BW500

Laboratory Setup Ball and Beam

Assembly and Start-Up

Date 21. March 2001

1 Assembly and Start-Up

1.1 Unpacking

After the BW500 has been unpacked, all components are to be checked visually for damages as well as for completeness. Complain any possible damage caused by transportation to the transporter as well as to us. In this case, please secure the packaging until final clarification.

The standard shipment of the BW500 consists of:

- The BW500 mechanics consisting of a DC motor, an incremental encoder, a camera with lighting equipment completely premounted and wired.
- A 19" plug-in box, the actuator, containing a power supply, a servo amplifier controlling the current of the motor, a signal adaption unit and a module with measurement outputs.
- A lead to connect the actuator to the BW500 mechanics.
- A mains supply lead.
- A metal ball and a squash ball.
- Two halogen bulbs.
- Detailed documentation of the hardware and software.

Depending on the desired option, the shipment is extended by the following items:

- Option 500-02, a PC adapter card DAC98 with a lead to connect to the actuator and an executable controller program on a 3,5" floppy disk.
- Option 500-03 extends the Option 500-02 by a 3,5" floppy disk containing an executable program with Fuzzy software.
- Option 500-05, a 3,5" floppy disk containing the C source files of the programs from the options 500-02 and 500-03.
- Option 500-06, a 12,7 cm monochrome monitor suitable for display of the camera image and for service incl. power supply and connection cable.

1.2 Setting up the System

1.2.1 The BW500 Mechanics

To avoid deformation of the Plexiglas parts, choose a place, where the system is not exposed to extreme temperatures. In particular direct sun light and direct heat radiation, e. g. by a radiator, are to be avoided. Direct sun light causes sensor failures especially of the camera.

The system must be placed on a solid surface.

Mount the two halogen bulbs into the corresponding lampholders located below the small platform on top of the system.

1.2.2 Actuator

The air must be able to circulate freely above, below as well as behind the actuator box.

Do not use a soft surface. Otherwise the ventilation slots located on the bottom of the actuator box could be covered due to its weight.

Do not place any objects, e. g. manuals on top of the actuator box. (heat exchange).

1.3 Description of the BW500 Mechanical Setup

Aluminium profiles form the platform and the framework of the laboratory setup which is covered at the side by four sheets of Plexiglas. A camera module and the lighting equipment is mounted below a small platform on top of the system. An aluminium box is mounted at the left side of the platform. The 30 polar terminal to connect to the actuator is located at its rear panel. The beam is located in the centre of the system. It is driven by a tooth-belt, a tooth-wheel and a DC motor so that a ball can be stabilized at a desired position. The motor is located bottom left of the system box. The angle of the beam is measured by an incremental encoder mounted at the rear end of the beam shaft in the centre of the system. Two micro switches are located below the beam. One of them is closed when the beam reaches its maximum angle.

1.4 Description of the BW500 Actuator

1.4.1 The Rear Panel

Figure 1.1 displays the components located on the rear panel. The mains input unit is located on the right. It contains two fuse holders, the mains inlet and the power switch. Located on the left side of the ventilation cover are:

- the 30-polar connector for the mechanical system
- the 50-polar socket to connect the actuator to the DAC98 PC-card (Options 500-02, 500-03),
- the cinch terminal to connect a monitor (option 500-06) for the camera signal.

1.4.2 The Front Panel

Figure 1.2 displays the components located on the front panel.

1.4.2.1 SERVO MOTOR

The servo amplifier functioning as a current controller for the motor is located on the left side of the front panel. Its transfer behaviour is proportional.

1.4.2.2 POWER SERVO

Light emitting diodes indicate different functions/states of the system:

- System (green): Mechanical system is connected to the actuator
- Ready (green): Indicates a successful system test

Controller:

- Manual (red): Manual control is enabled
- PC (red): PC control is enabled
- Extern (red): An external input is enabled
- +35V (green): +35V power supply for the servo amplifiers is available.
- -35V (green): -35V power supply for the servo amplifiers is available.
- Stop: Using this key will disconnect the input signals of the servo amplifiers.
- Stop: Pressing this key will disconnect the input signals from the servo amplifier.

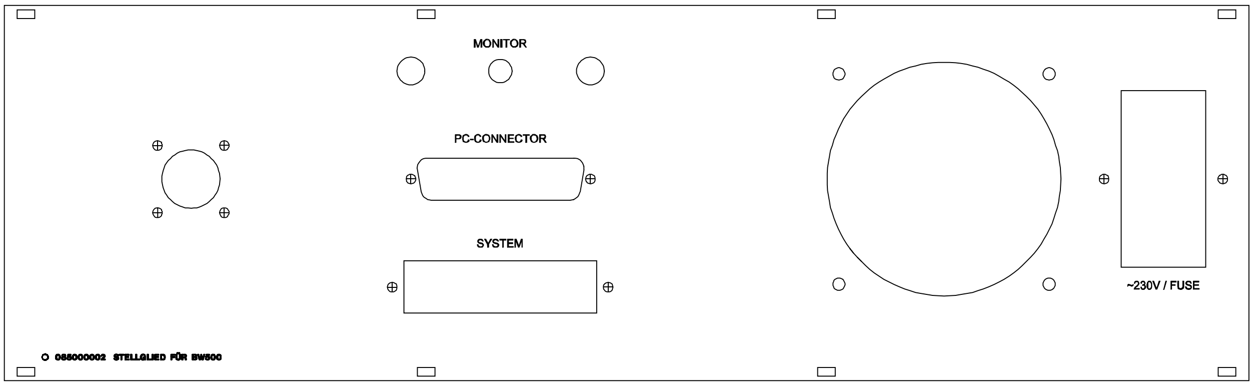


Figure 1.1 : Rear panel with denotations

Additionally this module contains the power supply unit for the servo amplifiers.

1.4.2.3 SENSOR

This module provides the following signals at its 4 BNC-terminals:

- Position: The position of the ball on the beam
- INC-CHA: Channel A of the incremental angle encoder
- INC-CHB: Channel B of the incremental angle encoder
- Control Signal: The control signal of the controller

Additionally this module contains the logic circuit to process the image of the camera.

1.4.2.4 Mains Supply (POWER module)

The power module contains the power supply for the amplifiers, the digital electronics and the incremental encoder. Three light emitting diodes indicate the availability of the voltages.

- +15V (green): A voltage of +15V is available.
- -15V (green): A voltage of -15V is available.
- +5V (green): A voltage of +5V is available.

1.4.2.5 CONTROLLER

This module allows a direct control of the servo amplifier either by using a potentiometer or by connecting an external controller. It contains the control of the lighting of the beam in addition.

- Switch Light: Switches the lighting "ON" or "OFF".
- Potentiometer MANUAL: Allows for manual adjusting the control signal (force) of the beam.
- Key MANUAL START: Connects the control signal adjusted by the potentiometer MANUAL to the servo amplifier of the motor. This is true only as long as the key is pressed. Afterwards the control is at the state which was active before pressing the key.
- BNC socket CONTROLSIGNAL INPUT: Input for setpoint of the motor current (range ±10V, 1V=2.15N).
- Key EXTERN START: Connects the control signal provided at the socket CONTROLSIGNAL INPUT to the servo amplifier of the motor. This switch function is disabled when another controller is enabled. Switching the key again disconnects the signals.

This module contains the logic circuit to enable the control signals.

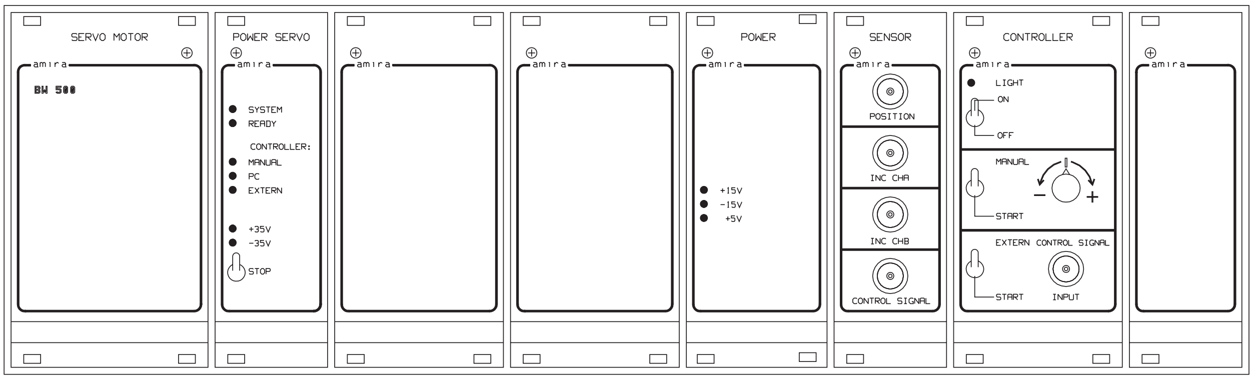


Figure 1.2 : Front panel with denotations

1.5 Connecting the System Components and Start Up

Before setting up the system, please check whether your mains supply provides a 230V, 50/60Hz voltage. Connect the actuator to the mains supply and switch on the actuator. The LED’s +15V, -15V, +5V, +35V and -35V should light up now. Then switch off the actuator and establish the lead connection between the actuator and the mechanical system. Now switch on again the actuator. Besides the LED’s mentioned above the LEDs "System" and "Ready" should light up. After pressing the "Stop" key, all LEDs should light up.

The system is now ready for operation.

An additional DC measuring instrument is required to perform a function test. Connect this instrument the terminal labelled "Position" on the module "Sensor". Switch on the lighting of the beam. Place a ball on the beam and change its position by manually taking the right end of the beam and manipulating its angle. The displayed voltage measured by the instrument should represent the ball position (range ±10V). The monitor from option 500-06 may be used to overview the camera signal. When the monitor is to be used, connect the device to the corresponding monitor socket at the actuator rear panel. The screen should now display the beam with the ball and a black line. This black line emphasizes the image row used to measure the ball position. The black line should be located in the middle of the beam and should end at the left side of the ball.

1.6 Manual Control

Please adjust the potentiometer "MANUAL" exactly to its middle position and press the key "MANUAL START". During pressing the key the LED "MANUAL" will light up. In this state you may use the potentiometer to carefully change the control signal (force) for the beam. The control signal may be measured at the terminal "CONTROL SIGNAL" (1V = 2,15N). Letting the key to its original position will cause an actuator operation mode as before pressing the key. Therefore manual system control may be used to disturb a controller.

1.7 Control with External Controller

The measured value of the ball position is provided at the terminal "Position" (range ±10V). A quadrature incremental encoder measures the beam angle and provides its signals channel A and B at the terminals "INC CHA" and "INC CHB" (360° = 20000 increments). Any external controller has to be able to process such signals and to perform a calibration of the angle zero. The output of an external controller is to be connected to the terminal "CONTROLSIGNAL INPUT" (range ±10V, 2.15 N/V). The control signal is connected to the servo amplifier of the beam motor after pressing the key "Start Extern". The external control operation mode is terminated either after pressing the key "Start Extern" again or by pressing the key "Stop" on the module "POWER SERVO".

1.8 PC Control

Controlling the ball and beam system by a PC is described in the chapter "Operating Instructions" (only available with option 500-02). To install the controller program the installation program SETUP.EXE from the enclosed floppy disk is to be started with Windows 3.1 or Windows 95/98 (arbitrary subdirectories may be entered in the installation dialog but their names must contain only 8 characters besides an extension). Following a successful installation the controller program **BW500W.EXE** may be started immediately.

1.9 Output Stage Release

If you use your own PC controller via the 50-pol. connector on the rear panel please think of the release of the output stage. The output stage release is a safety function so that in case of program failure the motor stops immediately.

You need two digital signals for the output stage release. DO1 (pin 35 of the 50-pol. connector) gets first a

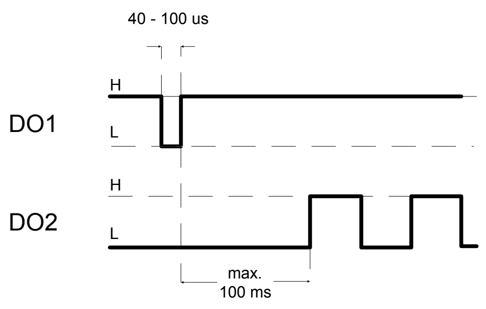


Figure 1.3 : Signals for the output stage release

high-level with pulse to low, duration 40 - 100 µs. After going high DO2 (pin 36 of the 50-pol. connector) needs within the next 100 ms a rect-signal in the range of 10 Hz and 1 kHz. (see fig. 1.3)

1.10 Locating Errors

First try to eliminate problems with the help of the following table. In case you cannot solve problems with your BW500 by yourself, please contact us.

Problem	Possible cause
The LEDs do not light up	Check the connection to the mains supply Check the fuses for the mains supply (rear panel)
LED + 35 V does not light up	Check the fuse of the module "POWER SERVO"
LED - 35 V does not light up	Check the fuse of the module "POWER SERVO"
LEDs + 15 V and - 15 V do not light up	Check the fuse of the module "POWER"
LED + 5 V does not light up	Check the fuse of the module "POWER"
LED "System" does not light up	Check the lead connection between mechanical system and actuator
Selected controller cannot be enabled	Check whether another controller, is enabled (LEDs on "POWER SERVO" module) Check LED "Ready" lights up

Technical Data

Date: 17.05.2000

1 Technical Data

1.1 BW500 Mechanics, Drive, Sensors

1.1.1 Dimensions and Weight of the BW500 Mechanics

Dimensions and Weight	Value	Unit
Length	1105	mm
Depth	215	mm
Height	1025	mm
Weight	18	kg

Dimensions and Weight Ball	Value	Unit
Diameter steal ball	40	mm
Weight steal ball	0,27	kg
Diameter Squash ball	37,5	mm
Weight Squash ball	0,025	kg

1.1.2 The Drive

Prinziple: The drive is a permanently exited DC motor with ball bearings. The carriers of the graphite brushes are accessible by opening the lateral plastic covers.

Drive Type GNM3125	Value	Unit
Rated voltage	24	V
Rated speed	3000	Rpm
Rated current	2	A
Rated power	30	W
Armature weight	0.19	kg
Motor weight	0.77	kg
Moment of inertia	0.177	kgcm^2
Rated torque	9.6	Ncm
Starting torque	40.0	Ncm
Max. continous torque during stillstand	10.5	Ncm
Friction torque	2.0	Ncm
Mechanical time constant	15.4	ms
Connected resistance	3.13	Ω
Armature inductance	3	mH
Armature resistance	2.6	Ω
Voltage constant	6.27	mV / $^1/_{\text{min}}$
Torque constant	6.0	Ncm / A
max. peak current	16	A
Electrical time constant	0.96	ms

1.1.3 The Incremental Encoder

Incremental Encoder Type RI58-D	Value	Unit
Mode of attachment	synchro flange	
Shaft diameter	10	mm
Max. speed RI58-D	10000	Rpm
Torque	<1	Ncm
Moment of inertia RI58-D	0.014	kgcm^2
Mass RI58-D	0.36	kg
Resolution	5000	lines/rot.
Interface	RS 422	
Supply voltage	5	V
Impulse channels	A/A;B/B;N/N	
Impulse wave shape	rectangle	
Phase shift A/B	90	°
Index signal N/N	1	1/Rotation

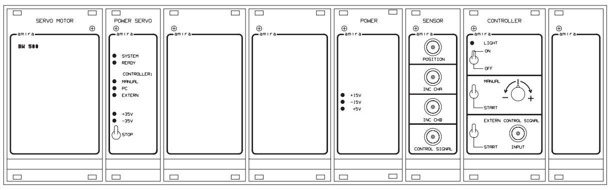
1.1.4 The Camera

Camera	Value	Unit
Supply voltage	9-12	V
Power consumption	2	W
Output sig. level on 75 Ω	1	V_{ss}
Horizontal resolution	420	lines
Sensitivity to light	0,05	Lux
Focal distance of lens	f=3,6 F1,4	mm
Weight	20	g
Dimensions (B x H x T)	32 x 28,5 x 32	mm
Lighting (halogen)	2 x 20 (36°)	W

1.1.5 The Monitor (Option 500-06)

Monochr. Monitor	Value	Unit
Picture tube	12,7	cm
Line termination	75	Ω
Working voltage	230	V~
Weight	1,4	kg
Dimensions (B x H x T)	190x143x245	mm

1.2 Actuator



Dimensions and weight	Value	Unit
Length	465	mm
Depth	308	mm
Height	150	mm
Weight	10	kg

Mains supply	Value	Unit
Input voltage	230	V
Frequency	50/60	Hz
Power consumption max.	140	W
Fuses S1 and S2	2	A T

1.2.1 SERVO MOTOR Module

Inputs	Value	Unit
Supply voltages	+35 -35	V V
Current consumtion	2	A
Control signal from signal adaption unit	±10	V
Amplification	0.4	A/V
Time constant	< 1	ms

Outputs	Value	Unit
Motor terminal voltage	± 30	V
Current monitor for signal adation unit	0.2	V/A

1.2.2 POWER SERVO Module

Inputs	Value	Unit
Supply voltages Servo power unit	2*24 V~	V
Current consumption	2.5	A

Outputs	Value	Unit
Servo supply voltage	± 35, 2.5	V A
Reset signal for enable circuit	TTL signal	

1.2.3 SENSOR Module

Measuring signals (BNC Sockets)	Value	Unit
Ball position from image processing (Position)	± 10, ~25	V V/m
Inkremental encoder signal of beam angle channel A (INC CHA)	TTL signal 20000	V Inkr./Rot.
Inkremental encoder signal of beam angle channel B (INC CHB)	TTL signal 20000	V Inkr./Rot.
Control signal for motor servo (Control Signal)	± 10, 2,15	V N/V

1.2.4 POWER Module

Mains supply	Value	Unit
Input voltage 2 primary fuses	230 200	V mA T
Frequency	50/60	Hz
Output voltage	± 15 +5	V V

1.2.5 CONTROLLER Module

Input	Value	Unit
BNC socket Extern Control Signal	± 10	V

1.2.6 Rear Panel Connections

PC-Connector (DAC98)

Pin-No.	Pin-Den.	Reservation	bit fr. Dout3
1	INC0 CHA	beam angle channel A	
2	INC0 CHB	beam angle channel B	
3	INC1 CHA	n.c.	
4	INC1 CHB	n.c.	
5	INC2 CHA	n.c.	
6	INC2 CHB	n.c.	
7	INC3 CHA	n.c.	
8	INC3 CHB	n.c.	
9	DIN0	/Stop left	0
		camera Bit 0	1
10	DIN1	/Stop right	0
		camera Bit 1	1
11	DIN2	/LED-Ready	0
		camera Bit 2	1
12	DIN3	/PC-Ready	0
		camera Bit 3	1
13	OUT1	n.c.	
14	OUT2	n.c.	
15	AGND	AGND	
16	n.c.	n.c.	
17	n.c.	n.c.	
18	INC0 /CHA	beam angle channel A (inverted)	

PC-Connector (DAC98)

Pin-No.	Pin-Den.	Reservation	Bit fr. Dout3
19	INC0 /CHB	beam angle channel B (inverted)	
20	INC1 /CHA	n.c.	
21	INC1 /CHB	n.c.	
22	INC2 /CHA	n.c.	
23	INC2 /CHB	n.c.	
24	INC3 /CHA	n.c.	
25	INC3 /CHB	n.c.	
26	DIN4	camera Bit 8	0
		camera Bit 4	1
27	DIN5	camera Bit 9	0
		camera Bit 5	1
28	DIN6	camera Bit 10	0
		camera Bit 6	1
29	DIN7	camera Bit 11	0
		camera Bit 7	1
30	n.c.	n.c.	
31	AGND	AGND	
32	n.c.	n.c.	
33	n.c.	n.c.	
34	Dout0	execution time	
35	Dout1	pulse (enable servo)	
36	Dout2	rectangle (enable servo)	
37	Dout3	camera control register	
38	Dout4	camera hold	
39	Dout5	n.c.	

PC-Connector (DAC98)			
Pin-No.	Pin-Den.	Reservation	Bit fr. Dout3
40	Dout6	n.c.	
41	Dout7	n.c.	
42	DGND	DGND	
43	DGND	DGND	
44	IN1	n.c.	
45	IN2	n.c.	
46	IN3	n.c.	
47	Aout0	motor control signal	
48	Aout1	n.c.	
49	n.c.	n.c.	
50	n.c.	n.c.	

Functional Description of the electrical Connections between the Actuator and the PC Card DAC98:

INC0 CHA (Pin No. 1):
INC0 CHB (Pin No. 2):

- These signals deliver the 4Q square wave signal of the angle encoder to the PC adapter card. The decoding of these signals is made on the DAC98.

DIN0(Pin No. 9):

- If the digital output ’camera control register’ is low, this pin delivers the state of the left limit switch (low == switch is actuated). This switch is used to disconnect the motor if the beam is at its left limit position.

- If the digital output ’camera control register’ is high, this pin delivers the bit 0 of the 12 bit word for the determination of the ball position.

DIN1(Pin No. 10):

- If the digital output ’camera control register’ is low, this pin delivers the state of the right limit switch (low == switch is actuated). This switch is used to disconnect the motor if the beam is at its right limit position.

- If the digital output ’camera control register’ is high, this pin delivers the bit 1 of the 12 bit word for the determination of the ball position.

DIN2(Pin No. 11):

- If the digital output ’camera control register’ is low, this pin delivers the result of the system self test (low == system is OK).

- If the digital output ’camera control register’ is high, this pin delivers the bit 2 of the 12 bit word for the determination of the ball position.

DIN3(Pin No. 12):

- If the digital output ’camera control register’ is low, this pin delivers the state of the output stage release (low == system is released) (see also chapter 1.9 of ’Assembly and Start-Up’).

- If the digital output ’camera control register’ is high, this pin delivers the bit 3 of the 12 bit word for the determination of the ball position.

INC0 /CHA (Pin No. 18):

- This signal is the inverted signal of INC0 CHA (Pin No. 1).

INC0 /CHB (Pin No. 19):

- This signal is the inverted signal of INC0 CHB (Pin No. 2).

DIN4(Pin No. 26):

- If the digital output 'camera control register' is low, this pin delivers the bit 8 of the 12 bit word for the determination of the ball position.

- If the digital output 'camera control register' is high, this pin delivers the bit 4 of the 12 bit word for the determination of the ball position.

DIN5(Pin No. 27):

- If the digital output 'camera control register' is low, this pin delivers the bit 9 of the 12 bit word for the determination of the ball position.

- If the digital output 'camera control register' is high, this pin delivers the bit 5 of the 12 bit word for the determination of the ball position.

DIN6(Pin No. 28):

- If the digital output 'camera control register' is low, this pin delivers the bit 10 of the 12 bit word for the determination of the ball position.

- If the digital output 'camera control register' is high, this pin delivers the bit 6 of the 12 bit word for the determination of the ball position.

DIN7(Pin No. 29):

- If the digital output 'camera control register' is low, this pin delivers the bit 11 of the 12 bit word for the determination of the ball position.

- If the digital output 'camera control register' is high, this pin delivers the bit 7 of the 12 bit word for the determination of the ball position.

Execution time (Pin No. 34):

- At that moment this signal is not used. For future extensions this signal will be set to 1 during the calculation of the control algorithm.

Pulse(enable servo) (Pin No. 35):
Rectangle(enable servo) (Pin No. 36):

- The functionality of these signals are described in chapter 1.9 of 'Assembly and Start-Up'.

Camera control register (Pin No. 37):

- With this digital output (as seen from the PC) the functionality of the signals DIN0-DIN7 is switched over.

Camera hold (Pin No. 38):

- If this digital output is set to 0, the measurment signals from the image processing unit will not be updated. This is necessary because the reading of the 12 bit data word happens in two steps. An update of the measurment signal between these readings would cause errors.

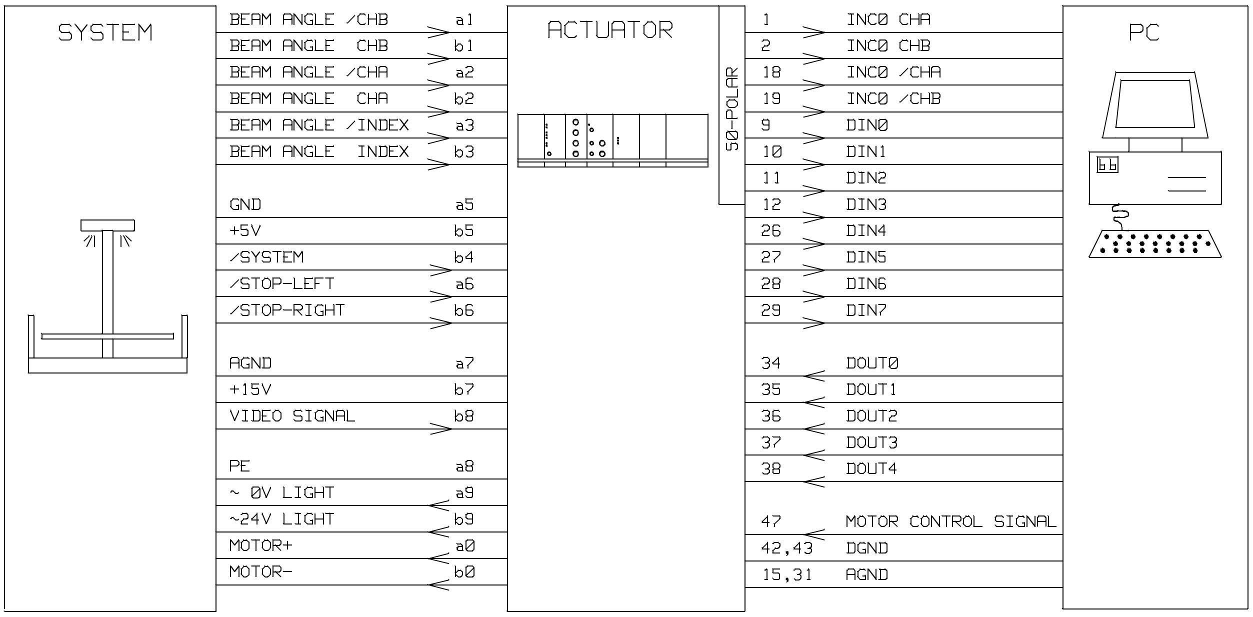
Aout0 (Pin No. 47):

- This analog output of the DAC98 delivers the control signal for the motor.

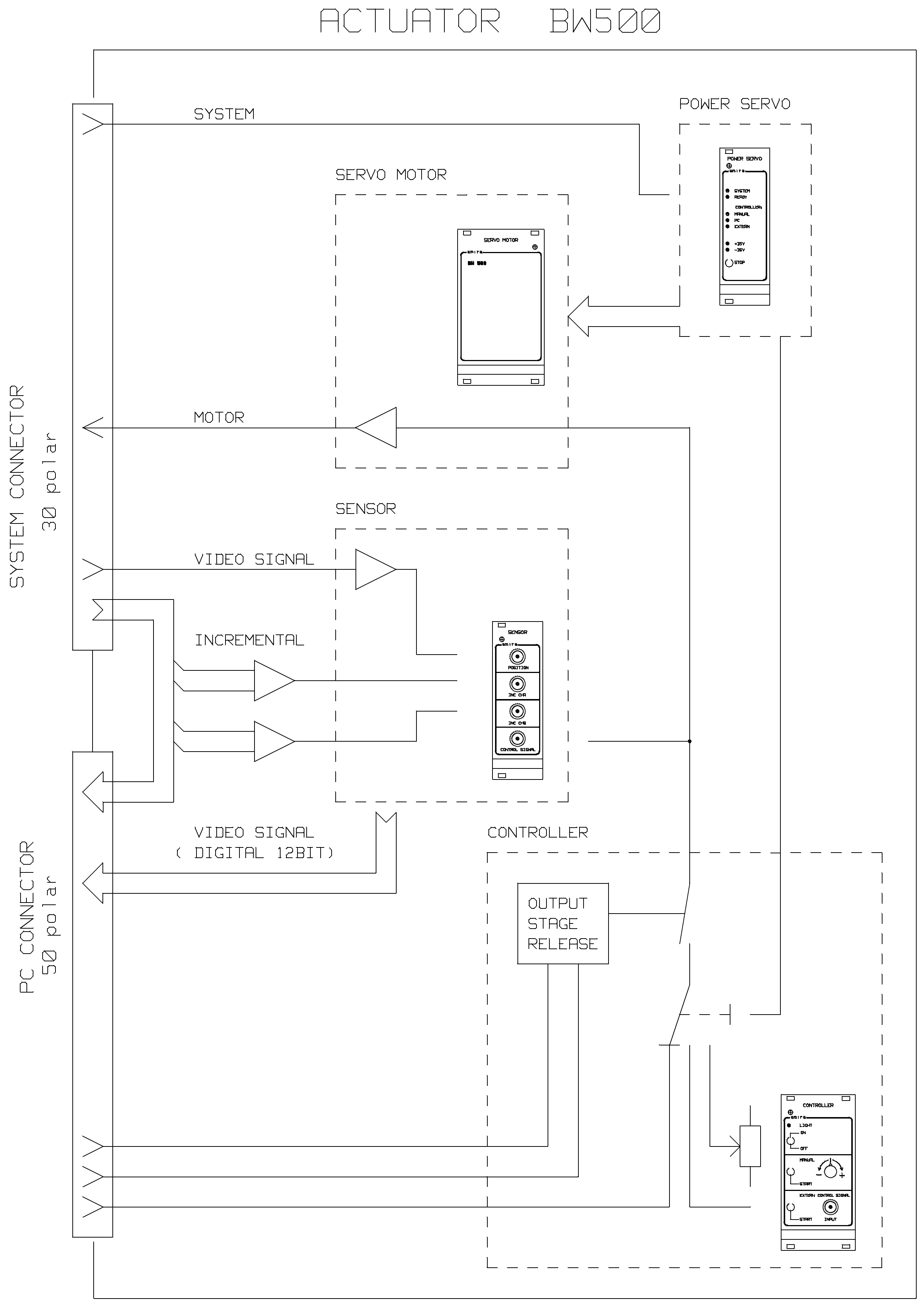
The image processing system for the position measuring of the ball

It is sufficient to evaluate one single line of the picture information from the camera because the ball position is described by an one-dimensional value. This happens inside the hardware of the BW500 system. The line which has to be evaluated, will detect by means of a counter and a comparator. In the beginning of the scanning of the picture line a timer is started. This timer is stopped as soon as the black ball is recognized on the white bar. The state of the timer is copied into a buffer and transferred to the PC. The timer value can linearly be mapped on the distance which has to be measured because the system was calibrated by means of the PC software. The position of the picture line which has to be evaluated as well as the threshold value for the detection of the ball will be calibrated on the corresponding mechanic during the fabrication of the image processing system. **This calibration may not be changed**. The position measuring of the ball is continuously executed after switching on the system. The updating of the data in the output buffer can be stopped by the PC (camera hold).

System Connector					
Pin-No.	Reservation	Pin-No.	Reservation	Pin-No.	Reservation
a1	/CH B	b1	CH B	c1	n.c.
a2	/CH A	b2	CH A	c2	n.c.
a3	/Index	b3	Index	c3	n.c.
a4	Screen	b4	/System	c4	n.c.
a5	DGND	b5	+5V D	c5	n.c.
a6	/Stop left	b6	/Stop right	c6	n.c.
a7	AGND	b7	+15V	c7	n.c.
a8	PE	b8	Video signal	c8	n.c.
a9	Light	b9	Light	c9	n.c.
a0	Motor+	b0	Motor-	c0	n.c.



Connections between actuator, mechanics and PC


ACTUATOR BW500
SYSTEM
POWER SERVO
SERVO MOTOR
POWER SERVO
SYSTEM
READY
CONTROLLER:
MANUAL
PC
EXTERN
+35V
-35V
STOP
SERVO MOTOR
BW 500
MOTOR
SYSTEM CONNECTOR 30 polar
SENSOR
VIDEO SIGNAL
INCREMENTAL
SENSOR
POSITION
INC CH-A
INC CH-B
CONTROL SIGNAL
VIDEO SIGNAL (DIGITAL 12BIT)
CONTROLLER
PC CONNECTOR 50 polar
OUTPUT STAGE RELEASE
CONTROLLER
LIGHT
ON
OFF
MANUAL
START
EXTERN CONTROL SIGNAL
START
INPUT

Mathematical Model of the System

Date: 14.03.1996

1 Mathematical Model of the Ball and Beam System

This chapter describes the mathematical modelling of the system ball and beam. Here an application of the Lagrange principle known from the analytical mechanic is considered especially.

1.1 System Description Used for the Model

The following denotations (see also figure 1.1) will be used to derive the mathematical model.

The abbreviations have the following meaning:

m	:	mass of the ball
g	:	gravity
r	:	roll radius of the ball
I_b	:	inertia moment of the ball
I_w	:	inertia moment of the beam
M	:	mass of the beam
b	:	friction coefficient of the drive mechanics
K	:	stiffness of the drive mechanics
$u(t)$	:	force of the drive mechanics
l	:	radius of force application
l_w	:	radius of beam
x'	:	ball co-ordinate with respect to the beam
y'	:	ball co-ordinate with respect to the beam
ψ	:	angle of the ball to the beam
α	:	angle of the beam to the horizontal

A linear sliding friction in the drive mechanics is observable during a rotation, which is described by the friction coefficient b.

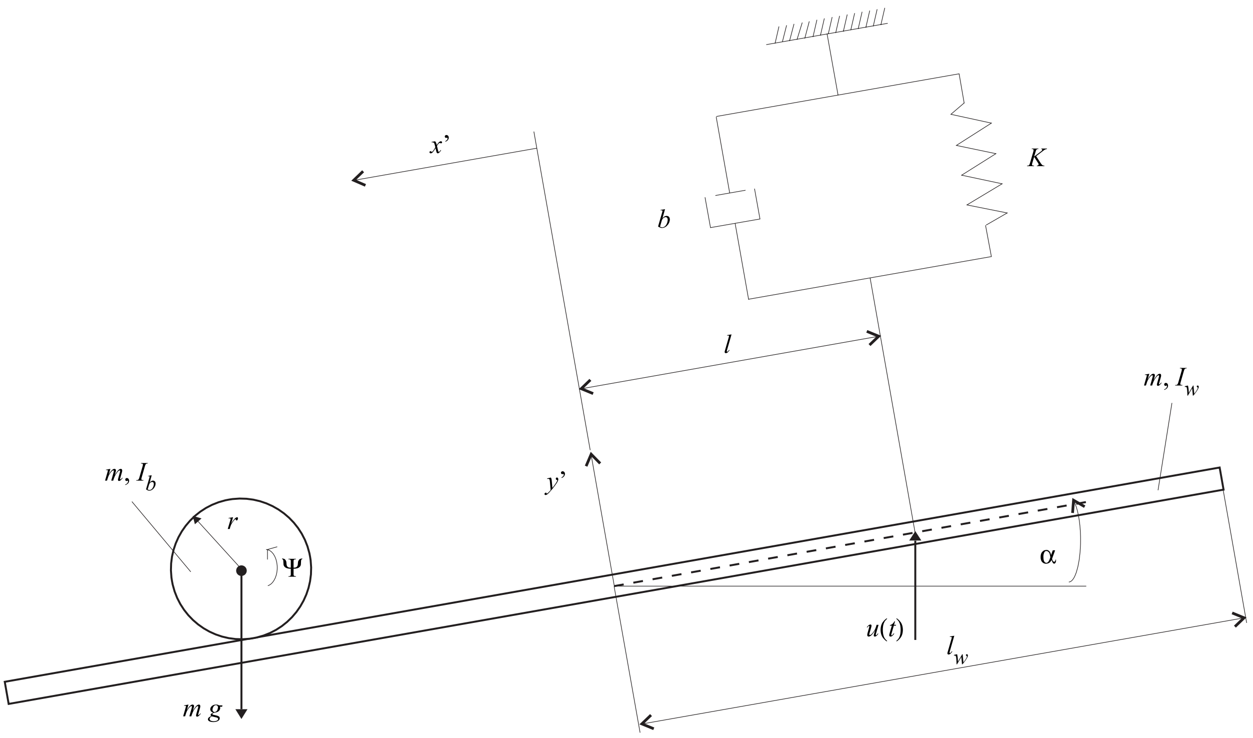


Figure 1.1: System elements for the Ball and Beam system

The spring with the stiffness K takes into account the delay behaviour of the driving belt which however may be neglected in the realized laboratory setup. With this the driving force $u(t)$ is the input variable of the system.

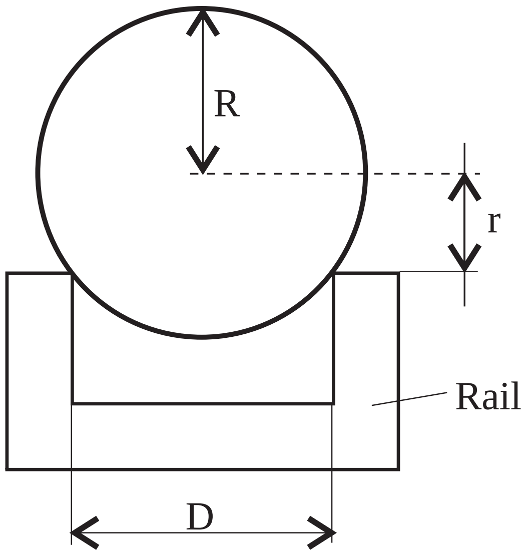


Figure 1.2: The different radius of the ball with respect to the rail of the beam

Because the ball does not roll in the plane, but in a groove (u-type profile, see figure 1.2), two different radius of the ball have to be considered on setting up the motion equations. One is the radius R of the ball the other is the roll radius r, given by the distance from the middle of the ball to the surface of the beam.

1.2 Methods for Modelling

At first the method of d' Alembert would provide a solution for the mathematical modelling of the mechanic system on hand. Here balances of forces and moments of inertia are used to establish the motion equations after cutting free each single mass. Applying this method on the ball and beam system will show problems when the system is analyzed in more details.

According to figure 1.1 $\kappa = 4$ free co-ordinates x', y', ψ and α are definable. The mechanical degree of freedom f results with respect to Göldner and Holzweissig (1989) from reducing the $\kappa = 4$ possible motions to $f = 2$.

Therefore the result are the following $\sigma = \kappa - f = 2$ constraint conditions, i.e. kinematic linkages:

$$x' = r\psi \qquad y' = l\alpha \quad .$$

A linkage results from reducing the degree of freedom of a mechanic system. So the dynamic behaviour of the ball is to be described in a moving reference system x', y' and with respect to a static co-ordinate system ξ, ζ (inertial system). For this reason the formulation of the linear and angular momentums is no longer simple (Frik 1994), which increases the effort setting up the motion equations considerably. In addition the reactions from cutting free the masses are contained in the motion equations when the d' Alembert method is applied. Further more these reactions are out of interest for the examination of the dynamic behaviour of the ball and beam system.

The Lagrange method (Göldner, Holzweissig 1989) however provides an alternative solution to establish the motion equations without cutting free the masses. For this reason the problem of the relative motion is reduced considerably. This method is described in the following section and is applied to the ball and beam system.

1.3 The Lagrange Equations of 2. Kind

The Langrange equations of 2. kind consider the energy to result with the motion equations of an arbitrary mechanic multiple-body system. To do this at first all possible motions of the system have to be defined by the free co-ordinates

$$x_i , \quad i = 1,..., \kappa \qquad (1.1)$$

with respect to predefined steady positions. In case of geometric and/or kinematic links in a mechanic system the number of co-ordinates required to describe the motion is reduced by the number σ of constraint conditions. The remaining co-ordinates are denoted as generalized co-ordinates:

$$q_j\,,\quad j=1,...,f\ . \tag{1.2}$$

The number of independent generalized co-ordinates is given by the degree of freedom *f*.

Therefore the κ free system co-ordinates x_i may be formulated as a function of the *f* generalized co-ordinates q_j as follows:

$$x_i \ =\ x_i\,(\,q_1\,,q_2\,,...,q_f\,,t\,),\quad i=1,...,\kappa \tag{1.3}$$

where *t* denotes the explicit effect of the time.

With this altogether *f* independent variations

$$\delta\,q_1\,,...,\delta\,q_f \tag{1.4}$$

are definable. Because the position vectors $\underline{r}$ depend on the generalized co-ordinates q_j, the virtual active part of an external force is describable as follows:

$$\delta\,W \ =\ \underline{F}\cdot\delta\,\underline{r} \ =\ \sum_{j=1}^{f}\underline{F}\cdot\frac{\delta\,\underline{r}}{\delta\,q_j}\,\delta\,q_j$$

$$=\sum_{j=1}^{f} Q_j\ \delta\,q_j\ . \tag{1.5}$$

Q_j are the generalized forces with respect to the generalized co-ordinates. Q_j is a force, when q_j is a distance. When Q_j describes a moment, q_j stands for an angle. All the active forces and moments are to be considered in Q_j, than means the impressed and elastic link forces.

Typical elastic link forces like linear spring forces or potential forces are formulated as follows:

$$Q_j \ =\ -\frac{\delta\,V}{\delta\,q_j}\ . \tag{1.6}$$

where *V* denotes the potential of the impressed forces and the elastic link forces.

The function *L* called Lagrange function may therefore be defined by (Glödner, Holzweissig 1989):

$$L=T-V\ . \tag{1.7}$$

T is the sum of the kinetic energies of each single rigid body of the system. For the *i*-th rigid body the kinetic energy is assembled by the translation of the *i*-th centre of mass and the rotation of the *i*-th mass point around the *i*-th centre of mass (Frik 1994):

$$T_i \ =\ \frac{1}{2}\,m_i\,v_{si}^2+\frac{1}{2}\,\omega_i\,I_{si}\,\omega_i\ . \tag{1.8}$$

With this the Langrange equations of 2. kind are formulated as:

$$\frac{d}{d\,t}\left(\frac{\delta\,L}{\delta\,\dot{q}_j}\right)-\frac{\delta\,L}{\delta\,q_j}=\overline{Q}_j\ ,\ \ j=1,...,f\ . \tag{1.9}$$

When $\overline{Q}_j$ contains dissipative forces (i.e. linear friction forces) in addition, which reduce the system energy during motion, $\overline{Q}_j$ may be stated as follows (Meirovitch 1970):

$$\overline{Q}_j=Q_j^*-\frac{\delta\,\Upsilon}{\delta\,\dot{q}_j} \tag{1.10}$$

where

Υ	:	dissipative Rayleigh function
Q_j^*	:	nondissipative potential-free generalized forces

Knowing the equation (1.9) the motion equations of the mechanic system are obtained by differentiation with respect to the generalized co-ordinates and with respect to the time.

1.3.1 Application on the Ball and Beam System

According to section 1.2 the ball and beam system is describable by $\kappa=4$ free co-ordinates $x\,'$, $y\,'$, ψ and α

with $\sigma = 2$ constraint conditions:

$$x' = r\psi\ , \quad y' = l\alpha \tag{1.11}$$

This leads to $f = 2$ independent variations. With respect to the given sensors to measure the position of the ball x' as well as the angle of the beam α for the state controller

$\delta q_1 = \delta x'$ variation of the position of the ball

$\delta q_2 = \delta \alpha$ variation of the angle (1.12)

are selected.

Kinetic Energy of the System

The kinetic energy T is given by the following 2 portions:

Ball: $$T_b = \frac{1}{2} m v_s^2 + \frac{1}{2} I_b \omega_b^2 \tag{1.13}$$

Beam: $$T_w = \frac{1}{2} I_w \dot{\alpha}^2 \tag{1.14}$$

leading to

$$T = T_b + T_w\ . \tag{1.15}$$

The velocity of the centre of mass v_s as well as the angular velocity ω_b of the ball have to be defined as a function of the generalized co-ordinates.

