b,FracPart,b)
```

In diesem Fall würde in c der Wert von b^2 abgelegt. Der Vorteil des zweiten Parameters ist, daß Sie ganz einfach

```
c = MUL(5,0,b)
```

schreiben können, um b mit 5 zu multiplizieren. Auch richtig, aber unübersichtlicher und für Fließpunktzahlen ungeeignet (da FracPart nicht definiert wird!) wäre:

```
c = MUL(5<<FracPart,FracPart,b)
```

Division von Festpunktzahlen

Ebenso wie es für die Multiplikation von Festpunktzahlen eine vorgefertigte Funktion gibt, gibt es diese auch für die Division von Festpunktzahlen:

```
SVartyp DIV(SVartyp a, unsigned char fa, SVartyp b)
```

Da diese Funktion die gleichen Parameter wie die für die Multiplikation hat, brauchen wir hier nicht näher darauf einzugehen. Die Funktion selbst unterliegt den gleichen Bedingungen und wird auf gleiche Art und Weise erstellt wie die Multiplikation.

Änderung von Multiplikation und Division

Die Funktionen für die Multiplikation und Division von Festpunktzahlen sind in Funktionscodeabschnitten der Datei *Common.c* abgelgt. In Abhängigkeit von den eingestellten Parametern beim Generieren von C-Code wird ein entsprechender Abschnitt eingefügt.

Stellen Sie fest, daß Ihr Compiler nicht unbedingt den schnellsten Code für diese Funktion erstellt, so macht es durchaus Sinn, diese Funktionen als Inline-Funktionen bzw. ganz in Assembler zu schreiben. Dazu sollten Sie den alten Funktionscodeabschnitt kopieren und diese Kopie bearbeiten. Anschließend müssen Sie den Namen dieses modifizierten Funktionscodeabschnittes sowie den Verweis auf den Funktionscodeabschnitt innerhalb der Verweisdatei ändern.

Anpassung von Blöcken an Benutzerwünsche

In diesem Kapitel sollen anhand des PID-Reglers benutzereigene Anpassungen des Funktionscodeabschnittes aufgeführt werden. Wir werden zunächst die Abänderung eines Funktionscodes bei Fließpunktzahlen und anschließend eine Modifikation bei Festpunktzahlen vornehmen. Zu guter Letzt wollen wir einen

Funktionscodeabschnitt für den PID-Regler entwerfen, der sowohl bei Festpunkt- als auch bei Fließpunktzahlen Gültigkeit hat.

Modifikation bei Fließpunktzahlen

In der ausgelieferten Version arbeitet der PID-Regler bei Fließpunktzahlen exakt so wie derjenige unter BORIS. Da die Verzögerung des D-Anteils approximiert werden kann gemäß

$$e^{-\frac{\Delta T}{T_{Vz}}} \approx \frac{1}{\left(\frac{\Delta T}{T_{Vz}}\right)^2 + \frac{\Delta T}{T_{Vz}} + 1}$$

läßt sich durch die Einsparung der Exponentialfunktion Rechenzeit sparen. Hier soll gezeigt werden, wie der neue Funktionscode integriert werden kann. Dazu wollen wir, um später auf den ursprünglichen Stand zurückgreifen zu können, eine Kopie des Funktionscodeabschnitts des PID-Reglers modifizieren. Laden Sie dazu die mitgelieferte Datei PID.C aus dem Unterverzeichnis *\AutoCode*; diese enthält die notwendigen Änderungen als Kommentar und braucht daher nur geringfügig geändert zu werden.

Der Funktionscodeabschnitt des PID-Reglers sieht folgendermaßen aus:

```
1   /*PID-controller (floatingpoint)*/
2   #include <math.h>
3
4   void PIDT1_SetParam(PIDT1Struct *p, int id, SVartyp value)
5   {
6     switch (id){
7     case 1 :
8       if (value!=p->Kr){
9         p->Kr=value;
10        p->calcParams=1;
11      }
12      break;
13    case 2 :
14      if ( ((value>0) ? value : 1.0E-10) != p->Tn){
15        p->Tn=(value>0)? value : 1.0E-10;
16        p->calcParams=1;
17      }
```

```
18      break;
19    case 3 :
20      if (value!=p->Tv){
21        p->Tv=value;
22        p->calcParams=1;
23      }
24      break;
25    }
26  }
27
28
29  SVartyp PIDT1_GetParam(PIDT1Struct *p, int id)
30  {
31    switch (id){
32    case 1 : return MUL(p->Kr,FracPart,One);
33    case 2 : return MUL(p->Tn,FracPart,One);
34    case 3 : return MUL(p->Tv,FracPart,One);
35    }
36  }
37
38  void PIDT1_CalcInnerParams(PIDT1Struct *p)
39  {
40    if (!p->calcParams) return;
41    if (p->IOn) p->Ki = p->Kr/p->Tn/#InsertINVDELTAT;
42    if (p->DOn) {
43      p->alpha=exp(- p->INVTvz/#InsertINVDELTAT);
44      /*
45        p->alpha=p->INVTvz/#InsertINVDELTAT;
46        p->alpha=1/(zw*zw+zw+1)
47      */
48      p->Kd = p->Kr * p->Tv * #InsertINVDELTAT;
49      p->Kd = p->Kd * (One-p->alpha);
50    }
51    p->calcParams=0;
52  }
53
54
55  MEM_ATTRIBUTE SVartyp PIDT1_init(PIDT1Struct *p, SVartyp *e)
56  {
57    SVartyp zw;
```

```
58    p->calcParams=1;
59    PIDT1_CalcInnerParams(p);
60    if (!p->POn) p->yP = 0; else p->yP=p->Kr * *e;
61    if (!p->IOn) p->yI = 0; else p->yI=p->y0;
62    if (!p->DOn) p->yD = 0; else p->yD=p->Kd * *e;
63    if (p->LimOn){
64      zw = p->yP + p->yI;
65      if (zw<p->ymin) zw=p->ymin;
66      if (zw>p->ymax) zw=p->ymax;
67      if (p->AW==1) {
68        if (p->yI<p->ymin) p->yI=p->ymin;
69        if (p->yI>p->ymax) p->yI=p->ymax;
70      }else
71        p->yI = zw - p->yP;
72      zw+=p->yD;
73      if (zw<p->ymin) zw=p->ymin;
74      if (zw>p->ymax) zw=p->ymax;
75    }else
76      zw = p->yP + p->yI + p->yD;
77    p->X1=*e;
78    return zw;
79  }
80
81  MEM_ATTRIBUTE SVartyp PIDT1_fct(PIDT1Struct *p, SVartyp *e)
82  {
83    SVartyp zw;
84    PIDT1_CalcInnerParams(p);
85    if (!p->POn) p->yP = 0; else p->yP = p->Kr * *e;
86    if (!p->IOn) p->yI = 0; else p->yI+= p->Ki * p->X1;
87    if (!p->DOn) p->yD = 0; else p->yD = p->yD * p->alpha + p-
    >Kd* (*e-p->X1);
88    if (p->LimOn) {
89      zw = p->yP + p->yI;
90      if (zw<p->ymin) zw=p->ymin;
91      if (zw>p->ymax) zw=p->ymax;
92      if (p->AW==1) {
93        if (p->yI<p->ymin) p->yI=p->ymin;
94        if (p->yI>p->ymax) p->yI=p->ymax;
95      } else
96        p->yI = zw - p->yP;
```

```
97      zw+=p->yD;
98      if (zw<p->ymin) zw=p->ymin;
99      if (zw>p->ymax) zw=p->ymax;
100   } else
101     zw = p->yP + p->yI + p->yD;
102   p->X1=*e;
103   return zw;
104 }
105 /*END*/
```

Die Zeilen 45 und 46 enthalten bereits die approximierte *e*-Term-Berechnung als Kommentarzeilen. Daher genügt es, diese Zeilen zu Programmzeilen zu machen und statt dessen die Zeile 43 auszukommentieren:

```
43      /* p->alpha=exp(- p->INVTvz/#InsertINVDELTAT); */
44      p->alpha=p->INVTvz/#InsertINVDELTAT;
45      p->alpha=1/(zw*zw+zw+1)
```

Als letztes könnten wir noch auf die Headerdatei MATH.H verzichten, die nur eingebunden wurde, damit dem Compiler die Exponentialfunktion bekannt ist. Dies ist aber nicht unbedingt erforderlich, da der Linker ohnehin ungebundene Referenzen entfernt.

Damit sind unsere Änderungen bezüglich der Funktionalität des PID-Reglers beendet. Nun muß noch dafür gesorgt werden, daß BORIS beim Generieren des Quellcodes eines PID-Blocks auf unseren abgeänderten Funktionscodeabschnitt zugreift. Dies erreichen wir, indem wir den zugehörigen Verweis auf die Datei in der Verweisdatei ändern:

vorher:

Blockname	**Datei**	**Funktionscodeabschnitt**
/*PID-controller (floatingpoint)*/	dynamic.c	/*PID-controller (floatingpoint)*/

nachher:

Blockname	**Datei**	**Funktionscodeabschnitt**
/*PID-controller (floatingpoint)*/	PID.C	/*PID-controller (floatingpoint)*/

Nun sind alle notwendigen Änderungen getätigt und wir können uns die Resultate zu Gemüte führen. Dazu nutzen wir die folgende Struktur:

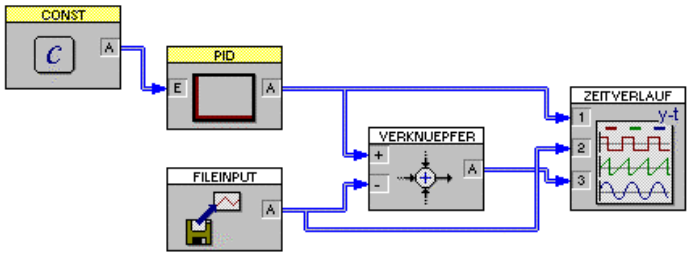

Bild 35. *Struktur zum Vergleich des PID-Reglers von BORIS mit dem generierten C-Code*

Damit die modifizierte Verzögerung des D-Anteils gut zu sehen ist, wählen wir folgende Einstellungen:

In der Simulationsumgebung: Simulationsdauer = 2; Schrittweite = 0.1

Im Block CONST: Ausgangswert = 1

Im Block PID: nur D-Anteil eingeschaltet; Vorhaltezeit = 2; Verzögerungszeit = 0.5

Einstellungen für die C-Code-Generierung:

Parameter:	**Wert:**
Zahlenformat	Fließpunktzahlen (float)
Rahmen-Funktion	Ausgabe in SIM-Dateien
Sonstige Parameter	-

Bild 36 zeigt zunächst die Differenz zwischen dem approximierten und dem exakten Exponentialterm des DT1-Anteils in Abhängigkeit von der Simulationsschrittweite ΔT und der Zeitkonstanten T_{Vz}. Bild 37 zeigt den Vergleich des modifizierten PID-C-Codes mit dem BORIS-PID-Regler.

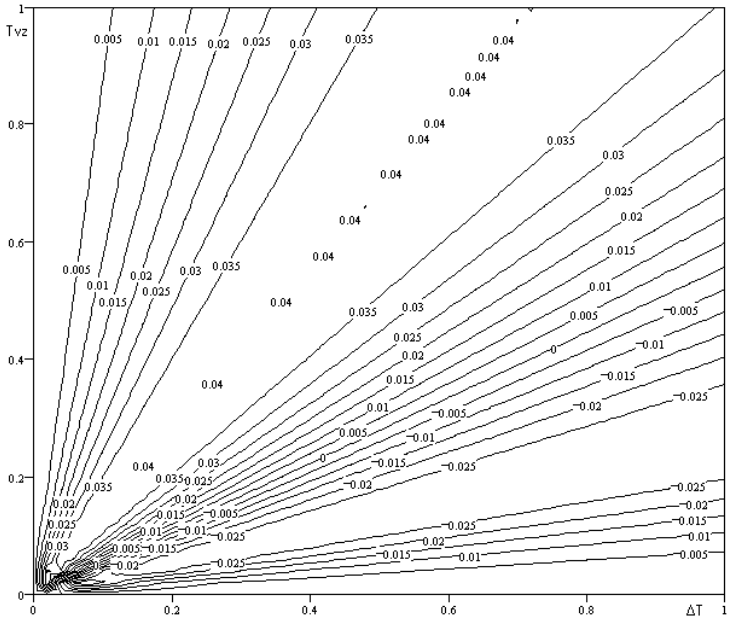

Bild 36. *Differenz zwischen nicht approximiertem und approximiertem DT1-Verhalten (in diesem Fall ≈ 0.01)*

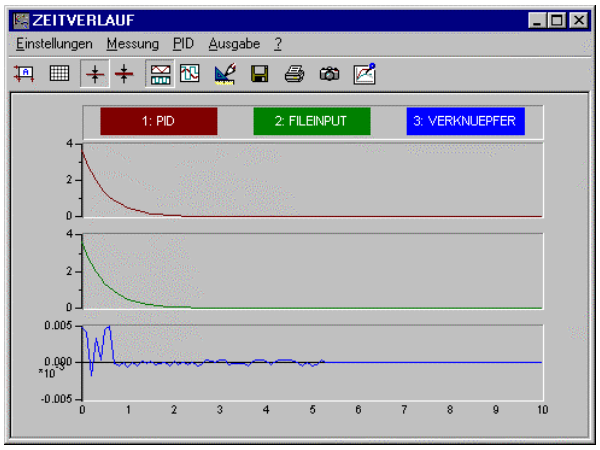

Bild 37. *Vergleich beider DT1-Verhalten*

Wie zu erwarten war, ist der Fehler so gering, daß er vernachlässigbar ist. Es macht also durchaus Sinn, das approximierte DT1-Verhalten statt des exakten

einzuführen (dies ist natürlich auch nicht exakt, aber für uns exakt genug, um es als dieses zu bezeichnen).

Zusammenfassung:

Sinnvoll wäre der Einsatz des approximierten DT1-Verhaltens, wenn die Exponentialfunktion die Zielhardware ins "Schwitzen" brächte. Diese ist aber bei den heutigen PC's mit wenigen Maschinenzyklen (dank der Fließkommaprozessoren) berechenbar. Die Approximation lohnt sich also für Rechnerarchitekturen, die diese Funktionalität nicht besitzen.

Modifikation bei Festpunktzahlen

In diesem Kapitel wollen wir einen *Geschwindigkeitsalgorithmus* für den PID-Regler entwerfen. Hierzu sind die Kenntnisse der Parameter, die in der Struktur für den PID-Regler abgelegt sind, notwendig. Inspizieren Sie dazu den *Anhang B*, in dem alle Blöcke, die von BORIS zu C-Code generiert werden können, aufgeführt sind. Des weiteren sollten Sie mit der Darstellung und der Arithmetik von Festpunktzahlen vertraut sein (Kapitel *Festpunktzahlen und ihre Arithmetik*).

Beim Geschwindigkeitsalgorithmus wird vom Regler statt des Stellgrößenwerts selbst die Stellgrößenänderung im Vergleich zum vorangegangenen Abtastschritt ausgegeben. Die zugehörige *z*-Übertragungsfunktion für den Standard-PID-Regler lautet

$$\frac{\Delta Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1}}$$

bzw. als Differenzengleichung

$$\Delta y_k = b_0 x_k + b_1 x_{k-1} + b_2 x_{k-2} - a_1 \Delta y_{k-1}$$

Dabei ist:

$$b_0 = zw_0 \cdot \left(1 + \frac{\Delta T + T_{Vz}}{2 \cdot T_N} + zw_1 \right)$$

$$b_1 = zw_0 \cdot \left(-1 + \frac{\Delta T}{2 \cdot T_N} - 2 \cdot zw_1 \right)$$

$$b_2 = zw_0 \cdot \left(zw_1 - \frac{T_{Vz}}{2 \cdot T_N} \right)$$

$$a_1 = -\frac{T_{Vz}}{\Delta T + T_{Vz}}$$

$$zw_1 = \frac{T_{Vz} + T_V}{\Delta T}$$

$$zw_0 = \frac{K_R}{1 + \frac{T_{Vz}}{\Delta T}}$$

Wie zu sehen ist, macht der hier dargestellte Algorithmus des PID-Reglers nur Sinn, wenn die Abtastzeit als konstant angesehen werden kann. Ist dies nicht der Fall, müßten für jeden Abtastschritt alle Koeffizienten neu berechnet werden. Zum Einsatz kann der Geschwindigkeitsalgorithmus nur kommen, wenn das Stellglied hinter dem PID-Regler speicherndes (integrales) Verhalten hat. Dies wäre beispielsweise bei einem Schrittmotor der Fall.

Unsere Aufgabe besteht nun darin, diesen Algortihmus zu programmieren. Dazu werden wir eine C-Funktion schreiben, die aus den von BORIS gelieferten Parametern die für diesen Algorithmus benötigten berechnet. Diese sollen in neuen Feldern derselben Struktur abgelegt werden. Somit genügt als Parameter ein Zeiger auf die neue PID-Struktur (die normale Struktur für den C-Quellcode eines PID-Reglers kann dem *Anhang B* entnommen werden). Weiterhin benötigen wir zwei Funktionen, die den von BORIS erzeugten Funktionsaufrufen genügen (s. *Anhang B*). Außer Acht lassen wir hier die *PIDT1_SetParam*- und *PIDT1_GetParam*-Funktionen, die nur von den Blökken Parameter-Modifizierer und Parameter-Wert verwendet werden. Hier die Funktionsköpfe der einzelnen, mit Anweisungen zu füllenden Funktionen:

```
MEM_ATTRIBUTE void CalculateCoefficients(PIDT1Struct *p)
MEM_ATTRIBUTE SVartyp PIDT1_init(PIDT1Struct *p, SVartyp *e)
MEM_ATTRIBUTE SVartyp PIDT1_fct(PIDT1Struct *p, SVartyp *e)
```

Da die Funktionen aber eine abgeänderte Struktur benötigen, wenden wir uns zunächst dieser zu. Fügen Sie der Struktur `PIDT1Struct` folgende Felder hinzu (die Struktur finden Sie in der Datei DYNAMIC.H):

```
SVartyp b0,b1,b2,a1;
```

für die Koeffezienten und

```
SVartyp Xk[2];
```

für die internen Speicher.

Nach der Änderung hat sie folgenden Aufbau:

```
1   typedef struct
2     { char    POn, IOn, DOn, LimOn, AW;/*AW==1 -> AntiWindUp-
    Hold; AW==2 -> AWU-Reset*/
3       char    calcParams;
4       SVartyp Kr,Tn,Tv,INVTvz;
5       SVartyp y0;
6       SVartyp ymin,ymax;
7       SVartyp Ki,Kd, alpha;
8       SVartyp yP, yI, yD;
9       SVartyp X1;
10      SVartyp Xk[2];
11      SVartyp b0,b1,b2,a1; } PIDT1Struct;
```

Nun können wir die erste Funktion, `CalculateCoefficients`, mit Anweisungen füllen. Beachten Sie dabei, daß die Multiplikation und die Division durch die Aufrufe MUL(a,Fa,b) und DIV(a,Fa,b) zu implementieren sind, da es sich ja um Festpunktzahlen handelt (s. *Multiplikation von Festpunktzahlen* und *Division von Festpunktzahlen*). Ebenso ist es wichtig, daran zu denken, daß im C-Code nicht die Abtastschrittweite, sondern ihr Kehrwert gegeben ist. Konstante Werte müssen in den richtigen Typ `SVartyp` mit Hilfe des Cast-Operators gewandelt werden, da sonst Vorzeichenfehler auftreten können. Das nachfolgende Listing zeigt die Funktion.

```
1   MEM_ATTRIBUTE void CalculateCoefficients(PIDT1Struct *p)
2   {
3     SVartyp zw0, zw1;
4     /*Now calculate coefficients*/
5     zw0=One+DIV(#InsertINVDeltaT,FracPart,p->INVTvz);
6     zw0=DIV(p->Kr,FracPart,zw0);
7     zw1=One+MUL(p->Tv,FracPart,p->INVTvz);
8     zw1=DIV(zw1,FracPart,DIV(p-
    >INVTvz,FracPart,#InsertINVDeltaT));
9     p->b0=One+DIV(p->INVTvz,FracPart,#InsertINVDeltaT);
10    p->b0=One+DIV(p->b0,FracPart,MUL(p->Tn,FracPart,p-
    >INVTvz)<<1)+zw1;
```

```
11    p->b0=MUL(zw0,FracPart,p->b0);
12    p->b1=-One+DIV(One,FracPart,MUL(p-
    >Tn,FracPart,#InsertINVDeltaT)<<1)-(zw1*2);
13    p->b1=MUL(zw0,FracPart,p->b1);
14    p->b2=MUL(p->INVTvz,FracPart,MUL(p->Tn,FracPart,One+One));
15    p->b2=zw1-DIV(One,FracPart,p->b2);
16    p->b2=MUL(zw0,FracPart,p->b2);
17    p->a1=-DIV(One,FracPart,One+DIV(p-
    >INVTvz,FracPart,#InsertINVDeltaT));
18  }
```

Listing 19. *Funktion zur Berechnung der Parameter b0, b1, b2 und a1 aus den gegebenen Parametern des PID-Reglers bei Festpunktzahlen*

Nachdem die Parameter *b0*, *b1*, *b2* und *a1* nun feststehen, können wir die Funktionen `PID_init()` und `PID_fct()` schreiben. In der "Init"-Funktion rufen wir einmalig die Funktion zur Berechnung der Koeffizienten auf. Danach kann die Funktion des PID-Reglers selbst, `PID_fct()`, zur Berechnung des Abtastschrittes zum Zeitpunkt $t = 0$ erfolgen. Beide Funktionen sind im nachfolgenden Listing aufgeführt.

```
1   MEM_ATTRIBUTE SVartyp PIDT1_fct(PIDT1Struct *p, SVartyp *e)
2   {
3     SVartyp zw;
4     zw=MUL(p->b0,FracPart,*e);
5     zw+=MUL(p->b1,FracPart,p->Xk[0]);
6     zw+=MUL(p->b2,FracPart,p->Xk[1]);
7     zw-=MUL(p->a1,FracPart,p->y0);
8     if (p->LimOn)zw=(zw>p->ymax) ? p->ymax:(zw<p->ymin) ? p-
    >ymin:zw;
9     p->Xk[1]=p->Xk[0];
10    p->Xk[0]=*e;
11    p->y0=zw;
12    return zw;
13  }
14
```

```
15  MEM_ATTRIBUTE SVartyp PIDT1_init(PIDT1Struct *p, SVartyp *e)
16  {
17    CalculateCoefficients(p);
18    p->Xk[0]=0;
19    p->Xk[1]=0;
20    return PIDT1_fct(p,e);
21  }
```

Listing 20. *Funktionen, deren Aufrufe durch den AutoCode-Generator in die Systemfunktionen eingebaut werden.*

Nach dieser Arbeit wollen wir uns nun die Ergebnisse ansehen. Doch zuvor noch eine kleine Checkliste, um Pannen zu vermeiden:

1. Ist der Funktionscodeabschnitt `/*PID-controller*/` inklusive der Endemarke `/*END*/` in der Datei PID.C (genau einmal) vorhanden ?
2. Ist die Datei PID.C in der Verweisdatei für den Verweis des PID-Reglers bei Festpunktzahlen eingetragen?
3. Ist der Verweis auf den Funktionscodeabschnitt korrekt in der Verweisdatei eingetragen?
4. Sind die Änderungen innerhalb der Struktur `PIDT1Struct,` die die Funktionen in dem Funktionscodeabschnitt verwenden, korrekt getätigt?
5. Sind die Funktionsköpfe einwandfrei (Groß-/Kleinschreibung) ?

Sind alle Punkte abgehakt, so kann mit der Simulationsstruktur nach Bild 38 der C-Code erzeugt werden. Damit die Verzögerung gut zu sehen ist, tätigen Sie folgende Einstellungen:

In der Simulationsumgebung: Simulationsdauer = 2; Abtastschrittweite = 0.05

Im Block CONST: Ausgangswert = 1

Im Block PID: Verstärkung = 1, Nachstellzeit = 1, Vorhaltezeit = 2, Verzögerungszeit = 0.5

Zusätzlich sollte im Block UNITDELAY Abtastrate gleich Simulationsschrittweite und im Integrierer die Integrierzeit gleich der Simulationsschrittweite sein. Die folgenden Parameter stellen Sie zur Generierung des C-Codes ein:

Parameter:	**Wert:**
Zahlenformat	15 Vor- und 15 Nachkommastellen
Rahmen-Funktion	Ausgabe in SIM-Dateien
Sonstige Parameter	-

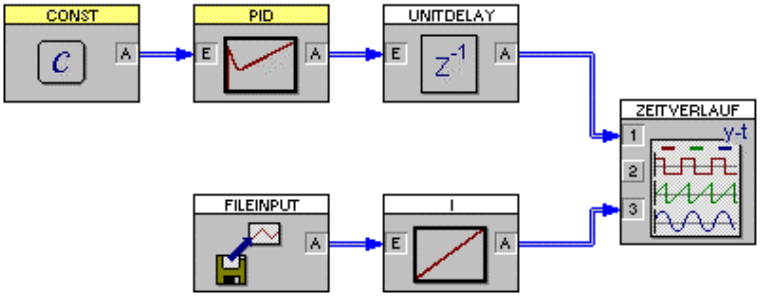

Bild 37. *Struktur zum Testen des Geschwindigkeitsalgorithmus für einen PIDT1-Regler*

Der Vergleich von C-Code und BORIS liefert das folgende Ergebnis:

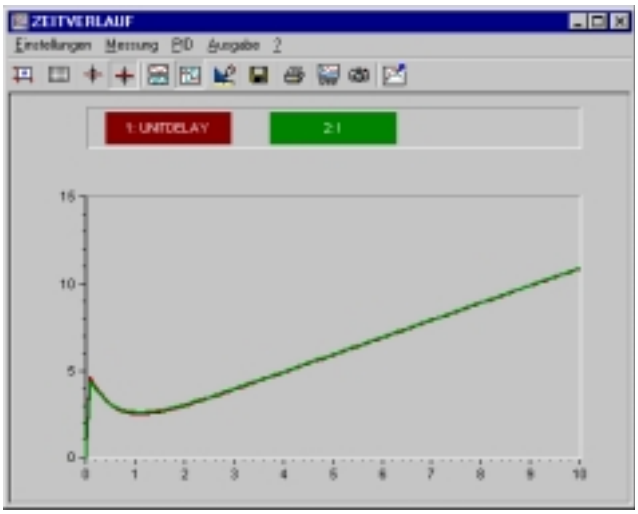

Bild 38. *Vergleich der Algorithmen*

In Bild 38 ist deutlich zu sehen, daß beide Zweige, der des PID-Reglers unter BORIS und der des FILEINPUT-Blocks, der ja das Ergebnis aus dem Simulationslauf des C-Codes einliest, im Rahmen der Bildschirmauflösung übereinstimmen. Wichtig für diese Übereinstimmung ist die Einstellung der richtigen Abtastzeit, die bei einem Zehntel der dominierenden Zeitkonstanten, in diesem Fall T_{Vz}, liegt.

Erstellen einer Funktion für Fest- und Fließpunktzahlen

Beim Erstellen einer Funktion, die gleichzeitig für Fließpunktzahlen und für Festpunktzahlen Gültigkeit hat, gelten einige Restriktionen:

- Die Schiebeoperatoren '<<' und '>>' sind unzulässig und müssen durch eine äquivalente Division ersetzt werden.
- Die Definitionen von `FracPart, FracMask, IntPart, IntMask` erfolgen nur bei Festpunktzahlen. Ihre Verwendung ist also bei Fließpunktzahlen unzulässig, es sei denn, sie werden als zweiter Parameter bei `MUL` und `DIV` eingesetzt (s. u.).

Hingegen sind die Funktionen `MUL(a,Fa,b)` und `DIV(a,Fa,b)` auch bei Fließpunktzahlen gültig, da der Parameter `Fa` bei Fließpunktzahlen durch ein entsprechendes Makro entfällt:

```
#define MUL(a,Fa,b) ((a)*(b))
#define DIV(a,Fa,b) ((a)/(b))
```

Werden diese Einschränkungen befolgt, so hat Ihr geschriebener C-Code für Fest- und Fließpunktzahlen Gültigkeit. Ist es nicht möglich, unter den gegebenen Restriktionen einen C-Code für beide Parteien zu schreiben, so kann mit einer Compilerdirektive Abhilfe geschaffen werden:

```
#ifdef FLOATINGPOINT
  /*C-Code für Fließpunktzahlen*/
#else
  /*Festpunktzahlen-Implementation*/
#endif
```

Im vorangegangenen Kapitel wurde ein Geschwindigkeitsalgorithmus für einen PID-Regler erstellt. Soll dieser so geändert werden, dass er auch für Fließpunktzahlen eingesetzt werden kann, so sind innerhalb des Funktionscodeabschnittes lediglich die Zeilen 10 und 12 zu änderen.

```
10  p->b0=One+DIV(p->b0,FracPart,MUL(p->Tn,FracPart,p-
    >INVTvz)*2)+zw1;
11    p->b0=MUL(zw0,FracPart,p->b0);
12    p->b1=-One+DIV(One,FracPart,MUL(p-
    >Tn,FracPart,#InsertINVDeltaT)*2)-(zw1*2);
```

Es wurden die Shift-Operationen gegen die Multiplikation ausgetauscht.

Des weiteren muß nun dafür gesorgt werden, dass dieser Funktionscodeabschnitt auch bei Fließpunktzahlen zum Einsatz kommt. Dazu ist der Eintrag

*/*PID-controller (floatingpoint)*/....*

in der Verweisdatei zu ändern in

*/*PID-controller (floatingpoint)*/ PID.C /*PID-controller*/*

Benutzerdefinierte Systemblöcke (User-DLLs)

Mit Hilfe von User-DLLs können Sie unter BORIS Ihre eigenen Systemblöcke programmieren; hierzu enthält das WinFACT-Benutzerhandbuch einige Beispiele. In diesem Kapitel wird Ihnen gezeigt, wie Sie auch diese Blöcke zu C-Code generieren können. Voraussetzung für das Verständnis dieses Abschnittes ist der Abschnitt *Benutzerdefinierte Systemblöcke* in Kapitel 10 (Modul BORIS) des WinFACT-Benutzerhandbuchs.

Schnittstelle des AutoCode-Generators für User-DLLs

Die Schnittstelle des AutoCode-Generators muß in der User-DLL in Gestalt der Funktion *GenerateCCode* verankert werden. Diese Funktion hat folgenden Aufbau:

Signatur: **Pascal:**

```
Function GenerateCCode(
                    D1 : PParameterStruct;
```

```
                    D2 : PNumberOfInputsOutputs;
                    step : Integer;
                    pc   : PChar;
                    size : Integer):Integer;
                                export stdcall;
```

C:

```
int _export _stdcall GenerateCCode(
                    PParameterStruct       D1,
                    PNumberOfInputsOutputs D2,
                    int                    step,
                    PChar                  pc
                    int                    size);
```

Parameter: *D1* ist ein Zeiger auf die Struktur *TParameterStruct*

D2 ist ein Zeiger auf die Struktur *TNumberOfInputsOutputs*

step ist ein 32-Bit-Integer-Wert; die unteren 16 Bit dienen als Zähler, der angibt, welche Stufe der Code-Generierung gerade bearbeitet wird (s. u.). Enthalten die oberen 16 Bit den Wert 1, so soll dies zeigen, daß *Konstanten zulässig* sind.

pc ist ein Zeiger auf eine Zeichenkette, in die C-Quelltext geschrieben werden kann. Gleichzeitig dient *pc* als Aufrufzähler für die jeweilige Stufe, d. h. bei jedem Aufruf von *GenerateC-Code* übergibt BORIS in *pc* die aktuelle Aufrufnummer für die jeweilige Stufe (beginnend bei null); dieser Wert kann dann benutzt werden, um bei mehreren Aufrufen einer Stufe (z. B. umfangreichen Quelltextgenerierungen) die jeweils benötigte Ausgabe zu steuern (siehe Beispiel 1 an späterer Stelle).

size gibt die Größe der Zeichenkette an. Es können maximal *size* Zeichen in die Zeichenkette, die durch *pc* referenziert wird, geschrieben werden. **Die Zeichenkette sollte mit einem Nullbyte-Zeichen abgeschlossen werden.**

Funktion: *GenerateCCode* wird pro Stufe (insgesamt neun verschiedene Stufen s. u.) der C-Code-Generierung so oft aufgerufen, bis der Rückgabewert der Funktion null ist. Die durch *pc* von der Funktion gelieferten Zeichenketten werden dabei direkt (also ohne Zeilenumbruch o. ä.) aneinander gehängt. So kann durch diese Funktion für jede Stufe der Quelltexterzeugung ein eigener Satz an Quelltextzeilen geschrieben werden.

Der Aufruf und die Implementierug sind nicht trivial, dafür erhöhen sie die Flexibilität der Code-Generierung für DLL-Blöcke aber enorm. Die Code-Generierung erfolgt für jeden Block in verschiedenen Stufen oder Durchläufen, die folgende Aufgaben haben:

1. Schreiben der #defines
2. Schreiben der #includes
3. Schreiben der globalen Variablen
4. Schreiben von Integrationsmethoden und sonstigen speziellen Funktionen.
5. Schreiben der Funktionsdeklarationen (xx_Init(), xx_fct(), xx_free(); dies sind in fast allen Fällen bislang auch gleichzeitig die Implementierungen der Funktionen)
6. Schreiben der Parameterzuweisungen
7. Schreiben des Initialisierungs-Funktionsaufrufes
8. Schreiben des Funktionsaufrufes
9. Schreiben des Freigabe-Funktionsaufrufes

Für jede dieser Stufen wird mindestens einmal die in der DLL vorhandene Funktion *GenerateCCode* aufgerufen. Wie oft sie tatsächlich pro Stufe aufgerufen wird, hängt von dem Ergebnis der Funktion ab. Solange der Rückgabewert ungleich 0 ist, wird die Funktion für diese Stufe erneut aufgerufen. Der Inhalt der Zeichenkette *pc* wird dabei aneinander gehängt, so daß eine lange Zeichenkette daraus entsteht (Sie müssen dafür Sorge tragen, daß ein Zeilenumbruch am Ende jeder Zeile entsteht). Zusätzlich werden spezielle Zeichenketten wie in der folgenden Tabelle beschrieben ersetzt:

Zeichenkette (Groß-/Kleinschreibung bleibt unbeachtet)	**Ersetzung**
`#BlckStr`	modifizierter Blockname, s. o.
`#SpaceOfBlckStr`	Anzahl an Leerzeichen, die der Länge des modifizierten Blocknamens entsprechen.

Zusätzlich können innerhalb der Zeichenkette für jede Stufe unterschiedliche Anweisungen für BORIS enthalten sein, die nicht direkt in den zu erzeugenden C-Quelltext geschrieben werden, sondern BORIS veranlassen, bestimmte Din-

ge zu verrichten und die Ergebnisse dieser Arbeit in einem bestimmten Format in den Quelltext einzufügen. Hier nun die neun Stufen von oben mit ihren einzelnen Anweisungen für BORIS:

1 Die komplette Zeichenkette wird in den Quelltextabschnitt der Define-Anweisungen angefügt.

2 Die Zeichenketten