The figure 1.3 shows all the variables required to formulate the velocity v_s with respect to the inertial system ξ , ζ. The following correlation is valid (Hagedorn 1990, Hering u.a. 1992):

$$\underline{v}_s = \underline{v}'_s + \underline{\omega} \times \underline{r}_s\ . \tag{1.16}$$

With the position vector $\underline{r}_s = [-x'\ ,\ r]^T$ with respect to the relative system x', y' and

$$\underline{v}'_s = \frac{d'\underline{r}_s}{dt} = [-\dot{x}'\ \ 0]^T$$

it follows for equation (1.16):

$$\frac{d\underline{r}_s}{dt} = \begin{bmatrix} -\dot{x}' \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\alpha} \end{bmatrix} \times \begin{bmatrix} -x' \\ r \\ 0 \end{bmatrix}$$

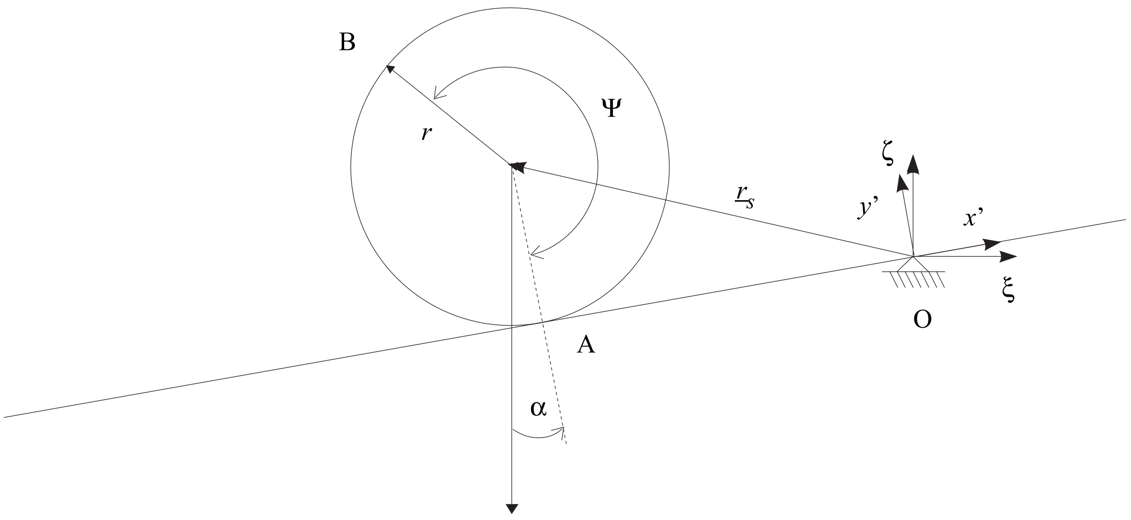


Figure 1.3: The definition of the position vector of the ball

$$= \begin{bmatrix} -\dot{x}' - \dot{\alpha}\, r \\ -x'\, \dot{\alpha} \\ 0 \end{bmatrix} . \qquad (1.17)$$

With this the ball velocity is formulated by

$$v_s^2 = \dot{x}'^2 + 2\,\dot{x}'\,\dot{\alpha}\, r + (\dot{\alpha}\, r)^2 + (x'\,\dot{\alpha})^2 \quad . \qquad (1.18)$$

An alternative determination of v_s^2 is given by the two differentiations

$$\dot{\xi}_s = \frac{d}{dt}(-x' \cos\alpha - r \sin\alpha)$$

and

$$\dot{\zeta}_s = \frac{d}{dt}(-x' \sin\alpha + r \cos\alpha)$$

of the Cartesian components of the centre of the ball with respect to the inertial system. Calculating its absolute value results in the correlation from equation (1.18).

For the determination of the angular velocity ω_b it is to be considered that it is combined by the rotation of the ball itself and the rotation of the beam. The current angular velocity $\underline{\omega}$ of a body executing two angle variations $d\,\underline{\varphi}_1$ and $d\,\underline{\varphi}_2$ is given according to Frik (1994) by the formulation $\underline{\omega} = \underline{\omega}_1 + \underline{\omega}_2$. For the ball and beam system this leads to:

$$\omega_b = \dot{\psi} + \dot{\alpha}$$
$$= \frac{\dot{x}'}{r} + \dot{\alpha} \quad . \qquad (1.19)$$

Inserting Eq. 1.18 and Eq. 1.19 in Eq. 1.15 results in the kinetic energy with respect to the generalized co-ordinates:

$$T = \frac{1}{2}\Big(I_w\,\dot{\alpha}^2 + m\,(\dot{x}'^2 + 2\,\dot{x}'\,\dot{\alpha}\, r + \dot{\alpha}^2\, r^2 + x'^2\,\dot{\alpha}^2)$$
$$+ I_b\,(\frac{\dot{x}'}{r} + \dot{\alpha})^2\Big) \quad . \qquad (1.20)$$

Potential Energy of the System

The potential energy of the system is combined by the following conservative portions:

1. the potential energy of the ball results in (see figure 1.4):

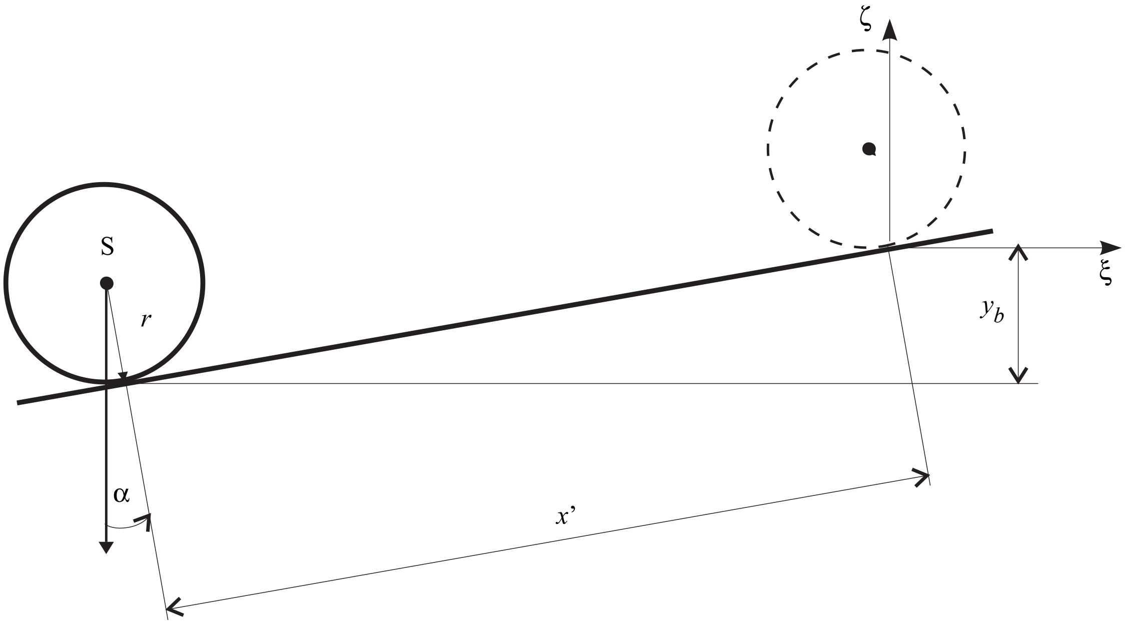


Figure 1.4: The potential energy of the ball

$$V_b = -m\,g\,y_b = -m\,g\,x'\sin(\alpha) \quad . \tag{1.21}$$

2. the potential energy of the driving spring:

$$V_f = \frac{1}{2}K\,\Delta y'^2 = \frac{1}{2}K\,l^2\,\alpha^2 \quad . \tag{1.22}$$

The Eq. 1.21 as well as Eq. 1.22 yield the potential energy of the system:

$$V = V_b + V_f = -m\,g\,x'\sin(\alpha) + \frac{1}{2}K\,l^2\,\alpha^2 \quad . \tag{1.23}$$

Dissipative Forces of the System

The single dissipative force in the system is the linear sliding friction of the driving mechanic:

$$F_R = -b\,l\,\dot{\alpha} \quad .$$

Therefore the dissipative Rayleigh function is given by:

$$\Upsilon = \frac{1}{2}b\,l^2\,\dot{\alpha}^2 \quad . \tag{1.24}$$

Generalized Forces of the System

The driving force *u*(*t*) is the only non-conservative force in the given system. To determine the generalized force the method of virtual active force (Göldner, Holzweissig 1989) is applied:

$$\delta W = \underline{u}\cdot\delta\underline{r} = \begin{bmatrix} 0 \\ u(t) \end{bmatrix}\cdot\begin{bmatrix} 0 \\ l\cos(\alpha)\,\delta\alpha \end{bmatrix}$$

$$= u(t)\,l\cos(\alpha)\,\delta\alpha = Q^*_\alpha\,\delta\alpha + Q^*_{x'}\,\delta x' \quad . \tag{1.25}$$

Comparing the coefficients leads to the generalized force:

$$Q^*_\alpha = u(t)\,l\cos(\alpha) \;; \quad Q^*_{x'} = 0 \quad . \tag{1.26}$$

Motion Equations of the System

Now all the components of the Langrange equation (see Eq. 1.9) are known. To derive the motion equations only the differentiations with respect to the two generalized co-ordinates from Eq. 1.12 and with respect to the time are to be calculated. This results in the two following coupled non-linear differential equations for the generalized co-ordinates *x*'and α:

$$\left(m + \frac{I_b}{r^2}\right)\ddot{x}' + (m\,r^2 + I_b)\frac{1}{r}\ddot{\alpha} - m\,x'\dot{\alpha}^2$$
$$= m\,g\sin(\alpha) \tag{1.27}$$

$$(m\,x'^2 + I_b + I_w)\,\ddot{\alpha} + (2\,m\,\dot{x}'\,x' + b\,l^2)\,\dot{\alpha}$$
$$+ K\,l^2\,\alpha + (m\,r^2 + I_b)\frac{1}{r}\ddot{x}' - m\,g\,x'\cos(\alpha)$$
$$= u(t)\,l\cos(\alpha) \tag{1.28}$$

Eq. 1.27 is the equation for the motion of the ball and Eq. 1.28 is the equation for the motion of the beam.

In the following section the general non-linear state space description of the system is derived from the equations 1.27 and 1.28.

1.4 State Space Description of the Non-linear Model

The general form of a state space description of an arbitrary dynamic system is given as follows (Föllinger 1994):

$$\dot{\underline{x}} = \underline{f}(\underline{x}(t), \underline{u}(t), t) \tag{1.29}$$

$$\underline{y} = \underline{g}(\underline{x}(t), \underline{u}(t), t) \tag{1.30}$$

To simplify the equations the following abbreviations are introduced:

$$a_1 = m + \frac{I_b}{r^2}$$

$$a_2 = (m\,r^2 + I_b)\frac{1}{r}$$

$$a_3 = m\,g$$

$$b_1 = I_b + I_w$$
$$b_2 = 2\,m$$
$$b_3 = b\,l^2$$
$$b_4 = K\,l^2$$
$$b_5 = (\,m\,r^2 + I_b\,)\,\frac{1}{r}$$
$$b_6 = m\,g \quad . \tag{1.31}$$

With these abbreviations it follows for Eq. 1.27:

$$a_1\,\ddot{x}\,' + a_2\,\ddot{\alpha} - m\,x'\dot{\alpha}^2 = a_3\,\sin(\alpha) \tag{1.32}$$

and for Eq. 1.28:

$$(\,m\,x'^2 + b_1\,)\,\ddot{\alpha} + (\,b_2\,\dot{x}\,'\,x' + b_3\,)\,\dot{\alpha} + b_4\,\alpha + b_5\,\ddot{x}\,' - b_6\,x'\,\cos(\alpha) = u(\,t\,)\,l\,\cos(\alpha) \quad . \tag{1.33}$$

To obtain the state space description Eq. 1.33 is solved for $\ddot{\alpha}$ at first.

$$\ddot{\alpha} = \frac{-(b_2\,\dot{x}'\,x' + b_3\,)\,\dot{\alpha} - b_4\,\alpha - b_5\,\ddot{x}\,'}{m\,x'^2 + b_1} + \frac{b_6\,x'\,\cos(\alpha) + u(\,t\,)\,l\,\cos(\alpha)}{m\,x'^2 + b_1} \quad . \tag{1.34}$$

Inserting Eq. 1.34 in Eq. 1.32 and solving the result for $\ddot{x}\,'$ yields:

$$\ddot{x}\,' = \frac{a_2\,[(b_2\,\dot{x}'\,x' + b_3\,)\,\dot{\alpha} + b_4\,\alpha - b_6\,x'\cos(\alpha)]}{a_1\,(\,m\,x'^2 + b_1\,) - a_2\,b_5} + \frac{(\,m\,x'^2 + b_1\,)\,(\,a_3\,\sin(\alpha) + m\,x'\dot{\alpha}^2\,)}{a_1\,(\,m\,x'^2 + b_1) - a_2\,b_5} - \frac{a_2\,l\,\cos(\alpha)\,u(\,t\,)}{a_1\,(\,m\,x'^2 + b_1) - a_2\,b_5} \quad . \tag{1.35}$$

Now Eq. 1.35 is inserted in Eq. 1.34. The result is solved for $\ddot{\alpha}$ yielding:

$$\dot{x}_1 = x_2 \qquad \text{Eq. 1.37}$$

$$\dot{x}_2 = \frac{a_2\,[\,(\,b_2\,x_1\,x_2 + b_3\,)x_4 + b_4\,x_3 - b_6\,x_1\cos(x_3\,)\,] + (\,m\,x_1^2 + b_1\,)\,(\,a_3\,\sin(x_3\,) + m\,x_1\,x_4^2\,) - a_2\,l\,\cos(\,x_3\,)\,u(\,t\,)}{a_1\,(\,m\,x_1^2 + b_1\,) - a_2\,b_5} \qquad \text{Eq. 1.38}$$

$$\dot{x}_3 = x_4 \qquad \text{Eq. 1.39}$$

$$\dot{x}_4 = \frac{-(\,b_2\,x_1\,x_2 + b_3\,)\,x_4 - b_4\,x_3 + b_6\,x_1\cos(x_3\,)}{m\,x_1^2 + b_1} - \frac{b_5\,(\,a_3\,\sin(x_3\,) + m\,x_1\,x_4^2\,)}{a_1\,(\,m\,x_1^2 + b_1\,) - a_2\,b_5} - \frac{a_2\,b_5\,[(\,b_2\,x_1\,x_2 + b_3\,)\,x_4 + b_4\,x_3 - b_6\,x_1\,\cos(x_3\,)]}{(\,m\,x_1^2 + b_1\,)\,(\,a_1\,(\,m\,x_1^2 + b_1\,) - a_2\,b_5\,)} + \left(1 + \frac{a_2\,b_5}{a_1\,(\,m\,x_1^2 + b_1\,) - a_2\,b_5}\right)\frac{l\,\cos(x_3\,)\,u(\,t\,)}{m\,x_1^2 + b_1} \qquad \text{Eq. 1.40}$$

Figure 1.5: The nonlinear state space description

$$\ddot{\alpha} = \frac{-(b_2 \dot{x}' x' + b_3)\,\dot{\alpha} - b_4\,\alpha + b_6\,x'\cos(\alpha)}{m\,x'^2 + b_1} - \frac{b_5\,(a_3 \sin(\alpha) + m\,x'\,\dot{\alpha}^2)}{a_1\,(m\,x'^2 + b_1) - a_2\,b_5} - \frac{a_2\,b_5\,[\,(b_2 \dot{x}' x' + b_3)\,\dot{\alpha} + b_4\,\alpha - b_6\,x'\cos(\alpha)\,]}{(m\,x'^2 + b_1)\,(a_1\,(m\,x'^2 + b_1) - a_2\,b_5)} + \left(1 + \frac{a_2\,b_5}{a_1\,(m\,x'^2 + b_1) - a_2\,b_5}\right)\frac{l\cos(\alpha)\,u(t)}{m\,x'^2 + b_1} \quad . \tag{1.36}$$

Now the state space description is defined as follows:

$x_1 = x'$	Position of the ball
$x_2 = \dot{x}'$	Velocity of the ball
$x_3 = \alpha$	Angle of the beam
$x_4 = \dot{\alpha}$	Angularvelocity of the beam

With these denotations and Eq. 1.35 resp. Eq. 1.36 one obtains the nonlinear state space description of the ball and beam system as shown in figure 1.5.

1.5 Linearization of the State Space Description of the Nonlinear Model

The concept of a state control applied on the ball and beam system requires a linear plant model. This section describes the linearization of the equations 1.37-1.40 around an operating point $\underline{x}_0$, u_0 by deriving a Taylor series which is truncated after the first term. The operating point is selected as follows:

$$\underline{x}_0 = \begin{bmatrix} x_{10} \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad u_0 \quad . \tag{1.41}$$

Near to the operating point the following approximation is valid in addition:

$$\sin(x_3) \approx x_3\,, \quad \cos(x_3) \approx 1 \quad . \tag{1.42}$$

Ackermann (1988) states the following formulation for small deviations $\underline{x}_0$, u_0:

$$\dot{\underline{x}} = \underline{A}_0\,\underline{x} + \underline{b}_0\,u \tag{1.43}$$

with

$$\underline{A}_0 = \left.\frac{\delta \underline{f}}{\delta \underline{x}}\right|_{x_0,\,u_0}, \quad \underline{b}_0 = \left.\frac{\delta \underline{f}}{\delta u}\right|_{x_0,\,u_0} \quad . \tag{1.44}$$

Calculating the differential quotients results in a linearized system matrix $\underline{A}$ of the ball and beam system with the following component filling:

$$\underline{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ A_{21} & 0 & A_{23} & A_{24} \\ 0 & 0 & 0 & 1 \\ A_{41} & 0 & A_{43} & A_{44} \end{bmatrix} \tag{1.45}$$

and for the control matrix $\underline{b}$:

$$\underline{b} = \begin{bmatrix} 0 \\ B_2 \\ 0 \\ B_4 \end{bmatrix} \quad . \tag{1.46}$$

The components of the $\underline{A}$-matrix are defined as shown in figure 1.6.

The $\underline{b}$-matrix is combined by the components

$$B_2 = -\frac{a_2\,l}{(a_1\,(m\,x_{10}^2 + b_1) - a_2\,b_5)} \tag{1.53}$$

$$B_4 = \left(1 + \frac{a_2\,b_5}{(a_1\,(m\,x_{10}^2 + b_1) - a_2\,b_5)}\right)\frac{l}{m\,x_{10}^2 + b_1} \tag{1.54}$$

The output matrix $\underline{C}$ of the system is given by

$$\underline{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.55}$$

It is noted at this point that the position of the ball x_1 as well as the angle of the beam x_3 are measurable system signals.

With this the linear state space description

$$\dot{x} = \underline{A}\, x + \underline{b}\, u \ , \quad \underline{x}(t_0) = x_0 \qquad (1.56)$$

$$\underline{y} = \underline{C}\, \underline{x} \qquad (1.57)$$

is defined. The following chapter will describe the design of the state control.

1.6 References

/1/ : J.Ackermann :
Abtastregelung, Springer - Verlag, Berlin 1972

/2/ : O. Föllinger :
Regelungstechnik, Hüthig - Verlag, Heidelberg 1994

/3/ : M.Frick :
Mechanik III, Vorlesungssript, Uni-GH Duisburg FB 7, Duisburg 1994

$$A_{21} = \frac{-a_2 b_6 (a_1 (m x_{10}^2 + b_1) - a_2 b_5) + 2 m a_1 a_2 x_{10} (b_6 x_{10} + l u_0)}{(a_1 (m x_{10}^2 + b_1) - a_2 b_5)^2} \qquad (1.47)$$

$$A_{23} = \frac{a_3 (m x_{10}^2 + b_1) + a_2 b_4}{a_1 (m x_{10_2} + b_{1)} - a_2 b_5} \qquad (1.48)$$

$$A_{24} = \frac{a_2 b_3}{(a_1 (m x_{10}^2 + b_1) - a_2 b_5)} \qquad (1.49)$$

$$A_{41} = \frac{b_6 (- m x_{10}^2 + b_1)}{(m x_{10}^2 + b_1)^2} - \left[1 + a_2 b_5 \frac{2 a_1 m x_{10}^2 + 2 a_1 b_1 - a_2 b_5)}{(a_1 (m x_{10}^2 + b_1) - a_2 b_5)^2} \right] \frac{2 m l x_{10} u0}{(m x_{10}^2 + b_1)^2}$$
$$- a_2 b_5 b_6 \frac{m x_{10}^2 (3 a_1 m x_{10}^2 + 2 a_1 b_1 - a_2 b_5) + b_1 (- a_1 b_1 + a_2 b_5)}{[(m x_{10}^2 + b_1) (a_1 (m x_{10}^2 + b_1) - a_2 b_5)]^2} \qquad (1.50)$$

$$A_{43} = \frac{b_4}{m x_{10}^2 + b_1} - \frac{a_3 b_5}{a_1 (m x_{10}^2 + b_1) - a_2 b_5} - \frac{a_2 b_4 b_5}{(m x_{10}^2 + b_1) (a_1 (m x_{10}^2 + b_1) - a_2 b_5)} \qquad (1.51)$$

$$A_{44} = - b_3 \left(\frac{1}{(m x_{10}^2 + b_1)} + \frac{a_2 b_5}{(m x_{10}^2 + b_1) (a_1 (m x_{10}^2 + b_1) - a_2 b_5)} \right) . \qquad (1.52)$$

Figure 1.6: The linear state space description

/4/ : H. Göldner und F. Holzweissig :
Leitfaden der technischen Meschanik,
VEB Fachbuch - Verlag, Leipzig 1989

/5/ : P. Hagedorn :
Technische Meschanik III, Harri Deutsch -
Verlag, Frankfurt a. M. 1990

/6/ : E. Hering u. a :
Physik für Ingenieure, VDI - Verlag,
Düsseldorf 1992

/7/ : L. Meirovith :
Methods of Analytical Dynamics,
McGraw-Hill, Inc., New York 1970

Theoretical Background of the State Controller

Date: 17.03.1997

1 Controller Design in the State Space

This chapter describes the theoretical backgrounds of a state controller design. At first it goes into the state space description of linear sampled data systems.

1.1 State Equations of Sampled Data Systems

In the following it is assumed that the vector differential equation of the linear time-invariant system is known. In this case the state and output equations are stated as (Frank 1994a):

$$\dot{\underline{x}}(t) = \underline{A}\,\underline{x}(t) + \underline{B}\,\underline{u}(t) \quad , \underline{x}(t_0) = \underline{x}_0 \tag{1.1}$$

$$\underline{y}(t) = \underline{C}\,\underline{x}(t) + \underline{D}\,\underline{u}(t) \quad . \tag{1.2}$$

Where

$$\underline{u}(t) = \begin{bmatrix} u_1(t) \\ \cdot \\ \cdot \\ \cdot \\ u_p(t) \end{bmatrix} \tag{1.3}$$

is the input resp. control vector,

$$\underline{x}(t) = \begin{bmatrix} x_1(t) \\ \cdot \\ \cdot \\ \cdot \\ x_n(t) \end{bmatrix} \tag{1.4}$$

is the state vector and

$$\underline{y}(t) = \begin{bmatrix} y_1(t) \\ \cdot \\ \cdot \\ \cdot \\ y_q(t) \end{bmatrix} \tag{1.5}$$

is the output vector of the system. Furthermore one calls the $(n \times n)$-matrix $\underline{A}$ the system matrix, the $(n \times p)$ -matrix $\underline{B}$ the control matrix, the $(q \times n)$-matrix $\underline{C}$ the output matrix and the $(q \times p)$-matrix $\underline{D}$ the feed through matrix.

The general solution of the state differential equation results by means of the Laplace transformation in (Frank 1994a):

$$\underline{x}(t) = \underline{\Phi}(t - t_0)\,\underline{x}(t_0) + \int_{t_0}^{t} \underline{\Phi}(t - \tau)\,\underline{B}\,\underline{u}(\tau)\,d\tau \tag{1.6}$$

with the matrix function

$$\underline{\Phi}(t) = e^{\underline{A}t} = \sum_{\nu=0}^{\infty} \frac{(\underline{A}t)^{\nu}}{\nu!} \quad , \tag{1.7}$$

called fundamental or transition matrix. The state description of linear sampled data systems can be computed from Eq. 1.6 and Eq. 1.7 when the input signal is a piecewise constant time function (Isermann 1987)

$$\underline{u}(t) = \underline{u}(kT), \quad kT \le t \le (k+1)T. \tag{1.8}$$

By integration over the sampling period T one obtains for the state vector at time $(k+1)T$:

$$\underline{x}((k+1)T) = \underline{A}_D(T)\,\underline{x}(kT) + \underline{B}_D(T)\,\underline{u}(kT) \tag{1.9}$$

with the abbreviations

$$\underline{A}_D(T) = \underline{\Phi}(T) = e^{\underline{A}T} \tag{1.10}$$

and

$$\underline{B}_D(T) = \int_0^T \underline{A}_D(\tau)\,\underline{B}\,d\tau \quad . \tag{1.11}$$

Accordingly, the output equation of the sampled data system becomes:

$$\underline{y}(kT) = \underline{C}\,\underline{x}(kT) + \underline{D}\,\underline{u}(kT) \quad . \tag{1.12}$$

In figure 1.1 the block diagram of the sampled data system according to Eq. 1.9 and Eq. 1.12 is displayed.

The solution of Eq. 1.9 is obtained by recursive computation of the state variables at the time $(k+1)\,T$ using the input and state variables at time $k\,T$.

When the eigenvalues of the matrix $\underline{A}_D$ are located inside the unit circle of the z-plane, the system is stable. Analogously to continuous time systems, the eigenvalues can be computed from the roots of the characteristic equation (Isermann 1987):

$$\det(z\,\underline{I} - \underline{A}_D) = 0 \quad . \qquad (1.13)$$

If the eigenvalues of the continuous time systems are already known, the transformation

$$z_i = e^{T\lambda_i}, \quad i = 1,\ldots,n \qquad (1.14)$$

yields the location of the eigenvalues inside the z-plane.

When a system description is given in the state space, the control is realizable by a feedback of all the state variables to the system input introducing the problem of selecting suitable feedback coefficients. There are two different possibilities to solve the mentioned problem (Frank 1994b):

1. Prescribing the dynamic behaviour by selecting locations for the poles of the closed loop system (pole placement).

2. Optimization of a quality criterion (optimum control).

The following section will describe the first method.

1.2 Sampled Data Control with Feedback of the State Vector

It is assumed in the following that a system description according to Eq. 1.9 and Eq. 1.12 is given. If the system is controllable meaning that

$$\text{Rang}\left[\underline{B}_D \;\; \underline{A}_D\,\underline{B}_D \;\; \cdots \;\; \underline{A}_D^{n-1}\,\underline{B}_D\right] = n \qquad (1.15)$$

is valid a state control using pole placement is realizable.

The general structure of a sampled data control system is depicted in figure 1.2. The controller is realized by the state feedback

$$\underline{u}_R(k\,T) = -\underline{F}\,\underline{x}(k\,T). \qquad (1.16)$$

The design of the state controller using pole placement requires that the complete state vector $\underline{x}$ is known. When this vector is not measurable completely, the missing state variables have to be determined by further algorithms

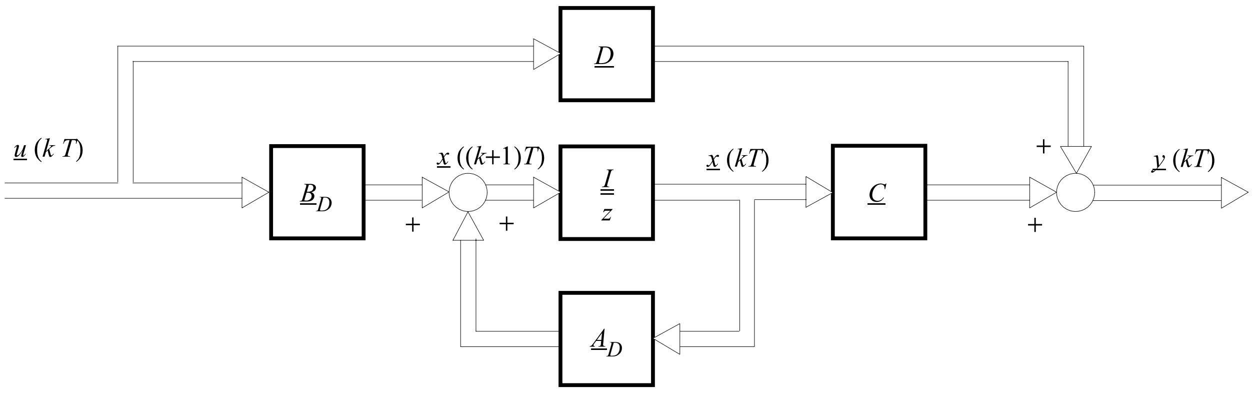


Figure 1.1 : Block diagram of a sampled data system

called observers. The following section 1.3 will deal with this problem. Here it is assumed at first that all the state variables are measurable.

The constant controller coefficients of the feedback matrix $\underline{F}$ are to be selected such that the poles of the open loop system are shifted to stable locations. When the number of inputs is equal to the number of outputs $p = q$, a prefilter given by the matrix $\underline{V}$ ensures that the output vector $\underline{y}$ is equal to the reference vector $\underline{w}$ in steady state conditions. According to figure 1.2 and with

$$\underline{u}(kT) = \underline{V}\,\underline{w}(kT) - \underline{F}\,\underline{x}(kT) \tag{1.17}$$

the system description of the closed control loop is given by:

$$\underline{x}((k+1)T) = (\underline{A}_D - \underline{B}_D\underline{F})\,\underline{x}(kT) + \underline{B}_D\underline{V}\,\underline{w}(kT) \tag{1.18}$$

$$\underline{y}(kT) = (\underline{C} - \underline{D}\,\underline{F})\,\underline{x}(kT) + \underline{D}\,\underline{V}\,\underline{w}(kT) \quad . \tag{1.19}$$

Comparing the equations 1.9 and 1.12 yields the relations between the open and the closed control loop as follows:

$$\begin{aligned} \underline{A}_D &\rightarrow \underline{A}_D - \underline{B}_D\underline{F} \\ \underline{B}_D &\rightarrow \underline{B}_D\underline{V} \\ \underline{C} &\rightarrow \underline{C} - \underline{D}\,\underline{F} \\ \underline{D} &\rightarrow \underline{D}\,\underline{V} \end{aligned} \quad . \tag{1.20}$$

Analogously to Eq. 1.13 the stability behaviour of the closed loop system can be evaluated by considering the eigenvalues of the new system matrix. The eigenvalues $\lambda_{R\,i}$, $i = 1,..., n$ can be computed from the characteristic equation

$$\det[z\,\underline{I} - (\underline{A}_D - \underline{B}_D\underline{F})] = 0. \tag{1.21}$$

1.2.1 Calculation of the Controller Feedback Matrix

Several methods (Ackermann 1988, Föllinger 1990, Frank 1994b and Isermann 1987) are known to compute the matrix $\underline{F}$. Here only two methods are described in detail. It is assumed in the following that the system has only one input signal ($p = 1$). In this case, the n components of the feedback matrix $\underline{F} = \underline{f}^T$ can be determined uniquely by assignment of the eigenvalues of the closed loop (Frank 1994b).

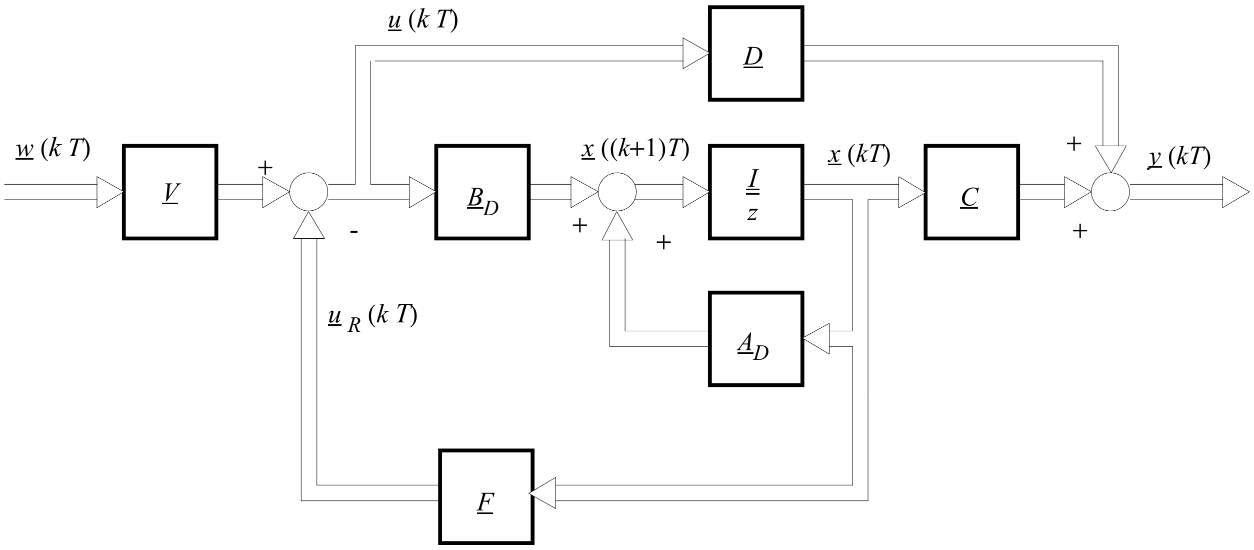


Figure 1.2 : Sampled data control loop with state feedback

Pole placement with preassigned characteristic polynomial

The eigenvalues $\lambda_{R\,1}, ..., \lambda_{R\,n}$ of the closed loop are preassigned. Then the characteristic polynomial is given by

$$P(z) = z^n + p_{n-1} z^{n-1} + p_{n-2} z^{n-2} + \ldots + p_0 \qquad (1.22)$$

A following comparison of the coefficients with

$$\det[z\underline{I} - (\underline{A}_D - \underline{b}_D \underline{f}^T)] = 0 \qquad (1.23)$$

results in the desired feedback vector.

Pole placement using the Ackermann equation

If the system is described in an arbitrary form and if it is controllable in addition the feedback vector is determined for an assigned characteristic polynomial according to Eq. 1.22 from the following relation:

$$\underline{f}^T = p_0 \underline{q}_s^T + \ldots + p_{n-1} \underline{q}_s^T \underline{A}_D^{n-1} + \underline{q}_s^T \underline{A}_D^n \; . \qquad (1.24)$$

The term $\underline{q}_s^T$ is here the last row of the inverse controllability matrix

$$\underline{Q}_s^{-1} = \begin{bmatrix} \underline{b}_D & \underline{A}_D \underline{b}_D & \cdots & \underline{A}_D^{n-1} \underline{b}_D \end{bmatrix}^{-1} \; . \qquad (1.25)$$

Eq. 1.24 is known from the literature as the Ackermann equation.

Determination of the prefilter

To determine the matrix $\underline{V}$ it is assumed in the following

1. The matrix $\underline{F} = \underline{f}^T$ is known already.

2. The feed through matrix $\underline{D} = 0$, which is the case for most of the real systems.

Transforming the equations 1.18 and 1.19 into the z-domain yields:

$$z\,\underline{X}(z) = (\underline{A}_D - \underline{B}_D \underline{F})\,\underline{X}(z) + \underline{B}_D \underline{V}\,\underline{W}(z) \qquad (1.26)$$

and

$$\underline{Y}(z) = \underline{C}\,\underline{X}(z) \; . \qquad (1.27)$$

Solving Eq. 1.26 for $\underline{X}(z)$ and substituting the result in Eq. 1.27 results in

$$\begin{aligned} \underline{Y}(z) &= \underline{C}\,(z\underline{I} - \underline{A}_D + \underline{B}_D \underline{F})^{-1} \underline{B}_D \underline{V}\,\underline{W}(z) \\ &= \underline{G}(z)\,\underline{W}(z) \; . \end{aligned} \qquad (1.28)$$

In this equation the term $\underline{G}(z)$ denotes the z-transfer function of the system. In the steady state the output vector can be computed by the final value theorem to

$$\begin{aligned} \underline{y}(+\infty) &= \lim_{z \to 1} [(z-1)\,\underline{G}(z)\,\underline{W}(z)] \\ &= \underline{G}(1) \lim_{z \to 1} [(z-1)\,\underline{W}(z)] \\ &= \underline{G}(1)\,\underline{w}(+\infty) \end{aligned} \qquad (1.29)$$

In order for the output to be equal to the setpoint in the steady state ($\underline{y} = \underline{w}$), it must be required for the Z-transfer function

$$\underline{G}(1) = \underline{I} \; . \qquad (1.30)$$

This condition yields together with Eq. 1.28 the desired relation for the prefilter $\underline{V}$

$$\underline{V} = [\underline{C}\,(\underline{I} - \underline{A}_D + \underline{B}_D \underline{F})^{-1} \underline{B}_D]^{-1} \; . \qquad (1.31)$$

Now all the matrices of the state controller are determined. The next section will describe how state variables which cannot be measured are reconstructed by means of observers.

1.3 State Observers

In the previous chapter 1.2 different methods to determine the feedback matrix $\underline{F}$ of a state control have been described. There it was assumed that the state vector of the system is known. In practice however, it is often very costly or altogether impossible to determine the state vector $\underline{x}$ by measurements. By means of state observers however, it is possible, to reconstruct the states of the system using the input and output signals. The use of state observers is always possible for observable systems, i.e. the following relation has to be valid (Frank 1994b):

$$\text{Rang}\left[\underline{C}^T \;\; \underline{A}_D^T \underline{C}^T \;\; \dots \;\; (\underline{A}_D^T)^{n-1} \underline{C}^T\right] = n \quad . \qquad (1.32)$$

At first the *Luenberger observer* will be described shortly to get a better understanding of a *reduced order observer* applied on the ball and beam system.

1.3.1 The Luenberger Observer

The background of this observer can be derived by the block diagram from figure 1.3. The difference between the measured output vector $\underline{y}$ and the output vector of the model $\hat{\underline{y}} = \underline{C}\,\hat{\underline{x}}$ is feed back via the matrix $\underline{F}_B$ to the input of the model. The matrix $\underline{F}_B$ is to be determined such that the following relation for the estimation error $\underline{e}(k\,T)$ is valid for arbitrary initial states $\underline{x}_0$, $\hat{\underline{x}}_0$:

$$\lim_{k\to\infty} \underline{e}(k\,T) = \lim_{k\to\infty} (\,\underline{y}(k\,T) - \hat{\underline{y}}(k\,T)\,) \equiv 0 \quad . \qquad (1.33)$$

For a system described as

$$\underline{x}(\,(k+1)\,T\,) = \underline{A}_D\,\underline{x}(\,k\,T\,) + \underline{B}_D\,\underline{u}(\,k\,T\,) \quad , \underline{x}(0) = \underline{x}_0 \qquad (1.34)$$

$$\underline{y}(\,k\,T\,) = \underline{C}\,\underline{x}(\,k\,T\,) \qquad (1.35)$$

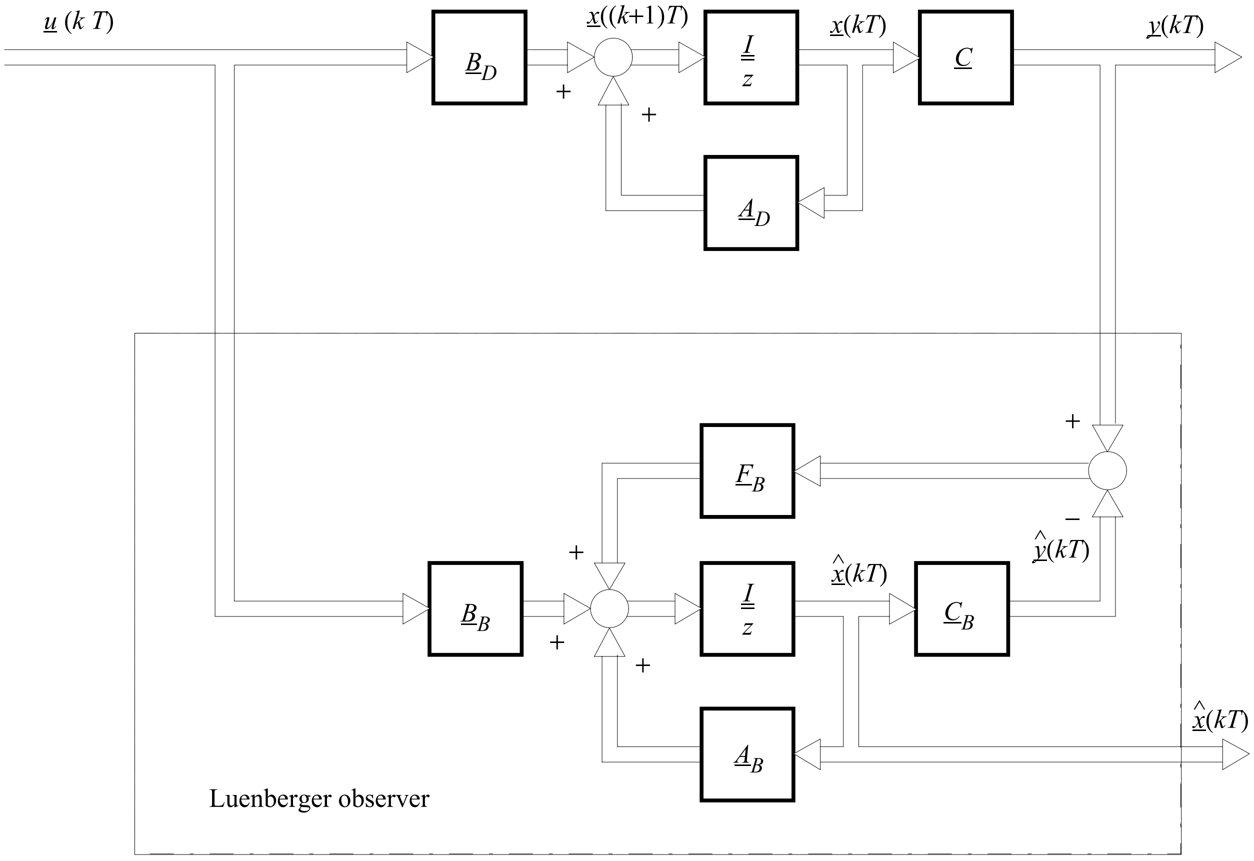


Figure 1.3 : Block diagram of the Luenberger state observer

a Luenberger observer is formulated as a dynamic system with the differential state equation

$$\underline{\hat{x}}((k+1)T) = (\underline{A}_D - \underline{F}_B\,\underline{C})\,\underline{\hat{x}}(kT) + \underline{B}_D\,\underline{u}(kT) + \underline{F}_B\,\underline{y}(kT) \quad . \tag{1.36}$$

Eq. 1.33 is valid if and only if the eigenvalues of the system matrix $\underline{A}_B$ of the observer

$$\underline{A}_B = \underline{A}_D - \underline{F}_B\,\underline{C} \tag{1.37}$$

are located inside the unit circle of the *z*-plane. The eigenvalues are determined from the characteristic equation

$$\det[z\underline{I} - \underline{A}_D + \underline{F}_B\,\underline{C}] = 0 \quad . \tag{1.38}$$

One method to determine the observer parameters is to transpose the dynamic matrix according to Eq. 1.37 and to design a fictitious state controller (Föllinger 1994). For the following methods it is assumed that the system is a single input-single output system ($q = 1$).

Pole placement by a preassigned characteristic equation

Here the eigenvalues $\lambda_{B\,1}, ..., \lambda_{B\,n}$ of the observer are preassigned. The characteristic polynomial is given by

$$P_B(z) = z^n + p_{n-1}z^{n-1} + p_{n-2}z^{n-2} + \ldots + p_0 \tag{1.39}$$

A comparison of the coefficients with Eq. 1.38 results in the desired feedback vector of the observer.

Pole placement using the Ackermann equation

When Eq. 1.38 is written in the form

$$\begin{aligned}\det[z\underline{I} - (\underline{A}_D - \underline{f}_B\,\underline{c}^T)]^T &= \det[(z\underline{I})^T - (\underline{A}_D - \underline{f}_B\,\underline{c}^T)^T] \\ &= \det[z\underline{I} - (\underline{A}_D^T - \underline{c}\,\underline{f}_B^T)] \quad ,\end{aligned} \tag{1.40}$$

a comparison with Eq. 1.23 yields the following relations:

$$\begin{aligned}\underline{A}_D &\rightarrow \underline{A}_D^T \\ \underline{b}_D &\rightarrow \underline{c} \\ \underline{f}^T &\rightarrow \underline{F}_B^T\end{aligned} \quad . \tag{1.41}$$

With this the observer parameters are determined as follows:

If the system is described in an arbitrary form and provided the system is observable the feedback matrix $\underline{f}_B$ is determined for a preassigned characteristic polynomial according to Eq. 1.39 from the following relation:

$$\begin{aligned}\underline{f}_B &= (p_0\,\underline{q}_B^T + \ldots + p_{n-1}\,\underline{q}_B^T(\underline{A}_D^T)^{n-1} + \underline{q}_B^T(\underline{A}_D^T)^n)^T \\ &= (p_0\,\underline{q}_B + \ldots + p_{n-1}\,\underline{A}_D^{n-1}\,\underline{q}_B + \underline{A}_D^n\,\underline{q}_B) \quad .\end{aligned} \tag{1.42}$$

Where the term $\underline{q}_B$ denotes the last column of the observabilty matrix

$$\underline{Q}_B^{-1} = \begin{bmatrix} \underline{c}^T \\ \underline{c}^T\underline{A}_D \\ \cdot \\ \cdot \\ \cdot \\ \underline{c}^T\underline{A}_D^{n-1} \end{bmatrix}^{-1} \quad . \tag{1.43}$$

For a system with multiple output signals the observer matrix can no longer be determined uniquely. In this case additional conditions besides the observer poles are to be assigned (Föllinger 1994).