```
AddToInclude("test.h")
AddToInclude(<stdio.h>)
```

veranlassen BORIS, diese Headerdateien in den *#Include*-Abschnitt der Quelltextdatei einzufügen, falls sie nicht schon vorhanden sind. Alle anderen Zeichenketten werden direkt in den Quelltext geschrieben.

3 Die Zeichenkette

```
WriteBlockFunction(/*Fkt-Codeabschnitt*/)
```

trägt dafür Sorge, daß BORIS den Funktionscodeabschnit `/*Fkt-Codeabschnitt*/` in der Verweisdatei sucht, dann den dort angegebenen Funktionscodeabschnitt in der ebenfalls dort angegebenen Datei sucht und einfügt. Dieses Einfügen eines Funktionscodeabschnittes erfolgt nur, wenn der Funktionscodeabschnitt nicht schon einmal eingefügt wurde.

Alle anderen Zeichenketten werden direkt in den Quelltext geschrieben.

4 Die Auswertung erfolgt exakt so wie unter Punkt 3.

5 Hier werden alle Zeichenketten direkt in den Quelltext geschrieben.

6 Die Zuweisungen an Parameter wurden so geschaffen, daß die unter BORIS getätigten Einstellungen berücksichtigt werden (Fließpunkt, Festpunkt etc.). Zusätzlich sind weitere Parameter der Anweisung eingeführt worden, um die korrekte Erzeugung von Zeichenketten, die einem Feld einer Struktur etwas zuweisen, zu vereinfachen. Die für diese Aufgabe zur Verfügung stehende Anweisungszeichenkette hat folgenden Aufbau:

CAssign { AS_SN,[p],Fz,Wert | AS_CSN,[..%s..],Fz,Wert | AS_PN,[p1,p2],Fz,Wert | AS_INT,[p],Fz,Wert | AS_CINT,[..%s..],Fz,Wert | AS_LONG,[p],Fz,Wert | AS_STRING,[p],Fz,Wert | AS_CHAR,[p],Fz,Wert | AS_ASSIGN,[p],Fz,Wert | AS_CODE,[],Fz,Wert }

Wie zu sehen ist, ist der erste Parameter für die Art der Zuweisung verantwortlich. Dabei wird der Parameter *Wert* dem Parameter *p* der Struktur dieses Blockes nach der Vorschrift, die in AS_xx verankert ist, zugewiesen (siehe nachfolgende Tabelle). *Fz* ist ein Fehlerstring, der von BORIS bei der Generierung als Hilfestellung ausgegeben wird, falls bei der entsprechenden Zuweisung ein Fehler auftritt; er kann von Ihnen frei vergeben werden.

AS_xx - Format	**Zuweisungs-Vorschrift**
AS_SN	Der in eckigen Klammern stehende Parameter erhält den Wert (interpretiert als Zahl) im Signalformat. D. h., wurden Festpunktzahlen gewählt, so erhielte dieser Parameter den entsprechend kodierten Wert. Der Parameter muß den Typ *SVartyp* besitzen.
AS_CSN	Der in eckigen Klammern stehende Parameter erhält eine Formatzeichenkette, in der das Zeichen *%s* durch die in *Wert* enthaltene Zahl, die in das Signalformat transformiert wird, ersetzt wird. Die Ausgabe wird in die Headerdatei geschrieben.
AS_FSN	Arbeitet wie AS_CSN ohne die Ausgabe in die Headerdatei zu lenken.
AS_PN	*Ist nur noch aus Kompatibilitätsgründen vorhanden. Verwenden Sie stattdessen*

	AS_SN. (AS_PN wird auf AS_SN umgeleitet!)
AS_INT	Der Parameter in eckigen Klammern erhält den Wert (interpretiert als Zahl) gerundet als Integer.
AS_CINT	Der in eckigen Klammern stehende Parameter erhält eine Formatzeichenkette, in der das Zeichen *%s* durch die in *Wert* enthaltene Zahl ersetzt wird. Die Ausgabe wird in die Headerdatei geschrieben.
AS_FINT	Arbeitet wie AS_CINT ohne die Ausgabe in die Headerdatei zu lenken.
AS_LONG	Der Parameter in eckigen Klammern erhält *Wert* (interpretiert als Zahl) gerundet als Integer mit dem Suffix 'L' für *long*-Konstanten.
AS_STRING	Dem Parameter in eckigen Klammern wird *Wert* (interpretiert als Zeichenkette) durch die strcpy-Funktion zugewiesen.
AS_CHAR	Der Parameter in eckigen Klammern erhält *Wert* (interpretiert als **ein** Zeichen) durch Zuweisung mit Hochkommata (Beispiel: p='a').
AS_ASSIGN	Dem Parameter wird direkt *Wert* zugewiesen. Dieser wird als Zeichenkette rechts vom Zuweisungsoperator interpretiert.
AS_CODE	Schreibt ungeachtet des Parameters *Wert* interpretiert als Zeichenkette in den Quelltext.

Die Verwendung dieser recht komplexen Anweisungszeichenkette wird durch die Beispiele deutlicher. Die Parameter AS_CSN und AS_CINT schreiben im Gegensatz zu allen anderen ihre Ausgabe in die Headerdatei. Es ist aber häufig erforderlich, daß sämtliche Ausgaben in diese Datei geschrieben werden können. Daher gibt es die Anweisungen

>*CONSTFILE* und

>NORMALFILE

die immer als einzige in *pc* enthalten sein sollten. Die erste lenkt die Ausgabe in die Headerdatei solange um, bis irgendwann die zweite auftritt. Standardmäßig ist das Schreiben in die normale C-Datei eingestellt; soll in die Headerdatei geschrieben werden, so ist dies explizit durch die obige Anweisung zu veranlassen (Beispiel 2 zeigt die Verwendung dieser Befehle).

Alle anderen Zeichenketten werden direkt in den Quelltext geschrieben.

7 Die Funktionsaufrufe werden als Formatzeichenkette ausgewertet. Sie enthalten die Form

```
fktname(&#Blckstr, %i1, %i2,...,
                   %o1, %o2,...);
```

dabei muß die Anzahl der %-Parameter gleich der Summe der Ein-/Ausgänge dieses Blockes sein. Die Zeichenkette *%ix* wird durch den *x*-ten Eingang ersetzt, die Zeichenkette *%ox* durch den *x*-ten Ausgang. Die Eingangsparameter werden als Kopie (call-by-value) übergeben, die Ausgangsparameter als Referenz (call-by-reference).

8 Die Auswertung erfolgt exakt so wie unter Punkt 6.

9 Alle Zeichenketten werden direkt in den Quelltext geschrieben.

Beispiele

Beispiel 1: Eine einfache User-DLL

Dieses Beipiel entspricht dem ersten Beispiel aus dem WinFACT-Benutzerhandbuch, Kapitel *Benutzerdefinierte Systemblöcke*. Hier wurde im Beispiel 1 eine simple User-DLL geschrieben, die es nun so zu erweitern gilt, daß sie bei der C-Code-Generierung Ihre Funktionalität in den Quelltext liefert. Dazu ist die Funktion *GenerateCCode* zu implementieren und in die Exportliste der DLL einzutragen (letzteres wird hier nicht gezeigt, s. WinFACT-Benutzerhandbuch). Nachfolgendes Listing zeigt die implementierte Funktion.

```
1   function  GenerateCCode(D1   : PParameterStruct;
                            D2   : PNumberOfInputsOutputs;
                            step : Integer; pc : PChar;
```

```
                          size : Integer)
                         :Integer; EXPORT STDCALL;
2   var n : Integer;
3   begin
4     n:=StrToInt(strpas(pc)); // Aufrufnummer der Stufe
5     fillchar(pc[0],size,0);  // pc initialisieren
6     Result:=0; // standardmäßig nur einmal pro Stufe aufrufen
7     case step of
8     //DEFINES
9     0: ;
10
11    //INCLUDES
12    1: StrLCopy(pc,'AddToInclude("DLLDemo.h")',size);
13
14    //GLOBALVARS
15    2: StrLCopy(pc,'MEM_ATTRIBUTE_VAR DLLDEMOStruct
    #BlckStr;',size);
16
17    //INTEGRATIONMETHOD
18    3: ;
19
20    //DECLARATIONS
21    4: StrLCopy(pc,'WriteBlockFunction(/*DLLDemo 1*/)',size);
22
23    //SETTINGS
24    5: begin
25         case n of
26           0:StrLCopy(pc,PChar('CAssign(AS_SN,[k1],Fehler bei
    k1,'+FloatToStr(D1^.E[0])+')'),size);
27           1:StrLCopy(pc,PChar('CAssign(AS_SN,[k2],Fehler bei
    k2,'+FloatToStr(D1^.E[1])+')'),size);
28           2:StrLCopy(pc,PChar('CAssign(AS_SN,[k3],Fehler bei
    k3,'+FloatToStr(D1^.E[2])+')'),size);
29           3:StrLCopy(pc,PChar('CAssign(AS_INT,[invert],Fehler
    bei invert,'+FloatToStr(D1^.B[0])+')'),size);
30         end;
31         Result:=n-3;
32       end;
33
34    //INITFUNCTIONCALL und FUNCTIONCALL
```

```
35    6,7: StrLCopy(pc,'dlldemo_fct(&#BlckStr, %i1, %i2,
    %o1);',size);
36    //FREEFUNCTION
37    8:   ;
38    end;
39  end;
```

Listing 21. *Funktion* GenerateCCode *innerhalb der User-DLL. Zur Verdeutlichung der an BORIS übergebenen Anweisungen wurden diese hervorgehoben.*

Mit Hilfe der hervorgehobenen Anweisungen läßt sich die Funktion recht einfach realisieren. Probleme wie die Berücksichtigung der AutoCode-Generierungs-Einstellungen lassen sich nach außen verlagern, wo sie durch BORIS (Anweisung *CAssign*) oder durch einen entprechenden Funktionscodeabschnitt (Anweisungen *AddToInclude*, *WriteBlockFunction*) behandelt werden können.

Damit die User-DLL nun bei der C-Code-Generierung korrekt arbeitet, sind noch weitere Dinge zu tun. Durch Zeile 12 des obigen Listings wird BORIS veranlaßt, die Datei *DLLDemo.h* über *#Include* in den Quelltext zu binden. In dieser Datei soll dann die Strukturdefinition *DLLDEMOStruct* erfolgen:

```
1   #ifndef dlldemo
2   #define dlldemo dlldemo
3
4     typedef struct {
5        char invert;
6        SVartyp k1,k2,k3;
7     } DLLDEMOStruct;
8
9   #endif
```

Listing 22. *Deklaration der Struktur* DLLDEMOStruct

In Zeile 15 des Listing 21 wurde die Ersetzungszeichenkette `#BlckStr` verwendet, um eine eindeutige Variable vom Typ *DLLDEMOStruct* für den C-Code zu erzeugen. Durch die `WriteBlockFunction`-Anweisung wurde der eigentliche Funktionsquelltext in einen Funktionscodeabschnitt mit dem Namen /*DLLDemo 1*/ verlagert. Daher muß die folgende Zeile in der Verweisdatei enthalten sein:

/*DLLDemo 1*/ DLLDemo.c /*DLLDemo 1*/

Der zugehörige Funktionscodeabschnitt in der Datei *DLLDemo.c* sieht folgendermaßen aus:

```
1   /*DLLDemo 1*/
2   MEM_ATTRIBUTE void dlldemo_fct(DLLDEMOStruct *p, SVartyp i1,
    SVartyp i2, SVartyp *o1)
3   {
4     if (i1<0) *o1=MUL(p->k1,FracPart,i2);
5     else if (i1==0) *o1=MUL(p->k2,FracPart,i2);
6     else *o1=MUL(p->k1,FracPart,i2);
7     if (p->invert) *o1= - *o1;
8   }
9   /*end*/
```

Listing 23. *Functionscodeabschnitt für die DLLDemo*

Die Zeilen 26 bis 29 in Listing 21 machen von der *CAssign*-Anweisung Gebrauch, so daß die Zuweisung von Werten an die Felder der Struktur *DLLDEMOStruct* vereinfacht wird. In Zeile 35 schließlich wurde die Formatzeichenkette eingesetzt, um den Funktionsaufruf *dlldemo_fct()* zu realisieren.

Das nachfolgende Listing zeigt den generierten C-Code des User-DLL-Blocks für folgende Einstellungen:

Parameter:	**Wert:**
Zahlenformat	Fließpunktzahlen (float)
Rahmen-Funktion	Standard-Ein-/Ausgabe
Sonstige Parameter	-

```
1   /************************************************************
    ***************
2     This file was created by the AutoCode-Generator of the
    program BORIS
3     which is a module of WinFACT (Windows Fuzzy And Control
    Tools)
4
5     Instead of editing this file, modify BORIS System-File
6     with BORIS and use the AutoCode-Generator once again.
7
8     Generated by : Michael
9                 on : MICHAELNT
10
11    Boris Version    : 6.1.1.218
12    Source-File(s)   : *.BSY from:21.11.2001 10:26:48
13                       THE FILE WAS MODIFIED AND NOT SAVED!!
```

```
14     refers to      :
15
    C:\WINFACT\WF2001\AUTOCODE\EXAMPLES\DLLDEMO1.DLL
    from:26.05.1999 16:28:46
16    Timestamp        : 22.11.2001 15:05:03
17
18  ************************************************************
    ***************/
19  /*********************************************************/
20  /*  all #define directives the c-compiler has to know   */
21  /*********************************************************/
22  #define Rgchk(a) (a)
23  #define MUL(a,b,c) Rgchk((a)*(c))
24  #define DIV(a,b,c) Rgchk((a)/(c))
25  /*********************************************************/
26  /*                  #include directives                  */
27  /*********************************************************/
28  #include "C:\temp\test.h"
29  #include "regdef.h"
30  #include <stdio.h>
31  #include "DLLDemo.h"
32  /*****************************************************/
33  /* declarations of global constants that are needed  */
34  /*****************************************************/
35    const SVartyp One=1;
36    const SVartyp SignalMax=1E32;
37    const SVartyp SignalMin=-1E32;
38    const SVartyp HIGHLEVEL=5;
39    const SVartyp LOWLEVEL=0;
40    const SVartyp THRESHOLD=2.5;
41    const SVartyp PI_DIV_2=1.570796327;
42    const SVartyp PI_MUL_2=6.283185307;
43    const SVartyp PI=3.141592654;
44    const SVartyp EXP1=2.718281828;
45  /*****************************************************/
46  /* declarations of global variables that are needed  */
47  /*****************************************************/
48    MEM_ATTRIBUTE_VAR SVartyp INVDELTAT;
49    MEM_ATTRIBUTE_VAR SVartyp X_VEC[MAX_X_VEC][2];
50    MEM_ATTRIBUTE_VAR SVartyp I_VEC[MAX_I_VEC][2];
```

```
51    MEM_ATTRIBUTE_VAR unsigned int TimeStepCounter;
52    MEM_ATTRIBUTE_VAR DLLDEMOStruct USER1_V0;
53  /*******************************************************/
54  /* specific global functions such as integrationmethods */
55  /*******************************************************/
56
57
58  /*********************************/
59  /* global functions of the blocks */
60  /*********************************/
61  MEM_ATTRIBUTE void dlldemo_fct(DLLDEMOStruct *p, SVartyp i1,
    SVartyp i2, SVartyp *o1)
62  {
63    if (i1<0) *o1=MUL(p->k1,FracPart,i2);
64    else if (i1==0) *o1=MUL(p->k2,FracPart,i2);
65    else *o1=MUL(p->k1,FracPart,i2);
66    if (p->invert) *o1= - *o1;
67  }
68
69  /*************************************/
70  /* the controller initialising function */
71  /*************************************/
72  void testinitcontrol(SVartyp USER1_0I1,
73                       SVartyp USER1_0I2,
74                       SVartyp *USER1_0O1)
75  {
76
77    {
78      /* initialising the state of the system and used exter-
    nals */
79      unsigned int i;
80      for (i=0; i<MAX_X_VEC; i++){X_VEC[i][0]=0;
    X_VEC[i][1]=0;}
81    }
82    {
83      unsigned int i;
84      for (i=0; i<MAX_I_VEC; i++){I_VEC[i][0]=0;
    I_VEC[i][1]=0;}
85    }
86    {
```

```
87      /* set state of the system from the external inputs */
88      I_VEC[0][0]=USER1_0I1;
89      I_VEC[1][0]=USER1_0I2;
90    }
91    USER1_V0.k1=1;
92    USER1_V0.k2=5;
93    USER1_V0.k3=10;
94    USER1_V0.invert=0;
95
96    /* calls of the used functions in the system */
97    dlldemo_fct(&USER1_V0, I_VEC[0][0], I_VEC[1][0],
    &X_VEC[0][0]);
98
99    {
100     /* set all external outputs from the state of the system
    */
101     unsigned int i;
102     for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
103     *USER1_0O1=X_VEC[0][0];
104   }
105 }
106
107 /*********************************/
108 /* the controller function itself */
109 /*********************************/
110 void testcontrol(SVartyp USER1_0I1,
111                  SVartyp USER1_0I2,
112                  SVartyp *USER1_0O1)
113 {
114
115   {
116     /* set state of the system from the external inputs */
117     unsigned int i;
118     for(i=0; i<MAX_I_VEC; i++) I_VEC[i][1]=I_VEC[i][0];
119     I_VEC[0][0]=USER1_0I1;
120     I_VEC[1][0]=USER1_0I2;
121   }
122   dlldemo_fct(&USER1_V0, I_VEC[0][0], I_VEC[1][0],
    &X_VEC[0][0]);
123
```

```
124   {
125     /* set all external outputs from the state of the system
    */
126     unsigned int i;
127     for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
128     *USER1_0O1=X_VEC[0][0];
129   }
130 }
131 void testfreecontrol(void)
132 {
133 }
```

Listing 26. *Generierter C-Code für Beispiel 1*

Die `main`-Funktion wurde aus dem Listing entfernt. Außerdem wurden alle Einträge, für die die Funktion *GenerateCCode* verantwortlich ist, hervorgehoben. Die Wirkung der einzelnen Anweisungen aus Listing 21 auf den generierten C-Quelltext dürften dadurch klar werden.

Beispiel 2: Benutzerdefiniertes Kennfeld

Dieses Beispiel entspricht Beispiel 2 aus dem WinFACT-Benutzerhandbuch. Die Besonderheit liegt hierbei darin, daß hier eine FWM-Datei (der Aufbau einer FWM-Datei kann ebenfalls der WinFACT-Dokumentation entnommen werden; zur Information: Eine FWM-Datei kann ein Kennfeld eines Fuzzy-Controllers mit 2 Eingängen und 1 Ausgang beherbergen; dazu kann das WinFACT-Modul FLOP herangezogen werden) eingelesen wird und die Daten in ein zweidimensionales Feld abgespeichert werden. Wenn nun der endgültige C-Code auf der Zielhardware zum Einstatz kommen soll, die kein Medium zur Speicherung einer Datei (im Sinne eines Dateisystems) aufweist, so müssen die Daten der Datei in die Parameterstruktur des Blockes eingetragen werden. Da dies hier recht viele sind[7], soll bei der Implementierung der *GenerateCCode*-Funktion die Möglicheit vorgesehen werden, das Kennfeld selbst als konstant und somit ROM-fähig abzulegen. Für andere Zwecke kann es geeigneter sein, das Kennfeld im RAM-abzulegen, um eventuell Veränderungen während des Programmlaufes daran durchzuführen (ist zwar eher unwahrscheinlich, aber es soll uns hier um das Prinzip gehen, beide Dinge in der *GenerateCCode*-Funktion zu vereinbaren).

[7] Ein 20 x 20 Kennfeld beansprucht bei 16 Bit pro Element 800 Bytes an Speicher, was für einige Microcontroller zur direkten RAM-Speichererschöpfung führt. Die meisten Controller haben relativ wenig Onboard-RAM und verhältnismäßig viel Onboard-ROM.

Ein Unterscheidungskriterium zur Differenzierung, ob die Erzeugung als konstantes Kennfeld oder veränderbares Kennfeld stattfinden soll, wird in den oberen 16 Bit des Parameters *step* der Funktion *GenerateCCode* geliefert. Ist hier eine 1 eingetragen, so ist das Kennfeld als konstantes Feld anzulegen. Wie muß nun die Parameterstruktur im C-Code aussehen, um einerseits ein konstantes Feld, andererseits ein variables Feld ansprechen zu können? Eine Deklaration der Form

```
typedef struct{
  ...
  SVartyp z[20][20];
  ...} DRDFELDStruct;
```

ließe nur eine Speicherung im RAM zu, womit alles andere hinfällig wäre. Nun könnte man aber einen zweiten Typ einführen, um die Speicherung als konstantes Feld vorzunehmen und in der *GenerateCCode*-Funktion zu unterscheiden, welchen Typ eine Variable für die Funktion annehmen soll:

```
typedef struct{
  ...
  SVartyp *z;
  ...} CONSTDRDFELDStruct;
```

Hier ließe sich das konstante Feld der Form

```
const SVartyp VariablenNameFuer_z[20*20]={
  1, 2, ..., 20,
  21, 22, ..., 40,
  ...
  381, 382, ..., 400};
```

durch eine einfache Zuweisung an das Feld *z*

```
  #Blckstr.z=(SVartyp*) VariablenNameFuer_z;
```

bewerkstelligen. Da man jedoch unterschiedliche Parameterstrukturen hat, müssen der Funktionskopf und der Zugriff auf das Feld *z* in der C-Funktion unterschiedlich erfolgen. Man muß also immer zwei Funktionen pflegen, was sicherlich nicht sinnvoll ist. Außerdem ist die zweite Parameterstruktur im Gegensatz zur ersten von der Größe des in der User-DLL verwendeten Kennfeldes unabhängig. Selbst für eine Erweiterung der User-DLL auf 80x80 Felder müßten Sie die erste Deklaration überarbeiten. Mit wenig Aufwand kann jedoch die zweite Parameterstruktur auch für das Erzeugen eines veränderbaren Kennfeldes erzielt werden. Dazu muß nur dem Zeiger `SVartyp *z` ein entsprechend großer Speicherbereich zugeordnet werden. Bei Beendigung des Programmlaufs des C-Programmes muß dieser natürlich wieder freigegeben werden. Dies darf allerdings nur erfolgen, wenn der Speicher explizit angelegt wurde. Er darf und kann nicht freigegeben werden, wenn der Kennfeldspeicher

als konstant vereinbart wurde. Es wird demzufolge ein Feld benötigt, welches diese Eigenschaft speichert. In dem unten stehenden Listing soll das Feld *cz* diese Aufgabe übernehmen. Die Zwecke der weiteren Felder dürften von der Namensgebung her offensichtlich sein.

Noch eine Anmerkung zu dynamisch allokiertem Speicher: Da wir in einem realen System davon ausgehen können, daß das C-Programm ständig läuft und meist nur durch Reset unterbrochen wird, entfällt das Problem der Speicherzerstückelung und somit der Garbage-Collection (Zusammenfassung nicht mehr verwendeten Speichers, um einer weiteren Speicherallokierung gerecht werden zu können). Sollte Ihr System aber so aufgebaut werden, daß das C-Programm immer wieder neu angestoßen wird, so müssen Sie sicherstellen, daß dieses auch dann noch definiert arbeitet, wenn eine Speicherallokation scheitert. Wir gehen hier davon aus, daß entweder eine Garbage-Collection vom Betriebssystem durchgeführt wird, immer genügend zusammenhängender Speicher zur Verfügung steht bzw. daß auf kleinen Zielsystemen von vornherein mit einem konstanten Kennfeld gearbeitet wird. Demzufolge steht der Lösung also nichts mehr in Wege. Die verwendete Parameterstruktur zeigt Listing 27.

```
1   typedef struct{
1      char cz;
2      unsigned int nx,ny;
3      SVartyp      dx,dy;
4      char         interpolate;
5      SVartyp      *z;
6      SVartyp      xmin,xmax;
7      SVartyp      ymin,ymax;} DRDFELDStruct;
```

Listing 27. *Parameterstruktur für die Funktion des 3D-Kennfeldes*

Sie soll in der Datei *DRDFeld.h* abgelegt sein. Diese muß also durch #include hinzugebunden werden, was durch

```
StrLCopy(pc,'ADDToInclude("DRDFeld.h")',size);
```

in der *GenerateCCode*-Funktion der User-DLL geschehen sollte. Die Funktion des 3D-Kennfeldes selbst kann in einen Funktionscodeabschnitt verlegt werden. Somit ist der Eintrag

```
/*3D-Kennfeld*/    drdfeld.c   /*3D-Kennfeld*/
```

in Ihrer Verweisdatei hinzuzufügen. Die Funktion selbst kann über

```
StrLCopy(pc,'WriteBlockFunktion(/*3D-Kennfeld*/)',size);
```

in den endgültigen C-Code eingebunden werden. Die Implementierung der Funktion entspricht ungefähr der Pascal-Implementierung der User-DLL, lediglich der Zugriff auf das Kennfeld ist ein anderer (Listing 28).

```
1   /*3D-Kennfeld*/
2   #include <malloc.h>
3   MEM_ATTRIBUTE void drdfeld_fct(DRDFELDStruct *p,
                                   SVartyp i1, SVartyp i2,
                                   SVartyp *o1, SVartyp *o2)
4   { SVartyp x0, y0, xp, yp;
5     int ix,iy;
6
7     if ((i1<p->xmin)||(i1>p->xmax)||
          (i2<p->ymin)||(i2>p->ymax)) *o2=HIGHLEVEL;
8     else *o2=LOWLEVEL;
9     ix = (int)((i1-p->xmin)/p->dx);
10    iy = (int)((i2-p->ymin)/p->dy);
11    if (ix < 0) ix = 0;
12    if (iy < 0) iy = 0;
13    if (p->interpolate){
14      if (ix > p->nx-2) ix = p->nx-2;
15      if (iy > p->ny-2) iy = p->ny-2;
16      x0 = p->xmin + ix * p->dx;
17      y0 = p->ymin + iy * p->dy;
18      xp = DIV(p->z[ix+1+iy*p->ny]-p->z[ix+iy*p->ny],
    FracPart, p->dx);
19      yp = DIV(p->z[ix+(iy+1)*p->ny]-p->z[ix+iy*p->ny],
    FracPart, p->dy);
20      *o1= p->z[ix+iy*p->ny] + MUL(i1-x0, FracPart, xp) +
    MUL(i2-y0, FracPart, yp);
21    }else{
22      if (ix > p->nx-1) ix = p->nx-1;
23      if (iy > p->ny-1) iy = p->ny-1;
24      *o1= p->z[ix+iy*p->ny];
25    }
26  }
27
28  MEM_ATTRIBUTE void drdfeld_init(DRDFELDStruct *p,
                                    SVartyp i1, SVartyp i2,
29                                  SVartyp *o1, SVartyp *o2)
30  { drdfeld_fct(p,i1,i2,o1,o2);}
```

```
31
32  MEM_ATTRIBUTE void drdfeld_free(DRDFELDStruct *p)
33  { /* z nur freigeben, wenn das Kennfeld dynamisch angelegt
    wurde:*/
34    if (! p->cz) free(p->z);
35  }
36  /*END*/
```

Listing 28. *Implementierung des Funktionscodeabschnittes zur 3D-Kennfeld-Funktion*

Nun zur Problematik der *GenerateCCode*-Funktion in der User-DLL. Zunächst muß geprüft werden, ob die Kennfeld-Datei vorhanden ist, wozu die Funktion *CanSimulateDLL()* dient (siehe WinFACT-Benutzerhandbuch, Kapitel *Benutzerdefinierte Systemblöcke, Beispiel 2: Benutzerdefiniertes Kennfeld*). Ist die Datei nicht vorhanden, so ist die User-DLL nicht simulierbar und somit muß auch kein C-Code erzeugt werden. Wenn aber alles o. k. ist, so muß einmalig die in der DLL verankerte Struktur zur Speicherung von Informationen bezüglich des Kennfeldes angelegt werden. Der Aufruf der ebenfalls schon vorhandenen Funktion *InitUserData()* erledigt dies für uns. Das Gegenstück *DisposeUserData()* wird nach Abschluß der C-Code-Generierung im letzten Aufruf von *GenerateCCode* ausgeführt. Nach der Speicherallokation durch *InitUserData()* muß die angelegte Datenstruktur mit der Information aus der FWM-Datei initialisiert werden. Dies erfolgt wie in der Funktion *InitSimulation()*. Die Zeilen der ersten Stufen können nun direkt formuliert werden:

```
37  function GenerateCCode(D1   :PParameterStruct;
                           D2   :PNumberOfInputsOutputs;
                           step :Integer;
                           pc   :Pchar;
                           size :Integer):Integer;
                          export stdcall;
38  var Datei     : Text;
39      n         : Integer;
40      i, j      : Integer;
41      ix, iy    : Integer;
42      sx,sy     : String;
43  begin
44    Result:=0;
45    n:=StrToInt(strpas(pc));
46    fillchar(pc[0],size,0);
47
48    if CanSimulateDLL(D1)=0 then exit;
```

```
49
50   case LoWord(step) of
51   // Defines:
52   0: begin
53       InitUserData(D1);
54       AssignFile(Datei, strpas(D1^.D));
55       {$I-}reset(Datei);{$I+}
56       with PUserData(D1^.UserDataPtr)^ do begin
57         readln(Datei,ny,nx);
58         for i:=1 to ny do
59           for j:=1 to nx do readln(Datei,z[j, i]);
60         dx:=(D1^.E[1]-D1^.E[0])/(nx-1);
61         dy:=(D1^.E[3]-D1^.E[2])/(ny-1);
62       end;
63       CloseFile(Datei);
64      end;
65   // Includes:
66   1: StrLCopy(pc,'AddToInclude("drdfeld.h")',size);
67   // Globale Variablendefinition:
68   2: StrLCopy(pc,'MEM_ATTRIBUTE_VAR DRDFELDStruct
   #BlckStr;',size);
69   // Integrationsmethoden und ähnliches:
70   3: ;
71   // Funktionsdeklarationen:
72   4: StrLCopy(pc,'WriteBlockFunction(/*3D-
   Kennfeld*/)',size);
```

Listing 29. *Erster Teil der Funktion* GenerateCCode *der User-DLL; dieser enthält überwiegend vorbereitende Maßnahmen. Lediglich die hervorgehobenen Anweisungen wirken sich auf den C-Code aus.*

Die 5. Stufe ist die komplexeste in der Funktion. Zunächst können hier die Felder initialisiert werden, die in beiden Generierungsarten (Kennfeld variabel oder Kennfeld konstant) identisch sind.

```
73   // Initialisieren der Felder der Parameterstruktur:
74   5: begin
75        Result:=1;  // GenerateCCode immer wieder aufrufen!
76        case n of
77          0:StrLCopy(pc, PChar('CAssign(AS_INT,
                [nx],Fehler bei nx,
```

```
                '+IntToStr(PUserData(D1^.UserDataPtr)^.nx)+
                ')'),size);
78          1:StrLCopy(pc, PChar('CAssign(AS_INT,
                [ny],Fehler bei ny,
                '+IntToStr(PUserData(D1^.UserDataPtr)^.ny)+
                ')'),size);
79          2:StrLCopy(pc, PChar('CAssign(AS_SN,
                [dx],Fehler bei dx,
                '+FloatToStr(PUserData(D1^.UserDataPtr)^.dx)+
                ')'),size);
80          3:StrLCopy(pc, PChar('CAssign(AS_SN,
                [dy],Fehler bei dy,
                '+FloatToStr(PUserData(D1^.UserDataPtr)^.dy)+
                ')'),size);
81          4:StrLCopy(pc, PChar('CAssign(AS_SN,
                [xmin],Fehler bei xmin,
                '+FloatToStr(D1^.E[0])+')'),size);
82          5:StrLCopy(pc, PChar('CAssign(AS_SN,
                [xmax],Fehler bei xmax,
                '+FloatToStr(D1^.E[1])+')'),size);
83          6:StrLCopy(pc, PChar('CAssign(AS_SN,
                [ymin],Fehler bei ymin,
                '+FloatToStr(D1^.E[2])+')'),size);
84          7:StrLCopy(pc, PChar('CAssign(AS_SN,
                [ymax],Fehler bei ymax,
                '+FloatToStr(D1^.E[3])+')'),size);
85          8:StrLCopy(pc, PChar('CAssign(AS_INT,
                [interpolate],Fehler bei interpolate,
                '+IntToStr(Ord(D1^.B[0]))+')'),size);
86          9:StrLCopy(pc, PChar('CAssign(AS_INT,
                [cz],Fehler bei cz,
                '+IntToStr(HiWord(Step))+')'),size);
```

Listing 30. *Zweiter Teil der Funktion* GenerateCCode *der User-DLL: Initialisierung der identischen Felder (Anfang der Stufe 5)*

Von nun an muß aber unterschieden werden, ob das Feld *z* der Parameterstruktur auf ein konstantes Feld ober ein variables Feld zeigen soll. Wie oben schon erwähnt, dienen zur Unterscheidung die höherwertigen 16 Bit von *step*. Für den Fall des variablen Kennfeldes muß *z* zunächst einen Speicherbereich erhalten, in dem anschließend jeder einzelne Wert des Kennfeldes abgelegt werden muß. Für den Fall des konstanten Kennfeldes muß *z* einfach auf den

Speicherbereich des Kennfeldes zeigen. Das konstante Feld soll in die Headerdatei geschrieben werden. Demzufolge wird vorher eine Ausgabeumleitung durch *>ConstFile* vorgenommen. Da im Anschluß an das konstante Kennfeld die Stufe 5 verlassen wird und somit keine weiteren Ausgaben in die C-Datei in dieser Stufe anfallen, brauchen wir *>NormalFile* nicht anzugeben! Alle anderen Stufen werden generell in die "normale" zu erzeugende C-Datei geschrieben.