In was already noted in the chapter modelling that two of the four state variables (ball position x_1 and beam angle x_3) are measured from the BW500 laboratory setup. According to Föllinger (1994) only $n - q$ state variables are to be reconstructed by the observer when q state variables are known from measurements. So a reduced order observer is sufficient in this case offering the advantage that the estimated signals converge faster to the real values. The following section will describe the design of a reduced order observer.

1.3.2 Reduced Order Observer

The initial consideration is to introduce the measured signals as state variables (Isermann 1987). The state vector $\underline{x}$ becomes

$$\underline{x} = \begin{bmatrix} \underline{y}(kT) \\ \hat{\underline{x}}_B(kT) \end{bmatrix} \qquad (1.44)$$

where $\hat{\underline{x}}_B(kT)$ denotes the vector containing the state variables which cannot be measured. Then it follows from Eq. 1.9

$$\begin{bmatrix} \underline{y}((k+1)T) \\ \hat{\underline{x}}_B((k+1)T) \end{bmatrix} = \begin{bmatrix} \underline{A}_{11} & \underline{A}_{12} \\ \underline{A}_{21} & \underline{A}_{22} \end{bmatrix} \begin{bmatrix} \underline{y}(kT) \\ \hat{\underline{x}}_B(kT) \end{bmatrix} + \begin{bmatrix} \underline{B}_1 \\ \underline{B}_2 \end{bmatrix} \underline{u}(kT) \qquad (1.45)$$

which becomes after some conversions:

$$\hat{\underline{x}}_B((k+1)T) = \underline{A}_{22}\hat{\underline{x}}_B(kT) + [\underline{A}_{21}\underline{y}(kT) + \underline{B}_2\underline{u}(kT)] \qquad (1.46)$$

$$\underline{y}((k+1)T) - \underline{A}_{11}\underline{y}(kT) - \underline{B}_1\underline{u}(kT) = \underline{A}_{12}\hat{\underline{x}}_B(kT) \quad . \qquad (1.47)$$

Considering that the term $\hat{\underline{x}}_B$ denotes the desired vector and that

$$\underline{y}((k+1)T) - \underline{A}_{11}\underline{y}(kT) - \underline{B}_1\underline{u}(kT)$$

is a known vector then this relation may be formulated in a typical state space description according to Eq. 1.34 and Eq. 1.35. This results in:

$$\begin{aligned} \underline{x}(kT) &\rightarrow \hat{\underline{x}}_B(kT) \\ \underline{A}_D &\rightarrow \underline{A}_{22} \\ \underline{B}_D\underline{u}(kT) &\rightarrow \underline{A}_{21}\underline{y}(kT) + \underline{B}_2\underline{u}(kT) \\ \underline{y}(kT) &\rightarrow \underline{y}((k+1)T) - \underline{A}_{11}\underline{y}(kT) - \underline{B}_1\underline{u}(kT) \\ \underline{C} &\rightarrow \underline{A}_{12} \quad . \end{aligned} \qquad (1.48)$$

Using these relations and the observer equation 1.36 to design a reduced order observer it follows:

$$\begin{aligned} \hat{\underline{x}}_B((k+1)T) = {} & (\underline{A}_{22} - \underline{L}_B\underline{A}_{12})\hat{\underline{x}}_B(kT) + \underline{A}_{21}\underline{y}(kT) \\ & + \underline{B}_2\underline{u}(kT) + \underline{L}_B[\underline{y}((k+1)T) \\ & - \underline{A}_{11}\underline{y}(kT) - \underline{B}_1\underline{u}(kT)] \quad . \end{aligned} \qquad (1.49)$$

In Eq. 1.49 the right side however still contains the vector $\underline{y}((k+1)T)$ which is unknown at time kT. Substituting the vector $\hat{\underline{x}}_B$ by the observer state variables

$$\underline{\eta}(kT) = \hat{\underline{x}}_B(kT) - \underline{L}_B\underline{y}(kT) \qquad (1.50)$$

in Eq. 1.49 results in the observer equation which corresponds to the equations 1.46 and 1.47 of the reduced system:

$$\begin{aligned} \underline{\eta}((k+1)T) &= (\underline{A}_{22} - \underline{L}_B\underline{A}_{12})\underline{\eta}(kT) \\ &\quad + [(\underline{A}_{22} - \underline{L}_B\underline{A}_{12})\underline{L}_B + \underline{A}_{21} - \underline{L}_B\underline{A}_{11}]\underline{y}(kT) \\ &\quad + (\underline{B}_2 - \underline{L}_B\underline{B}_1)\underline{u}(kT) \\ &= \underline{A}_B\underline{\eta}(kT) + \underline{F}_B\underline{y}(kT) + \underline{B}_B\underline{u}(kT) \end{aligned} \qquad (1.51)$$

where the estimated state variable $\hat{\underline{x}}_B$ can be calculated from:

$$\hat{\underline{x}}_B(kT) = \underline{\eta}(kT) + \underline{L}_B\underline{y}(kT) \quad . \qquad (1.52)$$

Here the following relations are valid:

$$\begin{aligned} \underline{A}_B &= \underline{A}_{22} - \underline{L}_B\underline{A}_{12} \\ \underline{F}_B &= \underline{A}_B\underline{L}_B + \underline{A}_{21} - \underline{L}_B\underline{A}_{11} \\ \underline{B}_B &= \underline{B}_2 - \underline{L}_B\underline{B}_1 \quad . \end{aligned} \qquad (1.53)$$

The elements of the observer matrix $\underline{L}_B$ can be determined according to the method described in section 1.3.1 by substituting the corresponding relations in Eq. 1.40 and Eq. 1.42.

1.4 State Observer in the Control Loop

Up to now the observer was considered as a single dynamic system providing the complete state of a system with respect to its input and output signals. The next step is to include the observer in a state control, i. e. to provide the controller from section 1.2 with the estimated state vector. Figure 1.4 displays the corresponding block diagram with an reduced order observer. The output matrices $\underline{C}_B$ and $\underline{V}_B$ combine the state vector $\underline{x}$ according to the design of the controller and observer. At this point it is to be considered if the dynamic of the closed control loop is modified after inserting the dynamic system observer, i. e. if the eigenvalues $\lambda_{R\,1},..., \lambda_{R\,n}$ as assigned during the controller design were shifted to undesired locations. It is relatively easy to show (Frank 1994b) that this shifting does not occur, instead the observer eigenvalues are only added to the eigenvalues of the closed loop without an observer. In case the system is controllable and observable the eigenvalues $\lambda_{B\,i}$, $i = 1,..., n-q$ of the reduced order observer as well as the eigenvalues $\lambda_{R\,i}$, $i = 1,..., n$ are assignable separately at arbitrary locations (see also Föllinger 1994).

The following section will describe a method to estimate an additional disturbance signal z_S which cannot be measured by using a *disturbance observer*.

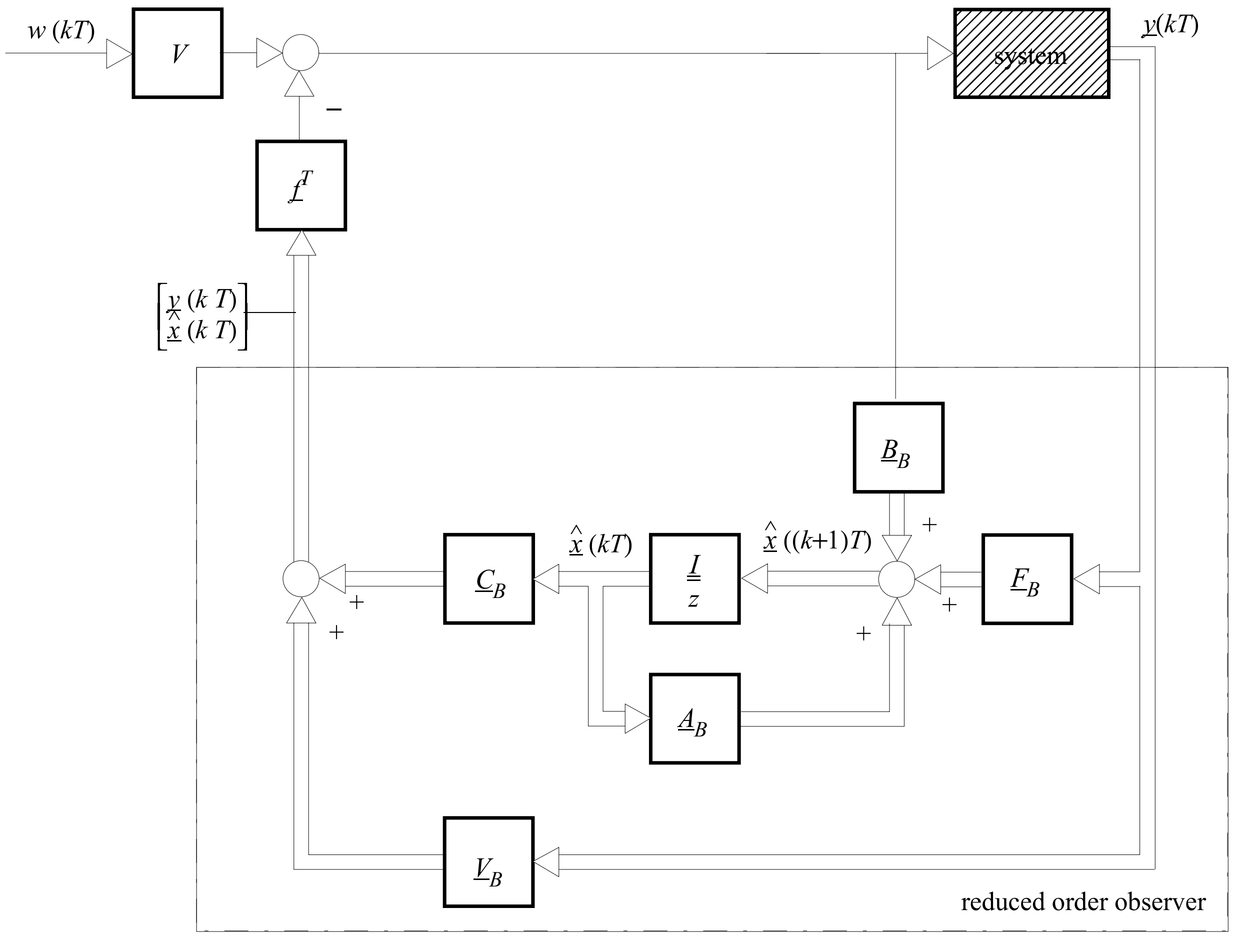


Figure 1.4 : Block diagram of the controlled system with state feedback and reduced order observer

1.5 Disturbance Observer

The previous sections did not consider disturbance effects which are active over a longer period of time. Only the initial vector $\underline{x}_0$ may be taken as a disturbance signal. It is therefore required to consider in addition disturbance signals acting on real plants. According to Föllinger (1994) this may be realized by two different methods:

1. Extending the method of feeding forward the disturbance signal for the state space.

2. Transferring the classical PI controller structure to the state space.

The following will describe the first method in case of a disturbance signal which cannot be measured.

A homogeneous differential equation may be formulated to estimate an unknown disturbance signal z_S described only by a poor knowledge of its shape. The disturbance signal is generated by random initial conditions of this equation. With this method the signal z_S is combined by a stochastic portion with respect to the unknown initial conditions and a deterministic portion represented by the typical signal shape resulting from the homogeneous differential equation. Converting this formulation to the standard state space description results in the discrete mathematical disturbance model for the general case:

$$\begin{aligned} \underline{x}_S((k+1)\,T) &= \underline{A}_S\,\underline{x}_S(k\,T)\quad,\ \underline{x}_S(0) = \underline{x}_{S0} \\ \underline{z}_S(k\,T) &= \underline{C}_S\,\underline{x}(k\,T)\quad, \end{aligned} \tag{1.54}$$

where x_S denotes the state vector and z_S denotes the output vector of this formulation.

Figure 1.5 displays the structure of the extended plant model including the disturbance model. With this the state equations of the extended plant model may be formulated as follows:

$$\begin{bmatrix} \underline{x}((k+1)\,T) \\ \underline{x}_S((k+1)\,T) \end{bmatrix} = \begin{bmatrix} \underline{A}_D & \underline{E}_D\,\underline{C}_S \\ \underline{0} & \underline{A}_S \end{bmatrix} \begin{bmatrix} \underline{x}(k\,T) \\ \underline{x}_S(k\,T) \end{bmatrix} + \begin{bmatrix} \underline{B}_D \\ \underline{0} \end{bmatrix} \underline{u}(k\,T)\quad, \tag{1.55}$$

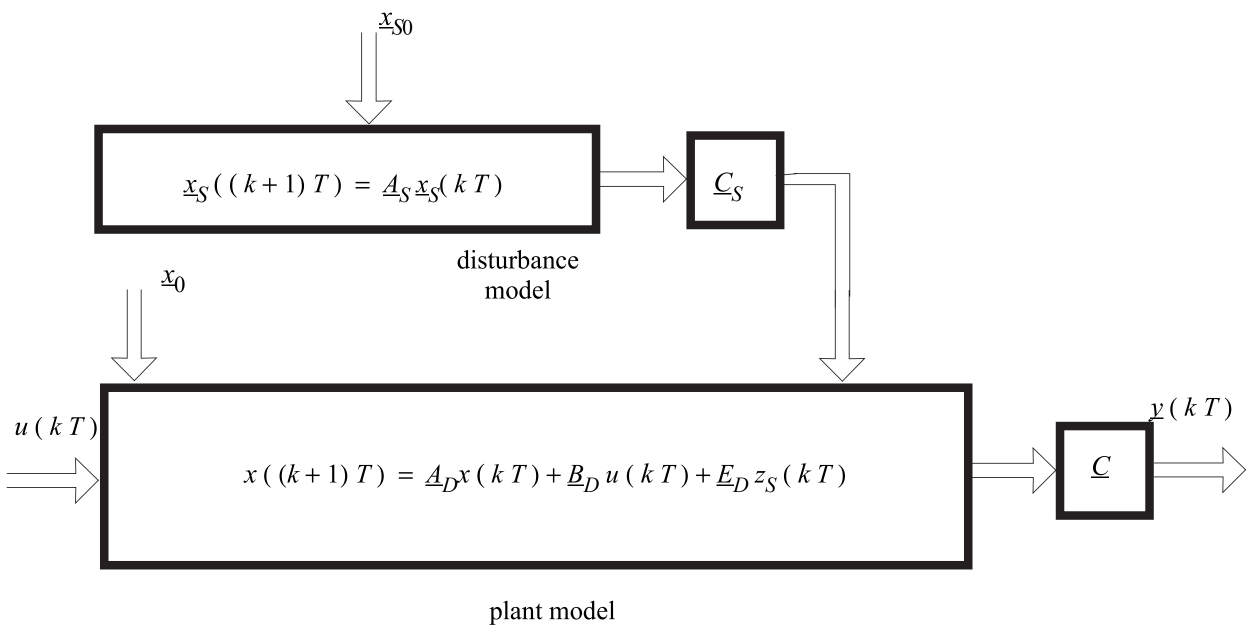

$\underline{x}_{S0}$
$\underline{x}_S((k+1)\,T) = \underline{A}_S\,\underline{x}_S(k\,T)$
$\underline{C}_S$
disturbance model
$\underline{x}_0$
$u(k\,T)$
$x((k+1)\,T) = \underline{A}_D x(k\,T) + \underline{B}_D\,u(k\,T) + \underline{E}_D\,z_S(k\,T)$
$\underline{C}$
$\underline{y}(k\,T)$
plant model

Figure 1.5: Extended plant model

$$\underline{y}(k\,T) = [\,\underline{C} \quad \underline{0}\,]\begin{bmatrix} \underline{x}(k\,T) \\ \underline{x}_S(k\,T) \end{bmatrix} . \qquad (1.56)$$

Eq. 1.55 formulates the standard state space description of the plant where only the control vector $\underline{u}$ and the initial condition

$$\underline{x}_{0E} = \begin{bmatrix} \underline{x}_0 \\ \underline{x}_{S0} \end{bmatrix} \qquad (1.57)$$

are acting as external signals. Therefore a controller as well as an observer are designable according to the methods described above. It is obvious from figure 1.5 that $\underline{u}$ cannot influence the disturbance model in any case. With respect to this the controller is selected as follows:

$$\underline{u}(k\,T) = -\underline{F}\,\underline{x}(k\,T) + \underline{u}_S \ . \qquad (1.58)$$

However the signal z_S was assumed to be not measurable. An observer for the extended plant model, called disturbance observer, provides an estimation $\hat{\underline{z}}_S$ of the disturbance vector. The disturbance observer is driven by the estimation error $\underline{e}$. An approximated disturbance compensation is realized using $\hat{\underline{z}}_S$ in the following relation (Föllinger 1994):

$$\underline{u}_S = -(\underline{B}_D^T\underline{B}_D)^{-1}\,\underline{B}_D^T\,\underline{E}_D\,\hat{\underline{z}}_S \ . \qquad (1.59)$$

After describing the theoretical background of the complete state control the following chapter will deal with the realization of the state control for the BW500 laboratory experiment.

1.6 References

/1/ : O. Föllinger :
Regelungstechnik, Hüthig - Verlag, Heidelberg 1994

/2/ : P. M. Frank :
Regelungstechnik II, Vorlesungssript, Uni-GH Duisburg FB 9, Duisburg 1994a

/3/ : P. M. Frank :
Regelungstechnik III, Vorlesungssript, Uni-GH Duisburg FB 9, Duisburg 1994b

/4/ : R. Isermann :
Digitale Regelsysteme I, Springer - Verlag, Berlin 1987

/5/ : A. J. Laub und J. N. Little :
Control System Toolbox for use with Matlab, User's Guide, The Math Works, Inc., Sherborn 1986

2 Realization of the State Controller

This chapter describes the realization of the state control for the ball and beam system. This includes the denotation of all the BW500 system variables. The pole placement for the controller and the observer is carried out by means of simulating the closed control loop containing the linear and the non-linear plant model. In addition an algorithm for the constant compensation of the sticking friction of the BW500 driving mechanic is derived. Then a simple filter method for the camera signal indicating the ball position is described. At first the model parameters introduced in the chapter "Modelling" are determined.

2.1 Model Parameters for the Ball and Beam System

Some of the parameters were determined by calculating the mean values of the results of extensive measurings taken from the laboratory setup. The values are shown in table 2.1.

Using the values from table 2.1 the still missing model parameters are determined as follows:

Roll Radius of the Ball

The roll radius of the ball r results from the formulation:

$$r = \sqrt{R^2 - (D/2)^2} = 0{,}018\, m \quad . \qquad (2.1)$$

Moment of Inertia of the Ball

Using a steal ball the moment of inertia I_b is calculated by (Frik 1994):

$$I_b = \frac{2}{5} m\, R^2 = 4{,}32\ 10^{-5}\, kg\, m^2 \quad . \qquad (2.2)$$

Description	Denotation	Value	Unit
Ball radius (steal)	R	0,02	m
Ball mass	m	0,27	kg
Beam mass	M	1,122	kg
Beam radius	l_w	050	m
Force-distance	l	0,49	m
Rail distance	D	0,017	m
Friction beam-drive	b	1,0	Ns/m
Stiffness driving-spring	K	0,001	N/m

Table 2.1 : Determined physical system values

Moment of Inertia of the Beam

The complete moment of inertia I_w of the beam is combined by the moment of inertia of the rail I_l and the moment of inertia of the driving mechanic I_m. For the rail with its u-profile the moment of inertia of a long rod is taken as a first approximation (Frik 1994):

$$I_l = \frac{1}{3} l_w^2 M = 9{,}35\ 10^{-2}\, kg\, m^2 \quad . \qquad (2.3)$$

The moment I_m denotes the resulting moment for the beam and not the primary moment of inertia of the drive, which is neglectable. On determination of I_m it is to be considered that the primary moment is transformed to the beam with the squared transmission ratio μ. I_m results with $\mu = 50$:

$$I_m = \mu^2\, 0{,}177\ 10^{-4}\, kg\, m^2 \approx 0{,}5\, I_l \quad . \qquad (2.4)$$

With this

$$I_w = I_l + I_m \approx 1{,}5\, I_l \quad . \qquad (2.5)$$

Now all the model parameters are known. They will be taken to calculate the matrices of the state space description according to the following section.

2.2 Linear State Space Model

A suitable operating point $\underline{x}_0, u_0$ is to be selected to linearize the equations 1.37-1.40. The non-linear state equations are linearized around

$$\underline{x}_0 = \begin{bmatrix} x_{10} \\ 0 \\ 0 \\ 0 \end{bmatrix} = \underline{0} \qquad (2.6)$$

with respect to a symmetric behaviour around the middle of the beam. This selection is the only possible one to obtain a beam symmetry for deviations of the ball position in either positive or negative direction. With this the driving force u_0 is given in the operating point by:

$$u_0 = 0 \quad . \qquad (2.7)$$

According to the correlations described in the chapter "Modelling" the result for the linearized system matrix $\underline{A}$ is:

$$\underline{A} = \begin{bmatrix} 0 & 1/s & 0 & 0 \\ -0{,}0342\ 1/s^2 & 0 & 6{,}592\ m/s^2 & 0{,}031\ m/s^2 \\ 0 & 0 & 0 & 1/s \\ 18{,}898\ 1/(ms^2) & 0 & -0{,}344\ 1/s^2 & -1{,}713\ 1/s \end{bmatrix}, \qquad (2.8)$$

and the result for the control vector $\underline{b}$ is:

$$\underline{b} = \begin{bmatrix} 0 \\ -0{,}0633\, m/(s^2\, N) \\ 0 \\ 3{,}4960\ 1/(s^2\, N) \end{bmatrix} \qquad (2.9)$$

Now the linear state space description of the ball and beam system is known completely. Therefore the design of the state controller is described in the following section.

2.3 Design of the State Controller

We start with a short consideration of the open control loop before the feedback coefficients of $\underline{F}$-matrix as well as the prefilter V are designed.

2.3.1 Eigenvalues of the Ball and Beam System

Figure 2.1 shows the eigenvalues λ_i, $i = 1,..., 4$ of the plant in the z-plane calculated by means of Eq. 1.13. The system is unstable because a pole is outside the unit circle.

The state feedback as shown in figure 2.2 stabilizes the plant as will be described in the following.

2.3.2 Controller Design Using Pole Placement

The aim of this section is to determine a control matrix $\underline{F} = \underline{f}^T$, so that the closed control loop obtains a desired dynamic behaviour. With respect to the dynamic it is to be considered that the BW500 system provides a maximum control force of $u_{max} = 6{,}45\ N$. Therefore the pole placement has to follow this limitation, i.e. the selected poles must not be located to far left of the $j\omega$-axis in the s-plane respectively corresponding locations in the

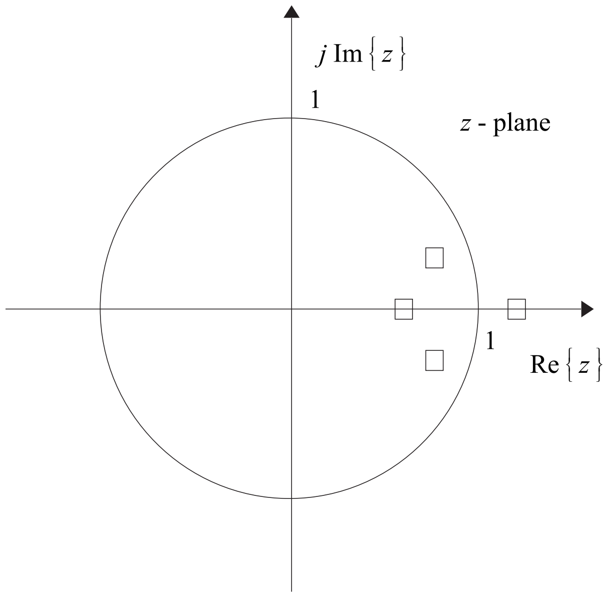


Figure 2.1 : Poles of the open loop system in the z-plane

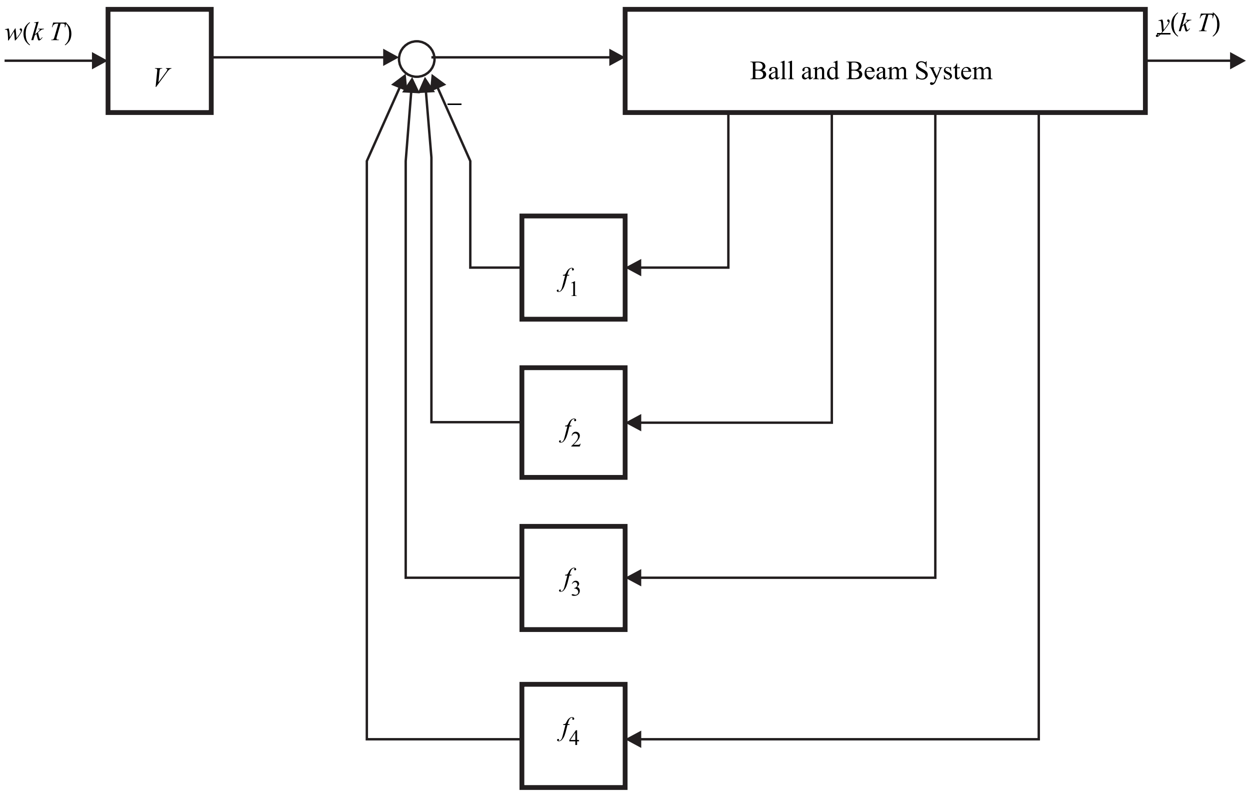


Figure 2.2: State feedback in the BW500 system

z-plane. Furthermore the linearization of the state space description required a small angle of the beam α. Performing the pole placement with respect to the mentioned constraints leads to the following eigenvalue configuration of the ball and beam system:

$$\lambda_{R\,1} = e^{-T\ 1/s} = 0{,}9512$$

$$\lambda_{R\,2} = e^{-T\,5\ 1/s} = 0{,}7788$$

$$\lambda_{R3} = e^{-T\,15\ 1/s} = 0{,}4724$$

$$\lambda_{R\,4} = \lambda_{R\,3}\ . \qquad (2.10)$$

With these values the $\underline{f}^T$-matrix results according to Eq. 1.24

$$\underline{f} = \begin{bmatrix} 27{,}170\ N/m \\ 29{,}82\ Ns^2/m \\ 58{,}78\ N/rad \\ 6{,}39\ Ns/rad \end{bmatrix} . \qquad (2.11)$$

It is to be noticed that the calculations were carried out for the z-plane using the program *Matlab* with control tool box. The z-plane had to be used because the control of the BW500 system with a PC realizes a sampled data control. Furthermore considerable differences between the design of the continuous or the discrete system may be obtained depending on the filling of the matrices of the closed control loop. The design used mainly the Matlab commands *c2d* and *place* (detailed description see Laub, Little 1986).

To determine the prefilter V it is not necessary for the ball and beam system to use the complex matrix equation 1.31. In this case the consideration of the steady state provides an alternative solution:

$$x_2(\infty) = x_3(\infty) = x_4(\infty) = 0 \quad . \qquad (2.12)$$

With this it follows from figure 2.2:

$$u(\infty) = V\,w(\infty) - f_1\,x_1(\infty) \quad . \qquad (2.13)$$

So the control force has to compensate the weight of the ball acting on the arm of the beam. In this case the following is valid:

$$u(\infty) = -\frac{m\,g\,x_1(\infty)}{l} \quad . \qquad (2.14)$$

According to Eq. 1.19 with $x_3(\infty) = 0$ it follows in addition:

$$y(\infty) = x_1(\infty) \quad . \qquad (2.15)$$

On the other hand the steady state requires $y(\infty) = w(\infty)$ yielding with Eq. 2.15:

$$x_1(\infty) = w(\infty) \quad . \qquad (2.16)$$

This correlation and the insertion of Eq. 2.14 in Eq. 2.13 results for the prefilter:

$$V = f_1 - \frac{m\,g}{l} = 21{,}7607\ N/m \quad . \qquad (2.17)$$

Now the design of the state controller is complete so that the realization of the state observer can follow as described in the following section.

2.4 Design of Reduced Order State Observer

For the BW500 laboratory setup the state variables x_2 and x_4 cannot be measured. Therefore they have to be reconstructed by an observer with reduced order. The theoretical background have been described in chapter 1.3.2.

All the measured variables as well as the control force are used to estimate the missing states as shown in figure 2.3.

So that the measured vector $\underline{y}$ as an input of the observer contains the elements $\underline{y} = [\,x_1 \quad x_3\,]^T$. With Eq. 1.45 the state equation of the ball and beam system may be stated as follows:

$$\begin{bmatrix} x_1\,((k+1)\,T) \\ x_3\,((k+1)\,T) \\ \cdots \\ \hat{x}_2\,((k+1)\,T) \\ \hat{x}_4\,((k+1)\,T) \end{bmatrix} = \left[\begin{array}{cc:cc} A_{D\,11} & A_{D\,13} & A_{D\,12} & A_{D\,14} \\ A_{D\,31} & A_{D\,33} & A_{D\,32} & A_{D34} \\ \cdots & \cdots & \cdots & \cdots \\ A_{D21} & A_{D\,23} & A_{D\,22} & A_{D\,24} \\ A_{D\,41} & A_{D43} & A_{D\,42} & A_{D\,44} \end{array}\right] \begin{bmatrix} x_1\,(k\,T) \\ x_3\,(k\,T) \\ \cdots \\ \hat{x}_2\,(k\,T) \\ \hat{x}_4\,(k\,T) \end{bmatrix} + \begin{bmatrix} B_{D\,1} \\ B_{D\,3} \\ \cdots \\ B_{D\,2} \\ B_{D\,4} \end{bmatrix} \underline{u}(k\,T) \quad . \qquad (2.18)$$

Where the symbols

$$A_{D\,i\,j}\,,\,B_{D\,i} \quad i,j = 1\,,\,...,\,4$$

denote the elements of the discrete system matrix $\underline{A}_D$.

The poles of the observer are selected as

$$z_i = e^{-T\,50\ 1/s} = 0{,}0821\ ,\ i = 1,\,2 \qquad (2.19)$$

With this the corresponding observer matrices are given as follows:

$$\underline{L}_B = \begin{bmatrix} 18{,}3520 & 0{,}1322 \\ 0{,}3248 & 17{,}4363 \end{bmatrix}$$

$$\underline{A}_B = \begin{bmatrix} 0{,}0821 & 0 \\ 0 & 0{,}0821 \end{bmatrix}$$

$$\underline{F}_B = \begin{bmatrix} -16{,}8555 & 0{,}0570 \\ 0{,}2069 & -16{,}0149 \end{bmatrix}$$

$$\underline{B}_B = \begin{bmatrix} -0{,}0018 \\ 0{,}0934 \end{bmatrix}$$

$$\underline{C}_B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\underline{V}_B = \begin{bmatrix} 1 & 0 \\ l_{11} & l_{12} \\ 0 & 1 \\ l_{21} & l_{22} \end{bmatrix} . \qquad (2.20)$$

It is to be noticed that the elements of the matrices above include units. So their values are not comparable directly.

The following section will extend the given plant model to consider the effects of friction on the BW500 driving mechanic.

2.5 Design of a Disturbance Observer

The sticking friction of the BW500 driving mechanic was neglected on the mathematical modelling of the plant. To establish a control for the real system the friction is estimated by a disturbance controller as described in chapter 1.5.

The sticking friction behaves like a piece-wise constant disturbance signal U_{S0} (see figure 2.4). However the distinct times t_v when U_{S0} changes its amplitude by the values h_v are unknown. So the disturbance signal may be interpreted as a solution of the homogenous differential equation

$$\dot{x}_S = 0. \qquad (2.21)$$

With this it follows for the matrices of the disturbance model from Eq. 1.54:

$$\underline{A}_S = 0 \quad \underline{C}_S = 1 \ . \qquad (2.22)$$

Figure 2.5 shows that U_{S0} acts via the control vector $\underline{E}_D = \underline{B}_D$ on $\underline{x}((k+1)\,T)$. So the extended system may be written in the form:

$$\begin{bmatrix} \underline{x}((k+1)\,T) \\ \hat{x}_S((k+1)\,T) \end{bmatrix} = \begin{bmatrix} \underline{A}_D & \underline{B}_D \\ \underline{0}^T & 0 \end{bmatrix} \begin{bmatrix} \underline{x}(k\,T) \\ \hat{x}_S(k\,T) \end{bmatrix} + \begin{bmatrix} \underline{B}_D \\ 0 \end{bmatrix} \underline{u}(k\,T). \qquad (2.23)$$

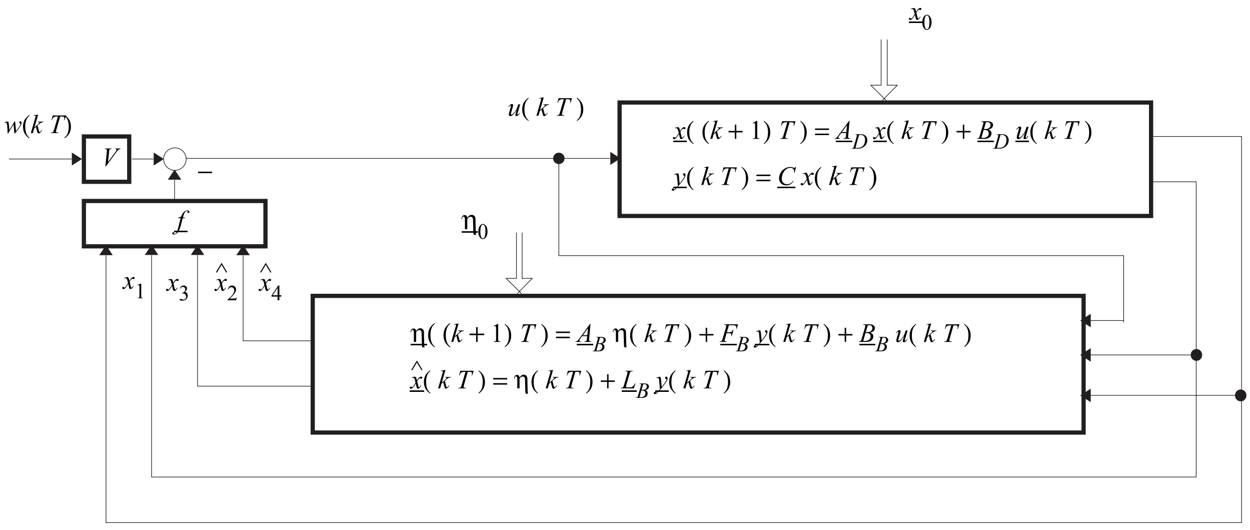

$\underline{x}_0$
$w(k\,T)$
V
$-$
$u(k\,T)$
$\underline{x}((k+1)\,T) = \underline{A}_D\,\underline{x}(k\,T) + \underline{B}_D\,\underline{u}(k\,T)$
$\underline{y}(k\,T) = \underline{C}\,x(k\,T)$
$\underline{f}$
η_0
x_1 x_3 $\hat{x}_2$ $\hat{x}_4$
$\underline{\eta}((k+1)\,T) = \underline{A}_B\,\eta(k\,T) + \underline{F}_B\,\underline{y}(k\,T) + \underline{B}_B\,u(k\,T)$
$\hat{\underline{x}}(k\,T) = \eta(k\,T) + \underline{L}_B\,\underline{y}(k\,T)$

Figure 2.3: Controlled BW500 system with a reduced order observer

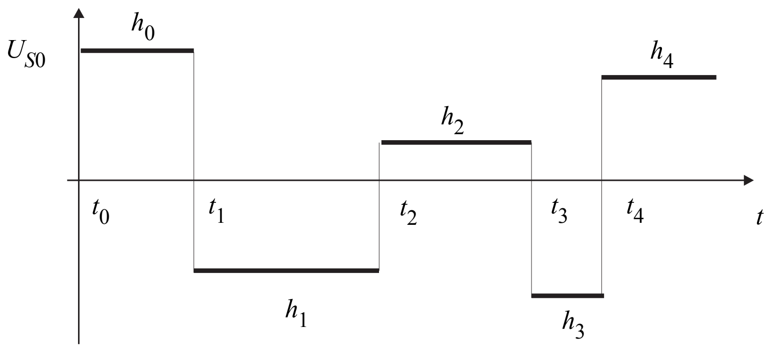


Figure 2.4: Characteristic behaviour of the disturbance signal

A reduced order observer is now designed for this system.

All the measured variables, all the state variables as well as the control force u_S are used to estimate the disturbance signal $\hat{x}_S = U_{S0}$ as shown in figure 2.6. The known vector $\underline{y}$ as an input of the disturbance observer contains the elements $\underline{y} = [\,x_1\ \hat{x}_2\ x_3\ \hat{x}_4]^T$. The pole of the disturbance observer is selected as

$$z_3 = e^{-T\,50\ 1/s} = 0{,}0821$$

With this the corresponding disturbance observer matrices are given as follows:

$$\underline{L}_{B\,S} = [0\ \ 0\ \ 0\ \ 5{,}4809]$$

$$\underline{A}_{B\,S} = [0{,}0821]$$

$$\underline{F}_{B\,S} = [-4{,}9618\ \ -0{,}1258\ \ 0{,}0764\ \ -4{,}5791]$$

$$\underline{B}_{B\,S} = [-0{,}9179]$$

$$\underline{V}_{B\,S} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ l_{B\,S\,1} & l_{B\,S\,2} & l_{B\,S\,3} & l_{B\,S\,4} \end{bmatrix}$$

$$\underline{C}_{B\,S} = [1] \quad .$$

As for the reduced order state observer the elements of the matrices above include units. So again the elements are not comparable directly.

A simple alternative is the constant disturbance compensation. Here a constant force is added to the control signal if the variance for the ball position exceeds the default threshold value (2 cm). The sign of this constant disturbance compensation depends on the variance, the value has to be determined experimentally.

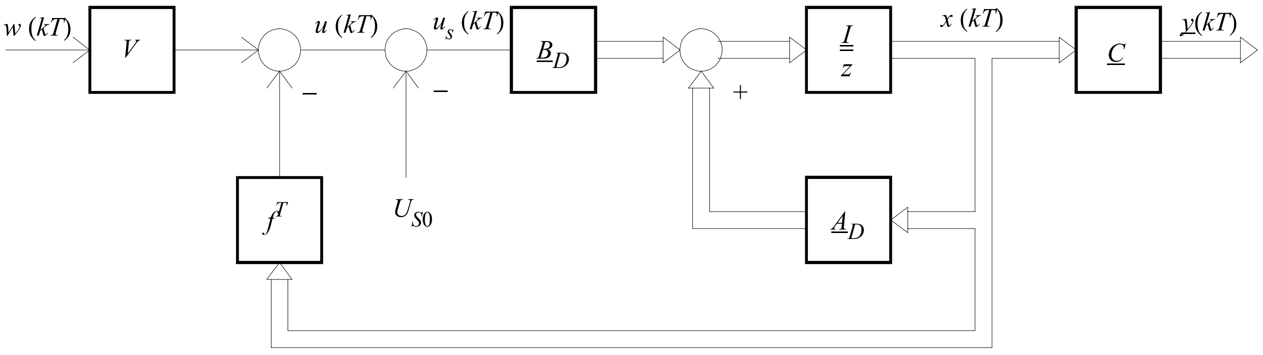


Figure 2.5: Closed control loop with disturbance signal

2.6 Filtering the Camera Signal for the Ball Position

On measuring the state variable x_1 irregular errors occur acting as considerable disturbances for the controller. To weaken this disturbance effect at least for a certain class of these errors a simple filter method will be used. The following will describe the strategy.

Filter method

During one sampling period the variation of the position Δx_1 is determined and compared with the maximum possible variation $\Delta x_{1\,\max}$. It is assumed that no error occurred when $\Delta x_1 \le \Delta x_{1\,\max}$. The current measurement value of x_1 is then used for the controller. However in the case of Δx_1 >~ DELTA x sub {1^max} the position value of the previous sampling period is used for the controller. The limit value $\Delta x_{1\,\max}$ is determined as follows:

$$\Delta x_{1\,\max} = x_{2\,\max}\, T = 1\, m/s\; 0{,}05\, s = 0{,}05\, m \quad . \quad (2.26)$$

However this method includes the disadvantage in the extreme case that the condition for an error is valid for all the following sampling periods. Under these circumstances the control may fail completely because the controller will always use only the last valid position value. To limit this failure to a short period the position values of the last ten sampling periods are compared in addition. When the comparison results in ten identical position values the filtering is interrupted for one sampling period. So the controller uses the current measurement value for the position x_1. This simple filter method guarantees that a measurement error is limited to $\Delta x_{1\,\max}$. The effect of this filter method will be shown during the control of the BW500 laboratory setup.