```
87          else with PUserData(D1^.UserDataPtr)^ do begin
88            if HiWord(Step)=0 then begin
89              if n=10 then
90                 StrLCopy(pc,
                     PChar('#BlckStr.z=(SVartyp *)
                     malloc(sizeof(SVartyp)*'
                     +IntToStr(nx)+'*'+IntToStr(ny)+');'),size)
91              else begin
92                n:=n-11;
93                if n<nx*ny then begin
94                  ix:=n mod nx;
95                  iy:=n div ny;
96                  sx:=IntToStr(ix);
97                  sy:=IntToStr(iy);
98                  StrLCopy(pc,
                      PChar('CAssign(AS_SN,
                      [z['+sx+']['+sy+']],
                      Fehler bei z,
                      '+FloatToStr(z[ix+1][iy+1])+')'),size);
99                end else Result:=0;
100             end;
101           end else begin
102             n:=n-10;
103             case n of
104               0:StrLCopy(pc,
                     '#BlckStr.z=(SVartyp*)#BlckStr_CZ;',size);
105               1:StrLCopy(pc,'>CONSTFILE',size);
106               2:StrLCopy(pc,
                     PChar('const SVartyp
                     #BlckStr_CZ['+IntToStr(nx*ny)+']={'),
                     size);
107               else begin
108                 n:=n-3;
```

```
109                ix:=n mod nx;
110                iy:=n div ny;
111                sx:=IntToStr(ix);
112                sy:=IntToStr(iy)+'* #BlckStr.ny';
113                if n<nx*ny-1 then
114                  StrLCopy(pc, PChar('CAssign(AS_CSN,
                       [/*['+sx+'+'+sy+']*/%s,'#13#10'],
                       Fehler bei z,
                       '+FloatToStr(z[ix+1][iy+1])+')'), size)
115                else begin
116                  StrLCopy(pc, PChar('CAssign(AS_CSN,
                       [/*['+sx+']['+sy+']*/%s};],Fehler bei
                       z,'+FloatToStr(z[ix+1][iy+1])+')'),
                       size);
117                  Result:=0;
118                end; {of else begin}
119              end; {of else begin}
120            end; {of case n of}
121          end; {of else begin}
122        end; { of else with PUserData(D1^.UserDataPtr)^
                 do begin }
123      end; {of case n of}
124    end; {of 5:begin}
```

***Listing 31**. Dritter Teil der Funktion* GenerateCCode *der User-DLL: Unterscheidung der Art der Initialisierung des Kennfeldes (Ende der Stufe 5)*

Nun bleiben lediglich noch die Funktionsaufrufe der C-Funktionen zu implementieren und die durch *InitUserData()* angelegte Datenstruktur in der User-DLL wieder freizugeben. Das folgende Listing komplettiert somit die endgültige *GenerateCCode*-Funktion der User-DLL.

```
125   6: StrLCopy(pc,
          'drdfeld_init(&#BlckStr, %i1, %i2, %o1, %o2);',size);
126   7: StrLCopy(pc,
          'drdfeld_fct(&#BlckStr, %i1, %i2, %o1, %o2);',size);
```

```
127   8: begin
128        StrLCopy(pc,'drdfeld_free(&#BlckStr);',size);
129        DisposeUserData(D1);
130      end;
131   end;
132 end;
```

Listing 32. *Vierter und letzter Teil der Funktion* GenerateCCode *der User-DLL: Funktionsaufrufe der Funktionen des Funktionscodeabschnittes*

Lassen wir nun den AutoCode-Generator von BORIS arbeiten und erzeugen zunächst den Code für ein einfaches variables Kennfeld auf der Basis folgender Matrix:

$$\underline{M} = \begin{pmatrix} 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 32 & 33 & 34 & 35 \\ 41 & 42 & 43 & 44 & 45 \\ 51 & 52 & 53 & 54 & 55 \end{pmatrix}$$

Dieses Kennfeld läßt die Indizierung des Feldes *z* gut nachvollziehen, so daß das nachfolgende Listing ein wenig transparenter wird. Zunächst die "variable Kennfeld-Version":

```
1   /*********************************************************
    ***************
2     This file was created by the AutoCode-Generator of the
    program BORIS
3     which is a module of WinFACT (Windows Fuzzy And Control
    Tools)
4
5     Instead of editing this file, modify BORIS System-File
6     with BORIS and use the AutoCode-Generator once again.
7
8     Generated by : Michael
9                 on : MICHAELNT
10
11    Boris Version    : 6.1.1.218
12    Source-File(s)   : *.BSY from:18.05.1999 14:48:42
13                       THE FILE WAS MODIFIED AND NOT SAVED!!
14      refers to      :
```

```
15                      C:\AUTOCODE\EXAMPLES\DRDFELD.DLL
    from:28.05.1999 13:06:06
16    Timestamp        : 15.11.2001 16:42:49
17
18  ************************************************************
    ***************/
19  /*********************************************************/
20  /*  all #define directives the c-compiler has to know   */
21  /*********************************************************/
22  #define Rgchk(a) (a)
23  #define MUL(a,b,c) Rgchk((a)*(c))
24  #define DIV(a,b,c) Rgchk((a)/(c))
25  /*********************************************************/
26  /*                  #include directives                  */
27  /*********************************************************/
28  #include "C:\temp\DRDDemoTest.h"
29  #include "regdef.h"
30  #include <stdio.h>
31  #include "drdfeld.h"
32  #include <malloc.h>
33  /*********************************************************/
34  /* declarations of global constants that are needed  */
35  /*********************************************************/
36    const SVartyp One=1;
37    const SVartyp SignalMax=1E32;
38    const SVartyp SignalMin=-1E32;
39    const SVartyp HIGHLEVEL=5;
40    const SVartyp LOWLEVEL=0;
41    const SVartyp THRESHOLD=2.5;
42    const SVartyp PI_DIV_2=1.570796327;
43    const SVartyp PI_MUL_2=6.283185307;
44    const SVartyp PI=3.141592654;
45    const SVartyp EXP1=2.718281828;
46  /*********************************************************/
47  /* declarations of global variables that are needed  */
48  /*********************************************************/
49    MEM_ATTRIBUTE_VAR SVartyp INVDELTAT;
50    MEM_ATTRIBUTE_VAR SVartyp X_VEC[MAX_X_VEC][2];
51    MEM_ATTRIBUTE_VAR SVartyp I_VEC[MAX_I_VEC][2];
52    MEM_ATTRIBUTE_VAR unsigned int TimeStepCounter;
```

```
53    MEM_ATTRIBUTE_VAR DRDFELDStruct USER1_V0;
54  /*********************************************************/
55  /* specific global functions such as integrationmethods */
56  /*********************************************************/
57
58
59  /*********************************/
60  /* global functions of the blocks */
61  /*********************************/
62  MEM_ATTRIBUTE void drdfeld_fct(DRDFELDStruct *p, SVartyp i1,
    SVartyp i2,
63                                                    SVartyp
    *o1, SVartyp *o2)
64  { SVartyp x0, y0, xp, yp;
65    int ix,iy;
66
67    if ((i1<p->xmin)||(i1>p->xmax)||(i2<p->ymin)||(i2>p-
    >ymax)) *o2=HIGHLEVEL;
68    else *o2=LOWLEVEL;
69    ix = (int)((i1-p->xmin)/p->dx);
70    iy = (int)((i2-p->ymin)/p->dy);
71    if (ix < 0) ix = 0;
72    if (iy < 0) iy = 0;
73    if (p->interpolate){
74      if (ix > p->nx-2) ix = p->nx-2;
75      if (iy > p->ny-2) iy = p->ny-2;
76      x0 = p->xmin + ix * p->dx;
77      y0 = p->ymin + iy * p->dy;
78      xp = DIV(p->z[ix+1+iy*p->ny]-p->z[ix+iy*p->ny],
    FracPart, p->dx);
79      yp = DIV(p->z[ix+(iy+1)*p->ny]-p->z[ix+iy*p->ny],
    FracPart, p->dy);
80      *o1= p->z[ix+iy*p->ny] + MUL(i1-x0, FracPart, xp) +
    MUL(i2-y0, FracPart, yp);
81    }else{
82      if (ix > p->nx-1) ix = p->nx-1;
83      if (iy > p->ny-1) iy = p->ny-1;
84      *o1= p->z[ix+iy*p->ny];
85    }
86  }
```

```
87
88  MEM_ATTRIBUTE void drdfeld_init(DRDFELDStruct *p, SVartyp
    i1, SVartyp i2,
89                                                  SVartyp
    *o1, SVartyp *o2)
90  { drdfeld_fct(p,i1,i2,o1,o2);}
91
92  MEM_ATTRIBUTE void drdfeld_free(DRDFELDStruct *p)
93  { /* z nur freigeben wenn das Kennfeld dynamisch angelegt
    wurde:*/
94    if (! p->cz) free(p->z);
95  }
96
97  /***************************************/
98  /* the controller initialising function */
99  /***************************************/
100 void DRDDemoTestinitcontrol(SVartyp USER1_0I1,
101                             SVartyp USER1_0I2,
102                             SVartyp *USER1_0O1,
103                             SVartyp *USER1_0O2)
104 {
105
106   {
107     /* initialising the state of the system and used exter-
    nals */
108     unsigned int i;
109     for (i=0; i<MAX_X_VEC; i++){X_VEC[i][0]=0;
    X_VEC[i][1]=0;}
110   }
111   {
112     unsigned int i;
113     for (i=0; i<MAX_I_VEC; i++){I_VEC[i][0]=0;
    I_VEC[i][1]=0;}
114   }
115   {
116     /* set state of the system from the external inputs */
117     I_VEC[0][0]=USER1_0I1;
118     I_VEC[1][0]=USER1_0I2;
119   }
120   USER1_V0.nx=5;
```

```
121   USER1_V0.ny=5;
122   USER1_V0.dx=0,25;
123   USER1_V0.dy=0,25;
124   USER1_V0.xmin=0;
125   USER1_V0.xmax=1;
126   USER1_V0.ymin=0;
127   USER1_V0.ymax=1;
128   USER1_V0.interpolate=1;
129   USER1_V0.cz=0;
130   USER1_V0.z=(SVartyp *) malloc(sizeof(SVartyp)*5*5);
131   USER1_V0.z[0+0* USER1_V0.ny]=11;
132   USER1_V0.z[1+0* USER1_V0.ny]=12;
133   USER1_V0.z[2+0* USER1_V0.ny]=13;
134   USER1_V0.z[3+0* USER1_V0.ny]=14;
135   USER1_V0.z[4+0* USER1_V0.ny]=15;
136   USER1_V0.z[0+1* USER1_V0.ny]=21;
137   USER1_V0.z[1+1* USER1_V0.ny]=22;
138   USER1_V0.z[2+1* USER1_V0.ny]=23;
139   USER1_V0.z[3+1* USER1_V0.ny]=24;
140   USER1_V0.z[4+1* USER1_V0.ny]=25;
141   USER1_V0.z[0+2* USER1_V0.ny]=31;
142   USER1_V0.z[1+2* USER1_V0.ny]=32;
143   USER1_V0.z[2+2* USER1_V0.ny]=33;
144   USER1_V0.z[3+2* USER1_V0.ny]=34;
145   USER1_V0.z[4+2* USER1_V0.ny]=35;
146   USER1_V0.z[0+3* USER1_V0.ny]=41;
147   USER1_V0.z[1+3* USER1_V0.ny]=42;
148   USER1_V0.z[2+3* USER1_V0.ny]=43;
149   USER1_V0.z[3+3* USER1_V0.ny]=44;
150   USER1_V0.z[4+3* USER1_V0.ny]=45;
151   USER1_V0.z[0+4* USER1_V0.ny]=51;
152   USER1_V0.z[1+4* USER1_V0.ny]=52;
153   USER1_V0.z[2+4* USER1_V0.ny]=53;
154   USER1_V0.z[3+4* USER1_V0.ny]=54;
155   USER1_V0.z[4+4* USER1_V0.ny]=55;
156
157
158
159   /* calls of the used functions in the system */
```

```
160   drdfeld_init(&USER1_V0, I_VEC[0][0], I_VEC[1][0],
    &X_VEC[0][0], &X_VEC[1][0]);
161
162   {
163     /* set all external outputs from the state of the system
    */
164     unsigned int i;
165     for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
166     *USER1_0O1=X_VEC[0][0];
167     *USER1_0O2=X_VEC[1][0];
168   }
169 }
170
171 /*********************************/
172 /* the controller function itself */
173 /*********************************/
174 void DRDDemoTestcontrol(SVartyp USER1_0I1,
175                         SVartyp USER1_0I2,
176                         SVartyp *USER1_0O1,
177                         SVartyp *USER1_0O2)
178 {
179
180   {
181     /* set state of the system from the external inputs */
182     unsigned int i;
183     for(i=0; i<MAX_I_VEC; i++) I_VEC[i][1]=I_VEC[i][0];
184     I_VEC[0][0]=USER1_0I1;
185     I_VEC[1][0]=USER1_0I2;
186   }
187   drdfeld_fct(&USER1_V0, I_VEC[0][0], I_VEC[1][0],
    &X_VEC[0][0], &X_VEC[1][0]);
188
189   /* set all external outputs from the state of the system
    */
190   {
191     /* set all external outputs from the state of the system
    */
192     unsigned int i;
193     for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
194     *USER1_0O1=X_VEC[0][0];
```

```
195     *USER1_0O2=X_VEC[1][0];
196   }
197 }
198 void DRDDemoTestfreecontrol(void)
199 {
200   drdfeld_free(&USER1_V0);
201 }
```

Listing 34. *C-Quelltext des 3D-Kennfeldes bei der Einstellung "Konstanten zulässig"*

Die Zuweisung des konstanten Feldes an das Feld *z* ist alles, was in der C-Datei erfolgen muß. Die Definition der Konstanten selbst wurde mit Hilfe der *#>ConstFile*-Anweisung in die Headerdatei portiert (dies ist zwar normalerweise wenig sinnvoll, hier soll aber nur die Wirkungsweise dargestellt werden).

```
1   /*************************************************************
    ***************
2     This file was created by the AutoCode-Generator of the
    program BORIS
3     which is a module of WinFACT (Windows Fuzzy And Control
    Tools)
4
5     Instead of editing this file, modify BORIS System-File
6     with BORIS and use the AutoCode-Generator once again.
7
8     Generated by : Michael
9                on : MICHAELNT
10
11    Boris Version    : 6.1.1.218
12    Source-File(s)   : *.BSY from:18.05.1999 14:48:42
13                       THE FILE WAS MODIFIED AND NOT SAVED!!
14      refers to      :
15                       C:\AUTOCODE\EXAMPLES\DRDFELD.DLL
    from:28.05.1999 13:06:06
16    Timestamp        : 15.11.2001 16:47:06
17
18  *************************************************************
    **************/
19  #ifndef DRDDemoTes
20
21  #define DRDDemoTes DRDDemoTes
22
```

```
23
24  #define MAX_X_VEC 2
25  #define MAX_I_VEC 2
26  #define IntMethod Euler
27  #define DGLIntMethod EulerDGL
28  #define MAX_DYNAMIC_ORDER 8
29  #define MAX_DGLSYS_ORDER 8
30  #define MAX_INPUT 50
31  #define FLOATINGPOINT
32  #define USES_CONST_PARAMS
33
34  typedef double SVartyp;
35  typedef double CastVartyp;
36  #include "regdef.h"
37
38
39  extern const SVartyp One;
40  extern const SVartyp SignalMax;
41  extern const SVartyp SignalMin;
42  extern const SVartyp HIGHLEVEL;
43  extern const SVartyp LOWLEVEL;
44  extern const SVartyp THRESHOLD;
45  extern const SVartyp PI_DIV_2;
46  extern const SVartyp PI_MUL_2;
47  extern const SVartyp PI;
48  extern const SVartyp EXP1;
49
50  const SVartyp USER1_V0_CZ[25]={/*[0][0]*/11,
51  /*[1][0]*/12,
52  /*[2][0]*/13,
53  /*[3][0]*/14,
54  /*[4][0]*/15,
55  /*[0][1]*/21,
56  /*[1][1]*/22,
57  /*[2][1]*/23,
58  /*[3][1]*/24,
59  /*[4][1]*/25,
60  /*[0][2]*/31,
61  /*[1][2]*/32,
62  /*[2][2]*/33,
```