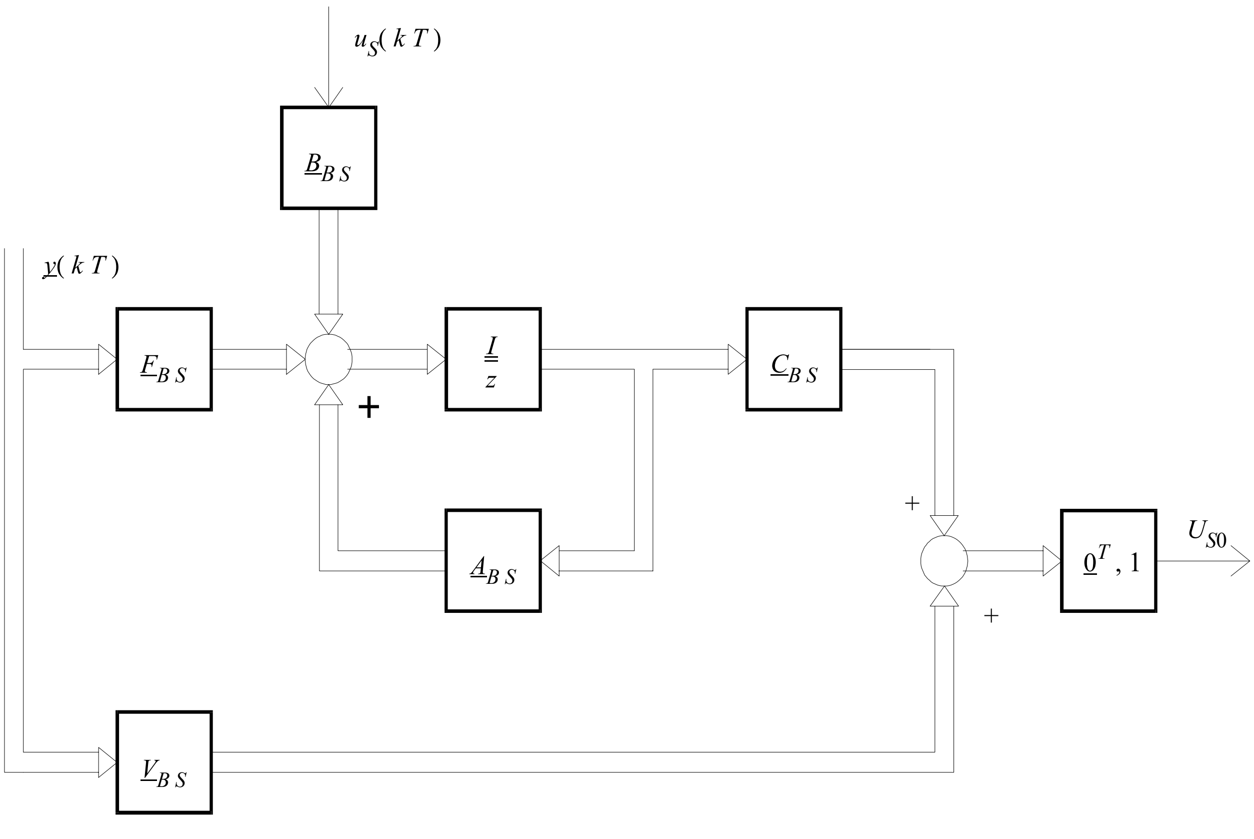


Figure 2.6: Observer for the disturbance U_{S0}

Theoretical Background of the Fuzzy Controller

Date: 08.11.1995

1 Backgrounds of the Fuzzy Controller

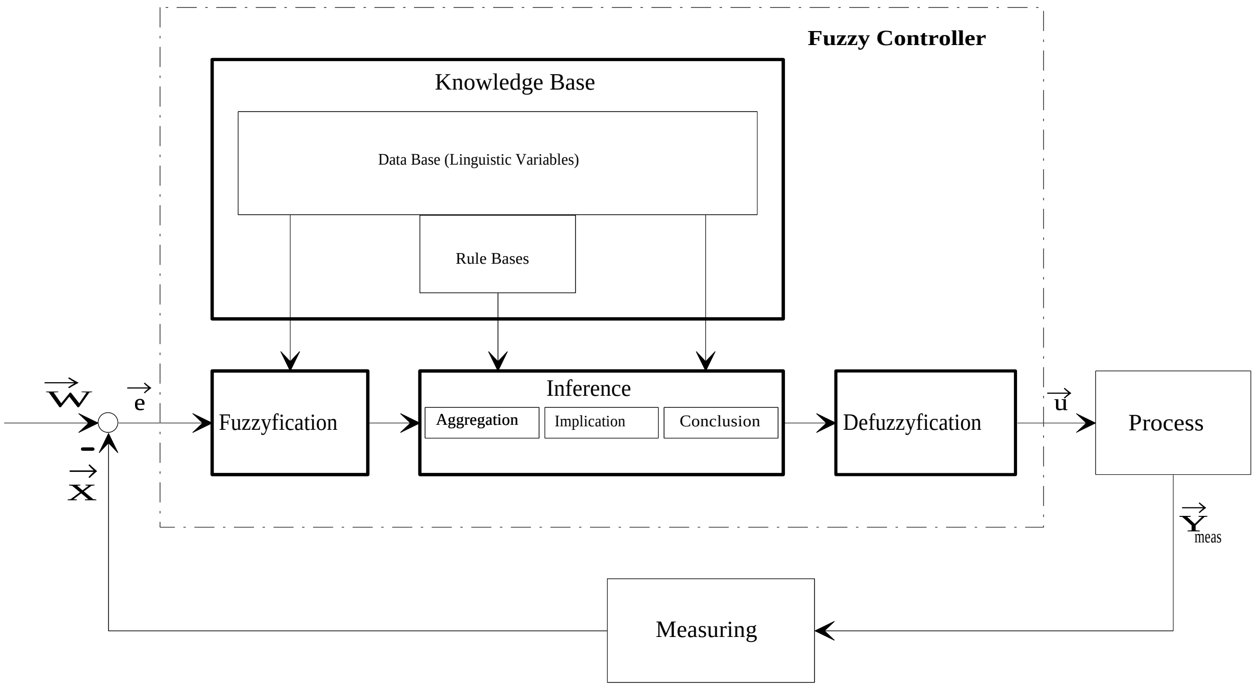


Figure 1.1: Components of the fuzzy Controller

1.1 The Fuzzy Set

The role of the numbers for the arithmetic is played by the fuzzy sets, also known as fuzzy aggregates, for the fuzzy theory. They are the mathematical base objects for which corresponding operators are defined.

To control a process, the required data are provided by a measuring system. Those data include the unit of measuring, the measured variable and possibly some other values which are not of interest in this case. The unit of measuring is the physical unit i.e. meter, whereas the measured value is a non-dimensional measured result. To order the inordinate group of all possible data they could be mapped to the group of real numbers, using for example the corresponding number of the measured value. Those numbers are representable graphically by a straight line of numbers.

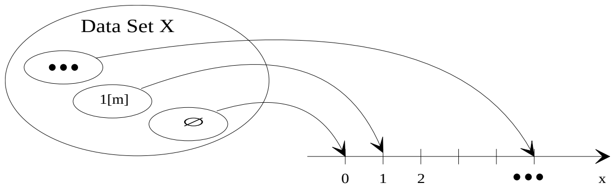


Figure 1.2: Mapping of the data set X to the real numbers

Correspondingly a set of expressions can be mapped. A special case is the set with the element *true*, which can be mapped to the numbers 0 and 1. In addition both sets are combinable in the case an expression *true* or *false* is assigned to each measured value. This representation corresponds to the well-known binary logic.

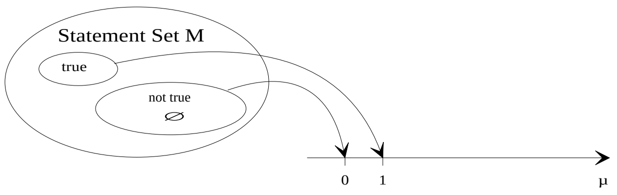


Figure 1.3: Mapping of the statement set M to the real numbers

Using this method a bar with the length less than 1 meter is clearly representable by the set of x contained in the data set X which results with the expression $\mu = 1$. Those are all values which apply to $0 \leq x \leq 1$.

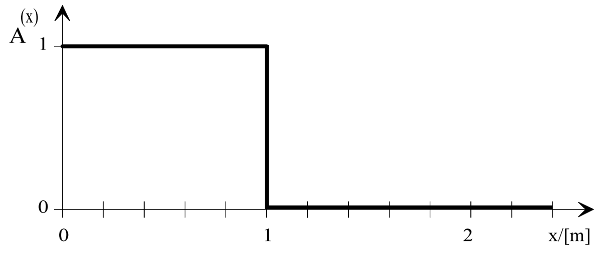


Figure 1.4: Example for a distinct set

Such a set, described by binary expressions, is called a distinct set.

The set of all bars, which are longer than 1 meter, can be described using the complement of the above mentioned set. This again is a distinct set.

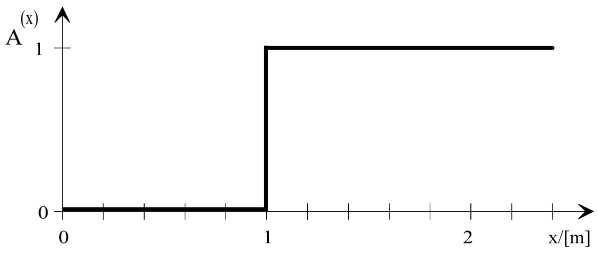


Figure 1.5: Example for the complement of the distinct set (see above)

Human meaning will consider the difference between 0.99 m and 1.01 m as being not significant so that the statement the bar with a length of 0.99 m does not belong to the set whereas the bar with 1.01 m belongs to it seems to be unnatural. A more plausible description is reachable by expanding the set of expressions. The representation of long bars could then look like the following:

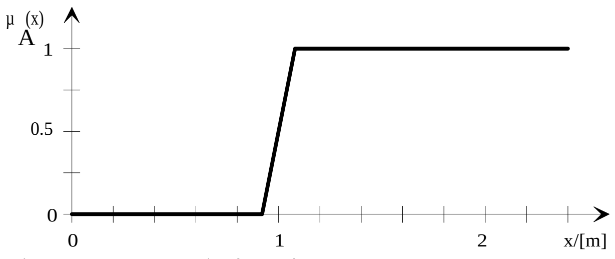


Figure 1.6: Example for a fuzzy set

The difference between the figures 1.5 and 1.6 can be seen from the transition region from $\mu_{A\,(x)} = 0$ to $\mu_{A\,(x)} = 1$. While the membership changes step-wise in the figure 1.5, a sliding transition of the membership values takes place in the figure 1.6. Doing this according to human mind a more natural description of the term length is possible. Such a set is called an indistinct set or a fuzzy set.

In the most general case the set of expressions is completely mappable to the interval of the real numbers. This is representable by the so-called membership function $\mu_{A\,(x)}$. It describes the degree of membership of all elements $a \in A \subseteq X$.

With that it is possible to describe indistinct terms like *nearly true, fairly true, quite false*, etc. mathematically.

So the fuzzy set is described by an ordered set of pairs of the form:

$$A = \{(x, \mu_{A\,(x)}) \mid x \ll X\}$$

Features of the Membership Functions:

- The membership function $\mu_{(x)}$ describes an onto mapping, that means for each picture point $\mu \in U$ there exists at least one original picture point $a \in A$. So only $A \rightarrow U$ describes a distinct mapping, whereas the reverse mapping $U \rightarrow A$ is indistinct in general.

- The range of values $\mu_{(x)}$ is the set of the positive, real numbers R^+. Usually $\mu_{(x)}$ is normalized to 1 when no other statements are made. So the fuzzy logic is a generalization of the classical logic.

1.2 The Linguistic Variable

It was shown in the previous section that with human mind fuzzy terms are describable mathematically. With this one must not forget that those terms always apply to the set $A \subseteq X$ with the membership function $\mu_{A\,(x)}$.

Example:

Let us assume the length $x \in X$ is given, so the term *very short* (Set *SK*) is definable using the fuzzy set $\mu_{SK(x)}$. Similar to this further terms like *short, normal, long* are definable using further fuzzy sets, but all the fuzzy sets have to belong to the same base set *X*. The fuzzy sets built in such a manner are combined to the so-called linguistic variable.

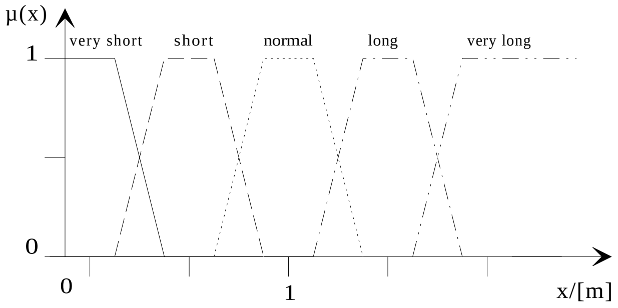


Figure 1.7: Example for a linguistic variable

So a linguistic variable can be taken as a group of fuzzy sets, which are defined with respect to the same base set *X* and the same range of values i.e. $\mu_{(x)} \in [0,1]$.

There are no limits for the overlapping regions so that an arbitrary number of fuzzy sets may overlap.

The totality of all input and output linguistic variables is called the data base which is used for control.

1.3 The Fuzzification

Since the fuzzy theory generally defines operators only for fuzzy sets, a fuzzy set is to be assigned to each distinct measured value (crisp value), i.e. the components of the control error vector $\vec{e}$. This process is called fuzzification. It can be seen easily that $\mu_{(x)}$ of a crisp value is identical to the normalized impulse function $\gamma_{o\,(x)}$ as well as the result is a square function *rect(x)* (see figure 1.8) in case the value is full of tolerances.

Unfortunately the following operation is called with the same name. This is the expression x_i *is* l_{ij} contained in the rules. In this case x_i is the fuzzificated measured value, i.e. x=1[m], and l_{ij} is the j-th fuzzy set of the i-th linguistic variable, i.e. $l_{0,0}$ = *very short*. The index *i* indicates that the fuzzy sets, which are to be compared, belong to the same base set and range of values. Since the connections and units in physical systems clearly define, which measured value is assigned to which linguistic variable the index *i* is neglectable.

It is also to be regarded that the keyword *is* describes an operator. It determines the maximum degree of correspondence a_{ij} of the both fuzzy sets using an *and* combination (minimum result) of both fuzzy sets followed by the determination of the maximum membership value. This is a number out of the range from Zero to the maximum value of $\mu_{(x)}$.

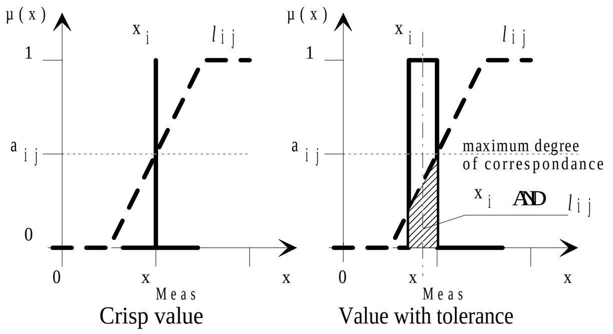


Figure 1.8: Fuzzification

1.4 The Rule and the Rule Base

A rule also called fuzzy implication has the general form:

if *Premise* then *Conclusion*

An arbitrary number of expressions x_i *is* l_{ij} is combined by the operators *and* or *or* in the premise.

Example for a premise:

x is *near* and *v* is *great*

The terms *near, great* are defined as fuzzy sets in the linguistic variables position and speed. The fuzzy sets *x, v* are the fuzzificated currently measured values.

The conclusion is an expression of the form x_o *is* l_{oj}. In this case l_{oj} is the j-th fuzzy set of the linguistic variable describing the output value and changed by the implication. The equal sign is here an assignment of the fuzzy set l_{oj} to the output variable x_o.

A rule base is a combination of an arbitrary number of rules.

The rule base describes the experience and the knowledge about the process and therefore must not be complete. For instance it could be the case that due to the ignorance of the developer not all physical actions are known. The lack of dominant rules is the result of the incompleteness of the rule base. This leads to a bad or useless control behaviour. This is repairable by the addition of further rules to the rule base.

The totality of all linguistic variables and rule bases, which characterize the fuzzy controller is called knowledge base.

1.5 The Inference Operation

1.5.1 The Aggregation

The aggregation (combination) of the degrees of membership a_{ij}, determined by the fuzzification, of a rule is performed by operators given in the premise.

Example of such a premise:

$a_{1,2}$ and $a_{2,1}$ or $a_{1,1}$

The degrees of membership $a_{i,j}$ are numbers with $0 \le a_{i,j} \le \mu_{\max}$.

Operators:

- or
  a_{ab} or $a_{cd} \Leftrightarrow$ MAX (a_{ab}, a_{cd})
- and
  a_{ab} and $a_{cd} \Leftrightarrow$ MIN (a_{ab}, a_{cd})

In general the order of the operators has to be regarded, because not all operators are commutative or associative at all. But the *and* and *or* operators meet this condition so that the premise can be interpreted recursively. The result of the a aggregation a_g is again a number. It is a measure of the fulfilment of the premise.

1.5.2 The Implication

The implication is used to infer the degree of fulfilment a'_{gr} at the output from the degree of fulfilment a_{gr} of the premise.

if a_{gr} then $a'_{gr} \quad \Leftrightarrow \quad a_{gr} \rightarrow a'_{gr} \le \mu_{\max}$

Usually the identity $a_{gr} \equiv a'_{gr}$ is taken as a mapping instruction.

1.5.3 The Conclusion

After the implication has determined the degree of fulfilment a'_{gr} of the output the conclusion determines the resulting fuzzy set.

Example:

x_o *is stark*

Doing this the fuzzy set of the output linguistic variable named in the rule is suitably combined with a'_{gr}. Usual combinations are the determination of the minimum or the product.

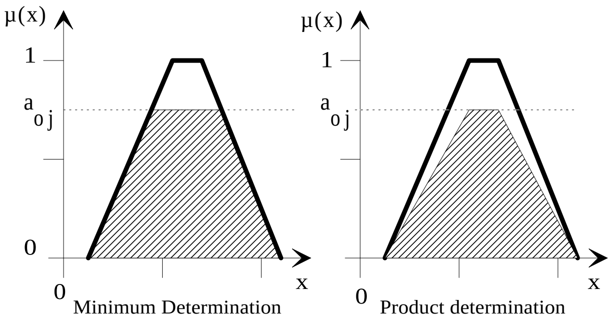


Figure 1.9: Min- and product method

1.5.4 The Inference

All rules of a rule base are interpreted in this way. The degree of fulfilment a'_{gr} is determined for each rule. The rules have to be combined because each rule delivers a resulting fuzzy set. This is performed by means of combining all the resulting fuzzy sets which apply to one and the same output linguistic variable.

In the special case that the conclusion is carried out using the product or minimum method and that the combination uses the maximum determination, the combination can be carried out previously to the conclusion using the following instruction:

$$a_{oj} = MAX\,(a_{gr_j\,1}, \ldots, a_{gr_j\,n}) \qquad 0 \le a_{gr} \le 1$$

The index j references to the fuzzy set of the output linguistic variable. This is a significant simplification. The fuzzy sets of the output linguistic variable converted in such a way are combined to one result. The combination is usually carried out by the maximum operator. Doing this the inference leads to the resulting fuzzy set, which describes the degrees of membership to the resulting set of all output values.

1.6 The Defuzzification

To come from the resulting fuzzy set determined by the inference to a crisp value, i.e. the control signal u a so-called defuzzification has to be carried out. That means a characteristic value of the resulting fuzzy set has to be found. To do this several methods are available, which can be distinguished with respect to the computing effort and the general application field.

- Maximum Choice (MAX)

$$y = min\ \{y \mid \mu_{(y)} = \mu_{\max}\}$$

- Mean Value of the Maximum (MOM, Mean of Maximum)

$$y = \sum_{i=1}^{l} \frac{w_i}{l} \cap \ \mu(w_i) = \mu_{\max}$$

- Center of Area (COA, Center of Area)
  This is the most general operation. Therefore this method of defuzzification is implemented in our controller. Here the abscissa co-ordinate of the Center of area is determined. The special advantage of this method is the result with continuous values in contrast to the first mentioned methods.

$$y = \frac{\int \mu_{(y)}\, y\, dy}{\int \mu_{(y)}\, dy}$$

$$\cup$$

$$y = \frac{\sum (\,\mu_{(y_i + \frac{\Delta y}{2})} + \mu_{(y_i - \frac{\Delta y}{2})}\,)\, y_i\, \Delta y}{\sum (\,\mu_{(y_i + \frac{\Delta y}{2})} + \mu_{(y_i - \frac{\Delta y}{2})}\,)\, \Delta y}$$

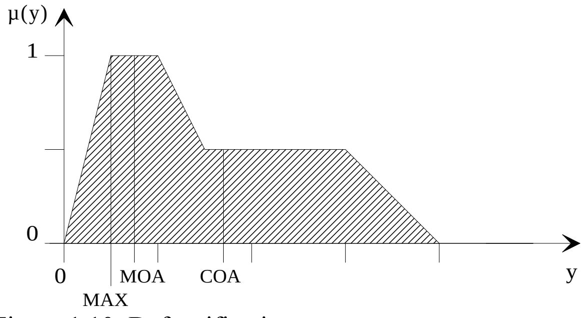


Figure 1.10: Defuzzification

1.7 Remarks

The fuzzy controller belongs to the non-linear steady controllers. So its input/output behaviour is definitely given by its control characteristic area. Unfortunately there is no general method available to find suitable fuzzy sets and rules for a given problem.

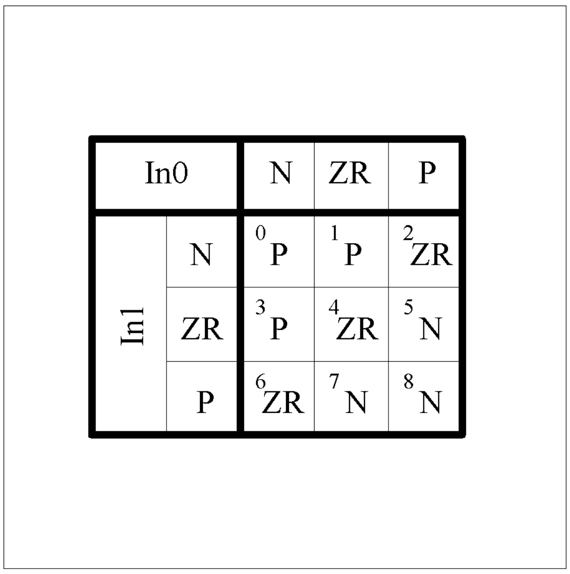
The fuzzy sets of the linguistic variables can be found with respect to

- the characteristic values of the actuator and measurement system,

- the knowledge of the control engineers,
- the observation of the process.

The rules can be found with respect to

- the experience of the operators,
- the knowledge of the control engineers,
- the observation of the process.



In0		N	ZR	P
In1	N	[0] P	[1] P	[2] ZR
	ZR	[3] P	[4] ZR	[5] N
	P	[6] ZR	[7] N	[8] N

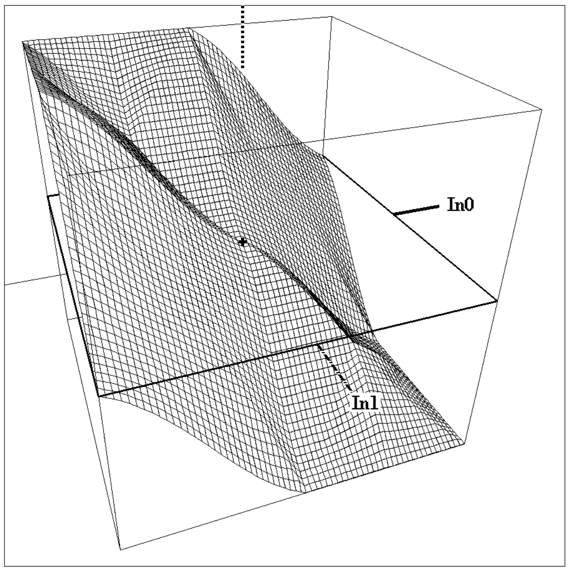
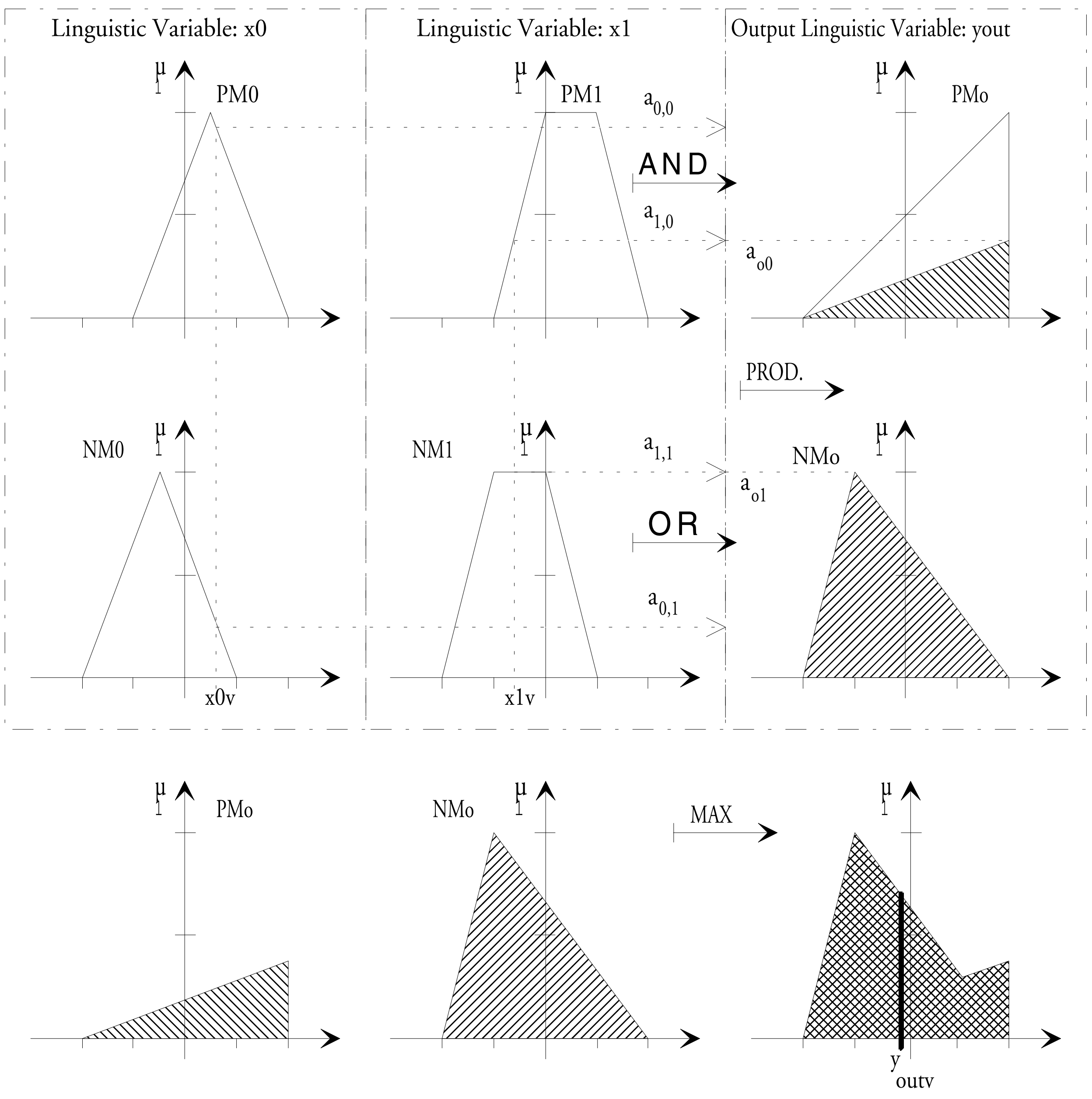


Figure 1.11: Example for a characteristic control area



```
Rule 0: if x0 is PM0  and  x1 is PM1  then  yout is PMo end
Rule 1: if x0 is NM0  or   x1 is NM1  then  yout is NMo end
Input variables: x0, x1
Sets of Input variables: PM0, NM0, PM1, NM1
Values of input variables: x0v, x1v
Output variable: yout
Sets of Output variables: PMo
Value of output variable: youtv
µ is degree of fulfilment
```

Figure 1.12: Example for the execution of a fuzzy algorithm using the Max-Product-Method

2 Realization of the Fuzzy Controller

Different to the state control a cascade structure is used for the fuzzy controller. Figure 2.1 displays this structure. The signal names correspond to the names of the fuzzy viariables.

A more simple definition of the fuzzy variables and rules as well as a shorter execution time are the advantages of the cascade structure. Each fuzzy block contains only two inputs and one output. Therefore the number of its rules is small. According to figure 2.1 the inner fuzzy controller realizes the angle control of the beam. The setpoint for this controller is provided by the outer position controller. The difference of the beam angle setpoint and its measured value as well as the beam angular velocity are the required input signals of the angle controller. Its output signal controls directly the force acting on the beam. The difference of the position setpoint and its measured value as well as the ball speed are the required input signals of the ball position controller. The angle setpoint of the beam is the output signal of this controller. The fuzzy control described up to now contains no elements to compensate the effects of the friction. Figure 2.2 displays an expanded structure of the fuzzy controller containing additional disturbance compensation. The disturbance signals are the beam friction and the ball friction which are estimated by two additional fuzzy blocks operating like fuzzy observers. To estimate the beam friction the corresponding fuzzy observer requires the current control signal for the force and the angular velocity of the beam as input signals. The fuzzy observer provides an offset value which is added to the control signal to compensate the beam friction. The fuzzy observer for the ball friction operates in a similar way.

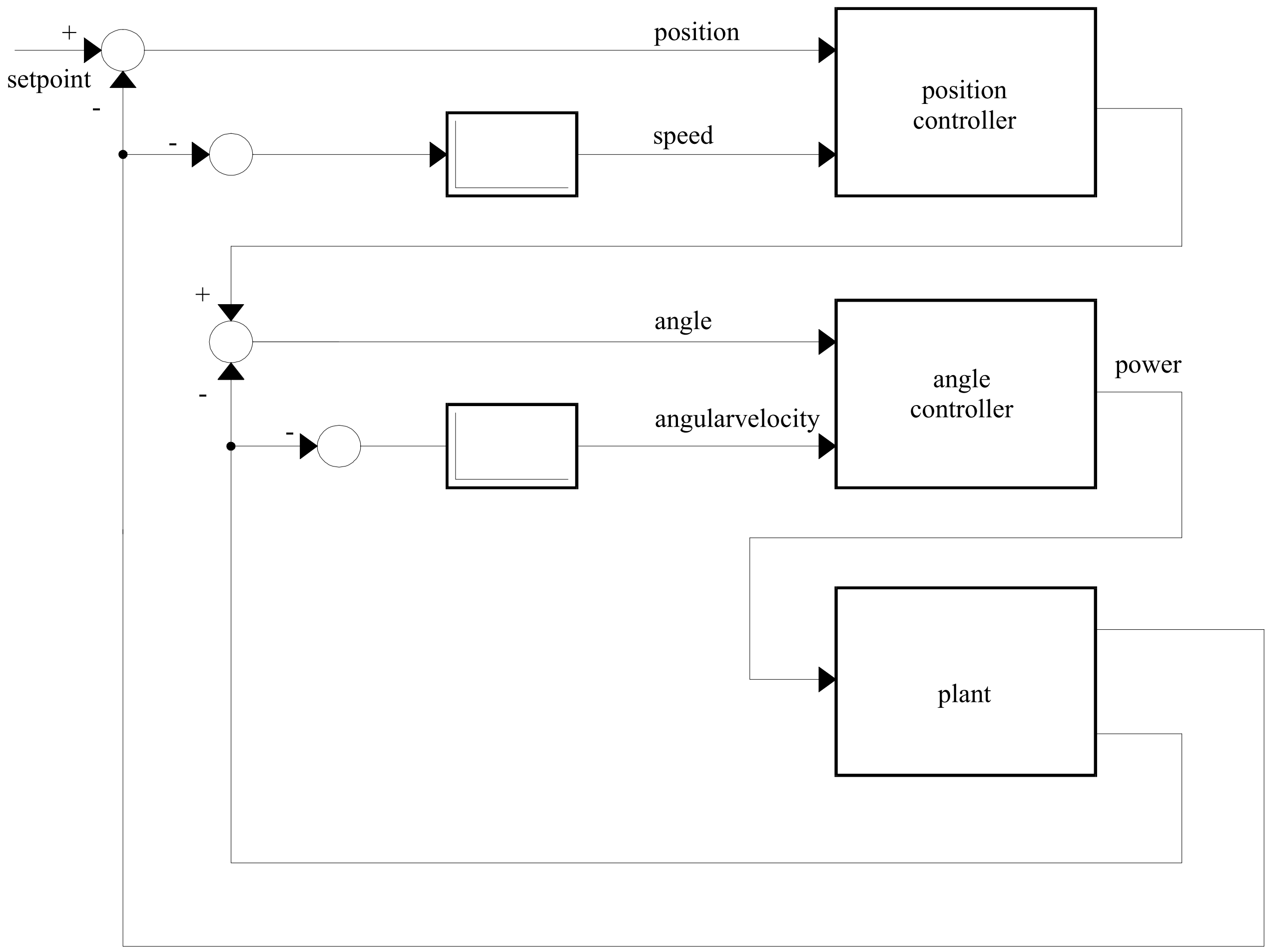


Figure 2.1: Simplified structure of a Fuzzy controller for the model "Ball and Beam"

The input signals angle of the beam and speed of the ball are used to determine an offset value. This offset value is added to the angle setpoint of the beam.

Difference quotients are used to determine the missing signals ball speed and angular velocity of the beam out of the measured signals ball position and beam angle.

The fuzzy controllers as well as the fuzzy observers used for this laboratory setup are realized by applying the library FUZZY.LIB of the company amira GmbH. It is a pure software realization of the methods described above.

A single ASCII file for each controller or observer is used to define the variables, their sets and the rules. The detailed format of this file is described in the chapter "Programming Instructions". Please regard the order of the input variables in addition to the syntax rules described in the mentioned chapter. This order is to be obligatory for each file and must not be changed in any case.

The ranges of the used variables are limited either by hardware or by software. The following table contains the variable names, their ranges and units.

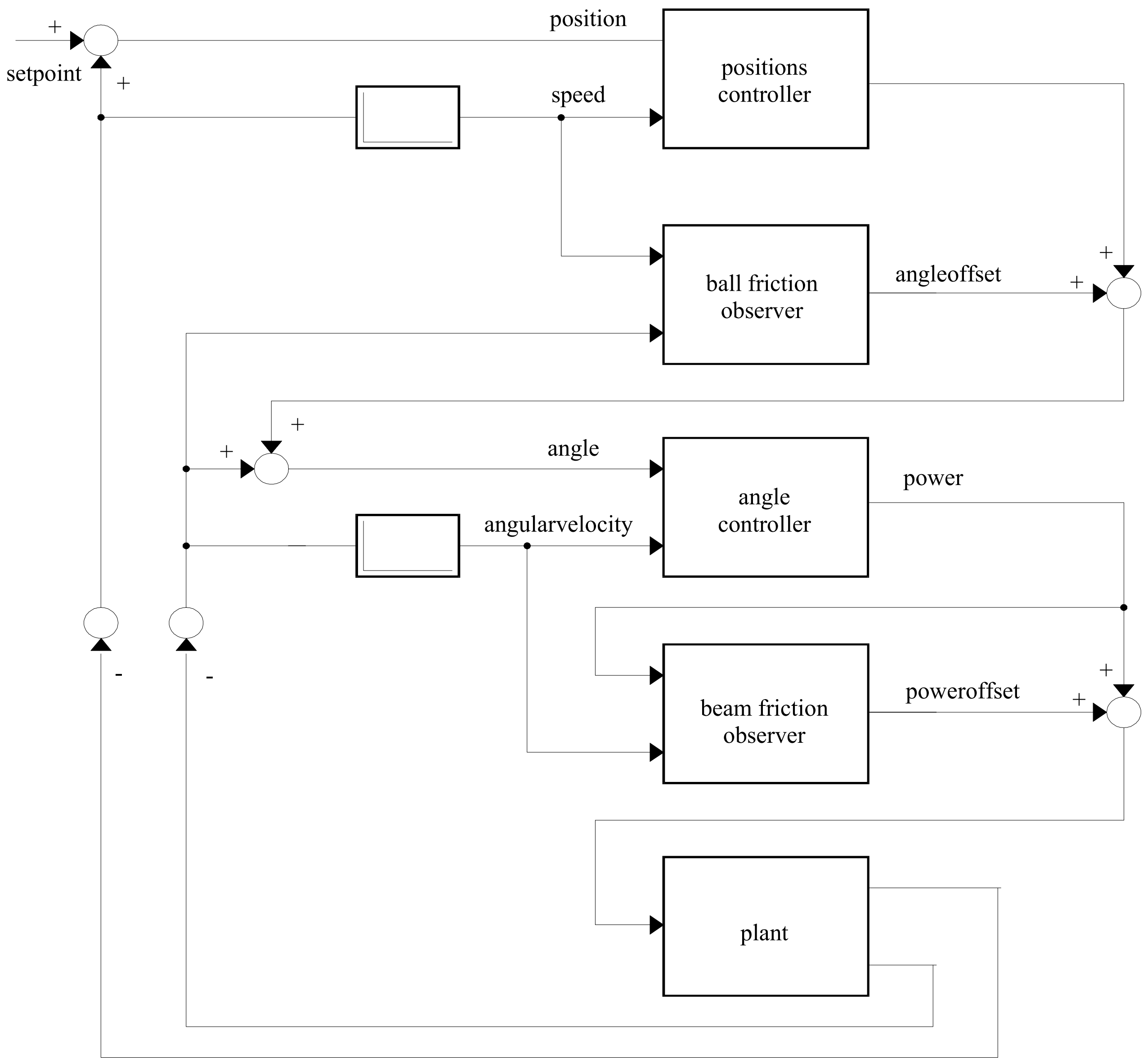


Figure 2.2: Structure of the fuzzy controller for model "Ball and Beam" with disturbance compensation

Name:	Unit	Minimum Value	Maximum Value
position	m	-0.7	0.7
speed	m/s	-0.5	0.5
angle	rad	-0.2	0.2
angleoffset	rad	-0.2	0.2
angular velocity	rad/s	-0.6	0.6
power	N	-6.0	6.0
poweroffset	N	-6.0	6.0

The following four fuzzy description files realize in example the fuzzy control of the system "Ball and Beam" according to figure 2.2:

1. The inner angle control
(file name: "wcontrol.fuz"):

```
/* variable definitions */
var input angle
set negativ    -0.2 1    0 0                    endset
set zero       -0.1 0    0 1       0.1 0        endset
set positiv    0 0       0.2 1                  endset
endvar

var input angularvelocity
set negativ    -0.6 1    -0.15 0                endset
set zero       -0.3 0    0 1       0.3 0        endset
set positiv    0.15 0    0.6 1                  endset
endvar

var output power
set bignegativ     -5 0      -4.5 1    -4 0     endset
set negativ        -3 0      -2.5 1    2 0      endset
set zero           -0.1 0    0 1       0.1 0    endset
set positiv        2 0       2.5 1     3 0      endset
set bigpositiv     4 0       4.5 1     5 0      endset
endvar

/* rule definitions */
if angle is zero and angularvelocity is zero then
power is zero end
if angle is zero and angularvelocity is negativ then
power is positiv end
if angle is zero and angularvelocity is positiv then
power is negativ end
if angle is negativ and angularvelocity is zero then
power is bigpositiv end
if angle is negativ and angularvelocity is negativ
then power is bigpositiv end
if angle is negativ and angularvelocity is positiv then
power is positiv end
if angle is positiv and angularvelocity is zero then
power is bignegativ end
if angle is positiv and angularvelocity is negativ then
power is negativ end
if angle is positiv and angularvelocity is positiv then
power is bignegativ end
```

2. The position control (file name: "xcontrol.fuz"):

```
/* variable definitions */
var output angle
set bignegativ     -0.2 0    -0.19 1   -0.18 0   endset
set negativ        -0.1 0    -0.09 1   -0.08 0   endset
set zero           -0.01 0   0 1       0.01 0    endset
set positiv        0.08 0    0.09 1    0.1 0     endset
set bigpositiv     0.18 0    0.19 1    0.2 0     endset
endvar

var input position
set negativ
     -0.2 1     0 0                                    endset
set zero
     -0.1 0     -0.03 0.5 0 1   0.03 0.5  0.1 0        endset
set positiv
     0 0        0.2 1                                  endset
endvar

var input speed
set negativ
     -0.3 1     0.0 0                                  endset
set zero
     -0.2 0     -0.05 0.5       0 1   0.05 0.5  0.2 0 endset
set positiv
     0.0 0      0.3 1                                  endset
endvar

/* rule definitions */
if position is zero and speed is zero then angle is
zero end
if position is zero and speed is negativ then angle is
positiv end
```

```
if position is zero and speed is positiv then angle is
negativ end
if position is negativ and speed is zero then angle is
positiv end
if position is negativ and speed is negativ then angle
is bigpositiv end
if position is negativ and speed is positiv then angle
is zero end
if position is positiv and speed is zero then angle is
negativ end
if position is positiv and speed is negativ then angle
is zero end
if position is positiv and speed is positiv then angle is
bignegativ end
```

3. The estimation of the beam friction
(file name: "werror.fuz"):

```
/* variable definitions */

var input angularvelocity
set zero       -0.03 0   0 1  0.03 0        endset
set notzero    -0.03 1   0 0  0.03 1        endset
endvar

var input power
set negativ    -3 1      -1 0               endset
set zero       -1 0      0 1      1 0       endset
set positiv    1 0       3 1                endset
endvar

var output poweroffset
set negativ    -3 0      -2.5 1   -2 0      endset
set zero       -0.1 0    0 1      0.1 0     endset
set positiv    2 0       2.5 1    3 0       endset
endvar

/* rule definitions */
if power is zero then poweroffset is zero end
if angularvelocity is notzero then poweroffset is zero
end
if power is negativ and angularvelocity is zero then
poweroffset is negativ end
if power is positiv and angularvelocity is zero then
poweroffset is positiv end
```

4. The estimation of the ball friction
(file name: "xerror.fuz")

```
/* variable definitions */
var input angle
set negativ    -0.1 1    -0.02 0               endset
set zero       -0.02 0   0 1       0.02 0      endset
set positiv    0.02 0    0.1 1                 endset
endvar

var input speed
set zero       -0.1 0    0 1       0.1 0       endset
set notzero    -0.1 1    0 0       0.1 1       endset
endvar

var output angleoffset
set negativ    -0.04 0   -0.03 1   -0.02 0     endset
set zero       -0.01 0   0 1       0.01 0      endset
set positiv    0.02 0    0.03 1    0.04 0      endset
endvar

/* rule definitions */
if angle is zero then angleoffset is zero end
if speed is notzero then angleoffset is zero end
if angle is negativ and speed is zero then angleoffset
is negativ end
if angle is positiv and speed is zero then angleoffset
is positiv end
```

The described files are contained in the program disk and will be loaded as standard files automatically after starting the program.

A runtime test is executed automatically after loading a fuzzy description file. This causes a short delay. In case the medium execution time of the corresponding fuzzy objects exceeds the sampling period of the digital controller, the rule base may not be used to control the system.

Program Operation

(WINDOWS Version)

Printed: 2. November 1999

1 Program Operation

The software package provides two versions to control the plant. The first version is a state controller as described in chapter "Realization of the state controller". Controller parameters are on-line adjustable. The second version to control the BW500 is a fuzzy controller as described in chapter "Realization of the Fuzzy Controller". Controller adjustments are storable to a hard disc and may be read at a later time. The setpoint for the ball position is adjustable as a constant value or a time function. Measurements of system variables are recordable with various trigger conditions. Recorded data are representable in a graphic on the screen.

1.1 Program Start

The correct execution of the program requires that besides **BW500W.EXE** the following files are available in the actual directory:

DEFAULT.CAL
BW500.HLP
BW500W16.INI
BWSERV16.DLL
TIMER16.DLL
PLOT16.DLL
BC450RTL.DLL
DAC98.DRV
DIC24.DRV
DEFAULT.FBW
XCONTRO.FUZ
WCONTROL.FUZ
XERROR.FUZ
WERROR.FUZ
DEFAULT.STA

The executable program requires at least all of the mentioned dynamic link libraries (*.DLL) as well as the IO-adapter card drivers (*.DRV), which may be contained in another directory but with a public path (like Windows/System).

Attention:
The files BWSERV16.DLL as well as TIMER16.DLL must reside in the actual directory when another Windows program from amira is stored on your hard disk!
An additional driver DUMMY.DRV is required for the DEMO version of the program.

The file DEFAULT.CAL is used to store the calibration data of the system. A detailed description of this file is given in chapter 1.10.6. The help file BW500.HLP allows for operating the program without having this manual at hand. The function key F1 or a specific ’Help’ button presented in a dialog is to be used to activate the corresponding help section. The initialization file BW500W16.INI is completely controlled by the executable program itself and should not be changed by the user. It serves for handling the IO-adapter card driver.

DEFAULT.FBW contains the file names list of the four fuzzy description files belonging to the fuzzy controller of the system. These fuzzy description files are loaded automatically during the program start. The file format is described in 1.10.3.

DEFAULT.STA contains the parameters of the state controller. This file is loaded automatically during program start. The file format is described in 1.10.4.

After starting the program **BW500W.EXE** the standard data files (see above) are loaded and checked, which can take some seconds. Missing files will result in corresponding error messages. The check procedure includes trying to open the recently selected driver (DAC98.DRV or DIC24.DRV) for the PC adapter card. When this driver could not be opened the TIMER16.DLL will present the error message ’StartTimer - InitDriver failed’. After prompting this message, the main window of the program will appear offering the menu item ’IO Interface’ to select another driver or to change the address of the adapter card.

1.2 Sensor Calibration

When the program started without any error the ’BW500 Calibration Dialog’ (see figure 1.2) will appear automatically on the screen.

This dialog allows for calibrating the position sensor (CCD camera) as well as the beam angle sensor (incremental encoder). The complete procedure is carried-out in three distinct steps as it is obvious from the three static fields in the window. At the beginning each of the static fields is emphasized with a blue (aqua) coloured background and a ’Start’ button is enabled only for the upmost field. The calibration steps are as follows:

The first calibration step maps the current incremental encoder signal to a beam angle of zero and at the same time the current camera signal is taken as a zero position signal of the ball. That means that the user is responsible at first to turn the beam to a horizontal position and then place a ball in the middle of the beam before pressing the ’Start’ button. The background colour will then turn to green until either valid measurements have been taken or errors have been detected.

A successful result is indicated by a white background colour, a check mark replacing the ’Start’ button and an automatic jump to the next calibration step.

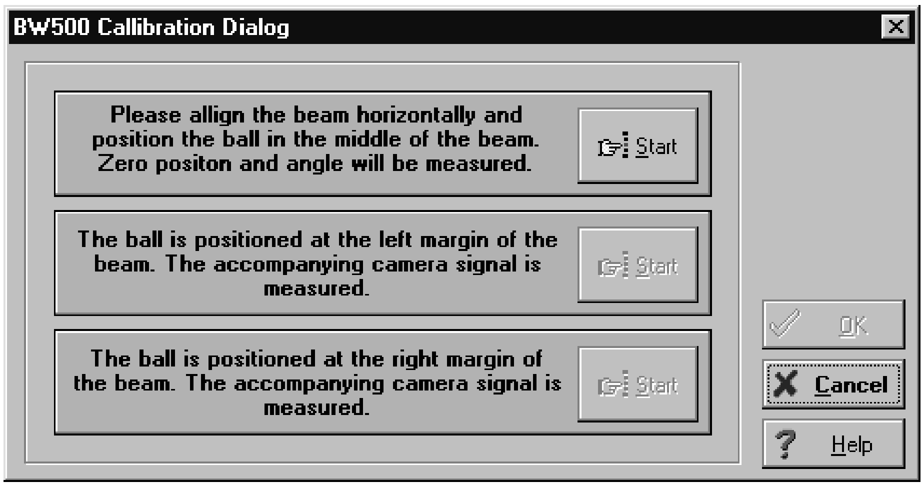
A false result is instead recognizable by a red background colour, a ’Retry’ button replacing the ’Start button and an additional error message (See below for possible error messages). The user is strictly recommended to ’repair’ the error before proceeding with the dialog.

The second calibration step, when activated, turns the background colour of the second static field to green and tries to position the ball at the left margin of the beam by starting the servo amplifier (enable output stage release) and then increasing the force (PI-controlled up to a limit of 3 [N]) acting on the beam until either the ball reaches the left margin (the current camera signal is taken as a sensor value for this position) or the measuring time exceeds a limit of 5 seconds.

The first case, a successful result, is indicated by a white background colour, a check mark replacing the ’Start’ button and an automatic jump to the next calibration step.

The second case, a false result, is instead recognizable by a red background colour, a ’Retry’ button replacing the ’Start button and an additional error message (See below for possible error messages). To ’repair’ the error the user has just to turn the horizontal beam manually until the ball reaches its left margin before pressing the ’Retry’ button.

The third calibration step is similar to the second calibration step but for a ball position at the right margin of the beam.


BW500 Callibration Dialog

Please allign the beam horizontally and position the ball in the middle of the beam. Zero positon and angle will be measured.

The ball is positioned at the left margin of the beam. The accompanying camera signal is measured.

The ball is positioned at the right margin of the beam. The accompanying camera signal is measured.

Figure 1.2: The ’BW500 Calibration Dialog’

The 'OK' button of the dialog is enabled only when all of the three calibration steps have been carried-out successfully. The 'Cancel' button may be used alternatively to terminate the dialog. But in this case none of the controllers can be started.

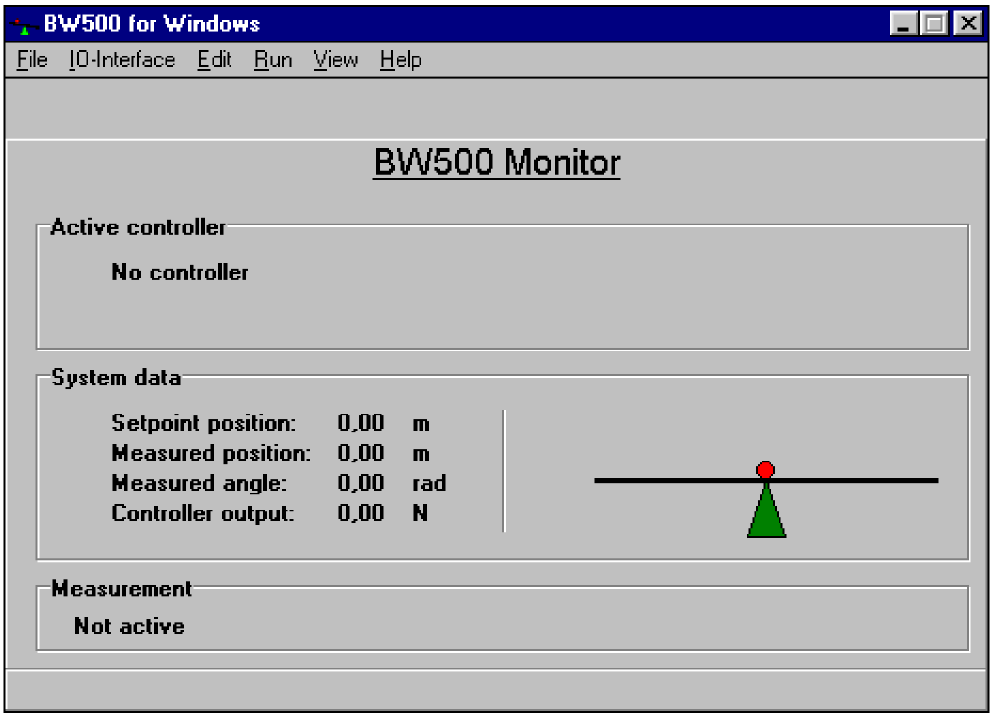
Possible error messages:

System not ready. Check Connections and Power.
Disengagement does not respond.
Camera signal out of range.
Unexpected limit switch.

Terminating the program itself will write the current calibration data, which are either from the calibration procedure or default data, to the file DEFAULT.CAL.

1.3 Main Window

Following a successful calibration the main window appears on the screen as shown in figure 1.3. The first screen row contains the main menu items. Its submenus are described in the following sections. The window **BW500 Monitor** is displayed in the middle of the screen. It displays the system state (controller type) and input as well as output signals of the controller.



BW500 Monitor

Active controller
No controller

System data
Setpoint position: 0,00 m
Measured position: 0,00 m
Measured angle: 0,00 rad
Controller output: 0,00 N

Measurement
Not active

Figure 1.3: The main BW500 window with monitor

1.4 Menu File

The pulldown menu **File** (see figure 1.2) provides functions for loading or saving of different files, to print plot windows as well as to terminate the program.

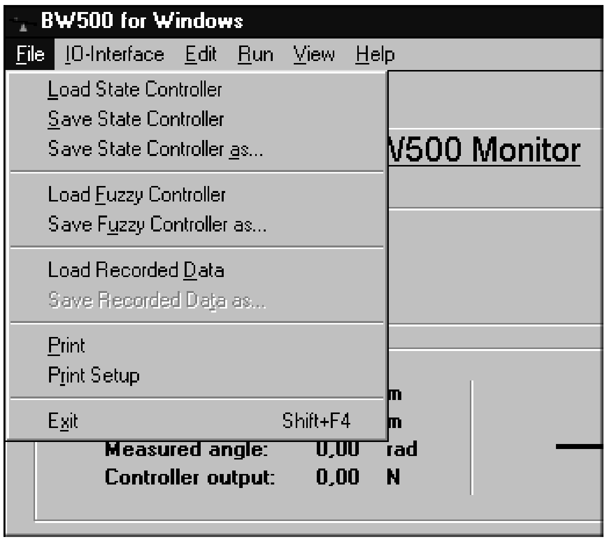


Figure 1.4: The sub menu 'File'

Load State Controller: Loads a parameter file (default extension *.STA) for the state controller. The file name is selected by the user from a file dialog window.

Save State Controller: Saves the adjusted parameters of the state controller in a disc file. Destination file is the parameter file of the state controller, which was recently opened. Please notice that the file "DEFAULT.STA" was opened and loaded automatically during the program start and may be overwritten by this command.

Save State Controller as ...: Operates similar to the item "Save State Controller", the name of the destination file is however selected by the user by means of a file dialog window.

Load Fuzzy Controller: Loads a parameter file (extension *.FBW) for the fuzzy controller. The file is selected by the user by means of a file dialog window. The parameter file contains the file names of four fuzzy description files. These fuzzy description files are loaded automatically and checked. A fuzzy rule base is generated if no errors were detected. Further information about the fuzzy description file can be found in the chapter 1.9.1 "Format of the Fuzzy Description File (*.FUZ)".

Save Fuzzy Controller as ...: Saves the names of the fuzzy description files of the fuzzy controller. The name of the destination file is selected by the user by means of a file dialog window.

Load Recorded Data: Opens a file dialog window for user selection of a data file containing recorded measurements (documentation file with extension *.PLD).

Save Recorded Data as ...: Saves the measurements previously recorded and the current system adjustments in a data file. The file name is selected by the user by means of a file dialog window (extension *.PLD).

Print: Opens the Print Window Dialog to select one or several plot windows for print output. This dialog presents a listbox containing the titles of all open plot windows. One or several windows may be selected for print output on the currently selected printer device (see **Print Setup ...**). A single window is printed on the upper half of a DIN A4 paper. The second window would be printed on the lower half of this paper. The following windows are printed on the next pages accordingly.

Print Setup ... Opens the Windows dialog to select a printer and to adjust its options.

Selecting the menu item **Exit** will terminate the program (equivalent to pressing Ctrl+F4).

1.5 Menu IO-Interface

The pulldown menu **IO-Interface** provides functions to manipulate the driver for the PC plug-in card (see figure 1.5a).

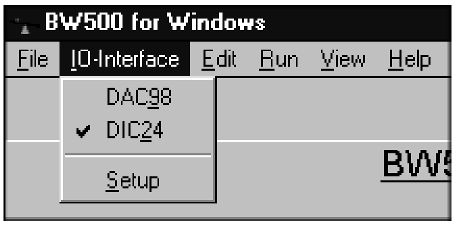
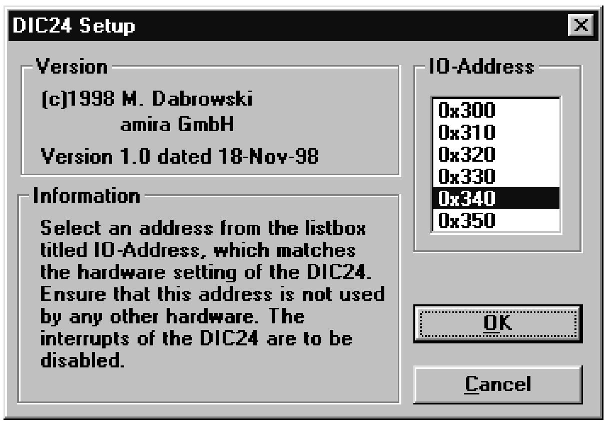


Figure 1.5a: The sub menu ’IO-Interface’

The first two items
DAC98
DIC24

represent the selectable drivers (DAC98.DRV, DIC24.DRV) for the IO-adapter cards which may be installed in the PC. Each driver is selectable only when it is contained in the same directory as the program BW500W.EXE (or in a directory with a public path like Windows/System). The recently selected driver is emphasized with a check mark. On program start the selected driver is read from the file BW500W16.INI which is controlled by the program automatically. When this file is missing the default driver is always the DAC98.DRV.


DIC24 Setup

Version
(c)1998 M. Dabrowski
amira GmbH
Version 1.0 dated 18-Nov-98

Information
Select an address from the listbox titled IO-Address, which matches the hardware setting of the DIC24. Ensure that this address is not used by any other hardware. The interrupts of the DIC24 are to be disabled.

IO-Address
0x300
0x310
0x320
0x330
[✓ 0x340]
0x350

Figure 1.5b: The card address setup dialog

The function **Setup** opens a dialog (see figure 1.5b) to adjust the drivers hardware address of the installed IO-adapter card. This address has to match the hardware settings !

This menu item is selectable only when no controller is active.

1.6 Menu Edit

The pulldown menu **Edit** (see figure 1.6a) contains items to edit parameters of the state controller as well as files for the fuzzy controller.

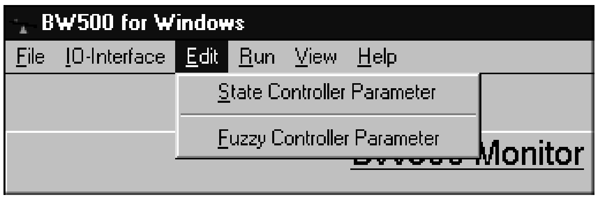
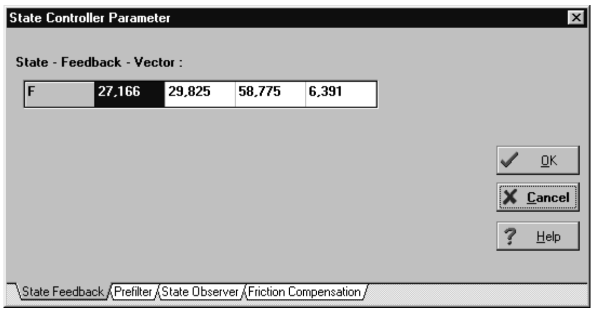

BW500 for Windows
File IO-Interface Edit Run View Help
State Controller Parameter
Fuzzy Controller Parameter
Monitor

Figure 1.6a: The sub menu 'Edit'

The menu item **State Controller Parameter** displays a notebook with four pages to edit all parameters of the state controller.

Selecting the first tab of this notebook allows for adjusting the elements of the **State Feedback**-vector (see figure 1.6b).

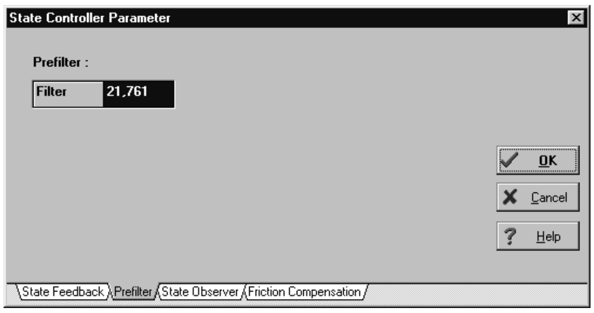

State Controller Parameter
OK
Cancel
Help
State Feedback Prefilter State Observer Friction Compensation

State - Feedback - Vector :

F	27,166	29,825	58,775	6,391

Figure 1.6b: The 'State Feedback' dialog

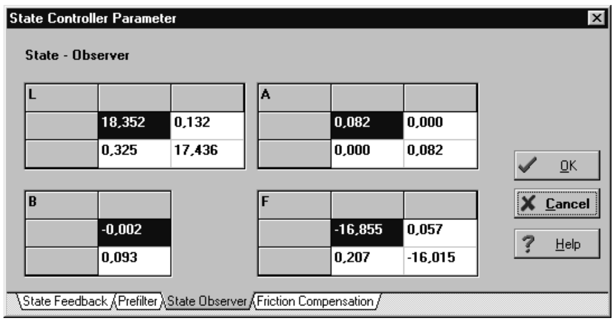
The second tab provides the adjustment of the **Prefilter** constant to obtain a steady state error of zero (see figure 1.6c).


State Controller Parameter
OK
Cancel
Help
State Feedback Prefilter State Observer Friction Compensation

Prefilter :

Filter	21,761

Figure 1.6c: The 'Prefilter' dialog


State Controller Parameter
OK
Cancel
Help
State Feedback Prefilter State Observer Friction Compensation

State - Observer

L		
	18,352	0,132
	0,325	17,436

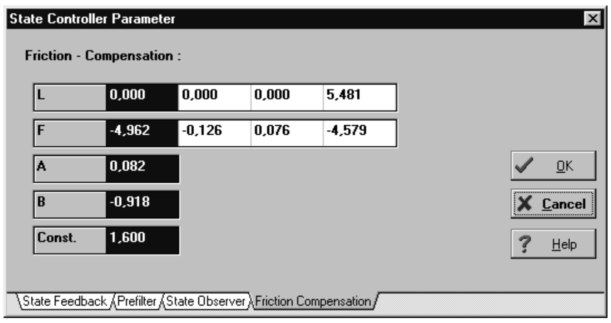
A		
	0,082	0,000
	0,000	0,082

B	
	-0,002
	0,093

F		
	-16,855	0,057
	0,207	-16,015

Figure 1.6d: The 'State Observer' dialog

Selecting the third tab labelled **State Observer** allows for manipulating the L, A and F matrix or the B vector of the observer (see figure 1.6d).


State Controller Parameter
OK
Cancel
Help
State Feedback Prefilter State Observer Friction Compensation

Friction - Compensation :

L	0,000	0,000	0,000	5,481
F	-4,962	-0,126	0,076	-4,579
A	0,082			
B	-0,918			
Const.	1,600			

Figure 1.6e: The 'Friction Compensation' dialog

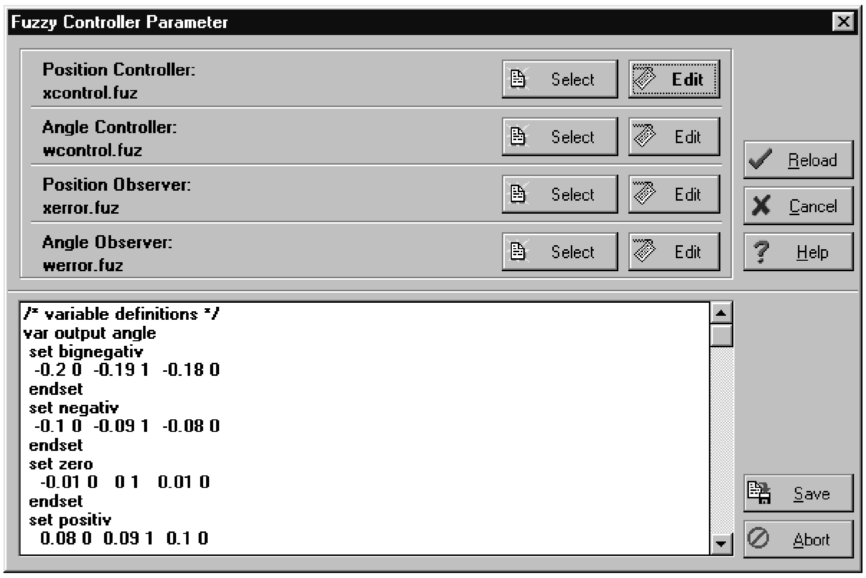
The fourth tab provides the adjustment of the parameters used by the **Friction Compensation** (see figure 1.6e).

The notebook dialog is terminated either by pressing the 'OK' button or the 'Cancel' button which are contained in each page. Any parameter changes become valid only if the 'OK' button was used to terminate the dialog.

The menu item **Fuzzy Controller Parameter** displays a dialog containing buttons labelled 'Select' and 'Edit' for each controller/observer providing methods either to select a new fuzzy description file or to edit the currently selected file which opens a new edit field below the dialog displaying the content of the fuzzy description file.

The following figure 1.6f displays this dialog with an opened edit field for the fuzzy description file named XCONTROL.FUZ of the fuzzy position controller.

The edit field itself is closed either by the button 'Save' which stores the content of the edit field to the file or by



```
/* variable definitions */
var output angle
 set bignegativ
  -0.2 0  -0.19 1  -0.18 0
 endset
 set negativ
  -0.1 0  -0.09 1  -0.08 0
 endset
 set zero
   -0.01 0   0 1   0.01 0
 endset
 set positiv
   0.08 0  0.09 1  0.1 0
```

Figure 1.6f: The ’Fuzzy Controller Parameter’ dialog

the button ’Abort’ which leaves the file unchanged.

Any changes of the fuzzy description files will become active only by pressing the button ’Reload’ of the dialog. That means that even the fuzzy controller file (*.FBW) is updated when a new fuzzy description file was selected. The dialog is terminated by means of the ’Cancel’ button.

1.7 Menu Run

The pulldown menu **Run** in figure 1.7a contains items to start and stop a controller, to calibrate sensors, to record measurements and to adjust the setpoint. The items to start any controller are enabled only when the sensors have been calibrated successfully which is indicated by a check mark.

State Controller: Starts the state controller. A dialog window is opened automatically to configure observers and the disturbance compensation before the controller is active (see figure 1.7b). These settings may be changed even for an active controller.

Fuzzy Controller: Starts the fuzzy controller defined by its fuzzy description files (*.FBW). A dialog window is opened automatically to configure observers for the

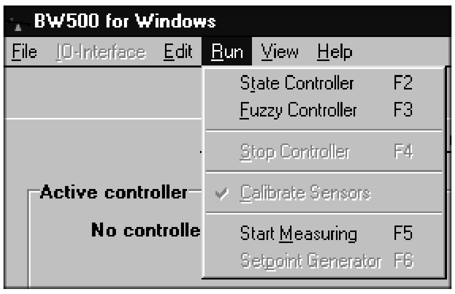


Figure 1.7a: The sub menu ’Run’

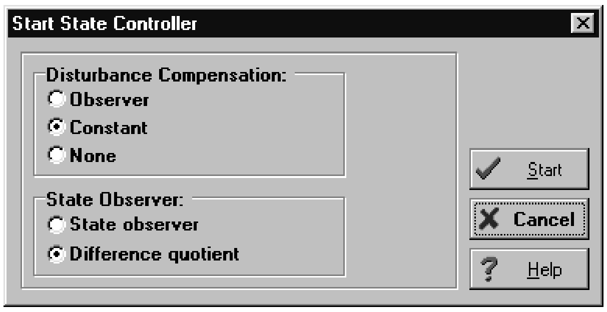
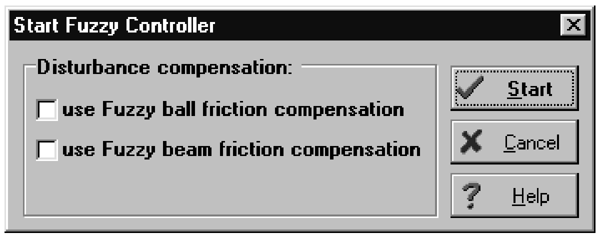


Figure 1.7b: The ’Start State Controller’ dialog

disturbance compensation before the controller is active (see figure 1.7c).


Start Fuzzy Controller

Disturbance compensation:

☐ use Fuzzy ball friction compensation

☐ use Fuzzy beam friction compensation

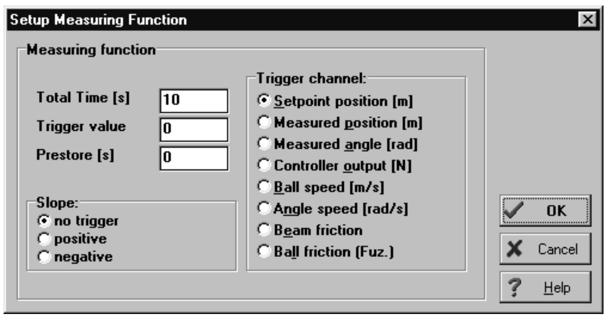
Figure 1.7c: The 'Start Fuzzy Controller' dialog

The menu item **Calibrate Sensors** carries out a menu driven calibration of the sensors as it was described with section 1.2. Any active controller will be stopped automatically.

Stop Controller: Stops any of the selected controllers and disables the menu item 'Setpoint Generator'.

The function **Start Measuring** is always enabled. It opens a dialog (see figure 1.7d) to adjust the measuring time and to assign trigger conditions to start recording the measurements. The measuring time in seconds is entered to the right to the title 'Total Time [s]:'. When 'Slope' is set to 'no trigger' measurement recording is started directly after closing the window using the 'Ok' button.

The trigger signal for conditional measuring ('Slope:' is set 'positive' or 'negative') is selected below the title 'Trigger Channel:'. The measurement recording starts after this signal raises above or falls below, depending on the settings of 'Slope', the limit value 'Trigger Value:'. In addition 'Prestore:' allows for adjustment of a time range for recording measurements before the trigger condition is valid. This time has always to be shorter than the adjusted measuring time.


Setup Measuring Function

Measuring function

Total Time [s] 10

Trigger value 0

Prestore [s] 0

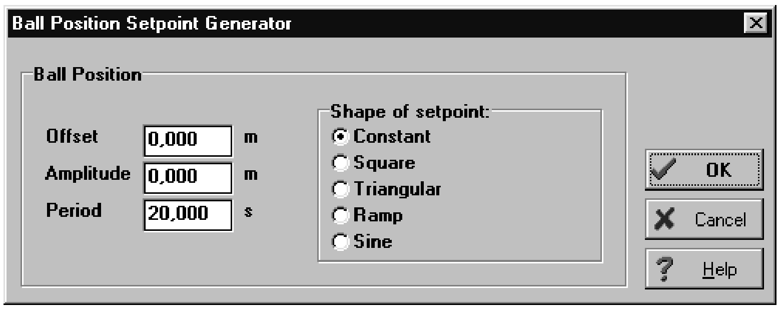
Slope: [✓ no trigger] positive negative

Trigger channel: [✓ Setpoint position [m]] Measured position [m] Measured angle [rad] Controller output [N] Ball speed [m/s] Angle speed [rad/s] Beam friction Ball friction (Fuz.)

Figure 1.7d: The 'Setup Measuring Function' dialog

The **Setpoint Generator** dialog (see figure 1.7e) is available only when one of the controllers is active. This dialog provides the adjustment of the setpoint of the ball position.

As can be seen from the figure, the setpoint is provided by a signal generator. The adjustable parameters are amplitude, offset, period and signal shape. In case the item 'Constant' is selected for the signal shape the corresponding setpoint value is offset + amplitude. The last is true also for periodic signals (rectangle, triangular, ramp, sine) with adjustable amplitude.


Ball Position Setpoint Generator

Ball Position

Offset 0,000 m

Amplitude 0,000 m

Period 20,000 s

Shape of setpoint: [✓ Constant] Square Triangular Ramp Sine

Figure 1.7e: 'Ball Position Setpoint Generator' dialog

1.8 Menu View

The pulldown menu **View** (see figure 1.8a) provides functions for graphic representations of recorded measurements, of data from a documentation file (*.PLD) as well as 3D-characteristics of a selectable fuzzy controller. Timing data may be displayed in addition.

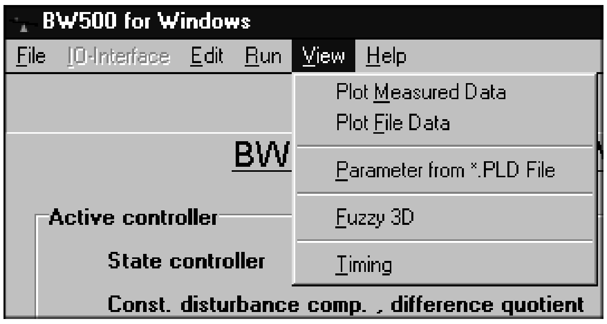


Figure 1.8a: The sub menu 'View'

The menu item **Plot Measured Data** is enabled only after the first measurement acquisition is started. It opens a dialog window (see figure 1.8b) to select the data which are to be displayed in a graphic representation. Terminating this dialog with 'Ok' will display the graphic window automatically on the screen. An example is shown in figure 1.8c.

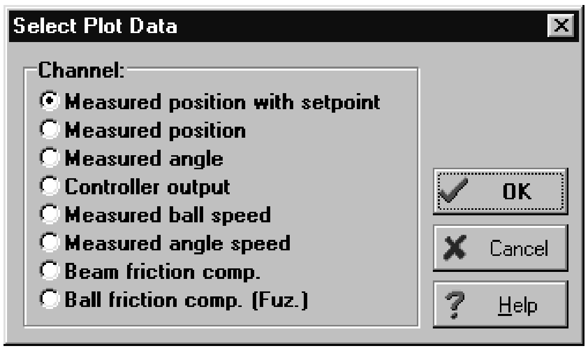


Figure 1.8b: The 'Select Plot Data' dialog

The menu item **Plot File Data** is enabled only when a documentation file (*.PLD) was recently loaded by means of the menu item 'Load Recorded Data'. The data of the documentation file are selected and displayed in a graphic representation as with the menu item 'Plot Measured Data'.

The menu item **Parameter From *.PLD File** generates an information box displaying the controller type and parameters read from the currently selected documentation file (*.PLD). This menu item is enabled only when such a file was loaded successfully.

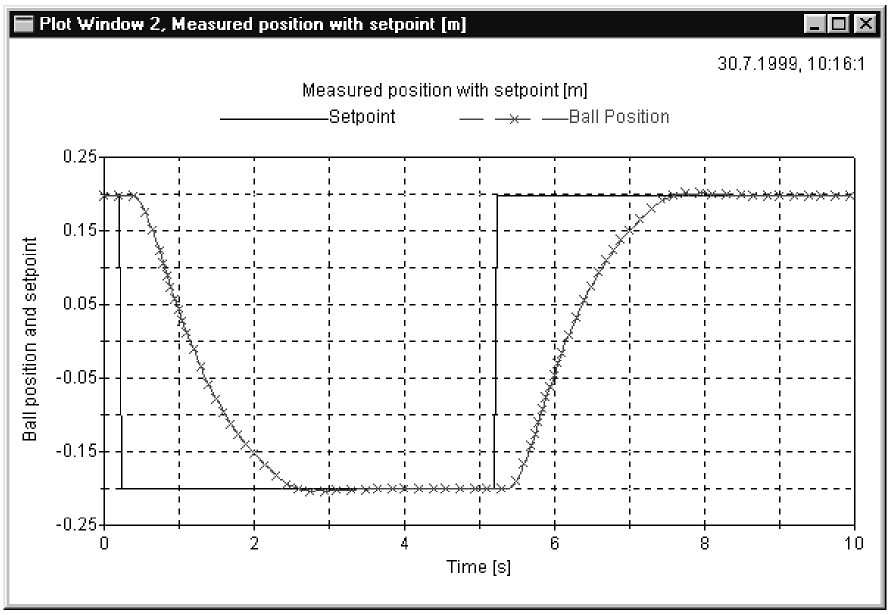


Figure 1.8c: Example of a 'Plot Window'

The menu item **Fuzzy 3D** opens a dialog (see figure 1.8d) displaying the controller characteristic of a selectable fuzzy controller in a three-dimensional graphic.

The fuzzy controller is referenced by its fuzzy description file. The X-axis and the Y-axis of the graphic represent the two inputs of a fuzzy controller while the Z-axis represent its output. The dimensions, that means the ranges of the input and output signals, of the resulting cube are displayed in a static field below the selector box for the fuzzy description file. The middle of the cube is indicated by a blue point while the minimum value for all axes is indicated by a red point.

The group of check boxes allows for manipulating the layout of the graphic:

With ’Grid’ marked a grid with either a higher or lower resolution depending on the setting of ’Low resolution’ is displayed along the surface of the characteristic.

The surface itself is displayed in a grey scale (darker areas indicate higher values along the Z-axis) when ’Surface’ is marked. The surface will be coloured if ’Colour’ is marked in addition (increasing values along the Z-axis are indicated by colour changes from red to blue).

The margins of the cube are displayed only when ’Co-ordinate box’ is marked. The same is valid for the three axes depending on the setting of ’Co-ordinate system’.

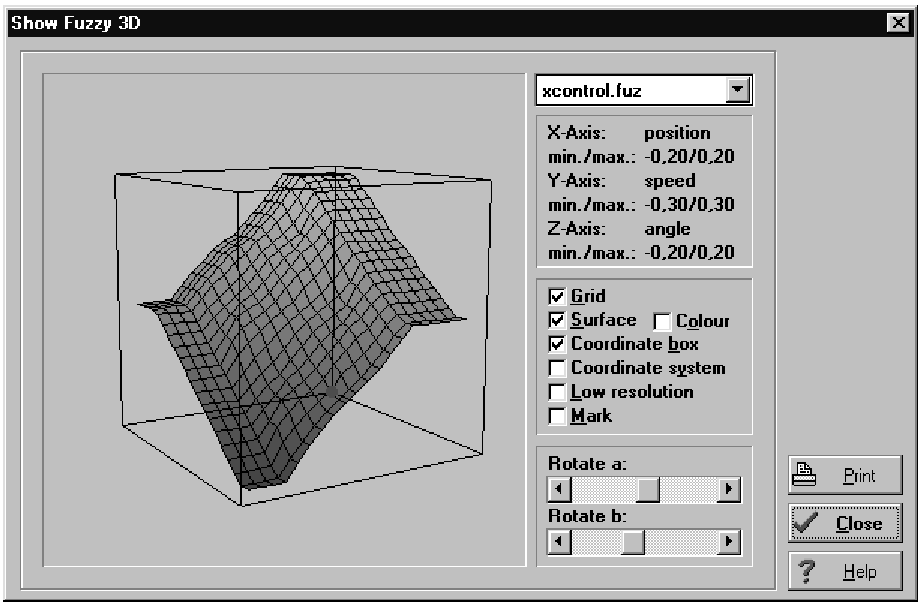
The check box ’Mark’ is selectable only when a fuzzy controller is active. If ’Mark’ is set the current operating point of the active fuzzy controller is indicated by a small green area. Its dimension corresponds to the currently selected grid width.

The scroll bars labelled ’Rotate a:’ and ’Rotate b:’ allow for rotating the graphic with respect to the X-axis and the Y-axis respectively. The crossing point of these axes is the middle of the cube. Alternatively the rotation is achieved by moving the mouse in the cube area accordingly.

The button ’Print’ starts a hardcopy output to the currently selected printer device.

The dialog is terminated by pressing the button ’Close’.

Activating the menu item **Timing** will present a window (see figure 1.8e) displaying the minimum and maximum values of the sampling period or the calculation time in milli seconds measured (with a resolution of 1ms) since the last start of a controller. The standard value is the



xcontrol.fuz

X-Axis: position
min./max.: -0,20/0,20
Y-Axis: speed
min./max.: -0,30/0,30
Z-Axis: angle
min./max.: -0,20/0,20

- [✓] Grid
- [✓] Surface
- [ ] Colour
- [✓] Coordinate box
- [ ] Coordinate system
- [ ] Low resolution
- [ ] Mark

Rotate a:
Rotate b:

Figure 1.8d: The ’Show Fuzzy 3D’ dialog

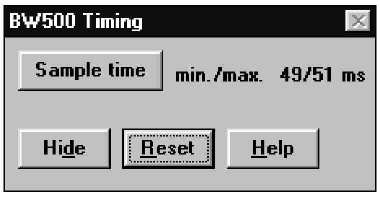


Figure 1.8e: The 'BW500 Timing' dialog

sampling period. While its minimum value is normally close to the nominal value of 50 ms, the maximum value may differ significantly from the nominal value especially in the case another Windows task with time consuming file accesses was started in the meantime.

Warning:
Starting another Windows task with too much file accesses while the controller program is running may cause a reset of the output stage release for the servo amplifier !!!

The calculation of the minimum and maximum values of the sampling period may be restarted by resetting the values by means of the button 'Reset'

The button labelled 'Sample time' resp. 'Calc time' switches between the two corresponding values.

The dialog will be terminated by pressing the button 'Hide'.

1.9 Menu Help

The pulldown menu **Help** as shown in figure 1.9a provides functions to control the Windows help function and to obtain general information about the program.

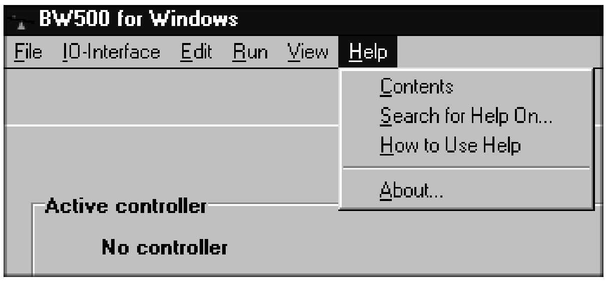
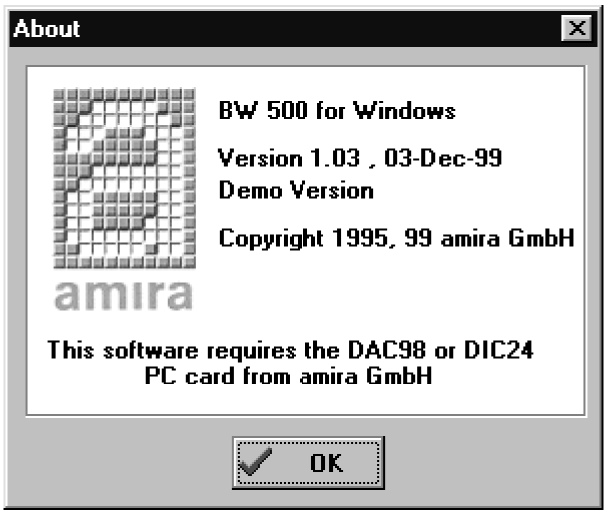


Figure 1.9a: The sub menu 'Help'

The menu item **Contents** displays the contents of the help file DTS200.HLP, while **Search for Help On ...** searches for keywords contained in this help file. The item **How to Use Help** opens the Use Help Dialog of Windows.

Activating the menu item **About** opens an information box displaying the program version, the copyright and the IO-adapter card requirements (see figure 1.9b).


About

BW 500 for Windows

Version 1.03 , 03-Dec-99
Demo Version

Copyright 1995, 99 amira GmbH

This software requires the DAC98 or DIC24 PC card from amira GmbH

Figure 1.9b: The 'About' dialog

1.10 Description of the File Formats

1.10.1 The Format of the Fuzzy Description File (*.FUZ)

The fuzzy description file with the extension FUZ is a file to configure a fuzzy controller. The file format is developed by the amira GmbH and is used by several products of the company amira.

The fuzzy description file is used to configure a fuzzy object, which i.e. may operate as a fuzzy controller.

The fuzzy description file is a simple ASCII file, which can be edited by a text editor. The length of a line is limited to 255 characters. Single assignments are separated by spaces or tabulators.

It contains four types of elements, which are described in the following sections:

Comments [optional]

The file can include a comment in classical C-style (’/*’ at the beginning and ’*/’ at the end) at every position except for the definition part of label. At least one space has to separate the comment string from the ’keywords’ ’/*’ and ’*/’.

The Definition of a Label [optional]

The definition of a label is limited to one line. It starts with the statement ’#define’. The next statement contains the label name and the last statement contains the label definition. Thus a label can be defined as follows:

```
#define name
 This_is_the_definition_of_the_label_name
```

The Definition of Fuzzy Sets and Variables

The definition of fuzzy sets is only allowed within the definition of variables. It is ignored in the other case. The definition of a variable starts with the statement ’var’. The next statement can hold two different names, either ’input’ in case an input variable is to be defined or ’output’ in case an output variable is to be defined. The third statement of a variable definition is its name. Now the definition of the fuzzy set follows. It begins with the statement ’set’ followed by the name of the fuzzy set. The name is followed by the x/y values as base points for a polygonal line. Similar to the statements the numbers are separated by spaces or tabulators. The definition of the fuzzy set ends with the statement ’endset’. The definition of a variable ends with the statement ’endvar’ after all the fuzzy sets of the fuzzy variable are defined. Such a definition may look like the following:

```
var input temperature
set cold     10 1     20 0            endset
set medium   10 0     20 1     30 0   endset
set warm     20 0     30 1            endset
endvar
```

The Definition of Fuzzy Rules

The definition of a fuzzy rule is recognized from its first statement ’if’. The last statement of a fuzzy rule is named ’end’. The definition of a fuzzy rule contains two parts,

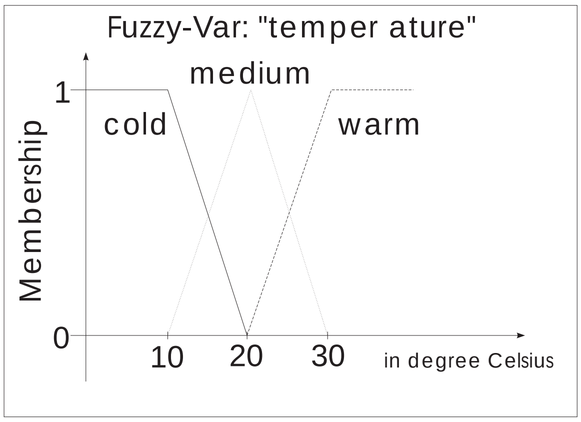


Figure 1.10: The Fuzzy variable ’temperatur’

the premise and the conclusion. Both parts are separated by the statement ’then’. The premise and the conclusion are built by a series of expressions which are combined by operators (further details are shown in the chapter of the theoretical backgrounds of a fuzzy controller). Permitted operators of the premise are ’and’ (Min-Operator) and ’or’ (Max-Operator) whereas the conclusion requires no operator to separate the expressions. An expression is the linkage of a fuzzy variable with one of its sets using the statement ’is’.

The formulation of a fuzzy rule requires that all the variables in use are defined previously since the fuzzy description file is interpreted only once from top to bottom. The syntax check of a fuzzy object tests whether the variables are defined, whether the used sets really belong to the variable and if the expressions are used correctly (input variables with the premise and output variables with the conclusion). A simple definition of a fuzzy rule may look like the following:

if temperature is cold then heating is high end

Table of the valid commands (keywords) and their explanation:

Command	Explanation
#define NAME TEXT	Defines a NAME, which is usable in the following statements and will be replaced by the definition TEXT automatically by the pre-processor.
/*	Begin of comment, ignored by the fuzzy controller kernel.
*/	End of comment.
var	Begin of linguistic variable definition. The statements "input" or "output" and the name of the variable must follow this keyword. Fuzzy sets are definable only in the following. The definition of the variable is terminated with the statement "endvar".
input	Defines the direction input for a variable.
output	Defines the direction output for a variable.
endvar	End of definition of a variable.
set	Begin of fuzzy set definition. A set name and a series of pairs of values must follow this keyword. The pairs of values are the base points of the set.
endset	End of set definition.
if	Begin of fuzzy rule definition. One or multiple premises separated by operators, the statement "then" and one or multiple conclusions must follow this keyword. The rule definition is terminated by the statement "end". A premise consists of a name of an input variable, the statement "is" and the name of the set belonging to this input variable. The conclusion is built in a similar way but the input variable is replaced by the output variable.
is	Separates variable and set in a premise or conclusion.
then	Separates the condition and the assignment part of a fuzzy rule.
and	Is the Minimum-Operator.
or	Is the Maximum-Operator.
end	End of rule definition.

Remark

The status and error messages which occur during the interpretation of the fuzzy description file are written to the file **ERROR.OUT** or appear on the screen.

1.10.2 Format of the File ERROR.OUT

The file ERROR.OUT may contain error messages as well as status messages. It may look like:

```
Fuzzy Parser Version 1.04 (07-DEC-94)

Fuzzy-Set <set_name> is already defined.
Fuzzy-Set <set_name> expects numerical value.
Unknown variable specification <string>.
Variable <var_name> is already defined.
Rule error, fuzzy variable <var_name> not found.
Rule error, fuzzy variable <var_name> is an
output variable.
Rule error, fuzzy variable <var_name> is an input
variable.
Rule syntax error, missing is.
Rule error, fuzzy set <set_name> is not member
of <var_name>.
Rule syntax error, unknown Operator <string>.
<label_name> is already defined.

<n> Errors detected.
```

1.10.3 Format of the Fuzzy Controller File for the Laboratory Experiment BW500 (*.FBW)

The fuzzy controller file for the BW500 contains four file names of fuzzy description files required for the fuzzy controller. Every file name begins in a new line, comments or empty lines are not allowed. Please change this file only using corresponding functions of the BW500 controller software.