```
63  /*[3][2]*/34,
64  /*[4][2]*/35,
65  /*[0][3]*/41,
66  /*[1][3]*/42,
67  /*[2][3]*/43,
68  /*[3][3]*/44,
69  /*[4][3]*/45,
70  /*[0][4]*/51,
71  /*[1][4]*/52,
72  /*[2][4]*/53,
73  /*[3][4]*/54,
74  /*[4][4]*/55};
75
76  #endif
```

Listing 35. *Konstantendefinition für das 3D-Kennfeld*

Anbindung der Ziel-Hardware

Die Anpassung an die Ziel-Hardware erfolgt zweckmäßigerweise durch Erstellung einer "passenden" I/O-Beschreibungsdatei. Der *Aufbau der I/O-Beschreibungsdatei* ist Gegenstand des gleichnamigen Kapitels. In diesem Abschnitt soll anhand zweier typischer Ziel-Hardwaresysteme eine Implementierung vorgestellt werden.

Im ersten Teil wird eine I/O-Beschreibungsdatei zur Anbindung an eine handelsübliche *PC-Einsteckkarte* (hier beispielhaft die *Advantech PCL-812*; der C-Code kann jedoch an andere Karten leicht angepaßt werden) erstellt. Dabei wird eine Rahmenfunktion vorgestellt, die die Karte im Echtzeitbetrieb (Karte löst über ihren programmierbaren Timer einen Interrupt zum PC aus) unter dem Betriebssystem DOS einsetzt.

Im zweiten Teil werden wir uns einem *Standard-Microcontroller*, dem C166, zuwenden. Dieser soll zur Echtzeitregelung verwendet werden, wobei hier nur auf die Ansteuerung der Analogein- und -ausgänge sowie die Programmierung

des Timer-Interrupts eingegangen wird. Die Programmierung von unterschiedlich privilegierten Interruptebenen wie zum Beispiel eines Notaus-Schalters wird dabei außer acht gelassen.

Der dritte Teil schließlich zeigt darüber hinaus, wie BORIS-Ausgangsblöcke vom Anwender für Ausgaben auf der Ziel-Hardware angepaßt werden können. Dazu wird die Programmierung einer (sehr primitiven) Grafikausgabe auf einem PC erläutert.

Anbindung von PC-Einsteckkarten

Zur Unterstützung einer PC-Einsteckkarte müssen vom Anwender folgende Grundfunktionen realisiert werden:

1. Auslösen einer A/D-Wandlung über Software-Triggerung und Einlesen des gewandelten Wertes.
2. Setzen eines Analogausgangs über den D/A-Wandler.
3. Programmierung der digitalen Ein-/Ausgänge (falls benötigt!).
4. Programmierung des auf der Karte befindlichen Timers, so daß dieser Interrupts zum PC auslöst (sofern Echtzeitbetrieb gewünscht wird).

Nachfolgend werden wir jeden einzelnen Punkt anhand der PCL-812 durchgehen. Die Informationen sind jedoch möglichst allgemein gehalten, damit die Übertragung auf andere Hardwaresysteme - insbesondere Karten anderen Typs - nicht schwer fällt. Alle Routinen befinden sich in der mitgelieferten Datei PCL812.C.

A/D-Wandlung

Die A/D-Wandlung erfolgt in der Regel über einen Analog-Multiplexer. Diesem muß bekannt gemacht werden, welcher analoge Eingangskanal auf den Wandler geschaltet werden soll. Anschließend muß die Wandlung softwaremäßig ausgelöst werden (Software-A/D-Trigger). Dies erfolgt in den meisten Fällen durch Beschreiben eines speziellen Registers auf der Karte. Ist die Wandlung beendet - was bei den meisten Karten durch Auslesen eines bestimmten Bits eines bestimmten Registers festgestellt werden kann (bei anderen Karten muß einfach eine kurze Zeitspanne gewartet werden) -, so muß der gewandelte Wert aus den entsprechenden Registern der Karte gelesen werden.

Es sind also folgende Schritte erforderlich:

1. Auswahl des einzulesenden Kanals durch den Analog-Multiplexer
2. Auslösen der Wandlung durch Beschreiben eines dafür vorgesehenen Registers der Karte
3. Warten, bis die Wandlung abgeschlossen ist
4. Auslesen des gewandelten Wertes

Bei der Implementierung kann also so vorgegangen werden, daß zunächst eine allgemeine Funktion zum Einlesen eines Analogwertes erstellt wird, die als Parameter den zu wandelnden Eingangskanal besitzt. Zusätzlich wird dann für jeden vorhandenen Eingangskanal eine spezifische Funktion erstellt, die diese allgemeine Funktion mit "ihrem" Kanal als Parameter aufruft. Diese Vorgehensweise spart Zeit, ist leichter zu pflegen und verringert die Größe des späteren Compilats bei Verwendung mehrerer Analogeingänge.

Nun zur Implementierung für die Wandlerkarte PCL-812. Das nachfolgende Listing zeigt den Teil der I/O-Beschreibungsdatei PCL812.C, der die Analogeingänge behandelt.

```
1   /*PCL812 AD*/
2   #InsertBlock /*Define addresses*/
3   MEM_ATTRIBUTE SVartyp PCL812AD(int e)
4   { unsigned int i,zw;
5     SVartyp zw1;
6
7     outp(PCL10,e);
8     outp(PCL11,1);
9     outp(PCL12,0);
10    for(i=0xffff;(i>100)&&(inp(PCL5)&16);i--);
11    zw=((unsigned int)(inp(PCL5) & 0x0f)<<8)+inp(PCL4);
12  #ifndef FLOATINGPOINT
13    zw=(CastVartyp)zw-2047;
14    zw1=MUL(10,0,One);    /*(-10V..10V) im internen Format*/
15    zw1=MUL((CastVartyp)zw>>11-FracPart,FracPart,zw1);
16  #else
17    zw1=10*(((CastVartyp)zw-2047)/2048);  /*12 Bit-Wandlung*/
18  #endif
```

```
19    outp(PCL11,6);
20    return zw1;
21  }
22  /*end*/
```

Listing 36. *Funktionscodeabschnitt zum Einlesen eines Analogkanals, der als Parameter übergeben wird*

In Zeile 7 des Listings wird der zu wandelnde Kanal dem Multiplexer bekannt gemacht. Der C-Code in den Zeilen 8 und 9 stellt auf der Karte eine Softwaretriggerung ein und löst die Wandlung aus. In Zeile 10 wird die *for*-Schleife solange ausgeführt, bis das fünfte Bit (Data ready-Bit) der Adresse PCL5 gleich null oder der Schleifenzähler unter 100 gesunken ist (letztere Bedingung dient lediglich dazu, ein "Aufhängen" des Programms zu verhindern, falls sich keine Einsteckkarte im Rechner befindet). Die symbolischen Konstanten, die die Registeradressen der Karte wiedergeben, sind im Funktionscodeabschnitt `/*Define addresses*/` definiert. In Zeile 11 wird der gewandelte Wert eingelesen und in den anschließenden Zeilen (bis Zeile 17) in das entsprechende Signalformat umgewandelt. Zeile 19 setzt die Karte wieder in den Interruptbetrieb zurück.

Nun zu den Funktionscodeabschnitten, die die einzelnen Kanäle wiedergeben. Da es insgesamt 16 Analogkanäle gibt, werden auch 16 Funktionscodeabschnitte benötigt (da jeder Kanal einzeln behandelt werden soll), die den Funktionscodeabschnitt aus dem obigen Listing einbinden. Nachfolgendes Listing zeigt diese Abschnitte.

```
1   /*AD Channel 0*/
2   #usedAddresses: ;
3   #Inputs:1
4   #InsertBlock /*PCL812 AD*/
5   MEM_ATTRIBUTE void ADChannel0(SVartyp *o1)
6   {  *o1=PCL812AD(0);}
7   /*END*/
8
9
10  /*AD Channel 1*/
11  #usedAddresses: ;
12  #Inputs:1
13  #InsertBlock /*PCL812 AD*/
14  MEM_ATTRIBUTE void ADChannel1(SVartyp *o1)
15  { *o1=PCL812AD(1);}
```

```
16  /*END*/
17
18
19  /*AD Channel 2*/
20  #usedAddresses: ;
21  #Inputs:1
22  #InsertBlock /*PCL812 AD*/
23  MEM_ATTRIBUTE void ADChannel2(SVartyp *o1)
24  {  *o1=PCL812AD(2);}
25  /*END*/
26
27
28  /*AD Channel 3*/
29  #usedAddresses: ;
30  #Inputs:1
31  #InsertBlock /*PCL812 AD*/
32  MEM_ATTRIBUTE void ADChannel3(SVartyp *o1)
33  {  *o1=PCL812AD(3);}
34  /*END*/
35
36
37  /*AD Channel 4*/
38  #usedAddresses: ;
39  #Inputs:1
40  #InsertBlock /*PCL812 AD*/
41  MEM_ATTRIBUTE void ADChannel4(SVartyp *o1)
42  {  *o1=PCL812AD(4);}
43  /*END*/
44
45
46  /*AD Channel 5*/
47  #usedAddresses: ;
48  #Inputs:1
49  #InsertBlock /*PCL812 AD*/
50  MEM_ATTRIBUTE void ADChannel5(SVartyp *o1)
51  { *o1=PCL812AD(5);}
52  /*END*/
53
54
55  /*AD Channel 6*/
```

```
56  #usedAddresses: ;
57  #Inputs:1
58  #InsertBlock /*PCL812 AD*/
59  MEM_ATTRIBUTE void ADChannel6(SVartyp *o1)
60  {  *o1=PCL812AD(6);}
61  /*END*/
62
63
64  /*AD Channel 7*/
65  #usedAddresses: ;
66  #Inputs:1
67  #InsertBlock /*PCL812 AD*/
68  MEM_ATTRIBUTE void ADChannel7(SVartyp *o1)
69  {  *o1=PCL812AD(7);}
70  /*END*/
71
72
73  /*AD Channel 8*/
74  #usedAddresses: ;
75  #Inputs:1
76  #InsertBlock /*PCL812 AD*/
77  MEM_ATTRIBUTE void ADChannel8(SVartyp *o1)
78  {  *o1=PCL812AD(8); }
79  /*END*/
80
81
82  /*AD Channel 9*/
83  #usedAddresses: ;
84  #Inputs:1
85  #InsertBlock /*PCL812 AD*/
86  MEM_ATTRIBUTE void ADChannel9(SVartyp *o1)
87  {  *o1=PCL812AD(9); }
88  /*END*/
89
90
91  /*AD Channel 10*/
92  #usedAddresses: ;
93  #Inputs:1
94  #InsertBlock /*PCL812 AD*/
95  MEM_ATTRIBUTE void ADChannel10(SVartyp *o1)
```

```
96   {  *o1=PCL812AD(10);}
97   /*END*/
98
99
100  /*AD Channel 11*/
101  #usedAddresses: ;
102  #Inputs:1
103  #InsertBlock /*PCL812 AD*/
104  MEM_ATTRIBUTE void ADChannel11(SVartyp *o1)
105  {  *o1=PCL812AD(11);}
106  /*END*/
107
108
109  /*AD Channel 12*/
110  #usedAddresses: ;
111  #Inputs:1
112  #InsertBlock /*PCL812 AD*/
113  MEM_ATTRIBUTE void ADChannel12(SVartyp *o1)
114  {  *o1=PCL812AD(12);}
115  /*END*/
116
117
118  /*AD Channel 13*/
119  #usedAddresses: ;
120  #Inputs:1
121  #InsertBlock /*PCL812 AD*/
122  MEM_ATTRIBUTE void ADChannel13(SVartyp *o1)
123  { *o1=PCL812AD(13);}
124  /*END*/
125
126
127  /*AD Channel 14*/
128  #usedAddresses: ;
129  #Inputs:1
130  #InsertBlock /*PCL812 AD*/
131  MEM_ATTRIBUTE void ADChannel14(SVartyp *o1)
132  {  *o1=PCL812AD(14);}
133  /*END*/
134
135
```

```
136  /*AD Channel 15*/
137  #usedAddresses: ;
138  #Inputs:1
139  #InsertBlock /*PCL812 AD*/
140  MEM_ATTRIBUTE void ADChannel15(SVartyp *o1)
141  {  *o1=PCL812AD(15);}
142  /*END*/
```

Listing 37. *Funktionscodeabschnitte der Analogeingänge*

Wie dem Listing unschwer zu entnehmen ist, gleichen sich alle Funktionscodeabschnitte bis auf die Nummer des im Funktionsaufruf `PCL812AD(.)` verwendeten Kanals. Da ein Kanal auch mehrfach in einem System auftreten darf, werden die Funktionen nicht gegeneinander verriegelt.

D/A-Wandlung

Die D/A-Wandlung ist, da hier kein Multiplexer verwendet wird und auch nicht auf eine Wandlung gewartet werden muß, einfacher zu programmieren. Es ist lediglich das entsprechende Signal in den Bereich zu konvertieren, der für die Wandlung relevant ist. Bei der PCL-812 wird der maximal für die Wandlung zulässige Signalwert 5 auf den Wert 4095 abgebildet, der Signalwert 0 auf den Wert 0 (hier wird von einem Eingangsspannungsbereich von 0-5 V ausgegangen). Es liegt also eine lineare Abbildung vor, was auch dem nachfolgenden Quellcodeabschnitt der Datei PCL812.C entnommen werden kann.

```
1   /*DA Channel 0*/
2   #usedAddresses: zero;
3   #outputs:1
4   #InsertBlock /*Define addresses*/
5   MEM_ATTRIBUTE void DAChannel0(SVartyp *o1)
6   { char a,b;
7     SVartyp zw,zw1;
8
9     zw=*o1;
10  #ifndef FLOATINGPOINT
11    zw1=MUL(5,0,One);    /*5 in signal-format*/
12    if (zw>zw1)zw=zw1;
13    else if (zw<0)zw=0;  /*0<=zw<=5*/
14  #else
15    if (zw>5) zw=5;
```

```
16    else if (zw<0)zw=0;
17  #endif
18    zw=MUL(4095,FracPart,zw)/5;
19    a=(char)((int)zw&0xff);
20    b=(char)((int)zw>>8);
21    outp(PCL4+0,a);
22    outp(PCL5+0,b);
23  }
24  /*END*/
25
26  /*DA Channel 1*/
27  #usedAddresses: one;
28  #outputs:1
29  #InsertBlock /*Define addresses*/
30  MEM_ATTRIBUTE void DAChannel1(SVartyp *o1)
31  { char a,b;
32    SVartyp zw,zw1;
33
34    zw=*o1;
35  #ifndef FLOATINGPOINT
36    zw1=MUL(5,0,One);    /*5 in signal-format*/
37    if (zw>zw1)zw=zw1;
38    else if (zw<0)zw=0;  /*0<=zw<=5*/
39  #else
40    if (zw>5) zw=5;
41    else if (zw<0)zw=0;
42  #endif
43    zw=MUL(4095,FracPart,zw)/5;
44    a=(char)((int)zw&0xff);
45    b=(char)((int)zw>>8);
46    outp(PCL4+2,a);
47    outp(PCL5+2,b);
48  }
49  /*END*/
```

Listing 38. *Implementierung der Funktionen für die beiden Analogausgänge der PCL-812*

Da die PCL-812 über zwei Analogausgänge verfügt, können zwei Funktionscodeabschnitte erstellt werden, die jeweils einen analogen Ausgangskanal verwenden. Dies ist im obigen Listing realisiert worden. Beide Funktionscode-

abschnitte sind gegen sich selbst verriegelt, so daß unter BORIS nur jeweils ein Funktionscodeabschnitt im System verfügbar ist. Denkbar ist aber auch eine Funktion mit zwei Parametern, von denen der erste den ersten Analogausgang und der zweite den zweiten Analogausgang setzt. Also sollte hier der Verriegelungsmechanismus eingesetzt werden, wie dies das untenstehende Listing zeigt.

```
1   /*DA Channel (zero, one)*/
2   #usedAddresses: zero,one;
3   #outputs:2
4   #InsertBlock /*Define addresses*/
5   MEM_ATTRIBUTE void DAChannel01(SVartyp *o1, SVartyp *o2)
6   {...}
7   /*END*/
```

Listing 39. *Mögliche Funktion, die beide Analogausgänge setzt*

Wir wollen uns aber die Mühe sparen und begnügen uns mit den in Listing 38 dargestellten Funktionen.

Digitale Ein-/Ausgänge

Wir wollen uns hier mit dem Einlesen einzelner Bits und dem Setzen bzw. Rücksetzen einzelner Ausgangsbits eines Ports befassen. Zuächst stellen wir einen Funktionscodeabschnitt vor, der von einem digitalen Eingangskanal das LSB (Bit 0) prüft und entsprechend High- oder Low-Pegel ausgibt.

```
1   /*Binary In 0*/
2   #usedAddresses: ;
3   #inputs:1
4   #InsertBlock /*Define addresses*/
5   MEM_ATTRIBUTE void BinaryIn0(SVartyp *o1)
6   { unsigned int zw;
7     zw=inp(PCL7)<<8 |inp(PCL6);
8     *o1=(zw&1)?HIGHLEVEL:LOWLEVEL;
9   }
10  /*end*/
```

Listing 40. *Binärer Eingangskanal 0*

Die restlichen 15 binären Eingangskanäle sind in gleicher Weise aufgebaut, so daß wir uns ein längeres Listing sparen können.

Gehen wir nun über zu den binären Ausgängen. Die PCL-812 hat 16 digitale Ausgänge. Jeder Ausgang wird über eine der beiden symbolischen Adressen

PCL13 und PLC14, die im Funktionscodeabschnitt `/*Define addresses*/` definiert sind, gesetzt bzw. rückgesetzt. Es ist also erforderlich, sich zu merken, welcher Ausgang schon gesetzt war und welcher nicht, damit ein Schreiben in die beiden 8 Bit-breiten Register nur den entsprechenden binären Ausgang manipuliert und die anderen unbeeinflußt läßt. Dies kann wiederum am besten durch eine Funktion erledigt werden, der die Nummer des Bits und der zu setzende Wert übergeben werden. Diese Funktion (Funktionscodeabschnitt `/*DigitalOutputs*/` des folgenden Listings) muß durch die Funktionen, die die einzelnen binären Ausgänge wiedergeben, aufgerufen werden. Im folgenden Listing soll nur ein binärer Ausgang (Funktionscodeabschnitt `/*Binary Out 0*/` des folgenden Listings) dargestellt werden. Die übrigen 15 Eingänge sind in gleicher Weise implementiert.