The fuzzy controller file DEFAULT.FBW looks like the following:

```
xcontrol.fuz
wcontrol.fuz
xerror.fuz
werror.fuz
```

1.10.4 Format of the State Controller File for the Laboratory Experiment BW500 (*.STA)

This file contains all parameters of the state controller as well as the corresponding observers for the laboratory experiment BW500. Each entry consists of two lines, an information block in square brackets in the first line and a data block in the second line. Further comments or empty lines are not allowed. The information block describes sufficiently the function and number of data inside the data block. Please change this file only using corresponding functions of the BW500 controller software.

The state controller DEFAULT.STA file looks like the following:

```
[Sampling Period]
0.05
[Feedback Vector]
-20 -40 -60 -2.3
[Observer Transformation Vector (l)]
29.8 0.084 -3467.8 28.574
[Observer System Matrix (a)]
4.54e-05 0 0 4.54e-05
[Observer Feedback Matrix (f)]
-29.83 -0.49 3467.6 173.4
[Observer Control Vector (b)]
-0.006 -0.256
[Disturbance Observer Transformation Vector (l)]
 0 0 0 -4.04
[Disturbance Observer System Matrix (a)]
 0.00674
[Disturbance Observer Feedback Matrix (f)]
0 0 -0.0005 3.525
[Disturbance Observer Control Vector (b)]
-0.99
[Constant Disturbance Compensation]
2.1
[Settings]
1 0
```

1.10.5 The Format of the Documentation File *.PLD

Measured data stored in a data file are reloadable and may be output in a graphic representation. In addition the system settings (CTRLSTATUS) which were active during the start of the data acquisition are stored in this file. They are displayable in a separate window.

The data file contains data in binary format stored in the following order:

The structure PROJEKT PRJ.
The structure CTRLSTATUS.
The structure DATASTRUCT.
The data array with float values (4 bytes per value)

The size of the data array is defined in the structure DATASTRUCT. With the BW500 the number of the stored channels is always 8 (the length of the measurement vector is 8, i.e. equal to 32 bytes). The vector contains the following signals:

the position setpoint of the ball [m],
the measured position of the ball [m],
the angle of the beam [rad],
the control force [N],
the velocity of the ball [m/s],
the angle velocity of the beam [rad/s],
the friction compensation of the beam [N],
the friction compensation of the ball [N].

The number of the stored measurement acquisitions (vectors) depends on the adjusted values for the sampling period and the measuring time. The maximum number of measurings is 1024. The time distance between two successive acquisitions is an integral multiple of the sampling period used by the controller.

1.10.6 Format of the Calibration Data File DEFAULT.CAL

The standard calibration data file DEFAULT.CAL is read automatically during the startup of the program BW500W.EXE. If it does not exist default values will be taken for the calibration data. Terminating the program will possibly create this file and write automatically the calibration data to it.

The file contains three parameters in ASCII format:

The first parameter represents the number of increments read for the zero angle of the beam.

The second parameter represents the decimal number corresponding to the camera signal read for the zero position of the ball.

The third parameter represents the decimal number corresponding to the camera signal range which maps the ball position range between the left margin and the right margin of the beam.

The file DEFAULT.CAL may have the following content:

```
-3  741  560
```

1.11 The DEMO Version

The demo version of the program BW500W.EXE is indicated by the title "BW500 - Monitor (Demo-Version)" in the monitor window. It operates with a non-linear mathematical model of the plant instead of reading sensor signals from the IO-adapter card or writing control signals to this card. Besides the functions to control the calibration and to select the IO-interface all of the menu items are available.

Remark:
Because the program names of the demo version and the standard version are the same the programs must reside in different subdirectories including the accompanying drivers and dynamic link libraries. Furthermore the dummy driver DUMMY.DRV must reside in the same directory as the demo version of the program.

The PC Plug-in Card DAC98

Date: 03.07.1998

1 The PC Plug-in Card DAC98 (PCA902)

1.1 Introduction

The DAC98 is a card for general purpose on an IBM-AT compatible PC. The different analog and digital inputs and outputs allow for a variance of applications in automatic measurement and control.

1.2 Features

- 8 analog inputs with a programmable input signal range for each channel
- 1 12 bit A/D converter: MAX197
- 2 bipolar/unipolar analog outputs
- 2 12 Bit D/A converter: AD 7542
- 3 quadrature incremental encoder inputs
- 16 bit counter for incremental encoder signals: DDM
- 8 TTL compatible inputs
- 8 TTL compatible outputs
- 32 bit timer/counter for interrupt control or time measurement
- 16 bit timer/counter for interrupt control or time measurement

1.3 Specifications

Analog Inputs:	
Number of inputs:	8
Converter:	1 MAX197
Resolution:	12 bit
Programmable input signal range :	5V
	10V
	+/- 5V
	+/- 10 V
Analog resolution:	max. 1.22mV
Low-pass filter:	10nF
Input resistance:	10k
Analog Outputs:	
Number of outputs:	2
Converter:	2 AD 7545
Resolution:	12 bit
Output signal range:	10V
	+/- 10 V
Analog resolution:	max. 2.44mV
Encoder inputs:	
Number of inputs:	3 (quadrature signals)
Decoder:	CPLD (DDM) developed by amira
Input signal level:	RS422
Counter width:	16 bit
Digital Inputs:	
Number of inputs:	8
Level:	TTL compatible
Digital Outputs:	
Number of outputs:	8
Level:	TTL compatible

1.4 Installation of the DAC98

1.4.1 Adjustment of the Base Address

One of 8 possible base addresses (0x300..0x370) is adjustable by means of a DIP switch providing a 3 bit coding. The meaning of the switch positions is as follows

1 = Switch position on
0 = Switch position off
(*) = Default configuration

Note: The base address is the start address of the I/O address range which must not be used by any other PC plug-in card.

The enclosed driver software requires the card with the base address 300 (hex). If you want to use this software without any changes please assure that none of the other PC plug-in cards in your PC uses the same base address. Otherwise you may change the base address in the software as well as for the PC plug-in card accordingly.

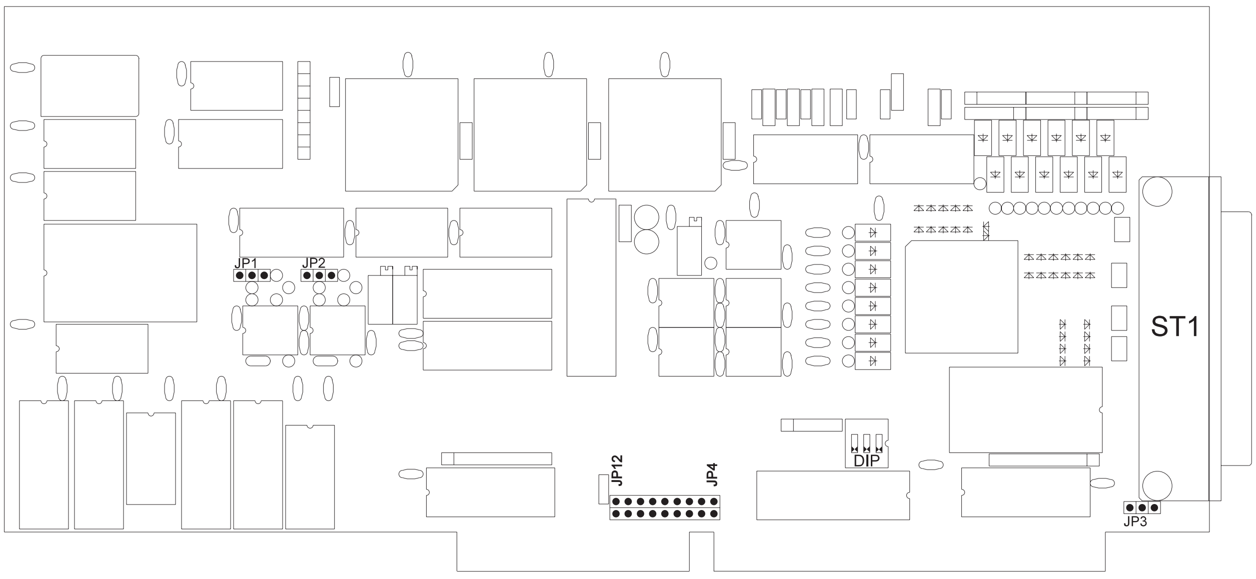
I/O Address (Hex)	3	2	1
300(*)	1	1	1
310	0	1	1
320	1	0	1
330	0	0	1
340	1	1	0
350	0	1	0
360	1	0	0
370	0	0	0

1.4.2 Adjustment of the Interrupt Channel

In case the interrupt feature of the DAC98 is to be used, a free interrupt channel of the PC hardware has to be identified. This channel number is then adjusted by means of the jumpers JP4 to JP12 (see table). The default interrupt channel setting is IRQ7. As for the base address it is to be assured that none of the other PC plug-in cards uses the same interrupt channel.

JP	4	5	6	7	8	9	10	11	12
IRQ	3	4	5	7	9	10	11	12	15

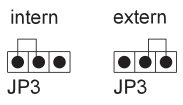
The PC hardware may be damaged when more than one jumper is installed or in case the selected interrupt channel is in use by another card.



Jumpers of the DAC98

1.4.3 The Operation Modes of the 16 Bit Timer/Counter

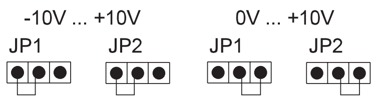
The 16 bit counter either counts external events or it counts the timer clock. The jumper JP3 adjusts the corresponding operation mode.



The default setting is timer clock counting mode.

1.4.4 Adjustment of the Analog Output Signal Range

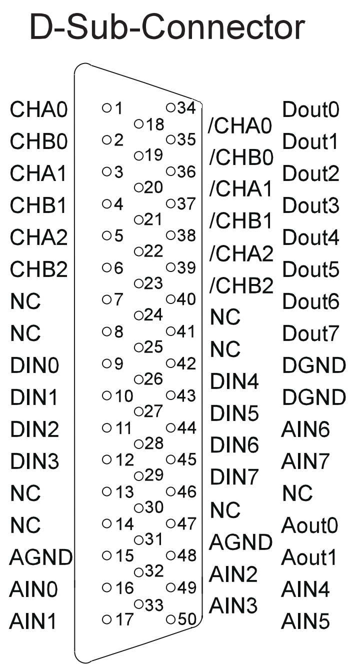
Two different output voltage ranges (-10 V to +10 V or 0 V to +10 V) are selectable for each D/A converter. The jumpers JP1 (analog out 0) and JP2 (analog out 1) provide this selection for each channel.



1.4.5 Pin-Reservations of the DAC98

The DAC98 is plugged in the PC with its slot connector and fixed with a screw at the rear of the PC casing. All of the input and output channels are accessible at the rear by a 50-polar D-Sub female connector.

ST1: 50-polar D-SUB connector



PC-Connector (DAC98)		
Pin-No.	Pin-Descr.	Reservation
1	CHA0	incremental encoder signal A 0
2	CHB0	incremental encoder signal B 0
3	CHA1	incremental encoder signal A 1
4	CHB1	incremental encoder signal B 1
5	CHA2	incremental encoder signal A 2
6	CHB2	incremental encoder signal B 2
7	CHA3	incremental encoder signal A 3
8	CHB3	incremental encoder signal B 3
9	DIN0	digital input 0
10	DIN1	digital input 1
11	DIN2	digital input 2
12	DIN3	digital input 3
13	n.c.	n.c.
14	n.c.	n.c.
15	AGND	analog ground
16	AIN0	analog input 0
17	AIN1	analog input 1
18	/CHA0	inverted incremental encoder signal A0
19	/CHB0	inverted incremental encoder signal B0
20	/CHA1	inverted incremental encoder signal A1
21	/CHB1	inverted incremental encoder signal B1
22	/CHA2	inverted incremental encoder signal A2
23	/CHB2	inverted incremental encoder signal B2
24	/CHA3	inverted incremental encoder signal A3
25	/CHB3	inverted incremental encoder signal B3
26	DIN4	digital input 4
27	DIN5	digital input 5
28	DIN6	digital input 6
29	DIN7	digital input 7

PC-Connector (DAC98)		
Pin-No.	Pin-Descr.	Reservation
30	n.c.	n.c.
31	AGND	AGND
32	AIN2	analog input 2
33	AIN3	analog input 3
34	Dout0	digital output 0
35	Dout1	digital output 1
36	Dout2	digital output 2
37	Dout3	digital output 3
38	Dout4	digital output 4
39	Dout5	digital output 5
40	Dout6	digital output 6
41	Dout7	digital output 7
42	DGND	digital ground
43	DGND	digital ground
44	AIN6	analog input 6
45	AIN7	analog input 7
46	Timer /Clk	input for external events
47	Aout0	analog output 0
48	Aout1	analog output 1
49	AIN4	analog input 4
50	AIN5	analog input 5

Note:

All the analog inputs which are not in use have to be connected to the analog ground.

1.4.6 Installation of the Card in the PC

a) Switch off the power to the PC and all other connected peripheral devices, e.g. monitor, printer.

b) Disconnect all cables of your PC.

c) Remove the top cover of your PC. (For details please refer to the manual of your PC).

d) Choose a free add-on slot (16 bit ISA) and remove the corresponding slot cover at the rear.

e) Plug in the DAC98 in the chosen slot and tighten the screw to hold the card's retaining bracket.

f) Replace the PC's top cover and fasten the screws. Connect all cables.

The card is now ready for operation. To test the function please install the software (ref. chapter 3.1).

1.5 Programming of the DAC98

Initialization and programming of the PC plug-in card DAC98 is described in the following to give a better understanding of its functions. The functions itself are realized by the drivers (see also chapter 4) included in the shipment.

1.5.1 The Registers of the DAC98

Mainly two ports are used to program the DAC98. One hardware address register (HWADR) for addressing the individual components of the card, and one data register (DATR). The access mode of the HWADR at the base address (BADR) is write only.

The DATR is addressable by BADR + 4 and can either be read or written, depending on the chosen HWADR.

The following table contains all hardware addresses of the card. The first column is the address (in Hex), the following column is the function of the DATR.

(r) = read the DATR, (w) = write to the DATR

HWADR (r/w)	
0x00(r)	read the identification string
0x08(r/w)	initialize A/D converter A/D converter low-byte
0x09(r)	A/D converter high-byte
0x10(w)	D/A converter channel 0
0x18(w)	D/A converter channel 1
0x20	Counter No. 0 of the 8254 timer
0x21	Counter No. 1 of the 8254 timer
0x22	Counter No. 2 of the 8254 timer
0x23	Mode of the 8254 timer
0x28	PortA of the 8255 IO-interface digital outputs
0x29	PortB of the 8255 IO-interface digital inputs
0x2A	PortC of the 8255 IO-interface digital outputs/inputs used internally
0x2B	Mode of the 8255
0x30	DDM No. 0 high-byte during read operation of the 16 bit increment counter, chip reset during write operation
0x31	DDM No. 0 low-byte of the 16 bit increment counter
0x38	DDM No. 1 high-byte during read operation of the 16 bit increment counter, chip reset during write operation
0x39	DDM No. 1 low-byte of the 16 bit increment counter
0x78	DDM No. 2 high-byte during read operation of the 16 bit increment counter, chip reset during write operation
0x39	DDM No. 2 low-byte of the 16 bit increment counter
0xBA	CSDDMALL
0xBB	all of the DDM chips are reset 0xF8 interrupt/clock signal

1.5.2 Configuration of the DAC98

The PC plug-in card DAC98 contains programmable chip devices which have to be initialized before using the functions of the card.

At first the digital inputs and outputs are to be configured by programming the 8255 chip containing 3 digital ports (PortA, PortB, PortC) either operating as inputs or outputs with 8 bits for each port. PortC is used internally and is to be programmed such that its bits 0...3 operate as inputs and its bits 4...7 operate as outputs. PortA has to operate as an output whereas PortB has to operate as an input.

The programming of the chip is performed in two steps. At first a value of 0x2B (address of the 8255 mode register) is written into the HWADR. Writing then a data value of 0x8A into DATR will program the input/output functions as described above.

At next the frequency to control the timer is to be programmed which will be described in section 1.5.11 .

1.5.3 Reading the Identification String

Reading the identification string (a preassigned bit map) is performed in two steps. At first a value of 0x00 (address of the identification string) is written into the HWADR. Reading then the DATR should result in a value of 0x55 when the card is installed correctly.

1.5.4 A/D Conversion

This section describes the procedures required for an A/D conversion. At first the channel which is to be read as well as its input signal range are to be selected. To do this a value of 0x08 (address of the A/D converter) is written into the HWADR. Then a value determining the channel and its signal range is sent to the DATR. As described in the following table the bits 0...2 define the selected channel and the bits 3 and 4 define the selected input signal range. Writing to the DATR in this case will be followed automatically by starting the conversion. The end of the conversion is indicated by the digital input No. 8 or by the interrupt channel 2. Since a running conversion is indicated by a "1" in digital input No. 8 the digital inputs are to be read until this bit is reset to "0" to detect the end of the conversion. The read operation for the digital inputs is described in section 1.5.7.

Now the result of the A/D conversion may be read:

At first a value of 0x08 (address of low-byte) is written into the HWADR. Reading the DATR in the following will result with the low-byte. Writing then a value of 0x09 (address of high-byte) into the HWADR will result with the high-byte after the next reading of the DATR. The described sequence of operations is to be obeyed absolutely during reading the converter.

Bit Pattern			Selected Channel
D2	D1	D0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Bit Pattern		Selected Range
D4	D3	
0	0	0..5V
0	1	0..10V
1	0	-5..+5V
1	1	-10..+10V

1.5.5 D/A Conversion

This section describes the sequence of operations required for a D/A conversion. After writing the address value of the selected D/A converter (0x10 for No. 0 or 0x18 for No. 1) to the HWADR only one further write operation to the DATR is required to start the conversion immediately. The lower right 12 bits of the DATR represent the value which is to be converted.

1.5.6 Programming the Timer Chip 8253

Since this chip offers a lot of features it is recommended to use its hardware manuals i.e. "Intel Microprocessor and Peripheral Handbook, Volume II Peripheral". Only a simple application example will be described in the following.

The 16 bit timer 0 and 1 of this chip are wired by hardware such that they operate as a cascaded 32 bit timer. This 32 bit timer counts the clock signal provided by a quartz base with a programmable divisor. The remaining timer 3 operates as a single 16 bit timer/counter either counting the clock signal mentioned above or external events. On counter overflow an interrupt may be requested in case this feature is enabled.

The following example describes the programming of the 32 bit timer operating as a square wave generator (suitable for interrupt triggering) with a period of 10 seconds.

According to the default clock signal with 2 MHz, the 32 bit timer has to be initialized with a value of 20.000.000. Since two 16 bit timer operate in a cascade, this value has to be separated in the product 4000 * 5000 (hexadecimal format: 0x0FA0 * 0x1388). At first the mode register of the 8253 is addressed by writing the value 0x0B3 in the register HWADR. Writing a value of 0x36 in the register DATR will program the mode register such that counter 0 is selected, low-byte-first is adjusted and the mode 3 is selected. Then the counter 0 is addressed by writing 0x08 in the register HWADR. Now the low byte of the desired divisor (in our case the low byte of 0x0FA0 is 0xA0), that means 0xA0 is written in the register DATR at first. Afterwards the high byte (0x0F in our case) is written in the register DATR. With this the programming of the counter 0 is completed and the similar programming of the counter 1 will be as follows. The mode register of the 8253 is addressed again by writing the value 0x0B in the register HWADR. Writing a value of 0x76 in the register DATR will program the mode register so that counter 1 is selected, low-byte-first is adjusted and the mode 3 is selected. Then the counter 1 is addressed by writing 0x09 in the register HWADR. Now the low byte of the desired divisor (in our case the low byte of 0x1388 is 0x88), that means 0x88 is written in the register DATR at first. Afterwards the high byte (0x13 in our case) is written in the register DATR. With this the programming of the 32 bit timer is completed.

You will find an application of this programming instruction in the "C++" respective "C" files in the functions **SetTimer**, **SetCounter**. When using the 8253 timer/counter to program interrupts a minimum sampling period should be regarded. This minimum sampling period depends on the available computing power, the used operating and bus system etc.. With a standard PC (80386 DX40, operating system DOS, ISA-Bus) this minimum sampling period is about 0.5 ms, in case the interrupt service routine has a very short execution time.

1.5.7 Reading the Digital Inputs

Reading the digital inputs requires two steps. At first a value of 0x29 (address of digital inputs) is written into the HWADR. Reading the DATR in the following will result with the state of the digital inputs. The single bits of the data byte correspond to the input channel numbers as follows:

Data bit	Assignment
D0	digital input 0
D1	digital input 1
D2	digital input 2
D3	digital input 3
D4	digital input 4
D5	digital input 5
D6	digital input 6
D7	digital input 7

For the digital inputs a 1 means the input has an high level signal.

1.5.8 Setting the Digital Outputs

Setting the digital outputs requires two steps. At first a value of 0x28 (address of digital outputs) is written into the HWADR. Writing a data byte to the DATR in the following will set the digital outputs accordingly. The single bits of the data byte correspond to the output channel numbers as follows:

Data bit	Assignment
D0	digital output 0
D1	digital output 1
D2	digital output 2
D3	digital output 3
D4	digital output 4
D5	digital output 5
D6	digital output 6
D7	digital output 7

1.5.9 Internal Digital Functions

The internal functions are initialized by a write operation followed by a read or write operation. At first a value of 0x2A (address of PortC of the 8255). is written into the HWADR. Reading the DATR in the following will result in a specific status information whereas writing to the DATR will result in specific settings according to the following table:

Data bit	Assignment
D0	set the gate of the 32-bit timer (output)
D1	set the gate of the 16-bit timer (output)
D2	not used
D3	not used
D4	busy signal of the A/D converter (input)
D5	not used
D6	not used
D7	not used

1.5.10 Reading an Incremental Encoder Input Channel

This section describes the read procedure of the incremental encoder input channel 0 in example. Using other channels requires the corresponding chip addresses.

Two steps are required:
a) Any arbitrary write access two a DDM chip results in changing its internal register sets such that the current data are available for reading. To do this a value of 0x31 (address of DDM No. 0) is written into the HWADR followed by a write operation to the DATR with an arbitrary value.
b) Now the content of the increment counter is ready for reading. At first a value of 0x30 (address of high byte of DDM No. 0) is written into the HWADR. Reading then the DATR will result with the high byte of the counter content. The low byte is accessed accordingly by using the address 0x31.

1.5.11 Interrupt / Clock Signal

The interrupt register is enabled and the clock signal is selected by the following operations. At first a value of 0xFA (address of interrupt / clock signal) is written into the HWADR. Writing a data byte to the DATR in the following will enable specific interrupts and select the clock signal according to the following table:

Data bit	Assignment
D0	interrupt of the 32-bit timer
D1	interrupt of the 16-bit timer
D2	interrupt on end of A/D conversion
D3	not used
D4	selected clock signal
D5	selected clock signal
D6	selected clock signal
D7	not used

Bit Pattern			Selected Clock
D6	D5	D4	
0	0	0	8MHz
0	0	1	4MHz
0	1	0	2MHz
0	1	1	1MHz
1	0	0	500kHz
1	0	1	250kHz
1	1	0	125kHz
1	1	1	62,5kHz

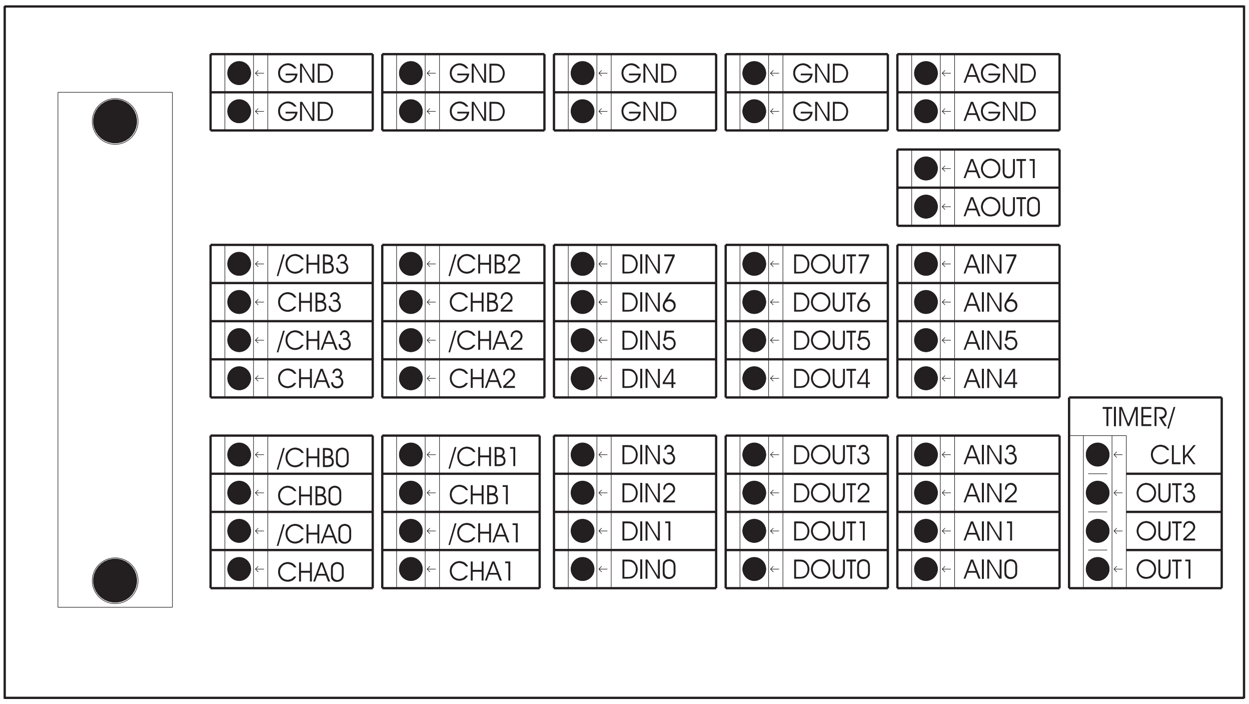
Note: To maintain any selected clock the corresponding bits have to be the same in following write operations to the interrupt / clock signal register.

The interrupt / clock signal register may also be read by writing a value of 0xFA (address of interrupt / clock signal) into the HWADR followed be reading the DATR.

2 DAC98 Adapter Card

2.1 Adapter Card EXPO-DAC98 (Opt. 902-01)

The adapter card EXPO-DAC98 contains screw terminals to provide the user with all the input/output signals of the DAC98. The adapter card is mounted in a aluminium case.



3 Operating Instructions for the Test Program

3.1 Installation

The test program requires an IBM compatible PC with Microsoft Windows 3.1 or Windows 95.

Now switch on your computer and start MS Windows.

Insert the **DAC98**-disk in the 3.5" disk drive of your computer. Now select the item "Run" from the menu "File" of the windows program manager from Windows 3.1 resp. the item "Run" of the start menu from Windows 95. Enter the command line

a:\install resp. **b:\install**

according to the drive assignment.

Prompt the input with "OK" or "Return". The running installation program now asks for the desired directory. The default setting **C:\DAC98** may be changed for the disk drive but without inserting additional sub-directories.

Attention !

Do not start DAC98TST.EXE directly from the floppy disk drive!

3.2 Program Start

After starting the program DAC98TST.EXE the main menu will appear on the screen as shown in figure 3.1. An error message will be displayed at first when the base address setting of the PC plug-in card does not match the corresponding address of the program. This address may be changed using the menu "IO-Interface" "Configuration".

The first line of the screen contains the menu bar. Its menu items are described in the following sections.

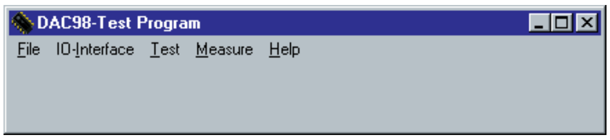


Figure 3.1: The main window of the DAC98 test software

3.3 Menu ’File’

The pull down menu ’File’ only contains the item ’Exit’ to terminate the test software as shown in figure 3.2.

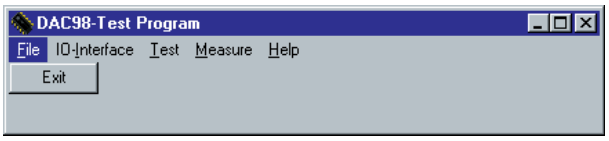
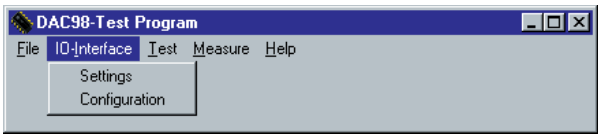


Figure 3.2: The menu ’File’

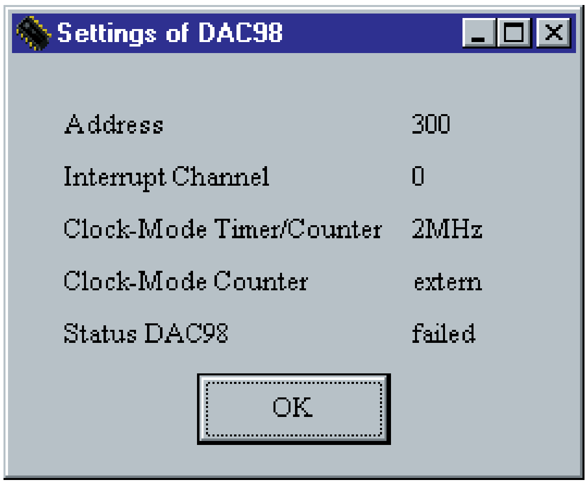
3.4 Menu ’IO-Interface’

The pull down menu ’IO-Interface’, see figure 3.3, provides two functions to manipulate the driver settings for the DAC98 PC plug-in card.



Figue 3.3: The menu ’IO-Interface’

The function ’Settings’ displays a window with the current driver settings as shown in figure 3.4. Any setting may be changed using one of the following sub-menus.



Address	300
Interrupt Channel	0
Clock-Mode Timer/Counter	2MHz
Clock-Mode Counter	extern
Status DAC98	failed

Figure 3.4: The window ’Settings’

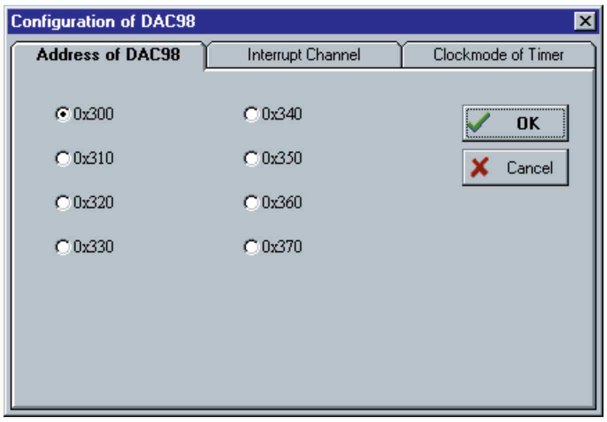
The function ’Configuration’ opens a window containing selectable dialogs as shown in figure 3.5.

The sub-menu ’Address of DAC98’ opens a menu to select one of the valid base addresses of the PC plug-in card.

The sub-menu ’Interrupt Channel’ opens a menu to select one of the useable interrupt channels.

The sub-menu ’Clockmode of Timer’ opens a menu to select the clock rate for the timer devices on the PC plug-in card.

The test software tries to access the base address to test the configuration when any selection is prompted using the OK button. Caution, any fault address setting may cause hardware conflicts!

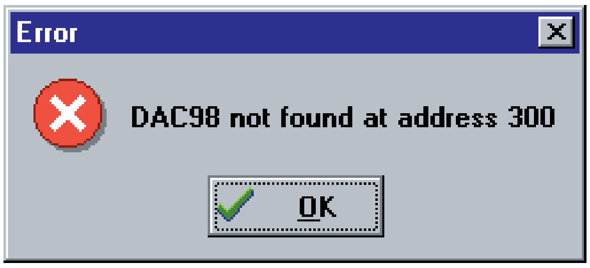


[✓ 0x300] 0x340
0x310 0x350
0x320 0x360
0x330 0x370

Figure 3.5: The window ’Configuration’

In case of a successfull test the current settings for the base address, the interrupt channel and the clock rate are stored in a file automatically. This file will be used during any start of the test software.

An error message will be displayed (see figure 3.3) when the PC plug-in card did not respond with the current base address settings.



DAC98 not found at address 300

Figure 3.6: The window ’Error’

3.5 Menu ’Test’

The pull down menu ’Test’ provides two functions to test the DAC98 PC plug-in card.

The menu ’Show all’ opens a window displaying the value of all signals of the DAC98 PC plug-in card. The analog signals are displayed in the left part whereas the digital signals are displayed in the right part of the window. The analog inputs will show a value of 0V and the digital inputs will show low level when the external connector of the PC plug-in card is open.

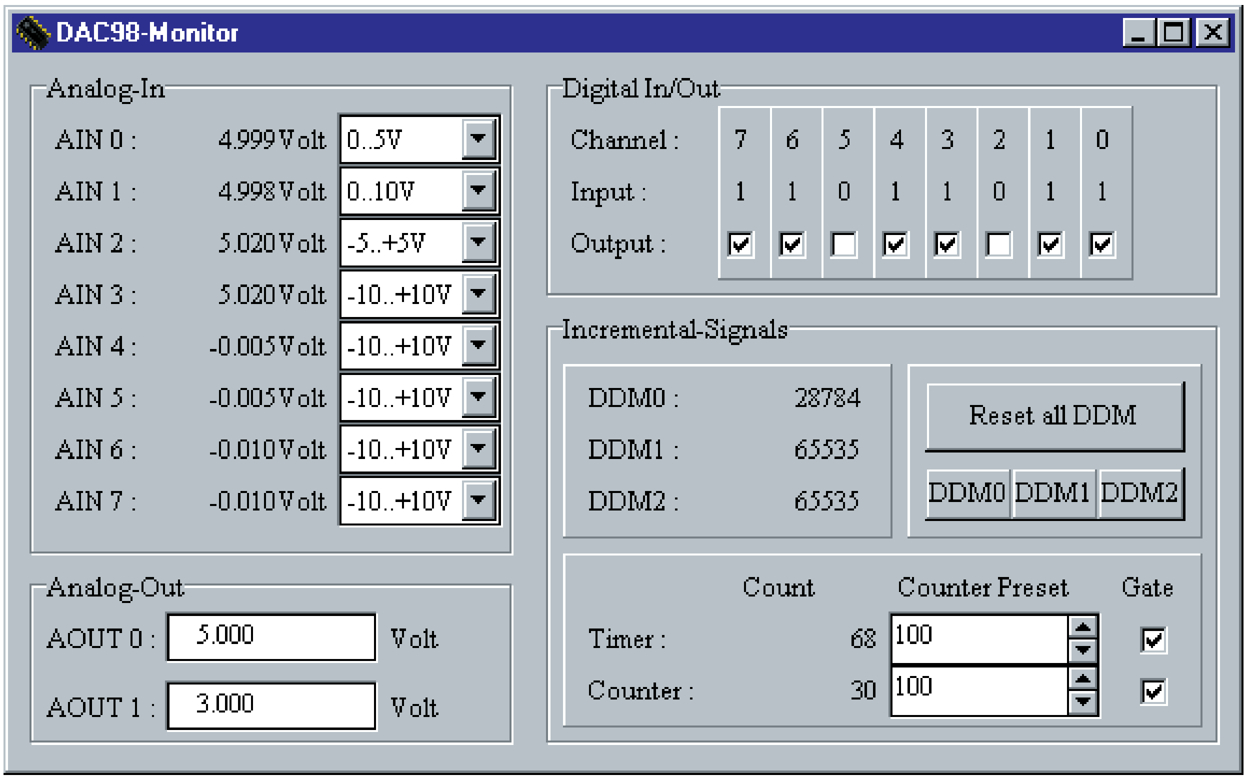
The input voltage range is selectable for each analog input left to the displayed measured value.

The analog outputs may be set after entering a valid number and prompting with ’Return’.

The digital outputs are manipulated by selecting the corresponding control fields.

The DDM devices may be reset using the displayed control buttons.

The timer resp. the counter decrement the preset value only when the ’Gate’ control button is active. The preset values may be changed at any time but they will be taken as start values only when the corresponding control button is active.


DAC98-Monitor

Analog-In

AIN 0 :	4.999 Volt	0..5V
AIN 1 :	4.998 Volt	0..10V
AIN 2 :	5.020 Volt	-5..+5V
AIN 3 :	5.020 Volt	-10..+10V
AIN 4 :	-0.005 Volt	-10..+10V
AIN 5 :	-0.005 Volt	-10..+10V
AIN 6 :	-0.010 Volt	-10..+10V
AIN 7 :	-0.010 Volt	-10..+10V

Analog-Out

AOUT 0 :	5.000	Volt
AOUT 1 :	3.000	Volt

Digital In/Out

Channel :	7	6	5	4	3	2	1	0
Input :	1	1	0	1	1	0	1	1
Output :	✓	✓	☐	✓	✓	☐	✓	✓

Incremental-Signals

DDM0 :	28784
DDM1 :	65535
DDM2 :	65535

	Count	Counter Preset	Gate
Timer :	68	100	✓
Counter :	30	100	✓

Figure 3.7: The window ’Showall’

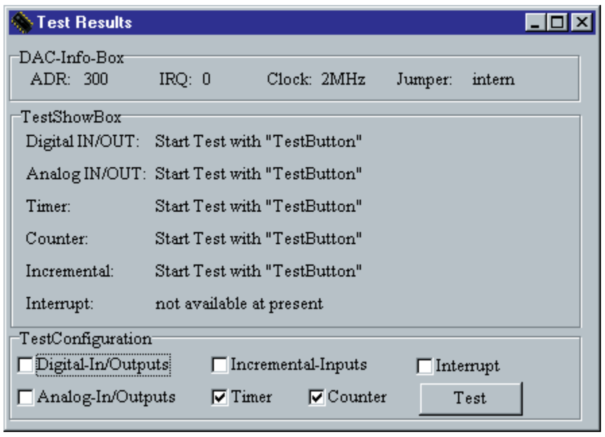
The menu item ’Test’ opens a window as shown in figure 3.8.

The field "DAC-Info-Box" displays the current configuration settings as well as a specific jumper setting of the PC plug-in card. For the jumper the message "intern" means that the timer counts the internal clock. The message "extern/undef" means that the timer either counts external events or that the jumper is missing.

The following fields allow for selection of single component tests and display the corresponding results. But all of these tests are meaningfull only when a special **test adapter from amira** is connected to the PC plug-in card.

The field "TestShowBox" displays the test results of the single components of the PC plug-in card.

The field "TestConfiguration" provides the selection of the components which are to be tested. The push button ’Test’ starts the test immediately. But all of these tests are meaningfull only when a special **test adapter from amira** is connected to the PC plug-in card.


Test Results

DAC-Info-Box
ADR: 300 IRQ: 0 Clock: 2MHz Jumper: intern

TestShowBox

Digital IN/OUT:	Start Test with "TestButton"
Analog IN/OUT:	Start Test with "TestButton"
Timer:	Start Test with "TestButton"
Counter:	Start Test with "TestButton"
Incremental:	Start Test with "TestButton"
Interrupt:	not available at present

TestConfiguration
☐ Digital-In/Outputs ☐ Incremental-Inputs ☐ Interrupt
☐ Analog-In/Outputs ☑ Timer ☑ Counter

Figure 3.8: The window ’Test Components’

3.6 Menu ’Measure’

The menu ’Measure’ opens a window displaying measured data from the analog inputs in a graphic. One or multiple data channels are selectable by corresponding control buttons. The input signal range is +/- 10V for each analog input. The width of the graphic corresponds to 380 samples taken in between a time which depends on the computing power of the PC.

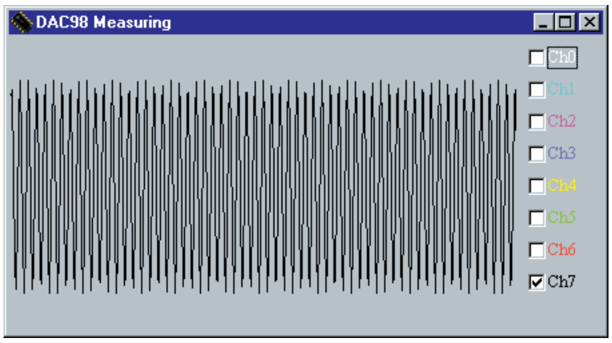
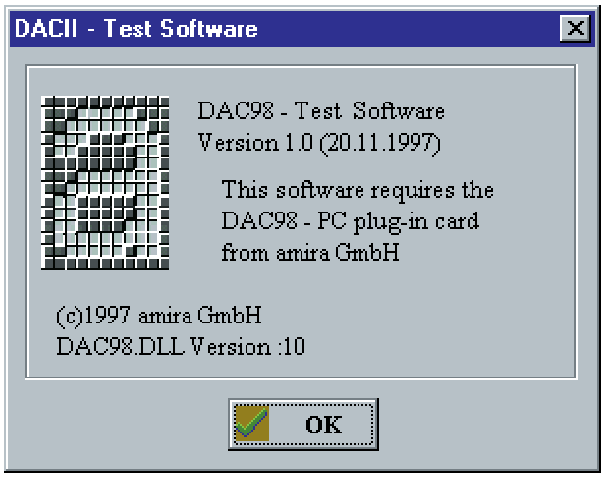


Figure 3.9: The window ’Measure’

3.7 Menu ’Help’

The pull down menu ’Help’ only contains the item ’Info’ as shown in figure 3.10. The selection of this item will display short information about the program version and the copyright.


DACII - Test Software

DAC98 - Test Software
Version 1.0 (20.11.1997)

This software requires the
DAC98 - PC plug-in card
from amira GmbH

(c)1997 amira GmbH
DAC98.DLL Version :10

Figure 3.10: The window ’Info’

4 Source Files of the DAC98 Driver

This chapter describes the contents as well as the functions of the driver modules written in C++ (16 bit version). The driver modules are contained in the file DAC98.CPP. A short DOS test program using some of the modules is given by the file TEST.CPP.

4.1 The Class DAC98

The class **DAC98** is used to control the PC plug-in card DAC98 of the company **amira** in a comfortable way. Several cards may be controlled without any problem by using as many driver objects.

Basic Classes:

none

Public Data:

unsigned int	ddm_counterr[3]	is an array containing the increment values counted by the three DDM devices.
enum	Clkmodes	defines the series of constant values for the clock rate of the timer device

Private Data:

unsigned char	ddm_adr[3]	is an array containing the addresses of the three DDM devices
unsigned long	timer_counter0	is the content of the first timer
unsigned long	timer_counter	is the content of the second timer
unsigned int	timer_counter2	is the content of the third timer
int	Base	is the base address of the DAC98
int	WR_DATA	is the offset which is to be added to the base address to write to the data register
int	RD_DATA	is the offset which is to be added to the base address to read the data register
int	intr	is the interrupt channel
double	Clock	is the timer clock
int	CounterGate	is the state of the counter gate
int	CounterJMP	is the state of the counter jumper
int	output_status_DAC98	is the register content of the digital outputs
int	input_status_DAC98	is the register content of the digital inputs
int	intr_status	is the content of the interrupt register

Public Element Functions:

Name:

DAC98

DAC98(int *adress*)

Class: DAC98

Description:

The constructor requires only the base address of the PC plug-in card

Parameter:

int	*adress*	is the base address of the PC plug-in card in the IO address range of the PC.

Return value:

none

Name:

GetAdress

int **GetAdress**(void);

Class: DAC98

Description:

The function **GetAdress** returns the variable *Base* which is the base address of the PC plug-in card adjusted by the constructor or by the function **SetAdress**.

Parameters:

none

Return value:

int	the adjusted base address of the PC plug-in card.

Name:

SetAdress

void **SetAdress**(int *adr*);

Class: DAC98

Description:

The function **SetAdress** adjusts the current base address to the new value of *Base*. This value has to match the base address which is configured on the hardware of the DAC98 to guarantee further accesses to the card.

Attention: Address conflicts may damage the PC hardware !

Parameters:

int	*adr*	is the new base address of the PC plug-in card DAC98.

Return value:

none

Name:

GetInterrupt

int **GetInterrupt**(void);

Class: DAC98

Description:

The function **GetInterrupt** returns a flag representing the number of the interrupt channel which was adjusted by the function **SetInterrupt** previously.

Attention: The jumper configuration on the hardware must be same. Interrupt conflicts may damage your hardware.

Parameters:

none

Return value:

int	number of interrupt channel.