```
1   /*DigitalOutputs*/
2   MEM_ATTRIBUTE void DigitalOutput(char s, char channel)
3   { char a,b;
4     if (s>THRESHOLD)
5       DigitalOut|=1<<channel;
6     else
7       DigitalOut&=~(1<<channel);
8     a=DigitalOut;
9     b=DigitalOut>>8 & 0x00FF;
10    outp(PCL13,a);
11    outp(PCL14,b);
12  }
13  /*end*/
14
15  /*Binary Out 0*/
16  #usedAddresses: B0;
17  #outputs:1
18  #InsertBlock /*Define addresses*/
19  #InsertBlock /*DigitalOutputs*/
20  MEM_ATTRIBUTE void BinaryOut0(SVartyp *o1)
21  {
22    DigitalOutput(*o1,0);
23  }
24  /*end*/
```

Listing 41. *Implementierung der Funktionscodeabschnitte für die binären Ausgänge der PCL-812*

Rahmenfunktion für den Echtzeitbetrieb

Damit überhaupt Echtzeitverhalten auf einer Hardware erreicht wird, muß auf der Ziel-Hardware ein Zeitgeber (Timer) vorhanden sein. Dieser muß so programmiert werden, daß er mit einer vorgebbaren Frequenz einen Interrupt auf der Recheneinheit auslöst, auf der das ausführbare Programm läuft. Dies ist im Fall der Karte PCL-812 der PC. Nachfolgendes Listing zeigt die komplette Rahmenfunktion, die anschließend näher besprochen werden soll.

```
1   /*used by main for realtime applications*/
2
3   /*****************************************/
4   /*  Interrupt Ctrl-C und timer routinen  */
5   /*****************************************/
6
7
8   MEM_ATTRIBUTE int FINISH=0;
9
10  void interrupt (*old_keyboard)(void);
11  void interrupt (*old_irq7)(void);
12
13  void InitPCLTimer(void)
14  { SVartyp a,b,c,w;
15    unsigned int zw1,zw,i;
16
17    /*calculate timer registers by frequency*/
18    w=#InsertINVDELTAT;
19    i=0;
20    a=b=One;
21    do{
22      a=(b+a)/2;
23      b=DIV(w,FracPart,a);
24      c=(a>b)?a-b:b-a;
25      i++;
26    }while((c>1)&&(i<23));
27    if(a<0){a=-a;b=-b;}
28    #ifndef FLOATINGPOINT
29      /*PCL812 works with an on board frequency of 2MHz */
30      if (b>a) {
```

```
31        zw=1414/(a>>FracPart);
32        zw1=1414/(b>>FracPart);
33      }else {
34        zw=1414/(b>>FracPart);
35        zw1=1414/(a>>FracPart);}
36    #else
37      if (b>a) {zw=1414.2135/a; zw1=1414.2135/b;}
38      else {zw=1414.2135/b; zw1=1414.2135/a;}
39    #endif
40    /*values for the timer registers are now in zw and in zw1
        zw1*zw=1/samplingfrequency*/
41    /*configure PCL812 for interrupt*/
42    outp(PCL11,6);
43    outp(PCL12,0);
44    /*set timer registers with values stored in zw and zw1*/
45    outp(PCL3,0xb4);
46    outp(PCL2,(char)(zw&0x00ff));
47    outp(PCL2,(char)(zw>>8));
48    outp(PCL3,0x74);
49    outp(PCL1,(char)(zw1&0x00ff));
50    outp(PCL1,(char)(zw1>>8));
51    /*Initiliaze the PC to work with IRQ 7 */
52    outp(0x20,0x20);
53    outp(0x21,inp(0x21)&0x7d);
54    /*start Timer on PCL812*/
55    outp(PCL8,0xff);
56  }
57
58
59  void interrupt keyboard()
60  { FINISH=1;
61    old_keyboard();
62  }
63
64  void interrupt IRQseven()
65  {
66    if(!Flag){                  /*Interrupt overflow ?*/
67      Flag=1;
68      outp(PCL8,0xff);          /*Reset PCL for interrupt */
69      outp(0x20,0x20);          /*Reset PC for interrupt*/
```

```
70      enable();
71      #InsertControlCall;
72      Flag=0;
73    } else if(Flag==1) FINISH=2; /*Interrupt overflow
                                     occured*/
74  }
75  /*END*/
76
77  /*main for realtime applications*/
78  #InsertBlock /*used by main for realtime applications*/
79
80  /*************************************/
81  /*    main for realtime applications     */
82  /*************************************/
83
84  MEM_ATTRIBUTE int main(void)
85  { char ps[80];
86    int i;
87    unsigned int far * screen;
88
89    screen=(unsigned int far * )MK_FP(0xb800,0);
90    old_keyboard = getvect(KEYB);
91    setvect(KEYB, keyboard);
92    old_irq7 = getvect(IRQ7);
93    setvect(IRQ7, IRQseven);
94    #InsertInitControlCall;
95
96    InitPCLTimer();
97    /*FINISH = dr cken von CTRL-C */
98    while(FINISH!=1) {
99      if (FINISH==2){    /*Interrupt overflow occured so*/
100       FINISH=0;        /*calculate new lower sampling
                             frequnezy*/
101       #InsertFreeControlCall;
102       #InsertINVDELTAT-=One;
103       #InsertInitControlCall;
104       InitPCLTimer();
105     }
106     #ifdef FLOATINGPOINT
```

```
107       sprintf(ps,"momentane Abtastzeit: %f
                     \r",(float)1/#InsertINVDELTAT);
108     #else
109       sprintf(ps,"momentane Abtastzeit: 1/%lu
                     \r",(long)#InsertINVDELTAT);
110     #endif
111     for(i=0;ps[i+1]!=0;i++)
112       screen[i]=(unsigned int)ps[i]+0x7900;
113   }
114
115   setvect(KEYB, old_keyboard);
116   setvect(IRQ7, old_irq7);
117   #InsertFreeControlCall;
118   return 0;
119 }
120 /*end*/
```

Listing 42. *Implementierung der Echtzeit-Rahmenfunktion für die PCL-812*

Der Funktionscodeabschnitt enthält folgende wesentliche Elemente:

1. Eine Initialisierungsfunktion *InitPCLTimer* (Zeilen 13-57). Diese Funktion nimmt zunächst in Abhängigkeit von der eingestellten Abtastzeit die Berechnung der Werte der Timer-Register vor (Zeilen 17-39). Anschließend wird die Karte derart initialisiert, daß sie einen Interrupt auslöst, wenn die Timer auf null heruntergezählt worden sind (timergesteuerter Interrupt, Zeilen 42 und 43) und die Timer-Register werden mit den zuvor berechneten Werten gesetzt (Zeilen 45-50). Der letzte Teil der Funktion sorgt dafür, daß der PC den Interrupt IRQ7 zuläßt (Zeilen 52 und 53) und der Timer gestartet wird (Zeile 55).
2. Da bei unserer Anwendung das Programm auf dem PC abläuft, muß dafür gesorgt werden, daß der Interrupt zum PC weitergeleitet wird. Ebenso muß der PC die Interruptverarbeitung zulassen und eine entsprechende Routine ausführen, wenn ein Interrupt eintrifft (Interrupt-Routine, Zeilen 64-74). Die Adresse der Interrupt-Routine wird in die Interruptvektortabelle eingetragen. Dabei wird die Adresse der alten Interrupt-Routine gespeichert (Zeilen 93 und 92). Die neue Interrupt-Routine selbst besteht im wesentlichen aus dem Aufruf der Systemfunktion (Zeile 71).

3. Einen Programmteil, der die alten Interrupt-Routinen des verwendeten Interrupts wieder einstellt und das Programm beendet (Zeilen 115 und 116).

4. Die `main`-Funktion, die die obigen Funktionen und die Init-Systemfunktion in geeigneter Weise aufruft und so das System startet (Zeilen 94-113).

Bei genauerer Analyse weist unsere Rahmenfunktion eine weitere Besonderheit auf: Da bei regelungstechnischen Anwendungen meist eine möglichst niedrige Abtastzeit erwünscht ist, die kleinstmögliche Abtastzeit aber in der Regel vorab nicht bekannt ist, stellt unsere Rahmenfunktion diese - ausgehend von der unter BORIS eingestellten Zeit - *automatisch* ein. Hierzu dienen die Zeilen 99-105 in Verbindung mit dem Flag *FINISH*. Ist die aktuelle Abtastzeit auf der Hardware nicht realisierbar (erkennbar daran, daß während der Interrupt-Verarbeitung ein weiterer Interrupt auftritt; Listing Zeile 66 und 73), so wird diese schrittweise erhöht. Dazu muß

1. die Interruptverarbeitung eingestellt werden (wird durch Nicht-Rücksetzen des PCL- und PC-Interrupts realisiert),

2. das System freigegeben werden, damit Speicherallokationen, die in Abhängigkeit der Abtastfrequenz stehen, korrigiert werden (falls keinerlei Abhängigkeiten entstehen können, kann dieser Punkt entfallen),

3. dann die Abtastzeit erhöht,

4. das System neu initialisiert (falls die Freigabe des Systems nicht notwendig war, entfällt auch dieser Punkt),

5. der Timer-Interrupt für die neue Abtastzeit eingestellt und

6. die Interruptfreigabe geschaltet werden.

Die aktuelle Abtastzeit wird zur Kontrolle zusätzlich auf dem Bildschirm ausgegeben (Zeile 111-112). Dabei ist es wichtig, daß direkt in den Bildschirmspeicher geschrieben wird, um Interrupts während dieser Phase zuzulassen. Ist eine automatische Anpassung der Abtastzeit nicht erforderlich, so können die entsprechenden Programmteile natürlich entfallen; der Programmcode vereinfacht sich dann erheblich.

Beispiel

Als letztes wollen wir eine Struktur zur Verwendung der PCL-812 als PID-Regler vorstellen. Dabei soll von Analogeingang 0 der Sollwert und von Ana-

logeingang 1 der Istwert der zu regelnden Strecke eingelesen werden. Der Stellwert wird auf dem Analogausgang 0 ausgegeben.

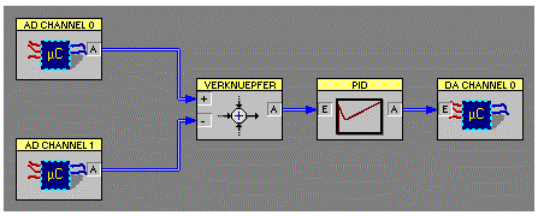

Bild 39. *Struktur zur Realisierung eines PID-Reglers auf der PCL-812*

Um den Aufbau dieser Struktur nachzuvollziehen, ist es erforderlich, daß als I/O-Beschreibungsdatei die Datei PCL812.C eingestellt wurde. Erst nach Einstellung dieser Beschreibungsdatei sind die C-Code Ein-/Ausgangsblöcke parametrierbar!

Ein Simulieren der in Bild 39 abgebildeten Struktur unter BORIS ist nicht sinnvoll, da die C-Code-Eingangsblöcke immer eine 0 als Wert an ihren nachfolgenden Block liefern. Ebenso zeigen die C-Code-Ausgangsblöcke keine Reaktion auf ihre Eingangswerte.

Anbindung an Microcontroller-Systeme

Die Anbindung an Microcontroller-Systeme kann auf nahezu gleiche Art erfolgen wie bei PC-Einsteckkarten. Wir wollen daher nicht mehr alles im Detail darlegen, sondern uns auf den wesentlichen Unterschied beschränken. Das Programm einschließlich der Interrupt-Routine wird nicht auf dem PC ausgeführt, sondern auf der Ziel-Hardware selbst. Dies ist eine Vereinfachung gegenüber der vorher besprochenen Anbindung von PC-Einsteckkarten. Von nun an wollen wir uns auf den Microcontroller 80C166 beziehen. Als C-Compiler wurde der C166-Compiler der Firma KEIL Elektronik eingesetzt.

A/D-Wandlung

Die A/D-Wandlung kann unter Verwendung der symbolischen Konstanten erfolgen, die in der dem Funktionscodeabschnitt `/*Define addresses*/` hinzugebundenen Datei *reg166.h* definiert sind.

```
1    /*AD166*/
2   #InsertBlock /*Define addresses*/
3   MEM_ATTRIBUTE SVartyp AD166(int e)
4   { unsigned int i,zw;
5     SVartyp zw1;
6
7     ADCON = e;      /* select A/D input,single conv.*/
8     ADST = 1;       /* start conversion             */
9     while (ADBSY);  /* wait for A/D result          */
10    zw=ADDAT&0x03FF;
11  #ifndef FLOATINGPOINT
12    if (11>FracPart) zw>>=11-FracPart;
13    else zw<<=FracPart-11;
14    zw1=MUL(5,0,One);
15    zw1=MUL(zw,FracPart,zw1);
16  #else
17    zw1=5.0*zw/1023;
18  #endif
19    return zw1;
20  }
21  /*end*/
22
23  /*AD In P50*/
24  #usedAddresses: ;
25  #Inputs:1
26  #InsertBlock /*Define addresses*/
27  #InsertBlock /*AD166*/
28  MEM_ATTRIBUTE void ADInP50(SVartyp *i1)
29  {  *i1=AD166(0);}
30  /*END*/
```

Listing 43. *Implementierung eines A/D-Kanals für den 80C166*

Da der 80C166 über zehn Analogeingänge verfügt (in obigem Listing ist nur die Implementierung des Kanals 0 gezeigt; die Programmierung der übrigen Kanäle erfolgt aber in gleicher Weise) und nur der zu wandelnde Analogkanal an den Multiplexer übergeben werden muß, ist es zweckmäßig, für die zehn Funktionscodeabschnitte, die die A/D-Kanäle darstellen, eine Funktion zu schreiben, der nur die entsprechende Kanalnummer übergeben wird. Diese liefert dann das Wandlungsergebnis. Die entsprechende Funktion ist im Abschnitt /*AD166*/ realisiert worden.

D/A-Wandlung

Da der 80C166 keine internen D/A-Wandler aufweist, wird einfach Port 2 des Controllers verwendet. Die unteren 8 Bit sollen einen, die oberen 8 Bit einen anderen D/A Wandler speisen (die Wandlerschaltung muß extern aufgebaut werden; entsprechende Schaltungen unter Verwendung von Standard-D/A-Wandlern findet man in der Literatur). Die Ausgabe des Signalwertes an den Port erfordert, da nur jeweils die obere oder untere 8-Bit-Hälfte beeinflußt werden soll, eine Variable, in der der letzte Zustand des P2-Registers gespeichert wird. Die hier verwendete Wandlerschaltung erzeugt, wenn alle 8 Bits High-Pegel führen, 2.5 Volt am Analogausgang.

```
1   /*DA Out on P2.0-P2.7*/
2   #usedAddresses: Lo(P2);
3   #outputs:1
4   #InsertBlock /*Define addresses*/
5   MEM_ATTRIBUTE void DA_P2_0To7(SVartyp *o1)
6   {
7     SVartyp zw,zw1;
8     zw=*o1;
9   #ifndef FLOATINGPOINT
10    zw1=MUL(5,0,One>>1); /*2.5 in signal-format*/
11    if (zw>zw1)zw=zw1;
12    else if (zw<0)zw=0;  /*0<=zw<=2.5*/
13  #else
14    if (zw>2.499) zw=2.499;
15    else if (zw<0)zw=0;
16  #endif
17    DP2=0xffff;
    /*port 2 is set as output port*/
18    PORT_P2&=0xff00;       /*erase low byte*/
19    PORT_P2|=(int)(MUL(511,FracPart,zw)/5);
20    P2=PORT_P2;
21  }
22  /*END*/
23
24  /*DA Out on P2.8-P2.15*/
25  #usedAddresses: Hi(P2);
26  #outputs:1
27  #InsertBlock /*Define addresses*/
```

```
28  MEM_ATTRIBUTE void DA_P2_8To15(SVartyp *o1)
29  {
30    SVartyp zw,zw1;
31    zw=*o1;
32  #ifndef FLOATINGPOINT
33    zw1=MUL(5,0,One>>1); /*2.5 in signal-format*/
34    if (zw>zw1)zw=zw1;
35    else if (zw<0)zw=0;  /*0<=zw<=2.5*/
36  #else
37    if (zw>2.499) zw=2.499;
38    else if (zw<0)zw=0;
39  #endif
40    PORT_P2&=0x00ff;       /*erase high byte*/
41    PORT_P2|=(int)(MUL(511,FracPart,zw)/5)<<8;
42    P2=PORT_P2;
43  }
44  /*END*/
```

Listing 44. *Ausgabe von Signalwerten an Port 2 (Low-Byte bzw. High-Byte) zur Speisung eines DA-Wandlers mit 2.5V Ausgangsspannung*

Binäre Ein-/Ausgänge

Die Programmierung binärer Ein- und Ausgänge zeigt das folgende Listing.