Name:

SetInterrupt

void **SetInterrupt**(int *i*);

Class: DAC98

Description:

The function **SetInterrupt** adjusts the flag representing the number of the interrupt channel which is configured by a jumper on the hardware.

Attention: The jumper configuration on the hardware must be same. Interrupt conflicts may damage your hardware.

Parameters:

int	*i*	is the number of the new interrupt channel

Return value:

none

Name:

Identifikation

int Identifikation(void);

Class: DAC98

Description:

The function **Identifikation** checks whether the **amira** DAC98 responds to the base address of the driver software. Any value unequal to zero is returned when the PC plug-in card returns a bit string preassigned on its hardware.

Parameters:

none

Return value:

int	Result = 1 indicates that the hardware responded to the base address, else the result = 0.

Name:

Init

void Init(void);

Class: DAC98

Description:

The function **Init** at first calls the function **Identifikation**. When this call is successfull the **amira** DAC98 is initialized to the default settings.

Parameters:

none

Return value:

none

Name:

Exit

int Exit(void);

Class: DAC98

Description:

The function **Exit** adjusts the analog and digital outputs to 0, the DDM devices are reset and the counter as well as the timer are stopped.

Parameters:

none

Return value:

int	Result is always = 1.

Name:

Setup

int Setup(void);

Class: DAC98

Description:

The function **Setup** searches for the **amira** DAC98 using all of the adjustable base addresses. This operation may cause hardware conflicts when any other card operates in the same address range, i. e. a network card. So the function must only be used when this case can be excluded.

Attention: Address conflicts may damage the PC hardware !

Parameters:

none

Return value:

int	Values unequal to zero indicate a successful function.

Name:

SetClock

void **SetClock**(int *mode*);

Class: DAC98

Description:

The function **SetClock** adjusts the clock rate for the timer device to the given value.

Parameters:

int	*mode*	is the desired clock rate. See the table for the adjustable values. The variable mode may be used or the string constant from the second column

mode	constant	clock rate
0	Clk8MHz	8MHz
1	Clk4MHz	4MHz
2	Clk2MHz	2MHz
3	Clk1MHz	1MHz
4	Clk500kHz	500kHz
5	Clk250kHz	250kHz
6	Clk125kHz	125kHz
7	Clk62kHz	62,5kHz

Return value:

none

Name:

GetClock

double **GetClock**(void);

Class: DAC98

Description:

The function **GetClock** returns the adjusted clock rate of the timer device.

Parameters:

none

Return value:

double clock rate of the timer device.

Name:

WriteDigital

void **WriteDigital**(int *channel*, int *value*);

Class: DAC98

Description:

The function **WriteDigital** resets the state of the output channel *channel* when the parameter *value* = 0 otherwise the state is set to 1.

Parameters:

in *channel* is the number of the digital output channel

Channel	Assignment
0	digital output 0
1	digital output 1
2	digital output 2
3	digital output 3
4	digital output 4
5	digital output 5
6	digital output 6
7	digital output 7
8	gate of the 32 bit timer (output 8)
9	gate of the 16 bit timer (output 9)
10	not used (output 10)
11	not used (output 11)

int *value* is the new state of the digital output channel.

Return value:

none

Name:

WriteAllDigital

void **WriteAllDigital**(int *value*);

Class: DAC98

Description:

The function **WriteAllDigital** adjusts the state of the 12 digital output channels according to the lower 12 bits of the parameter *value* .

Parameters:

int	*value*	state of the 12 output ports.

Data bit	Assignment
D0	digital output 0
D1	digital output 1
D2	digital output 2
D3	digital output 3
D4	digital output 4
D5	digital output 5
D6	digital output 6
D7	digital output 7
D8	set the gate of the 32 bit timer (output 8)
D9	set the gate of the 16 bit timer (output 9)
D10	no function (output 10)
D11	no function (output 11)

Return value:

none

Name:

ReadDigital

int **ReadDigital**(int *channel*);

Class: DAC98

Description:

The function **ReadDigital** returns the state (0 or 1) of the digital input channel *channel*.

Parameters:

int	*channel* is the number of the digital input channel.

Channel	Assignment
0	digital input 0
1	digital input 1
2	digital input 2
3	digital input 3
4	digital input 4
5	digital input 5
6	digital input 6
7	digital input 7
8	busy signal of the AD converter
9	not used
10	not used
11	not used

Return value:

int	state of the digital input channel.

Name:

ReadAllDigital

int **ReadDigital**(void);

Class: DAC98

Description:

The function **ReadAllDigital** returns the state of the 12 digital input channels in the lower 12 bits of the return value.

Parameters:

none

Return value:

int	state of the 12 digital input channels.

Data bit	Assignment
D0	digital input 0
D1	digital input 1
D2	digital input 2
D3	digital input 3
D4	digital input 4
D5	digital input 5
D6	digital input 6
D7	digital input 7
D8	busy signal of the AD converter
D9	not used
D10	not used
D11	not used

Name:

SetCounter

void **SetCounter**(unsigned int *count*);

Class: DAC98

Description:

The function **SetCounter** adjusts the initial value of the 16 bit counter to the given parameter *count*.

Parameters:

unsigned int	*count*	is the new initial value of the counter (will be decremented).

Return value:

none

Name:

GetCounter

unsigned int **GetCounter**(void);

Class: DAC98

Description:

The function **GetCounter** returns the current content of the 16 bit counter.

Parameters:

none

Return value:

unsigned int	is the counter content.

Name:

WaitCounter

int **WaitCounter**(double *time*);

Class: DAC98

Description:

The function **WaitCounter** provides a precise delay time by counting the internal timer clock. The function returns 1 only when the counter counts the internal timer clock signal otherwise it returns 0.

Attention: The delay time is correct only in case the counter is configured to count the (internal) timer clock.

Parameters:

unsigned long *time*	delay time in milli seconds.

Return value:

int	Result = 1, when the counter counts the internal timer clock signal otherwise it returns 0.

Name:

TestCounterJMP

int **TestCounterJMP**(void);

Class: DAC98

Description:

The function **TestCounterJMP** checks whether the counter is configured for counting internal or external events.

Parameters:

none

Return value:

int	Result = 1 indicates that the counter is connected to the internal timer clock, result = 0 means that the counter counts external events or the jumper is missing.

Name:

GateCounter

void **GateCounter**(int *val*);

Class: DAC98

Description:

The function **GateCounter** enables or disables the gate of the 16 bit counter. The counter is started with *val*=1.

Parameters:

int	*val*	is the new value for the counter gate.

Return value:

none

Name:

SetTimer

void **SetTimer**(unsigned long *time*);

Class: DAC98

Description:

The function **SetTimer** adjusts the initial value for the 32 bit counter (square wave operating mode). The upper 16 bit are written into the upper cascade of the timer register and the lower 16 bit are written into the lower cascade. The resulting period time is given by the product of the upper and lower cascade settings multiplied with the period time of the clock signal (default 1/2000000 s).

Parameters:

unsigned long *time*	is the new time value for the timer cascade.

Return value:

none

Name:

SetTimer

void **SetTimer**(double *time*);

Class: DAC98

Description:

The function **SetTimer** adjusts the initial value for the 32 bit counter (square wave operating mode). The parameter *time* is taken as a period time in milli seconds.

Parameters:

unsigned long *time*	timer value in milli seconds.

Return value:

none

Name:

GetTimer

unsigned long **GetTimer**(void);

Class: DAC98

Description:

The function **GetTimer** returns the current content of the 32 bit timer.

Parameters:

none

Return value:

unsigned long is the timer content.

Name:

GateTimer

void **GateTimer**(int *val*);

Class: DAC98

Description:

The function **GateTimer** enables or disables the gate of the 32 bit timer. The timer is started with *val*=1.

Parameters:

int	*val*	is the new value for the timer gate.

Return value:

none

Name:

SetINT

void **SetINT**(int *channel*, int *val*);

Class: DAC98

Description:

The function **SetINT** enables the interrupt channel determined by *channel* when the parameter *val* is unequal to zero. In the other case the interrupt channel is disabled.

Parameters:

int	*channel*	is the DAC98 interrupt channel. Valid channels are:
		0, 32 bit timer overflow
		1, 16 bit counter overflow
		2, end of conversion AD converter
int	*val*	is the interrupt enable flag.

Return value:

none

Name:

GetINT

int **GetINT**(void);

Class: DAC98

Description:

The function **GetINT** returns the state of the interrupt channel register. Any data bit reset to 0 indicates the source which requested the interrupt from the card previously.

Parameters:

none

Return value:

int state of the interrupt channel:

Data bit	Assignment
D0	32 bit timer overflow
D1	16 bit counter overflow
D2	end of conversion AD converter
D3	no function

Name:

ResetDDM

void **ResetDDM**(int *channel*);

Class: DAC98

Description:

The function **ResetDDM** resets a single DDM device indicated by the parameter *channel*.

Parameters:

int	*channel*	is the DDM device number.

Return value:

none

Name:

ResetAllDDM

void **ResetAllDDM**(void);

Class: DAC98

Description:

The function **ResetAllDDM** resets all of the DDM devices at once.

Parameters:

none

Return value:

none

Name:

ReadDDM

unsigned int **ReadDDM**(int *channel*);

Class: DAC98

Description:

The function **ReadDDM** reads the DDM device specified by the parameter *channel* and stores the results to the corresponding public data elements. The increment counter content is returned directly.

Parameters:

int	*channel*	is the DDM device number.

Return value:

unsigned int	is the 16 bit increment counter content of the DDM device.

Name:

ReadAllDDM

void **ReadAllDDM**(void);

Class: DAC98

Description:

The function **ReadAllDDM** reads all of the three DDM devices and stores the results to the corresponding public data elements. Before the read operation all the registers of the DDM devices are switched at the same time such that the results belong to the same time.

Parameters:

none

Return value:

none

Name:

ReadAnalogInt

int **ReadAnalogInt**(int *channel, int mode*=3);

Class: DAC98

Description:

The function **ReadAnalogInt** reads the analog input channel specified by *channel* and returns the corresponding integer value with respect to the input signal range given by *mode*.

Parameters:

int *channel*	is the number of the analog input channel.
int *mode*	is the mode defining the input signal range according to:
	0, 0..5V
	1,0..10V
	2, -5..+5V
	3, -10..+10V

Return value:

int	converted analog input value.

Name:

ReadAnalogVolt

float **ReadAnalogVolt**(int *channel*, int mode);

Class: DAC98

Description:

The function **ReadAnalogVolt** reads the analog input channel specified by *channel* and returns the corresponding voltage value with respect to the input signal range given by *mode*.

Parameters:

int *channel*	is the number of the analog input channel.
int *mode*	is the mode defining the input signal range according to:
	0, 0..5V
	1,0..10V
	2, -5..+5V
	3, -10..+10V

Return value:

float	converted analog input value in Volt.

Name:

WriteAnalogInt

void **WriteAnalogInt**(int *channel*, int *value*);

Class: DAC98

Description:

The function **WriteAnalogInt** sends an analog value to the desired channel (0 or 1). The parameter *value* has to be in a range from 0 to +4095.

Parameters:

int	*channel*	is the number of the analog output channel.
int	*value*	is the output value.

Return value:

none

Name:

WriteAnalogVolt

void **WriteAnalogVolt**(int *channel*, float *value*);

Class: DAC98

Description:

The function **WriteAnalogVolt** operates similar to **WriteAnalogInt**, but the output value is taken as a voltage. Its value has to be in the range from -10(V) to +10(V).

Parameters:

int	*channel*	is the number of the analog output channel.
int	*value*	is the output value in Volt.

Return value:

none

Private Element Function:

Name:

ReadDigitalInputs

unsigned int **ReadDigitalInputs**(void);

Class: DAC98

Description:

The function **ReadDigital** returns the state of the 12 digital input channels (8 external inputs + 4 internal states) in the lower 12 bits of the return value (similar to **ReadAllDigital** but with unsigned return value).

Parameters:

none

Return value:

unsigned int state of the 12 digital input channels.

Data bit	Assignment
D0	digital input 0
D1	digital input 1
D2	digital input 2
D3	digital input 3
D4	digital input 4
D5	digital input 5
D6	digital input 6
D7	digital input 7
D8	busy signal of the AD converter
D9	not used
D10	not used
D11	not used

4.2 The Class DIC

The class DIC is established to use existing software, written for the DIC24 PC plug-in card, now with the DAC98 PC plug-in card. That means this DIC class together with the DAC98 class replaces the "old" DIC class for the DIC24 PC plug-in card.

Since this DIC class only provides a subset of the features of the DAC98 it is strictly recommended to use only the DAC98 class for new projects.

This DIC class only contains those functions as an interface to "old DIC function calls" which are not provided directly by the DAC98 class.

Basic Class:

DAC98

Public Data:

unsigned char	ddm_status[4]	is an array containing the state register of the three DDM devices.
unsigned int	ddm_counter[4]	is an array containing the counted increments of the three DDM devices.
unsigned long	ddm_timer[4]	is an array containing the timer values of the three DDM devices.
int	aout0, aout1	are the values for the analog outputs
private:		
int	ident	is the state of the identification

Name:

DIC

DIC(int *adress*);

Class: DIC

Description:

The constructor only requires the base address of the card. The field *ddm_counter(3)* is reset to 0 because the DAC98 contains only 3 DDM devices instead of 4 on the DIC24.

Parameters:

int	*adress*	is the base address of the adapter card in the address range of the PC.

Return value:

none

Name:

~DIC

~DIC();

Class: DIC

Description:

The destructor requires no parameters.

Parameters:

none

Return value:

none

Name:

GetDigitalOut

unsigned int **GetDigitalOut**(void);

Class: DIC

Description:

The function **GetDigitalOut** returns the content of the shadow register for the digital outputs.

Parameters:

none

Return value:

int	state of the digital outputs.

Name:

GetAnalogOut

int **GetAnalogOut** (int *channel*);

Class: DIC

Description:

The function **GetAnalogOut** returns the integer value previously transfered to the specified analog channel.

Parameters:

int	*channel*	is the analog channel number.

Return value:

int	the 12 bit value of the previous analog output.

Name:

GetDDMCounter

unsigned int **GetDDMCounter**(int *channel*);

Class: DIC

Description:

The function **GetDDMCounter** returns the last counter content (global variable) of the DDM component specified by the given channel number, which was read by the functions **ReadDDM** or **ReadAllDDM**. For *channel* = 3 the return value is always = 0.

Parameters:

int	*channel*	is the DDM device number (0, 1, 2).

Return value:

unsigned int	is the 16 bit increment counter content of the DDM device.

Name:

GetDDMTimer

unsigned long **GetDDMTimer**(int *channel*);

Class: DIC

Description:

The function **GetDDMTimer** returns a timer value of 0 for any channel because this function is not implemented in the new DDM device. This dummy function is established only for compatibility reason.

Parameters:

int	*channel*	is the number of the DDM device.

Return value:

unsigned long	here always = 0.

Name:

GetDDMStatus

unsigned char **GetDDMStatus**(int *channel*);

Class: DIC

Description:

The function **GetDDMStatus** returns a state value of 0 for any channel because this function is not implemented in the new DDM device. This dummy function is established only for compatibility reason.

Parameters:

int	*channel*	is the number of the DDM device.

Return value:

unsigned char	here always = 0.

Name:

FilterINC

void **FilterINC**(int *channel*, int *val*);

Class: DIC

Description:

The function **FilterINC** is an empty function. This dummy function is established only for compatibility reason.

Parameters:

int	*channel*	is the number of the DDM device.
int	val	is the desired filter state (0==on, 1 == off)

Return value:

none

Name:

TimerDirINC

void **TimerDirINC**(int *channel*, int *val*);

Class: DIC

Description:

The function **TimerDirINC** is an empty function. This dummy function is established only for compatibility reason.

Parameters:

int	*channel*	is the number of the DDM device.
int	val	is the desired count direction state (0==increment, 1 == decrement)

Return value:

none

Name:

SetINT

void **SetINT**(int *channel*, int *val*);

Class: DIC

Description:

The function **SetINT** adjusts the interrupt enable register of the DAC98.

Parameters:

int	*channel*	is the interrupt channel. Valid channels are:
		4, 32 bit timer overflow
		5, 16 bit counter overflow
int	*val*	is the interrupt enable flag.

Return value:

none

Name:

SetTimer

void **SetTimer** (unsigned long *time*);

Class: DIC

Description:

The function **SetTimer** adjusts the initial value for the 32 bit counter (square wave operating mode). The upper 16 bit are written into the upper cascade of the timer register and the lower 16 bit are written into the lower cascade. The resulting period time is given by the product of the upper and lower cascade settings multiplied with the period time of the clock signal (default 1/2000000 s).

The identification procedure of the "old" DIC24 is simulated in addition.

Parameters:

unsigned long *time*	is the new time value for the timer cascade.

Return value:

none

Name:

GetTimer

unsigned long **GetTimer**(void);

Class: DIC

Description:

The function **GetTimer** returns the current content of the 32 bit timer.

The identification procedure of the "old" DIC24 is simulated in addition.

Parameters:

none

Return value:

unsigned long is the timer content.

Name:

GateTimer

void **GateTimer**(int *val*);

Class: DIC

Description:

The function **GateTimer** enables or disables the gate of the 32 bit timer. The timer is started with *val*=1.

Parameters:

int	*val*	is the new value for the timer gate.

Return value:

none

4.3 The Class PCIO

The class PCIO is established to use existing software, written for the DAC6214 PC plug-in card, now with the DAC98 PC plug-in card. That means this PCIO class together with the DAC98 class replaces the "old" PCIO class for the DAC6214 PC plug-in card.

Since this PCIO class only provides a subset of the features of the DAC98 it is strictly recommended to use only the DAC98 class for new projects.

This PCIO class only contains those functions as an interface to "old PCIO function calls" which are not provided directly by the DAC98 class.

Basic Class:

DAC98

Public Data:

unsigned int	ddm_counter[1]	is an array containing the increment count of the first DDM device.

Name:

PCIO

PCIO(int *adress*)

Class: PCIO

Description:

The constructor only requires the base address of the card.

Parameters:

int	*adress*	is the base address of the adapter card in the address range of the PC.

Return value:

none

Name:

~PCIO

~**PCIO**();

Class: PCIO

Description:

The destructor requires no parameter.

Parameters:

none

Return value:

none

Name:

DigitalOutStatus

unsigned char **DigitalOutStatus**(void),

Class: PCIO

Description:

The function **DigitalOutStatus** returns the state of the digital outputs.

Parameters:

none

Return value:

unsigned char	the state of the digital outputs.

Name:

IsPCIO

int **IsPCIO**(void);

Class: PCIO

Description:

The function **IsPCIO** calls the function **Identifikation** of the class DAC98 to check whether the DAC98 responds to the base address of the driver software. Any value unequal to zero is returned when the PC plug-in card returns a bit string preassigned on its hardware.

Parameters:

none

Return value:

int	Result of the test. Values unequal to zero indicate that the hardware responded to the base address.

Name:

ReadAnalogVoltMean

float **ReadAnalogVoltMean**(int *channel*, int *repeat*);

Class: PCIO

Description:

The function **ReadAnalogVoltMean** reads the analog input specified by *channel* *repeat* times and returns the mean value as a voltage.

Parameters:

int	*channel* is the analog input channel.
int	*repeat* is the number of read operations.

Return value:

float	mean value of analog input in Volt.

Name:

ResetHCTL

void **ResetHCTL**(void);

Class: PCIO

Description:

The function **ResetHCTL** calls the function **ResetDDM** to reset the first DDM device.

Parameters:

none

Return value:

none

Name:

ReadHCTL

int **ReadHCTL**(void);

Class: PCIO

Description:

The function **ReadHCTL** calls the function **ReadDDM** and returns the content of the increment counter of the first DDM device.

Parameters:

none

Return value:

int	increment counter content.

Name:

SetINT

void **SetINT**(int val);

Class: PCIO

Description:

The function **SetINT** adjusts the interrupt enable register of the DAC98.

Parameters:

int	*val*	is the interrupt enable flag.

Return value:

none

5 Windows Drivers for DAC98, DAC6214 and DIC24

The drivers are installable 16-Bit drivers applicable to 16- or 32-Bit programs with Windows 3.1 / 95 / 98. To exchange data with the drivers the following three 16-Bit API functions are used:

OpenDriver

HDRVR *hDriver* = **OpenDriver**(*szDriverName*, NULL, NULL)

Parameters *szDriverName* is the file name of the driver, valid names are "DAC98.DRV", "DAC6214.DRV" and "DIC24.DRV" (according to the PC adapter cards) possibly combined with complete path names.

Description The function **OpenDriver** initializes the driver and returns a handle for following accesses to this driver. If this function is called the first time the driver is loaded into the memory. Any further calls return another handle of an existing driver. The driver handle is valid only when the return value is unequal to NULL. In case the return value is equal to NULL, the function **OpenDriver** failed meaning that further driver accesses by the functions **SendDriverMessage** or **CloseDriver** are invalid. The parameter *szDriverName* of the function **OpenDriver** contains the DOS file name of the driver. The file name may include the disk name as well as the complete path names according to the 8.3 name convention but it must not exceed 80 characters. When only a single file name is used, the drivers location is expected in the standard search path of Windows. The other parameters are meaningless and should be equal to NULL.

Return Valid driver handle or NULL.

SendDriverMessage

LRESULT *result* = **SendDriverMessage**(*hDriver*, *DRV_USER*, *PARAMETER1*, *PARAMETER2*)

Parameters *hDriver* is a handle of the card driver.

DRV_USER is the flag indicating special commands.

PARAMETER1 is a special command and determines the affected channel number (see table below).

PARAMETER2 is the output value for special write commands.

Description The function **SendDriverMessage** transfers a command to the driver specified by the handle *hDriver*. The drivers for the adapter cards from **amira** expect the value *DRV_USER* for the second parameter (further commands can be found in the API documentation of **SendDriverMessage**). The third parameter *PARAMETER1* is of type ULONG specifying the command which is to be carried-out. The lower 8 bits of this parameter determine the channel (number) which is to be affected by the given command. The commands are valid for all of the three drivers. But the valid channel numbers depend on the actual hardware. The last parameter *PARAMETER2* is of type ULONG and is used with write commands. It contains the output value. The return value depends on the command. Commands and channel names are defined in the file "IODRVCMD.H".

Return Is equal to 0 in case of unsupported commands or special write commands. Otherwise it contains the result of special read commands.

Table of the supported standard API commands		
Command	Return	Remark
DRV_LOAD	1	loads the standard base address from SYSTEM.INI
DRV_FREE	1	
DRV_OPEN	1	
DRV_CLOSE	1	
DRV_ENABLE	1	locks the memory range for this driver
DRV_DISABLE	1	unlocks the memory range for this driver
DRV_INSTALL	DRVCNF_OK	
DRV_REMOVE	0,	
DRV_QUERYCONFIGURE	1	
DRV_CONFIGURE	1	calls the dialog to adjust the base address
DRV_POWER	1	
DRV_EXITSESSION	0	
DRV_EXITAPPLICATION	0	

Table of the special commands with the flag DRV_USER:

PARAMETER1				Return
Command	Channel Number			
	DAC98	DAC6214	DIC24	
DRVCMD_INIT initializes the card and has to be the first command				0
DRVINFO_AREAD returns the number of analog inputs				8 for DAC98, 6 for DAC6214, 0 for DIC24
DRVINFO_AWRITE returns the number of analog outputs				2 for all cards
DRVINFO_DREAD returns the number of digital inputs				8 for DAC98, DIC24 4 for DAC6214
DRVINFO_DWRITE returns the number of digital outputs				8 for DAC98, DIC24 4 for DAC6214
DRVINFO_COUNT returns the number of counters and timers				5 for DAC98 1 for DAC6214 6 for DIC24
DRVCMD_AREAD reads an analog input	0-7	0-5	no inputs	16 bit value from -32768 to 32767 according to the input voltage range
DRVCMD_AWRITE writes to an analog output	0-1	0-1	0-1	0
DRVCMD_DREAD reads a single digital input or all inputs (ALL_CHANNELS)	0-7 or ALL_CHAN	0-3 or ALL_CHAN	0-7 or ALL_CHAN	state (0 or 1) of a single input or states binary coded (channel0==bit0)
DRVCMD_DWRITE writes to a single digital output or to all outputs (channel0==bit0)	0-7 or ALL_CHAN	0-3 or ALL_CHAN	0-7 or ALL_CHAN	0
DRVCMD_COUNT reads a counter / timer	DDM0 DDM1 DDM2 COUNTER TIMER	DDM0	DDM0 DDM1 DDM2 DDM3 COUNTER TIMER	counter- / timer-content as an unsigned 32-bit value
DRVCMD_RCOUNT resets a counter / timer (counter, timer to the value -1) or all DDM's (ALL_CHANNELS)	DDM0 DDM1 DDM2 COUNTER TIMER ALL_CHAN	DDM0	DDM0 DDM1 DDM2 DDM3 COUNTER TIMER ALL_CHAN	0
DRVCMD_SCOUNT presets a counter / timer to an initial value	COUNTER TIMER		COUNTER TIMER	0

CloseDriver

CloseDriver(*hDriver*, NULL, NULL)

Parameters *hDriver* is a handle of the card driver.

Description The function **CloseDriver** terminates the operation of the driver specified by the handle *hDriver*. The driver is removed from the memory when all of its handles are released by the function **CloseDriver**.

BW500 Windows Software V1.0

Printed: 23. November 1999

1 Source Files of the BW500W Controller Program

1.1 General

The program is a 16-bit application, which may be started only once by the operating systems Windows 3.1 or Windows95. The desktop is created by means of the program language 'Pascal', while the actual controller is realized by a DLL developed with the program language 'C++'. Both program parts are available in source completely. The program package is completed by
the TIMER16.DLL to handle the cyclic controller calls,
the card drivers DAC98.DRV or DIC24.DRV to access the PC adapter card,
the PLOT16.DLL for the graphic output,
the help file BW500W.HLP,
the run-time library BC450RTL.DLL.

Generating a new executable program is possible only by means of the development systems 'Delphi' version 1.0 for the desktop and 'Borland C++' version 4.52 for the controller-DLL. The last may be generated using another 16-bit-C++ compiler in case a suitable project file can be created.

Prior to generating the program the first time please copy the complete content of the enclosed floppy disk to a new directory of your harddisk by keeping the directory structure (i.e. using the 'Explorer' to copy to the new directory BW500).

You will then find the following subdirectories:

BW500DSK
CONTRDLL
EXE

Where BW500DSK contains the *Delphi Project File* BW500W.DPR together with all the accompanying Pascal source files to generate the desktop, CONTRDLL contains the *Borland Project File* BWSERV.IDE with all the accompanying C++ source files to create the controller-DLL (BWSERV16.DLL). Finally the subdirectory EXE contains all the additional files required by the executable program (BW500.HLP, BW500W16.INI, DEFAULT.STA, DEFAULT.FBW, WCONTROL.FUZ, WERROR.FUZ, XCONTROL.FUZ, XERROR.FUZ, DAC98.DRV, DIC24.DRV,
DUMMY.DRV, TIMER16.DLL, PLOT16.DLL, BC450RTL.DLL).

Attention: After creating a new desktop or a new controller-DLL, the new results are to be copied later to the subdirectory EXE.

A DEMO version of the program (simulation of the mathematical model of the ball and beam system instead of accessing the PC adapter card) may be obtained simply by setting the macro __SIMULATION__ in the include file BWDEFINE.H and generating a new BWSERV16.DLL. Because the resulting DLL has the same name as the DLL controlling the real system, it should be copied together with all the required files (DUMMY.DRV is recommended instead of DAC98.DRV and DIC24.DRV)to a different subdirectory (i.e. DEMO) afterwards.

1.2 Global Data and Functions

The file **BWDEFINE.H** contains some definitions to clarify the readability of the source code and to adjust the program mode as well as the fixed sampling period. When __SIMULATION__ is defined all program functions besides system calibration are available for a simulated ball and beam system. The PC adapter card is not required in this program mode. To control the real system the macro __SIMULATION__ must not be defined!

Used definitions:

```
#define __SIMULATION__
#define __FUZZY__
#define ScopeBufSize        8
#define SIMTIME             0.05
```

The file **BW500DAT.H** contains global data structures which are used in different instances of the software. These structures are saved in the data files used to store measurements.

```
// Data structures:
struct PROJECT{
        char    number[10];         // P342
        char    name[10];           // BW500
        char    Titel[10];          // BW500 - Monitor
        char    Version[10];        // 1.00
        char    Date[10];           // 22.10.99
        char    Dummy[10];          // Reserve
};
static struct PROJECT PRJ = {
        "P342",
        "BW500",
        "BW500",
        "V2.00",
        "22-Oct-99",
        "res"
};
CTRLSTATUS {
        short controller;           // type of active controller
        double ta_ms;               // adjusted sampling period in [ms]
        char fuzname[80];           // "Fuzzy controller" file name
        short dummy;
        long timeofmeasure;         // Date and time of the measurement acquisition
};
```

```
struct DATASTRUCT {      // Structure to reconstruct the measured data
        short   nchannel;     // Length of the stored measurement vectors (number of channels)
        short   nvalues;      // Number of the measurement vectors (number of samples)
        float   deltatime;    // Time between two samples
};
```

The Format of the Documentation Data File *.PLD

Measured data stored in a data file are reloadable and may be output in a graphic representation. In addition the system settings (CTRLSTATUS) which were active during the start of the data acquisition are stored in this file. They are displayable in a separate window.

The data file contains data in binary format stored in the following order:

The structure PROJEKT PRJ.
The structure CTRLSTATUS.
The structure DATASTRUCT.
The data array with float values (4 bytes per value)

The size of the data array is defined in the structure DATASTRUKT. With the BW500 the number of the stored channels is always 8 (the length of the measurement vector is 8, i.e. equal to 32 bytes). The vector contains the following signals:

the position setpoint of the ball	in m,
the measured position of the ball	in m,
the angle of the beam	in rad
the control force	in N
the velocity of the ball	in m/s,
the angle velocity of the beam	in rad/s,
the friction compensation of the beam	in N,
the friction compensation of the ball	in rad.

The number of the stored measurement acquisitions (vectors) depends on the adjusted values for the sampling period and the measuring time. The maximum number of measurings is 1024. The time distance between two successive acquisitions is an integral multiple of the sampling period used by the controller.

1.3 Dialogs and Windows of the Desktop

The programs desktop is written in the program language Pascal. The main window with its menu bar as well as all of the following dialogs and message boxes are realized by the following files.

The file MAIN.PAS contains the procedures:

ShowHint(*Sender*: TObject)

FormCreate(*Sender*: TObject)

FormShow(*Sender*: TObject)

FormClose(*Sender*: TObject; var Action: TCloseAction)

FormDestroy(*Sender*: TObject)

FileMenuClick(*Sender*: TObject)

OpenStateClick(*Sender*: TObject)

SaveStateClick(*Sender*: TObject)

SaveStateasClick(*Sender*: TObject)

OpenFuzzyClick(*Sender*: TObject)

SaveFuzzyasClick(*Sender*: TObject)

LoadPlotData1Click(*Sender*: TObject)

SavePlot1Click(*Sender*: TObject)

Print1Click(*Sender*: TObject)

PrintSetup1Click(*Sender*: TObject)

ExitItemClick(*Sender*: TObject)

IOInterface1Click(*Sender*: TObject)

DAC98Click(*Sender*: TObject)

DIC24Click(*Sender*: TObject)

DACSetupClick(*Sender*: TObject)

Edit1Click(*Sender*: TObject)

StateControllerSetupClick(*Sender*: TObject)

FuzzyControllerSetupClick(*Sender*: TObject)

Run1Click(*Sender*: TObject)

StateController1Click(*Sender*: TObject)

FuzzyController2Click(*Sender*: TObject)

StopController1Click(*Sender*: TObject)

CalibrateSensors1Click(*Sender*: TObject)

StartMeasuring1Click(*Sender*: TObject)

SetpointGenerator1Click(*Sender*: TObject)

View1Click(*Sender*: TObject)

PlotMeasuredData1Click(*Sender*: TObject)

PlotFileData1Click(*Sender*: TObject)

ParametersfromPLDFile1Click(*Sender*: TObject)

Fuzzy3D1Click(*Sender*: TObject)

Timing1Click(*Sender*: TObject)

Contents1Click(*Sender*: TObject)

SearchforHelpon1Click(*Sender*: TObject)

HowtoUseHelp1Click(*Sender*: TObject)

About1Click(*Sender*: TObject)

Timer1Timer(*Sender*: TObject)

PaintBox1Click(*Sender*: TObject)

MeasLabelClick(*Sender*: TObject)

The file SINGLEIN.PAS contains the procedure:
WndProc(*var Msg*: TMessage)

The file ABOUT.PAS contains the procedure:
FormShow(*Sender*: TObject)

The file CALIB.PAS contains the procedures:
CenterBBtnClick(*Sender*: TObject)

LeftBBtnClick(*Sender*: TObject)

RightBBtnClick(*Sender*: TObject)

FormShow(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file FUZ3D.PAS contains the procedures:
rescale

recalc

calcrot

calctrans(*ix,iy,iz* : double; var *ox,oy,oz* : double)

FormShow(*Sender*: TObject)

FormHide(*Sender*: TObject)

FuzzyCBoxChange(*Sender*: TObject)

DrawSquare(*can* : TCanvas; *j, i, z0, z1, z2, z3* : Integer)

DrawCoors(*can* : TCanvas)

DrawCoors2(*can* : TCanvas)

DrawMark(*can* : TCanvas)

PaintBoxPaint(*Sender*: TObject)

ScrollBar1Change(*Sender*: TObject)

ScrollBar2Change(*Sender*: TObject)

PaintBoxMouseDown(*Sender*: TObject; *Button*: TMouseButton; *Shift*: TShiftState; *X, Y*: Integer);

PaintBoxMouseMove(*Sender*: TObject; *Shift*: TShiftState; *X, Y*: Integer);

PaintBoxMouseUp(*Sender*: TObject; *Button*: TMouseButton; *Shift*: TShiftState; *X, Y*: Integer);

KoorsCBoxClick(*Sender*: TObject)

Koors2CBoxClick(*Sender*: TObject)

ColorCBoxClick(*Sender*: TObject)

LowResCBoxClick(*Sender*: TObject)

PrintBBtnClick(*Sender*: TObject)

MarkCBoxClick(*Sender*: TObject)

Timer1Timer(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file FUZZYPAR.FUZ contains the procedures:

Big

Small

FormCreate(Sender: TObject)

FormDestroy(Sender: TObject)

FormShow(Sender: TObject)

Sel1BBtnClick(Sender: TObject)

Ed1BBtnClick(Sender: TObject)

CancelEdBBtnClick(Sender: TObject)

SaveEdBBtnClick(Sender: TObject)

HelpBtnClick(Sender: TObject)

The file MEASURE.PAS contains the procedures:

OKBtnClick(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file PLDINFO.PAS contains the procedures:

FormShow(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file PLOT.PAS contains the procedures:

OKBtnClick(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file PRINTPLT.PAS contains the procedures:

PrinterBitBtnClick(*Sender*: TObject)

OKBtnClick(*Sender*: TObject)

FormShow(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file SETPOINT.PAS contains the procedures:

OKBtnClick(*Sender*: TObject)

FormShow(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file STATEPAR.PAS contains the procedures:

FormCreate(Sender: TObject)

TabSetClick(Sender: TObject)

FormShow(Sender: TObject)

OKBtnClick(Sender: TObject)

HelpBtnClick(Sender: TObject)

The file STFUZ.PAS contains the procedures:

OKBtnClick(Sender: TObject)

HelpBtnClick(Sender: TObject)

The file STSTATE.PAS contains the procedures:

OKBtnClick(Sender: TObject)

HelpBtnClick(Sender: TObject)

The file TIMING.PAS contains the procedures:

UpdateData

Button1Click(Sender: TObject)

FormShow(Sender: TObject)

Button2Click(Sender: TObject)

Button3Click(Sender: TObject)

OnClickBtnSampleCalc(Sender: TObject)

The file TOOLS.PAS contains the functions:

FloatToStr2(*f* : Single) : string

FloatToStr3(*f* : Single) : string

FloatToStr4(*f* : Single) : string

StrToFloatMinMax(*s* : string; *min,max* : double) : double

StrToFloatStrMinMax(*s* : string; *var val* : double; *min,max* : double) : string

MinMaxi(*val, min, max* : Integer) : Integer

DetectNT : Boolean

The file DLLS.PAS contains besides the global data definitions the interface definitions for the DLL's BWSERV16 and TIMER16.

Global Data:

```
type ServiceParameter  = record
        controller :          WORD;
        stateobserver :       WORD;
        stateError :          WORD;
        fuzzyobserver :       WORD;
        fuzzyError :          WORD;
        dummy1 :              WORD;
        dummy2 :              WORD;
        spshape :             WORD;
        ft   :                array[0..3] of double;
        filter :              double;
        lbd  :                array[0..3] of double;
        abd  :                array[0..3] of double;
        fbd  :                array[0..3] of double;
        bbd  :                array[0..1] of double;
        dcon :                double;
        dabd :                double;
        dbbd :                double;
        dfbd :                array[0..3] of double;
        dlbd :                array[0..3] of double;
        name :                array[0..79] of char;
        spoffset, spamplitude, spperiode : double;
 end;
```

```
type ServiceData = record
        setpoint :                  double;
        pos, dpos, angle, dangle : double;
        out :                       double;
        fuzhelp :                   double;
        state :                     WORD;
        dummy :                     array[0..2] of WORD;
end;

type    Fuzzy3DInfo = record
        size :                      longint;
        idx, idy, idz :             Integer;
        incount, outcount :         Integer;
        xname :                     array[0..79] of char;
        yname :                     array[0..79] of char;
        zname :                     array[0..79] of char;
        xmin, xmax :                double;
        ymin, ymax :                double;
        zmin, zmax :                double;
end;


        param :                     ServiceParameter ;
        data :                      ServiceData;
```

TMainForm.ShowHint

ShowHint(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **ShowHint** is called, when the object of type TMainForm appears at the screen.

TMainForm.FormCreate

FormCreate(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormCreate** creates an instance of an object of type TMainForm. With this another instance (*single*) of type SingleInstance is created, which guarantees that this application (the BW500W program) may be started only once.

TMainForm.FormShow

FormShow(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TMainForm appears at the screen. At the same time the application BW500W is started with the selection of the controller DLL, here BWSERV16.DLL, and the driver for the adapter card (the drivers file name is contained in the file BW500W16.INI). The check marks below the menu item "IO-Interface" are set accordingly. The parameter structure *param* is read from the BWSER16.DLL. The variable *WithFuzzy* is set according to the existence of a fuzzy controller. Then the structure is initialized with default values (no controller, position setpoint to 0, period to 20s) and transferred (**SetParameter**) again to the controller inside the BWSERV16.DLL. The program version DEMO is detected, when the function **IsDemo** returns TRUE. Only in this case the menu items "IO-Interface" and "Calibrate Sensors" are disabled and the string "(Demo-Version)" is appended to the title line of the monitor window. The timers for the sampling period of the controller (**StartTimer**, TIMER16.DLL) as well as for the periodic update of the monitor window are started. The sensor inputs are calibrated automatically (**CalibrateSensors1Click**) in case of a real system.

TMainForm.FormClose

FormClose(*Sender*: TObject; var Action: TCloseAction)

Parameters: *Sender* is a reference to the calling object.

var Action is not used.

Description The procedure **FormClose** stops the timer for the sampling period of the controller.

TMainForm.FormDestroy

FormDestroy(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormDestroy** is called before the object of type TMainForm is removed from the memory. In this connection the animation picture as well as the application are released.

TMainForm.FileMenuClick

FileMenuClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FileMenuClick** is an event handler activated by clicking once on the menu item "File". The menu item "Save Recorded Data" is enabled when the memory contains data from a measurement acquisition. Otherwise this menu item is disabled.

TMainForm.OpenStateClick

OpenStateClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OpenStateClick** is an event handler activated by clicking once on the menu item "File/Load State Controller". A Windows system dialog appears allowing for the selection of a file name (extension *.STA) from which all the matrices of a state controller are to be loaded (**ReadStatePar**). An error message appears when a non-existing file is selected.

TMainForm.SaveStateClick

SaveStateClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **SaveStateClick** is an event handler activated by clicking once on the menu item "File/Save State Controller". All the matrices of the state controller contained in the memory are written to that file, from which the same matrices have been read previously (**OpenStateClick**). After starting the program this file is DEFAULT.STA. An error message will appear, when the data could not be written successfully to this file.

TMainForm.SaveStateasClick

SaveStateasClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **SaveStateasClick** is an event handler activated by clicking once on the menu item "File/Save State Controller as...". A Windows system dialog appears to select a new file name (extension *.STA) for storing (**WriteStatePar**) all the matrices of the state controller contained in the memory. An error message will appear, when the data could not be written successfully to this file.

TMainForm.OpenFuzzyClick

OpenFuzzyClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OpenFuzzyClick** is an event handler activated by clicking once on the menu item "File/Load Fuzzy Controller". A Windows system dialog appears a file name (extension *.FBW). Such a file is expected to be a "fuzzy-controller" file containing four other file names of fuzzy description files from which all the fuzzy variables and rules are to be read (**ReadFuzzy**). The selected "fuzzy-controller" file name is written to the parameter structure *param*. Selecting a file name of a file, which does not exist, will result in an error message.

TMainForm.SaveFuzzyasClick

SaveFuzzyasClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **SaveFuzzyasClick** is an event handler activated by clicking once on the menu item "File/Save Fuzzy Controller as...". A Windows system dialog appears allowing for the selection of a file name (extension *.FBW). The file names of the current four fuzzy description files are to be written to the selected file. These names are read from the current "fuzzy-controller" file (name is taken from *param*) and written to the selected "fuzzy-controller" file. The new "fuzzy-controller" file name is stored in *param*.

TMainForm.LoadPlotData1Click

LoadPlotData1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **LoadPlotData1Click** is an event handler activated by clicking once on the menu item "File/Load Recorded Data". A Windows system dialog appears allowing for the selection of a file name of a so-called documentation file (extension *.PLD), from which measured data are to be read (**ReadPlot**). Selecting a file, which does not exist will result in an error message.

TMainForm.SavePlot1Click

SavePlot1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **SavePlot1Click** is an event handler activated by clicking once on the menu item "File/Save Recorded Data". A Windows system dialog appears allowing for the selection of a file name of a so-called documentation file (extension *.PLD), to which measured data contained in the memory are to be written (**WritePlot**). An error message will appear, when the data could not be written successfully to the selected file.

TMainForm.Print1Click

Print1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Print1Click** is an event handler activated by clicking once on the menu item "File/Print". A modal dialog (**PrintPlotDlg**) appears to select previously created plot windows, which are to be printed to an output device (i.e. printer).

TMainForm.PrintSetup1Click

PrintSetup1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **PrintSetup1Click** is an event handler activated by clicking once on the menu item "File/Print Setup". The standard printer setup dialog of Windows is called to select and adjust the output device.

TMainForm.ExitItemClick

ExitItemClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **ExitItemClick** is an event handler activated by clicking once on the menu item "File/Exit" or by pressing Shift+F4. The current application, the program BW500W will be terminated.

TMainForm.IOInterface1Click

IOInterface1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **IOInterface1Click** is an event handler activated by clicking once on the menu item "IO-Interface". The following two menu items to select an adapter card driver for the DAC98 or DIC24 are enabled only, when the corresponding driver exists in the current directory. The menu item to select the dialog for adjusting the adapter card address is enabled only, when one of the above drivers is marked.

TMainForm.DAC98Click

DAC98Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **DAC98Click** is an event handler activated by clicking once on the menu item "IO-Interface/DAC98". The name of the driver file DAC98.DRV is written to the file BW500W16.INI. After stopping the timer (**StopTimer**) controlling the sampling period of the controller the driver DAC98.DRV is selected for the BWSERV16.DLL (**SelectDriver**). Error messages will appear, when stopping the timer or selecting the driver failed. The check marks of the corresponding menu item are set accordingly and the dialog **DACSetupClick** to adjust the adapter card address is called automatically.

TMainForm.DIC24Click

DIC24Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **DIC24Click** is an event handler activated by clicking once on the menu item "IO-Interface/DAC98". The name of the driver file DIC24.DRV is written to the file BW500W16.INI. After stopping the timer (**StopTimer**) controlling the sampling period of the controller the driver DIC24.DRV is selected for the BWSERV16.DLL (**SelectDriver**). Error messages will appear, when stopping the timer or selecting the driver failed. The check marks of the corresponding menu item are set accordingly and the dialog **DACSetupClick** to adjust the adapter card address is called automatically.