```
1   /*Binary In P3.0*/
2   #usedAddresses: ;
3   #inputs:1
4   #InsertBlock /*Define addresses*/
5   MEM_ATTRIBUTE void BitP3_0(SVartyp *i1)
6   { *i1=(P3&0x0001)?HIGHLEVEL:LOWLEVEL;}
7   /*END*/
8
9    /*Binary Out P3.8*/
10  #usedAddresses: P3.8;
11  #Outputs:1
12  #InsertBlock /*Define addresses*/
13  MEM_ATTRIBUTE void SetBitP3_8(SVartyp *i1)
14  { if (*i1>=THRESHOLD) P3|=0x0100; else P3&=~0x0100; }
15  /*END*/
```

Listing 45. *Programmierung binärer Ein-/Ausgänge*

Da der 80C166 frei als Ein- oder Ausgänge konfigurierbare digitale Kanäle aufweist, müssen zunächst die Ports entsprechend ihrer Funktion als Eingang oder als Ausgang konfiguriert werden, was in den *main*-Funktionen getätigt wird (s. u.).

Rahmenfunktion für den Echtzeitbetrieb

Die Programmierung des Timers zur Auslösung eines Interrupts mit einer vorgebbaren Frequenz ist im nachfolgenden Listing zu sehen, das anschließend näher besprochen werden soll.

```
1   /*Define addresses*/
2   #include <reg166.h>
3   MEM_ATTRIBUTE unsigned int PORT_P2=0;
4   /*end*/
5
6
7
8   /*used by main for realtime applications*/
9
10  /*****************************************/
11  /*          interrupt routines           */
12  /*****************************************/
13  void InitTimer0(void)
14  { char a;
15    int i,j;
16
17    j=i=0;
18    a=7;
19    while ((i<=j)&&(a>=0)){
20      i=j;
21      #ifndef FLOATINGPOINT
22        j=(2500000L>>a)/(#InsertINVDELTAT>>FracPart);
23      #else
24       j=(2500000L>>a)/(#InsertINVDELTAT);
25      #endif
26      a--;
27    }
28    if (j>i)i=j;
29    a++;
```

```
30    T0REL = -i;        /* set reload value */
31    T0    = -i;
32    T0IC  = 0x44;      /* set T0IE and ILVL = 1 */
33    IEN   = 1;         /* set global interrupt enable flag */
34    T01CON = 0x40+a;      /* start timer 0 */
35  }
36
37
38  void timer0(void) interrupt 0x20 using INTREGS
39   /* Int Vector at 0080H, other Reg Bank */
40  { IEN=0;  #InsertControlCall;
41    if (T0IC&0x80){
42      #InsertINVDELTAT-=One;
43      InitTimer0();
44    }
45    IEN=1;
46  }
47  /*END*/
48
49
50  /*realtime main*/
51
52  #InsertBlock /*Define addresses*/
53
54  #InsertBlock /*used By main for realtime applications*/
55
56  /***************************************/
57  /*    main for realtime applications      */
58  /***************************************/
59  MEM_ATTRIBUTE int main(void)
60  {
61    #InsertInitControlCall;
62    InitTimer0();
63    /*port 1 is set as output (hi byte) and input (lo byte):*/
64    DP1=0xff00;
65    /*port 2 is set as output port*/
66    DP2=0xffff;
67    /*port 3 is set as output (hi byte) and input (lo byte)
        P3.3 is PWM*/
68    DP3=0xff08;
```

```
69    T2CON=0x0025;  /*set PWM function*/
70    T4CON=0x0026;
71    T2=1000;
72    T4=2000;
73    T2R=1;
74    T4R=1;
75    T4IC=T2IC=0;
76
77    T3CON=0x0000;
78    T3OTL=1;
79    T3OE=1;
80    T3IC=0;
81    T3UD=1;
82    P3|=0x0008;     /*end of PWM settings*/
83    while(1) ;
84  }
85  /*end*/
```

Listing 46. *Rahmenfunktion für den Echtzeitbetrieb des 80C166*

Der Funktionscodeabschnitt enthält folgende wesentliche Elemente:

1. Eine Initialisierungsfunktion *InitTimer0* (Zeilen 13-35). Diese Funktion nimmt zunächst in Abhängigkeit von der eingestellten Abtastzeit die Ermittlung der Werte der Timer-Register vor (Zeilen 18-29). Anschließend werden die Register mit den Werten gesetzt (Zeile 30 und 31) und die Interrupt-Priorität eingestellt (Zeile 32). Der Interrupt ist nun zulässig (Zeile 33) und der Timer wird gestartet (Zeile 34).
2. Die Interrupt-Routine (Zeile 38-46) übernimmt in diesem Fall, da sich mit Hilfe des Interrupt-Control-Registers `TOIC` prüfen läßt, ob ein Interrupt-Überlauf eingetreten ist, ggfls. selbst die Verringerung der Abtastfrequenz (Zeilen 41-44).
3. Die `main`-Funktion, die die Init-Systemfunktion und die `InitTimer0()`-Funktion aufruft und so das System startet (Zeilen 59-84). In ihr werden nun die digitalen Ports des 80C166 für die Ein- oder Ausgabe konfiguriert und Einstellungen für den PWM-Ausgang vorgenommen (soll hier nicht weiter beschrieben werden). Letztendlich gerät das Programm in eine Endlosschleife, die nichts anderes macht, als zu warten, daß ein Interrupt sie für einen kurzen Augenblick unterbricht.

Anpassung von Ausgangsblöcken

Die meisten BORIS-Ausgangsblöcke (z. B. Analoginstrument, Digitalanzeige usw.) sind im C-Code zunächst "nackt" und müssen bei Bedarf vom Anwender "gefüllt" werden, damit sie auf der Ziel-Hardware ihre entsprechenden Funktionen übernehmen können. In diesem Kapitel soll eine "Primitiv"-Realisierung des Oszilloskops gezeigt werden. Zunächst begutachten wir, was uns der AutoCode-Generator in Bezug auf ein Oszilloskop liefert, ohne selber Hand angelegt zu haben. Dazu muß mindestens einer der Eingänge am Oszilloskop-Block mit einem zu C-Code zu generierenden Block versehen sein. Folgende Systemstruktur kann beispielsweise verwendet werden:

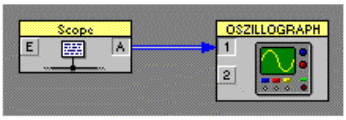

Bild 40. *Struktur zur Erzeugung eines Ausgangsblockes mit einem externen Eingang; diesen stellt das Label dar, da Ausgangsblöcke selbst keine externen Eingänge erzeugen können (s. o.)*

Diese erzeugt bei Code-Generierung folgenden Quelltext:

```
/*********************************************************/
/*  all #define directives the c-compiler has to know   */
/*********************************************************/
#define Rgchk(a) (a)
#define MUL(a,b,c) Rgchk((a)*(c))
#define DIV(a,b,c) Rgchk((a)/(c))
/*********************************************************/
/*                  #include directives                  */
/*********************************************************/
#include "\temp\test.h"
#include "regdef.h"
#include <stdio.h>
#include "outputs.h"
#include <stdarg.h>
#include <string.h>
/*********************************************************/
/* declarations of global constants that are needed  */
/*********************************************************/
  const SVartyp One=1;
  const SVartyp SignalMax=1E32;
  const SVartyp SignalMin=-1E32;
  const SVartyp HIGHLEVEL=5;
  const SVartyp LOWLEVEL=0;
  const SVartyp THRESHOLD=2.5;
```

```
  const SVartyp PI_DIV_2=1.570796327;
  const SVartyp PI_MUL_2=6.283185307;
  const SVartyp PI=3.141592654;
  const SVartyp EXP1=2.718281828;
/*****************************************************/
/* declarations of global variables that are needed  */
/*****************************************************/
  MEM_ATTRIBUTE_VAR SVartyp INVDELTAT;
  MEM_ATTRIBUTE_VAR SVartyp X_VEC[MAX_X_VEC][2];
  MEM_ATTRIBUTE_VAR SVartyp I_VEC[MAX_I_VEC][2];
  MEM_ATTRIBUTE_VAR unsigned int TimeStepCounter;
  MEM_ATTRIBUTE_VAR OSCILLOSCOPEStruct OSZILLOGRAPH_V1;
/*****************************************************/
/* specific global functions such as integrationmethods */
/*****************************************************/


/*********************************/
/* global functions of the blocks */
/*********************************/
MEM_ATTRIBUTE void Oscilloscope_init(OSCILLOSCOPEStruct
*p,SVartyp *e1,...)
{}
MEM_ATTRIBUTE void Oscilloscope_fct(OSCILLOSCOPEStruct
*p,SVartyp *e1,...)
{}
MEM_ATTRIBUTE void Oscilloscope_free(OSCILLOSCOPEStruct *p)
{}

/*************************************/
/* the controller initialising function */
/*************************************/
void testinitcontrol(SVartyp LABEL_0I1)
{

  {
    /* initialising the state of the system and used externals
*/
    unsigned int i;
    for (i=0; i<MAX_X_VEC; i++){X_VEC[i][0]=0; X_VEC[i][1]=0;}
  }
  {
    unsigned int i;
    for (i=0; i<MAX_I_VEC; i++){I_VEC[i][0]=0; I_VEC[i][1]=0;}
  }
  {
    /* set state of the system from the external inputs */
    I_VEC[0][0]=LABEL_0I1;
  }
  OSZILLOGRAPH_V1.name[0]=0;

strncat(OSZILLOGRAPH_V1.name,"OSZILLOGRAPH",sizeof(OSZILLOGRAPH
_V1.name)-1);
```

```
  /* calls of the used functions in the system */
  X_VEC[0][0]=I_VEC[0][0];
  Oscilloscope_init(&OSZILLOGRAPH_V1,
                    &X_VEC[0][0],NULL);

  {
    /* set all external outputs from the state of the system */
    unsigned int i;
    for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
  }
}

/*********************************/
/* the controller function itself */
/*********************************/
void testcontrol(SVartyp LABEL_0I1)
{

  {
    /* set state of the system from the external inputs */
    unsigned int i;
    for(i=0; i<MAX_I_VEC; i++) I_VEC[i][1]=I_VEC[i][0];
    I_VEC[0][0]=LABEL_0I1;
  }
  X_VEC[0][0]=I_VEC[0][0];
  Oscilloscope_fct(&OSZILLOGRAPH_V1,
                   &X_VEC[0][0],NULL);

  /* set all external outputs from the state of the system */
  {
    /* set all external outputs from the state of the system */
    unsigned int i;
    for(i=0; i<MAX_X_VEC; i++) X_VEC[i][1]=X_VEC[i][0];
  }
}
void testfreecontrol(void)
{
  Oscilloscope_free(&OSZILLOGRAPH_V1);
}
```

Listing 47. *Testquelltext, um herauszufinden, welche Zeilen vom AutoCode-Generator bei obiger Struktur für das Oszilloskop erzeugt werden (entsprechende Zeilen wurden hervorgehoben).*

Wie zu sehen ist, müssen lediglich die Parameterstruktur und die Funktionen des Oszilloskops gefüllt werden. Die Funktionsimplementierungen hängen natürlich von der verwendeten Hardware, dem darauf befindlichen Betriebssystem und dem Compiler ab. Generell empfiehlt sich folgende Vorgehensweise: In *Oscilloscope_init* muß ein Ausgabegerät vorbereitet werden und - um auf dieses in *Oscilloscope_fct* zuzugreifen - ein entsprechender Wert in der Parameterstruktur abgelegt werden. Dieser Wert, der als Bezug zum Ausgabekon-

text verwendet wird, kann in der Funktion *Oscilloscope_free* zur Freigabe des Ausgabegerätes verwendet werden. Besitzt Ihr System nur einen Ausgabekontext (bei Windows kann jedes Fenster als ein solcher dienen), so müssen Sie durch eine entsprechende Programmierlogik dafür sorgen, daß dieser nur einmalig erzeugt und zerstört wird.

Parameter-Identifier

Parameter-Identifier benötigen Sie nur dann, wenn sie auf eine Variable, die eine Struktur der Blockparameter enthält, zugreifen möchten und die Zugriffsmöglichkeit durch ein weiteres Programm erstellt werden soll (dieses Programm ist also noch zu schreiben). Da der Zugriff möglicherweise von Variable zu Variable unterschiedlich erfolgen soll, können keine direkten Zugriffsfunktionen erstellt werden. Vielmehr wird Ihnen nur ein sehr allgemeines Rüstzeug an die Hand gegeben, um den Rest zu bewerkstelligen.
Das Vorgehen ist recht einfach: Sie laden eine Parameter-Identifier-Spezifikation und ordnen die Identifier den Blöcken zu. Nach der Code-Generierung erhalten Sie dann eine Parameter-Identifier-Zuordnungsdatei, die Sie zur weiteren Verarbeitung heranziehen können.

Aufbau einer Parameter-Identifier-Spezifikation

Die Parameter-Identifier-Spezifikation ist in einer Datei mit Endung LST abzulegen. Sie enthält eine Tabelle, deren Spalten wahlweise durch Tabulatorzeichen oder Semikola getrennt sind. In der ersten Spalte werden die positiven ganzen Zahlen als Parameter-Identifier angegeben. In den nachfolgenden Spalten können beliebige Zeichen stehen, die Sie für die Weiterverarbeitung verwenden möchten (z. B.: Attribute wie Schreib-/Lesezugriffsrechte o. ä.). Die Zeile mit dem Parameter-Identifier 0 wird als Spaltenüberschrift verwendet.

In dem Kapitel *Einstellungen zur Code-Generierung* wurde bereits gezeigt, wie eine Parameter-Identifier-Spezifikation geladen werden kann. Hier soll nun der Inhalt der dort geladenen Datei gezeigt werden.

```
0; Spalte 1; Spalte 2; Spalte 3
1;a11;a12;a13
2;a12;a22;a23
3;a13;a23;a33
4;c1;c2;c3
5-8;a;b;c
9..12;x;y;z
```

Listing 48. *Beispiel einer Parameter-Identifier-Spezifikation*

Bemerkenswert an dem obigen Listing sind die letzten beiden Zeilen, die einen ganzen Bereich an Parameter-Identifizierern definieren. Die in dieser Datei beschriebenen Identifier müssen nicht lückenlos definiert werden!

Zuweisen eines Parameter-Identifiers

Die Zuweisung eines Parameter-Identifiers an einen Block kann im Popup-Menü zu diesem Block (Klick der rechten Maustaste) durch den Eintrag IDENTIFIER-ZUWEISUNG (NUR FÜR C-CODE)... erfolgen. Anschließend erscheint ein Dialog, in dem Sie die Nummer des Parameter-Identifiers angeben.

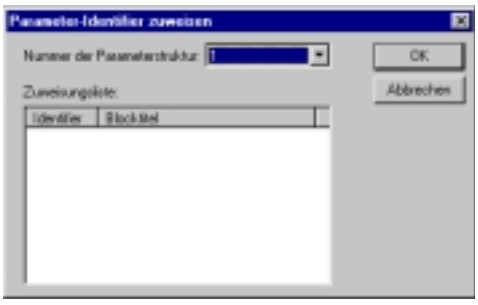

Bild 41 *Zuweisung eines Parameter-Identifiers an einen Block*

Hat der Block im C-Code keine Parameterstruktur, so erscheint im oben gezeigten Dialog ein entsprechender Hinweis. Die zugewiesenen Identifier werden mit dem System abgespeichert.

Enthält die BORIS-Struktur weitere Blöcke, denen schon Parameter zugewiesen worden sind, so werden diese in der Zuweisungsliste eingetragen. Es ist Ihnen freigestellt, ob Sie einen Parameter-Identifier für einen oder für mehrere

Blöcke benutzen möchten. Wir stellen Ihnen aber ein Hilfsmittel zur Überprüfung zur Verfügung.

Parameter-Identifier-Zuordnung prüfen

Zur Überprüfung, welcher Parameter-Identifier für welche Blöcke verwendet wird, dient der Menüpunkt CODE-GENERIERUNG | PARAMETER-IDENTIFIER-ZUORDNUNG PRÜFEN.... Es erscheint ein Dialog mit einem Textfenster, in dem genau beschrieben steht, welcher Block mit welchem Parameter-Identifier versehen wurde und wie oft dieser Identifier verwendet wurde. Da wir davon ausgehen, daß ein Parameter-Identifier auch wirklich etwas identifiziert, erzeugt eine Mehrfachverwendung eines Identifiers eine Warnung. Ein Fehler tritt auf, wenn ein Block einen Identifier hat, der nicht mehr in der Spezifikation vorhanden ist (weil diese beispielsweise verändert wurde).

Parameter-Identifier-Zuordnungsdatei

Diese Datei wird immer bei der Code-Generierung erstellt. Die Einstellung des Dateinamens kann in den Einstellungen zur Code-Generierung getätigt werden. Der Inhalt der Datei ist eine Tabelle, deren Spalten durch Semikola getrennt sind. Die ersten Spalten sind vordefiniert. Die Datei wird hier zunächst in tabellarischer Form gezeigt:

Pid	**Variablenname**	**Blockname**	**Blockindex**	**Blocktyp**	**Rest der Spezifikation zum Pid (weitere Spalten!)**
1	Regler_V2	Regler	2	FC	readonly
2	de_div_dt_V3	de/dt	3	D	readonly
3	I_V5	I	5	I	readonly

Die Datei hat demnach folgenden Inhalt:

1; Regler_V2; Regler; 2; FC; readonly

2; de_div_dt_V3; de/dt; 3; D; readonly

3; I_V5; I; 5; I; readonly

Das anfänglich erwähnte (noch zu schreibende) Programm könnte nun die Zeilen dieser Datei interpretieren, die entsprechend C-Funktionen für den Zugriff erzeugen und den ausgegebenen Quelltext bezüglich der Variablen manipulieren etc..