TMainForm.DACSetupClick

DACSetupClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **DACSetupClick** is an event handler activated by clicking once on the menu item "IO-Interface/Setup". After stopping the timer (**StopTimer**) controlling the sampling period of the controller the dialog **SetupDriver** of the TIMER16.DLL appears to adjust the adapter card address of the corresponding driver. Error messages will be presented, when stopping the timer failed or when the dialog could not adjust the address correctly. Terminating this dialog will start the sensor calibration dialog (**CalibrateSensors1Click**) automatically, which also restarts the timer.

TMainForm.Edit1Click

Edit1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Edit1Click** is an event handler activated by clicking once on the menu item "Edit". The following menu item "Edit/Fuzzy Controller Parameter" is enabled only, when the internal flag *WithFuzzy* is set.

TMainForm.StateControllerSetupClick

StateControllerSetupClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **StateControllerSetupClick** is an event handler activated by clicking once on the menu item "Edit/State Controller Parameter". The modal dialog (**StateParameterDlg**) will appear to display and adjust all the parameters of the state controller.

TMainForm.FuzzyControllerSetupClick

FuzzyControllerSetupClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FuzzyControllerSetupClick** is an event handler activated by clicking once on the menu item "Edit/Fuzzy Controller Parameters". The modal dialog (**FuzzyParameterDlg**) will appear to display and adjust all the parameters of the fuzzy controller.

TMainForm.Run1Click

Run1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Run1Click** is an event handler activated by clicking once on the menu item "Run". As long as no controller is selected the menu items to select the state controller and the fuzzy controller are enabled and the menu items to stop a controller and to adjust the setpoint generator are disabled. The last two menu items are enabled when any controller is active. The menu items to select a controller will remain disabled when the last sensor calibration failed.

TMainForm.StateController1Click

StateController1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **StateController1Click** is an event handler activated by clicking once on the menu item "Run/State Controller" or by pressing "F2". When the following dialog **StartStateDlg** is terminated with the "Ok" button, the check mark for the selected state controller and the corresponding parameter of the structure *param* are set. This structure is then transferred (**SetParameter**) to the BWSERV16.DLL.

TMainForm.FuzzyController2Click

FuzzyController2Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FuzzyController2Click** is an event handler activated by clicking once on the menu item "Run/Fuzzy Controller" or by pressing "F3". When the following dialog **StartFuzzDlg** is terminated with the "Ok" button, the check mark for the selected fuzzy controller and the corresponding parameter of the structure *param* are set. This structure is then transferred (**SetParameter**) to the BWSERV16.DLL.

TMainForm.StopController1Click

StopController1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **StopController1Click** is an event handler activated by clicking once on menu item "Run/Stop Controller" or by pressing "F4". The check marks as well as the corresponding controller flags of the structure *param* are reset. This structure is then transferred (**SetParameter**) to the BWSERV16.DLL.

TMainForm.CalibrateSensors1Click

CalibrateSensors1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **CalibrateSensors1Click** is an event handler activated by clicking once on the menu item "Run/Calibrate Sensors". When, after stopping (**StopController1Click**) the controller, the following modal dialog **CallibrateDlg** is terminated with the "Ok" button, the state of the sensor calibration is taken as successful and the corresponding check mark is set. The timer controlling the sampling period of the controller is restarted if it is not still running. If restarting the timer failed a corresponding error message will appear.

TMainForm.StartMeasuring1Click

StartMeasuring1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **StartMeasuring1Click** is an event handler activated by clicking once on the menu item "Run/Start Measuring" or by pressing "F5". The conditions of a measurement acquisition are adjusted by means of the following dialog **MeasureDlg**.

TMainForm.SetpointGenerator1Click

SetpointGenerator1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **SetpointGenerator1Click** is an event handler activated by clicking once on the menu item "Run/Start Measuring" or by pressing "F6". The conditions for the setpoint of the ball position are adjusted by means of the following dialog **GeneratorDlg**.

TMainForm.View1Click

View1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **View1Click** is an event handler activated by clicking once on the menu item "View". The following menu items "Plot Measured Data", "Plot File Data" and "Parameters from *.PLD File" are enabled only, when the memory contains data either from a previous measurement acquisition or after selecting a so-called documentation file.

TMainForm.PlotMeasuredData1Click

PlotMeasuredData1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **PlotMeasuredData1Click** is an event handler activated by clicking once on the menu item "View/Plot Measured Data". Those curves of a previous measurement acquisition are selected by means of the following dialog **PlotDlg**, which are to be presented in a plot window.

TMainForm.PlotFileData1Click

PlotFileData1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **PlotFileData1Click** is an event handler activated by clicking once on the menu item "View/Plot File Data". Those curves of a loaded documentation file are selected by means of the following dialog **PlotDlg**, which are to be presented in a plot window.

TMainForm.ParametersfromPLDFile1Click

ParametersfromPLDFile1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **ParametersfromPLDFile1Click** is an event handler activated by clicking once on the menu item "View/Parameters from *.PLD File". The parameters of the documentation file are displayed in a window by means of the following dialog **PLDInfoDlg**.

TMainForm.Fuzzy3D1Click

Fuzzy3D1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Fuzzy3D1Click** is an event handler activated by clicking once on the menu item "View/Fuzzy 3D". The following dialog **Show3DFuzDlg** will present a plot window containing the 3-dimensional characteristic of a selectable fuzzy description file.

TMainForm.Timing1Click

Timing1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Timing1Click** is an event handler activated by clicking once on the menu item "View/Timing". The visibility of a timing window (**TimingForm**) is toggled. It displays the minimum and maximum values of the sampling period or calculation time in [ms].

TMainForm.Contents1Click

Contents1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Contents1Click** is activated by clicking once on the menu item "Help/Contents" to display the contents of the help file BW500.HLP.

TMainForm.SearchforHelpon1Click

SearchforHelpon1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **SearchforHelpon1Click** is activated by clicking once on the menu item "Help/Search for Help On..." to start the Windows dialog to search for defined keywords.

TMainForm.HowtoUseHelp1Click

HowtoUseHelp1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HowtoUseHelp1Click** is activated by clicking once on the menu item "Help/How to Use Help" to start the Windows dialog displaying hints how to use the help function.

TMainForm.About1Click

About1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **About1Click** is activated by clicking once on the menu item "Help/About..." to display a window containing information about the program.

TMainForm.Timer1Timer

Timer1Timer(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Timer1Timer** is called every 200 ms by means of a timer. The data structure *data* of controller in the BWSERV16.DLL is read and the contents of the monitor in the main window are updated. This includes the current controller type, the position setpoint, the measured values for the ball position and beam angle, the control signal, the state of the measurement acquisition as well as an updated animation picture.

TMainForm.PaintBox1Click

PaintBox1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **PaintBox1Click** is activated by moving the mouse above the animation picture and pressing any mouse button. The setpoint generator dialog (by means of **SetpointGenerator1Click**) is called directly, when a controller is active.

TMainForm.MeasLabelClick

MeasLabelClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **MeasLabelClick** is activated by moving the mouse above the area displaying the progress of the measurement acquisition and pressing any mouse button. The measurement acquisition dialog (by means of **StartMeasuring1Click**) is called directly, when a controller is active.

TSingleInstance.WndProc

WndProc(*var Msg*: TMessage)

Parameters: *var Msg* is the current Windows system message received by this virtual window.

Description The procedure **WndProc** is a Windows message handler for the virtual window of type SingleInstance, which determines by checking the parameters *Msg*, *wParam* and *lParam* if an instance of this application was called already. In this case this application is terminated.

TAboutBox.FormShow

FormShow(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TAboutBox is displayed on the screen. Short information about the program (name, version, copyright, required PC adapter card) are presented in a window.

TCallibrateDlg.CenterBBtnClick

CenterBBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **CenterBBtnClick** is an event handler activated by clicking once on the upper "Start" button inside the "BW500 Calibration Dialog". The controller type is set to WAITING and the structure *param* is transferred to the BWSERV16.DLL. When the timer for the sampling period of the controller is running, the upper field in the dialog appears in a green background colour indicating an active sensor calibration for the zero-angle of the beam and for the zero-position of the ball. If the timer is not running or the sensor calibration failed the colour of the upper field is changed to red, the "Start" button is replaced by a "Retry" button and a corresponding error message is presented. In case of a successful sensor calibration, the colour of the upper field is changed to white and the next calibration step (**LeftBBtnClick**) is started automatically.

TCallibrateDlg.LeftBBtnClick

LeftBBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **LeftBBtnClick** is an event handler activated by clicking once on the "Start" button in the middle of the "BW500 Calibration Dialog". The field on the middle of the dialog appears in a green background colour indicating an active sensor calibration for left-most position of the ball. The required angle of the beam is initiated automatically. If the sensor calibration failed the colour of the field in the middle is changed to red, the "Start" button is replaced by a "Retry" button and a corresponding error message is presented. In case of a successful sensor calibration, the colour of the field in the middle is changed to white and the next calibration step (**RightBBtnClick**) is started automatically.

TCallibrateDlg.RightBBtnClick

RightBBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **RightBBtnClick** is an event handler activated by clicking once on the lower "Start" button inside the "BW500 Calibration Dialog". The lower field of the dialog appears in a green background colour indicating an active sensor calibration for right-most position of the ball. The required angle of the beam is initiated automatically. If the sensor calibration failed the colour of the lower field is changed to red, the "Start" button is replaced by a "Retry" button and a corresponding error message is presented. In case of a successful sensor calibration, the colour of the lower field is changed to white, the controller type is set to NONE and the structure *param* is transferred to the BWSERV16.DLL.

TCallibrateDlg.FormShow

FormShow(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TCallibrateDlg is displayed on the screen. The timer for the sampling period of the controller is started. No error is generated if the timer is still running. The three fields assigned to the calibration steps of the "BW500 Calibration Dialog" are displayed with a blue background colour. Only the "Start" button of the upper field is enabled. The user has to follow the instruction to "align the beam horizontally and position the ball in the middle of the beam", that means the positions are to be obtained manually, before pressing the enabled "Start" button. Otherwise the calibration of the camera signal for the ball position as well the calibration of the incremental encoder signal for the beam angle will fail.

TCallibrateDlg.HelpBtnClick

HelpBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the "BW500 Calibration Dialog". The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TShow3DFuzDlg.rescale

rescale

Description The procedure **rescale** calculates the scaling factors as well as initial values for the axes X, Y and Z such that the value ranges of the 3 fuzzy variables may be displayed in a cube with an edge length of 255.

TShow3DFuzDlg.recalc

recalc

Description The procedure **recalc** calculates the values of the fuzzy output variable for the complete value ranges of the fuzzy input variables with respect to the currently selected step width and based on the fuzzy object contained in the memory. The calculated vales are stored to the byte field *dat*.

TShow3DFuzDlg.calcrot

calcrot

Description The procedure **calcrot** calculates the values of the Euler rotation matrix depending on the rotation angles a and b. The angle a defines the rotation around the X-axis while the angle b defines the rotation around the Y-axis.

TShow3DFuzDlg.calctrans

calctrans(*ix,iy,iz* : double; var *ox,oy,oz* : double)

Parameters: *ix* x-co-ordinate original point.

iy y-co-ordinate original point.

iz z-co-ordinate original point.

var *ox* x-co-ordinate after rotation and projection.

var *oy* y-co-ordinate after rotation and projection.

var *oz* z-co-ordinate after rotation and projection.

Description The procedure **calctrans** calculates new co-ordinates for the 3-dimensional Cartesian co-ordinates of an original point by applying the Euler rotation matrix and calculating the projection to a fixed Y-plane.

TShow3DFuzDlg.FormShow

FormShow(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TShow3DFuzDlg is displayed on the screen. The dialog "Show Fuzzy 3D" appears to present the 3-dimensional characteristic of a fuzzy controller with two input variables and one output variable. The attributes of the characteristic itself are changeable interactively by setting checkboxes, scroll bars or by moving the mouse. Initial values are defined for a bitmap of suitable size, the step width for the characteristic, the co-ordinates of the observer position as well as for the projection plane. The step width determines the number of partial areas along the X-axis and the Y-axis. The listbox to select a fuzzy description file is filled with the four names contained in the "fuzzy-controller" file. The name of this file is taken from the structure *param*. The checkbox to mark the current operating point is enabled only, when the active controller is a fuzzy controller. The characteristic for the first fuzzy description file is displayed automatically by **FuzzyBoxChange** as a bitmap in the left field of the dialog. The mapping of the characteristic is calculated such that an observer looks in the middle of a cube placed behind the screen, where the cube is surrounding the characteristic.

TShow3DFuzDlg.FormHide

FormHide(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormHide** is called, when the object of type TShow3DFuzDlg has to disappear from the screen. The checkbox to mark the operating point in the characteristic as well as the timer are disabled, the bitmap is released.

TShow3DFuzDlg.FuzzyCBoxChange

FuzzyCBoxChange(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FuzzyCBoxChange** generates a new fuzzy object with respect to the currently fuzzy description file, which was selected from the listbox. When the selected file does not exist, a corresponding error message is presented. The text field below the listbox will contain the updated names and value ranges of the fuzzy variables belonging to the fuzzy object with two input variables and one output variable. The fuzzy variables are assigned to the X-, Y- and Z-axis accordingly. The corresponding characteristic is calculated and displayed as a bitmap. At the end the fuzzy object is removed again from the memory. A possibly running timer is stopped during the run-time of this procedure.

TShow3DFuzDlg.DrawSquare

DrawSquare(*can* : TCanvas; *j, i, z0, z1, z2, z3* : Integer)

Parameters: *can* is an identifier for a device context.

j is the index of the partial area along the X-axis.

i is the index of the partial area along the Y-axis.

z0 is the edge point 1 (Z-value) of the partial area.

z1 is the edge point 2 (Z-value) of the partial area.

z2 is the edge point 3 (Z-value) of the partial area.

z3 is the edge point 4 (Z-value) of the partial area.

Description The procedure **DrawSquare** draws the partial area defined by its edge points (*zo, z1, z2, z3*) taken as base points with respect to the Z-axis and defined by the indexes (*j, i*) along the X- and Y-axis as a polygon on the projection plane, which is identified by the device context *can*. The appearance of the polygon depends on the setting of further checkboxes. The polygon gets a black coloured frame, when "Grid" (**GridCBox**) is set, is displayed as a surface, when "Surface" (**SurfaceCBox**) is set, the surface is drawn with a grey scale with decreasing darkness for increasing Z-values or with a colour changing from red to blue, when "Colour" (**ColorCBox**) is set in addition.

TShow3DFuzDlg.DrawCoors

DrawCoors(*can* : TCanvas)

Parameters: *can* is an identifier for a device context.

Description The procedure **DrawCoors** draws the edges of the cube surrounding the characteristic as black coloured lines on the projection plane identified by the device context *can*.

TShow3DFuzDlg.DrawCoors2

DrawCoors2(*can* : TCanvas)

Parameters: *can* is an identifier for a device context.

Description The procedure **DrawCoors2** draws the 3-dimensional axes crossing in the middle of the cube surrounding the characteristic as black coloured lines on the projection plane identified by the device context *can*.

TShow3DFuzDlg.DrawMark

DrawMark(*can* : TCanvas)

Parameters: *can* is an identifier for a device context.

Description The procedure **DrawMark** draws the partial area, which is nearest to the current operating point of the fuzzy controller, as a green coloured area on the projection plane identified by the device context *can*. The operating point results from the current sensor values or its differentiations (see parameter structure *data*) with respect to the currently selected fuzzy description file.

TShow3DFuzDlg.PaintBoxPaint

PaintBoxPaint(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **PaintBoxPaint** calculates a new bitmap representing a complete characteristic and copies this bitmap to the screen memory only, when *needRedraw* is set. The complete characteristic consists of the partial areas defined by its base points contained in the byte field *dat*. The characteristic is completed by the edges of the cube, by the axes crossing or marking of the operating point, when the checkboxes "Co-ordinate box" (**KoorsCBox**), "Co-ordinate system" (**Koors2CBox**) or "Mark" (**MarkCBox**) are set accordingly.

TShow3DFuzDlg.ScrollBar1Change

ScrollBar1Change(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **ScrollBar1Change** maps the current position of the scroll mark to an angle in the range from -180 to +180, which is taken as a rotation angle around the X-axis. If the absolute change of the rotation angle is greater than 18, the characteristic is updated by means of **PaintBoxPaint**.

TShow3DFuzDlg.ScrollBar2Change

ScrollBar2Change(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **ScrollBar2Change** maps the current position of the scroll mark to an angle in the range from 0 to 360, which is taken as a rotation angle around the Y-axis. If the absolute change of the rotation angle is greater than 18, the characteristic is updated by means of **PaintBoxPaint**.

TShow3DFuzDlg.PaintBoxMouseDown

PaintBoxMouseDown(*Sender*: TObject; *Button*: TMouseButton; *Shift*: TShiftState; *X, Y*: Integer);

Description The procedure **PaintBoxMouseDown** is called, when the mouse is moved above the bitmap field to display the characteristic and when a mouse button is pressed. If the left mouse button is pressed the global co-ordinates *lastX, lastY* are set equal to the mouse position and the variable *lastBtn* is set to TRUE.

TShow3DFuzDlg.PaintBoxMouseMove

PaintBoxMouseMove(*Sender*: TObject; *Shift*: TShiftState; *X, Y*: Integer);

Description The procedure **PaintBoxMouseMove** is called, when the mouse is moved above the bitmap field to display the characteristic. If the variable *lastBtn* is set at the same time the current mouse position is mapped to new positions of the scrollbars to define the rotation angles around the X-axis and the Y-axis. The modified scrollbar positions (be means of **ScrollBar1Change**, **ScrollBar2Change**) will then result in an updated output with a rotated characteristic.

TShow3DFuzDlg.PaintBoxMouseUp

PaintBoxMouseUp(*Sender*: TObject; *Button*: TMouseButton; *Shift*: TShiftState; *X, Y*: Integer);

Description The procedure **PaintBoxMouseUp** is called, when the mouse is moved above the bitmap field to display the characteristic and when none of the mouse buttons is pressed. The variable *lastBtn* is reset.

TShow3DFuzDlg.KoorsCBoxClick

KoorsCBoxClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **KoorsCBoxClick** is an event handler activated by clicking once on the checkbox "Co-ordinate box". The output of the characteristic is with or without a surrounding cube according to the new setting of the checkbox.

TShow3DFuzDlg.Koors2CBoxClick

Koors2CBoxClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Koors2CBoxClick** is an event handler activated by clicking once on the checkbox "Co-ordinate system". The output of the characteristic is with or without an axes crossing according to the new setting of the checkbox.

TShow3DFuzDlg.ColorCBoxClick

ColorCBoxClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **ColorCBoxClick** is an event handler activated by clicking once on the checkbox "Colour". The output of the characteristic is with grey or coloured partial areas according to the new setting of the checkbox.

TShow3DFuzDlg.LowResCBoxClick

LowResCBoxClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **LowResCBoxClick** is an event handler activated by clicking once on the checkbox "Low resolution". The updated output displays the characteristic with smaller (step width = 12) or greater (step width = 25) partial areas according to the new setting of the checkbox.

TShow3DFuzDlg.PrintBBtnClick

PrintBBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **PrintBBtnClick** is an event handler activated by clicking once on the "Print" button of the "Show Fuzzy 3D" dialog. The Windows system dialog appears to select an output device for the hardcopy of the complete "Show Fuzzy 3D" dialog.

TShow3DFuzDlg.MarkCBoxClick

MarkCBoxClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **MarkCBoxClick** is an event handler activated by clicking once on the checkbox "Mark". The updated output of the characteristic will contain a partial area indicating the current operating point according to the setting of the checkbox. The timer state is set equal to the setting of the checkbox, that means when the operating point is to be indicated the timer will produce an updated characteristic periodically.

TShow3DFuzDlg.Timer1Timer

Timer1Timer(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Timer1Timer** is called by a timer every 200ms, as long as this timer is enabled. The state of the timer is set equal to the setting of the checkbox "Mark". The output of the characteristic is updated.

TShow3DFuzDlg.HelpBtnClick

HelpBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the "Show Fuzzy 3D" dialog. The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TFuzzyParametersDlg.Big

Big

Description The procedure **Big** expands the dialog by another edit field below the dialog and two additional button. The content of the edit field is erased.

TFuzzyParametersDlg.Small

Small

Description The procedure **Small** removes the edit field and its accompanying two buttons from the lower part of the dialog. The content of the edit field is erased. The name of the fuzzy description file belonging to the edit field is reset to NONAME.FUZ.

TFuzzyParametersDlg.FormCreate

FormCreate(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormCreate** creates an instance of type TFuzzyParametersDlg. A string list is generated to store the names of the fuzzy description files.

TFuzzyParametersDlg.FormDestroy

FormDestroy(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormDestroy** is called before the object of type TFuzzyParametersDlg is removed from the memory. The string list containing the names of the fuzzy is erased.

TFuzzyParametersDlg.FormShow

FormShow(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TFuzzyParametersDlg is displayed on the screen. At first only the upper part of the "Fuzzy Controller Parameters" dialog is presented. The four fields (labelled "Position Controller", "Angle Controller", "Position Observer" and "Angle Observer") display the names of the accompanying fuzzy description files. These names are load from the "fuzzy-controller" file, the name of which is read from the structure *param*. If this file does not exist a corresponding error message will appear.

TFuzzyParametersDlg.Sel1BBtnClick

Sel1BBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Sel1BBtnClick** is an event handler activated by clicking once on one of the "Select" buttons of the "Fuzzy Controller Parameters" dialog. A Windows system dialog will appear to select the name of a fuzzy description file (extension *.FUZ). The selected name will be displayed left to the "Select" button, when an existing file was chosen.

TFuzzyParametersDlg.Ed1BBtnClick

Ed1BBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Ed1BBtnClick** is an event handler activated by clicking once on one of the "Edit" buttons of the "Fuzzy Controller Parameters" dialog. The dialog will be expanded by an edit field and two additional buttons ("Save", "Abort"). If the fuzzy description file with the name displayed at the left side in the field of the activated "Edit" button exists its content is shown in the edit field (variable *Memo1*). Typical edit functions are now allowed inside the edit field.

TFuzzyParametersDlg.CancelEdBBtnClick

CancelEdBBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **CancelEdBBtnClick** is an event handler activated by clicking once on the "Abort" button of the dialog extension. The edit field will be removed from the dialog and its content will be erased.

TFuzzyParametersDlg.SaveEdBBtnClick

SaveEdBBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **SaveEdBBtnClick** is an event handler activated by clicking once on the "Save" button of the dialog extension. The content of the edit field is stored to the open fuzzy description file. An error message will appear, when the saving procedure fails. Then the edit field will be removed from the dialog and its content will be erased.

TFuzzyParametersDlg.OKBtnClick

OKBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OKBtnClick** is an event handler activated by clicking once on the "Reload" button of the "Fuzzy Controller Parameters" dialog. The names of all currently selected fuzzy description files are written to the open "fuzzy-controller" file. The corresponding fuzzy descriptions are reloaded and the accompanying objects are generated. Errors occurred during writing to the "fuzzy-controller" file or errors generated with the new creation of the fuzzy objects are displayed in corresponding error messages.

TFuzzyParametersDlg.HelpBtnClick

HelpBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the "Fuzzy Controller Parameters" dialog. The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TMeasureDlg.OKBtnClick

OKBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OKBtnClick** is an event handler for clicking once on the "Ok" button from the dialog "Setup Measuring Function" to adjust the conditions for the measurement acquisition. The contents of three input fields are converted to numbers for the total measuring time (*time* = 0 to 1000 sec), for the time before reaching the trigger condition (*prestore* = 0 to measuring time) and for the trigger level (*trigger* = -0.4 to 0.4) only when none of the numbers exceeds the valid range. Two further groups of radio buttons are used to determine the trigger channel *tchannel* as well as the trigger condition *slope*. The trigger condition is either not existing or defined as a slope, meaning that the measured value of the trigger channel has to exceed the trigger level either in positive or in negative direction. The measuring is started directly after terminating the dialog. Measured values are the setpoint for the ball position, the measured values for the ball position and the beam angle, the differentiations of the measured signals as well as the control signal including additional friction compensations.

TMeasureDlg.HelpBtnClick

HelpBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the "Setup Measuring Function" dialog. The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TPLDInfoDlg.FormShow

FormShow(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TPLDInfoDlg is displayed on the screen. The controller settings of a loaded so-called documentation file are displayed in a "PLD Information" window. The controller settings include the controller type (state controller, fuzzy controller, calibration mode, no controller). the time of the measurement acquisition as well as the sampling rate of the measurement.

TPLDInfoDlg.HelpBtnClick

HelpBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the "PLD Information" dialog. The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TPlotDlg.OKBtnClick

OKBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OKBtnClick** is an event handler for clicking once on the "Ok" button from the dialog "Select Plot Data" to select the channels of a measuring which are to be represented in a plot window. The selectable channels are the measured value and setpoint of the ball position, the measured ball position, the measured beam angle, the control signal, the calculated ball speed and angle speed, the friction compensation for the beam and ball (the last only with the fuzzy controller).

TPlotDlg.HelpBtnClick

HelpBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the dialog "Select Plot Data" to select the channels of a measuring which are to be represented in a plot window. The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TPrintPlotDlg.PrinterBitBtnClick

PrinterBitBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **PrinterBitBtnClick** is an event handler for clicking once on the "Printer" button from the dialog to select previously created plot windows. A modal Windows system dialog appears that permits the user to select which printer to print to, how many copies to print and further print options.

TPrintPlotDlg.OKBtnClick

OKBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OKBtnClick** is an event handler for clicking once on the "Ok" button from the dialog to select previously created plot windows. All of the plot windows selected from the list box are printed directly to the current output device (by means of the function **PrintPlotMeas**). When multiple plot windows are selected an offset of 150 mm (counted from the upper margin of a DIN A4 page) is added before every second print output and a form feed follows this output.

TPrintPlotDlg.FormShow

FormShow(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TPrintPlotDlg is displayed on the screen. At first all titles of the previously created plot windows are inserted in a listbox.

TPrintPlotDlg.HelpBtnClick

HelpBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler for clicking once on the "Help" button from the dialog to select previously created plot windows. The accompanying section of the help file BW500.HLP will be displayed in a window on the screen.

TGeneratorDlg.OKBtnClick

OKBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OKBtnClick** is an event handler for clicking once on the "Ok" button from the dialog "Ball Position Setpoint Generator" to adjust the setpoint for the ball position. For the setpoint the signal shape is selectable by radio buttons (constant, rectangle, triangle, ramp, sine) and an offset, an amplitude as well as a time period are adjustable by input fields. The period is meaningless in case of a constant signal shape. The real signal is always built by the sum of offset and amplitude. The valid value ranges are -0.4 to +0.4m for the setpoint's offset and amplitude and 0 - 1000 sec for the period. Only when none of the corresponding number exceeds the valid value range, the numbers are stored in the parameter structure *param* which is then transferred to the controller in the BWSERV16.DLL. Finally the dialog is terminated and the generator starts operating with the next sampling period.

TGeneratorDlg.FormShow

FormShow(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TGeneratorDlg is displayed on the screen. The input fields as well as the radio buttons to adjust the generator for the setpoint of the ball position are preset according to the parameters of the global parameter structure *param*.

TGeneratorDlg.HelpBtnClick

HelpBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the dialog "Ball Position Setpoint Generator". The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TStateParametersDlg.FormCreate

FormCreate(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormCreate** creates an instance of an object of type TStateParametersDlg. With this a multiple-page dialog with four pages is generated.

TStateParametersDlg.TabSetClick

TabSetClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **TabSetClick** is an event handler for clicking once on one of the tabs of a page in the "State Controller Parameters" dialog. The selected page will appear inside the dialog.

TStateParametersDlg.FormShow

FormShow(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TStateParametersDlg is displayed on the screen. All the matrices of the state controller are read from the parameter structure *param*. The edit fields of the multiple-page dialog "State Controller Parameters" are preset accordingly. The first page of the dialog with the tab stop "State Feedback" is displayed allowing to inspect and change the components of the feedback vector *F*. The second page "Prefilter" provides editing the parameter of the pre-filter. The components of the matrices *L, A, B, F* of the reduced-order state observer are changeable with the third page "State Observer". The fourth page with the tab stop label "Friction Compensation" allows for editing the matrices *L, A, B, F* of the (friction) disturbance observer and the parameter *Const.* of the constant friction compensation.

TStateParametersDlg.OKBtnClick

OKBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OKBtnClick** is an event handler for clicking once on the "Ok" button of one of the pages of the "State Controller Parameters" dialog. As long as all of the contents of the edit fields are convertible to binary numbers all the matrices of the state controller are copied to the parameter structure *param* which is then transferred to the BWSERV16.DLL. An error message is presented in the other case.

TStateParametersDlg.HelpBtnClick

HelpBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in one of the pages of the dialog "State Controller Parameters". The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TStartFuzzDlg.OKBtnClick

OKBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OKBtnClick** is an event handler activated by clicking once on the menu item "Run/Fuzzy Controller F3". The "Start Fuzzy Controller" dialog will appear displaying two checkboxes to select the friction compensation for the ball and/or the beam by means of a fuzzy disturbance observer. The element *param.fuzzyobserver* is set accordingly.

TStartFuzzDlg.HelpBtnClick

HelpBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the dialog "Start Fuzzy Controller". The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TStartStateDlg.OKBtnClick

OKBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OKBtnClick** is an event handler activated by clicking once on the menu item "Run/State Controller F2". The "Start State Controller" dialog will appear containing two groups of radio buttons to select the friction compensation for the ball (be means of an disturbance observer, a constant compensation or none compensation) and to select the way the differentiations of the state variables are to be determined (by means of a state observer or by difference quotients). The element *param.stateobserver* is set accordingly.

TStartStateDlg.HelpBtnClick

HelpBtnClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpBtnClick** is an event handler activated by clicking once on the "Help" button in the dialog "Start State Controller". The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TTimingForm.UpdateData

UpdateData

Description The procedure **UpdateData** is called periodically with the update rate of the main window (timer in **TMainForm**). The current minimum and maximum values of the real sampling period or of the calculation time during the sampling period are obtained by means of **GetMinMaxTime** from the TIMER16.DLL. The recently selected values are displayed in the "BW500 Timing" dialog. The flag *ResFlag* to reset the minimum and maximum values is reset.

TTimingForm.Button1Click

Button1Click(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Button1Click** is an event handler activated by clicking once on the "Reset" button in the "BW500 Timing" dialog. The flag *ResFlag* to reset the minimum and maximum values is set. The content of the dialog is updated (**UpdateData**).

TTimingForm.FormShow

FormShow(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when an object of type TTimingForm is displayed on the screen. The dialog "BW500 Timing" to display the minimum and maximum values of the real sampling period or of the calculation time during the sampling period in [ms]. A reset operation (**Button1Click**) is carried-out. The default display values are from the real sampling period.

TTimingForm.Button2Click

Button2Click(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Button2Click** is an event handler activated by clicking once on the "Hide" button in the "BW500 Timing" dialog. The procedure **TMainForm.Timing1Click** will reset the visibility of the "BW500 Timing" dialog.

TTimingForm.Button3Click

Button3Click(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Button3Click** is an event handler activated by clicking once on the "Help" button in the dialog "BW500 Timing". The corresponding section of the help file BW500.HLP will be displayed in a window on the screen.

TTimingForm.OnClickBtnSampleCalc

OnClickBtnSampleCalc(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **OnClickBtnSampleCalc** is an event handler activated by clicking once on the "Sample time"/"Calc time" button in the "BW500 Timing" dialog. The label of the button as well as the displayed values are toggled accordingly (from sampling period to calculation time and vice versa).

FloatToStr2

FloatToStr2(*f* : Single) : string

Parameters: *f* is the floating point value, which is to be converted.

Description The function **FloatToStr2** converts a floating point value (4 bytes for single) to a its string representation with a maximum of 7 significant digits and 2 digits behind the decimal point.

Return Is the string representation of the floating point value.

FloatToStr3

FloatToStr3(*f* : Single) : string

Parameters: *f* is the floating point value, which is to be converted.

Description The function **FloatToStr3** converts a floating point value (4 bytes for single) to a its string representation with a maximum of 7 significant digits and 3 digits behind the decimal point.

Return Is the string representation of the floating point value.

FloatToStr4

FloatToStr4(*f* : Single) : string

Parameters: *f* is the floating point value, which is to be converted.

Description The function **FloatToStr4** converts a floating point value (4 bytes for single) to a its string representation with a maximum of 7 significant digits and 4 digits behind the decimal point.

Return Is the string representation of the floating point value.

StrToFloatMinMax

StrToFloatMinMax(*s* : string; *min,max* : double) : double

Parameters: *s* is the string representation of a floating point value.

min is the lower limit for a floating point value.

max is the upper limit for a floating point value.

Description The function **StrToFloatMinMax** converts a string to the corresponding floating point value. When this value exceeds the lower or upper limit it is set equal to the exceeded limit and a corresponding message appears on the screen.

Return Is the possibly limited floating point value.

StrToFloatStrMinMax

StrToFloatStrMinMax(*s* : string; *var val* : double; *min,max* : double) : string

Parameters: *s* is the string representation of the floating point value.

var val is on return the possibly limited floating point value.

min is the lower limit for a floating point value.

max is the upper limit for a floating point value.

Description The function **StrToFloatStrMinMax** converts a string to the corresponding floating point value. When this value exceeds the lower or upper limit it is set equal to the exceeded limit and a corresponding message appears on the screen. The possibly limited floating point value is again converted to its string representation with a maximum of 7 significant digits and 3 digits behind the decimal point.

Return Is the string representation of the possibly limited floating point value.

MinMaxi

MinMaxi(*val, min, max* : Integer) : Integer

Parameters: *val* is the integer value, which is to be checked.
min is the lower limit for an integer value.
max is the upper limit for an integer value.

Description The function **MinMaxi** checks if an integer value is inside a limited range. When the integer value exceeds the lower or upper limit it is set equal to the nearest limit.

Return Is the possibly limited integer value.

DetectNT

DetectNT : Boolean

Description The function **DetectNT** returns TRUE, when the file NTOSKRNL exists in the Windows system directory, otherwise it returns FALSE. An error message is presented if the Windows system directory does not exist. It is assumed that the existence of the file means an operating NT system and a 16 bit application will not run with a NT system.

Return Returns TRUE, when NTOSKRNL.EXE exists, else FALSE.

1.4 Overview of Classes and DLL Interfaces

The files BWSERV.H, BWSERV.CPP contain:

BOOL CALLBACK **DoService**(DWORD *counter*)

BOOL CALLBACK **SetParameter**(WORD *wSize*, LPSTR *lpData*)

BOOL CALLBACK **GetParameter**(WORD *wSize*, LPSTR *lpData*)

BOOL CALLBACK **GetData**(WORD *wSize*, LPSTR *lpData*)

BOOL CALLBACK **LockMemory**(BOOL *bStart*, HDRVR *hDrv*)

BOOL CALLBACK **SetDriverHandle**(HDRVR *hDrv*)

BOOL CALLBACK **ReadFuzzy**(void)

BOOL CALLBACK **ReadStatePar**(char* *name*)

BOOL CALLBACK **WriteStatePar**(char* *name*)

BOOL CALLBACK **IsDemo**(void)

int CALLBACK **CalibrateSen**(int *mode*)

int CALLBACK **RawSensor**(int *mode*)

int CALLBACK **MeasureStart**(double *time*, double *trigger*, double *prestore*, int *tchannel*, int *slope*)

double CALLBACK **MeasureLevel**(void)

int CALLBACK **MeasureStatus**(void)

int CALLBACK **OpenFuzzy3D**(char* *filename*)

int CALLBACK **CloseFuzzy3D**(void)

Fuzzy3DInfo* CALLBACK **InfoFuzzy3D**(void)

double CALLBACK **CalcFuzzy3D**(double *x*, double *y*)

The files BW500STA.H, BW500STA.CPP contain the class **BW500STA** with:

void **BW500STA** (void)

void **Calc**(double *w*, double *Position*, double *Winkel*)

void **SetTa**(float *ta*)

void **Reset**(void)

int **Load**(char* *name*)

int **Save**(char* *name*)

void **SetStateObserver**(Observer *o*)

void **SetDistObserver**(Observer *o*)

int **geterrors**(void)

double* **GetV**(void)

double* **GetFt**(void)

double* **GetLBD**(void)

double* **GetABD**(void)

double* **GetFBD**(void)

double* **GetBBD**(void)

double* **GetDCON**(void)

double* **GetDLBD**(void)

double* **GetDABD**(void)

double* **GetDFBD**(void)

double* **GetDBBD**(void)

The files BW502FUZ.H, BW502FUZ.CPP contain the class **BW502FUZ** :

void **BW502FUZ** (void)

void **~BW502FUZ** (void)

void **Calc**(double *w*, double *Position*, double *Winkel*)

double **SelectFuzzyFile**(char* *name*)

void **Save**(char* *name*)

void **SetAngObserver**(Observer *o*)

void **SetPosObserver**(Observer *o*)

char* **getname**(void)

char* **getfname**(void)

int **geterrors**(void)

The files ARINGBUF.H, ARINGBUF.CPP contain:

class **STOREBUF**

void **ResetBufIndex**(void)
STOREBUF(int, float *)
~STOREBUF()
void **StartMeasure**(int, float, float, float, int , float)
void **WriteValue**(void)
void **SetOutChan**(int)
float **ReadValue**(void)
int **GetBufLen**(void)
float **GetBufTa**(void)
int **GetStatus**(void)

The files DRSIGNAL.H, DRSIGNAL.CPP contain:

class **AFBUF**

- **AFBUF**()
- **~AFBUF**()
- int **NewFBuf**(int)
- float **ReadFBuf**(void)
- int **WriteFBuf**(float)

class **TWOBUFFER**

- **TWOBUFFER**()
- void **New2Buffer**(int , int, int)
- int **Write2Buffer**(float)
- float **Read2Buffer**(void)

class **SIGNAL**

- **SIGNAL**()
- float **InitTime**(float)
- int **MakeSignal**(int , float, float, float, int)
- float **ReadNextValue**(void)
- void **SetRange**(float, float)
- void **WriteBuffer**(float)
- int **Stuetzstellen**(float, int)

The file PLOT.CPP contains:

class **PLOT**

- int CALLBACK **ReadPlot**(char **lpfName*)
- int CALLBACK **WritePlot**(char **lpfName*)
- int CALLBACK **Plot**(int *command*, int *channel*)
- int CALLBACK **GetPlot**(int *start*, char **lpzName*)
- int CALLBACK **PrintPlot**(int *idx*, HDC *dcPrint*, int *iyOffset*)
- int CALLBACK **GetPldInfo**(int &*controller*, char ***s*, int &*n*, int &*c*, double &*d*)

1.5 References of the DLL Interfaces

Global Data:

```
typedef struct{
        WORD        controller;           // controller type (state, fuzzy controller, none)
        WORD        state_observer;       // flag for type of friction compensation in state controller
        WORD        stateError;           // flag for error in state controller
        WORD        fuzzy_observer;       // flag for type of friction compensation in fuzzy controller
        WORD        fuzzyError;           // flag for error in fuzzy controller
        WORD        dummy1
        WORD        dummy2
        WORD        sp1shape;             // shape of setpoint signal (constant, rectangle, sine etc.)
        double      ft[4];                // state feedback vector
        double      filter;               // pre-filter
        double      lbd[4];               // Luenberger observer matrix L
        double      abd[4];               // Luenberger observer matrix A
        double      fbd[4];               // Luenberger observer matrix F
        double      bbd[2];               // Luenberger observer vector b
        double      dcon;                 // parameter of constant friction compensation
        double      dabd;                 // disturbance observer matrix A
        double      dbbd;                 // disturbance observer matrix B
        double      dfbd[4];              // disturbance observer matrix F
        double      dlbd[4];              // disturbance observer matrix L
        char        name[80];             // name of the "fuzzy-controller" file
        double      spoffset;             // offset of setpoint signal
        double      spamplitude;          // amplitude of setpoint signal
        double      spperiode;            // period of setpoint signal
}ServiceParameter ;

typedef struct{
        double      setpoint;             // setpoint for ball position
        double      pos;                  // measured value of ball position
        double      dpos;                 // calculated value of ball speed
        double      ang;                  // measured value of beam angle
        double      dang;                 // calculated value of beam angle speed
        double      out;                  // control signal for beam drive
        double      fuzhelp;              // ouput fuzzy object
        WORD        state;
        WORD        dummy[3];
}ServiceData;
```

```
struct Fuzzy3DInfo{
        long        size;           // current size of this structure
        int         idx;            // index for the first input variable
        int         idy;            // index for the second input variable
        int         idz;            // index for the output variable
        char        xname[80];      // name of the first input variable
        char        yname[80];      // name of the second input variable
        char        zname[80];      // name of the output variable
        double      xmin;           // minimum value of the first input variable
        double      xmax;           // maximum value of the first input variable
        double      ymin;           // minimum value of the second input variable
        double      ymax;           // maximum value of the second input variable
        double      zmin;           // minimum value of the output variable
        double      zmax;           // maximum value of the output variable
};
```

ServiceParameter *par*	is a global structure of type ServiceParameter (see also BWSERV.H)
ServiceData *dat*	is a global structure of type ServiceData (see also BWSERV.H)
HGLOBAL *mHnd*	is a handle for the code memory of the BWSERV16.DLL
UINT *mData* = 0	is a handle for the data memory of the BWSERV16.DLL
HDRVR *hDriver* = NULL	is a handle for the adapter card driver (*.DRV)
DWORD *dwCounter* = 0L	is a counter for calling the function **DoService**
DICDRV drv	is an instance of the class **DICDRV** (driver interface)
float *scopebuf*[ScopeBufSize] is the measurement-vector	
STOREBUF *scope*(ScopeBufSize, scopebuf, 1024)	is an object to handle the storage of a maximum of 1024 elements of type *scopebuf*
SIGNAL *SpGen*	is a setpoint generator object
BW500STA *StateCon*	is a state controller object
BW500FUZ *FuzzyCon*	is a fuzzy controller object
double *sWinkel*	is an beam angle setpoint during calibration
int *stopped*	is a flag for detecting multiple output stage release errors in **DoService**.
Fuzzy* *fuzzy3d*	is a pointer to an instance of the class **Fuzzy** (fuzzy object served for output of 3-dimensional characteristic).
Fuzzy3DInfo *fuzzy3dinfo*	is a global structure of type Fuzzy3DInfo (see also F3DINFO.H)

1.5.1 The DLL Interface BWSERV16

DoService

BOOL CALLBACK **DoService**(DWORD *counter*)

Parameters *counter* is a counter for the number of calls.

Description The function **DoService** is the service routine called with periodic timer events (see also TIMER16.DLL). Timer events occur with a (nearly) constant sampling period as long as they are enabled. The following operations are carried-out in sequence:

Reset the flag *stopped*
Reset the trigger pulse for the servo and the release counter for the first call or after changing the controller
Trigger the output stage (rectangle signal),
Read sensors for ball position, beam angle,
Correct the ball position,
Get the ball position setpoint from the generator,
Calculate the control signal (state controller/fuzzy controller/none),
Output of control signal,
Store the measurement-vector,
If PC control is disabled increment release counter, else reset
If release counter is equal to 5 set flag *stopped*.

Attention: This function is to be called only by the TIMER16.DLL!

Return Is equal to FALSE, when flag *stopped* is set (PC control no longer enabled), else TRUE.

SetParameter

BOOL CALLBACK **SetParameter**(WORD *wSize*, LPSTR *lpData*)

Parameters *wSize* is the size (in bytes) of the data structure pointed to by *lpData*.

lpData is a pointer to a data structure of type ServiceParameter.

Description The function **SetParameter** copies the data structure pointed to by *lpData* to the global structure *par* (type ServiceParameter) only when the size of the source structure is less than or equal to the size of the destination structure. In this case the matrices of the state controller, the type of friction compensation as well as the setpoint generator are set accordingly.

Return Is equal to TRUE when the size of the source structure is less than or equal to the size of the destination structure, else return is equal to FALSE.

GetParameter

BOOL CALLBACK **GetParameter**(WORD *wSize*, LPSTR *lpData*)

Parameters *wSize* is the size (in bytes) of the data structure pointed to by *lpData*.

lpData is a pointer to a data structure of type ServiceParameter.

Description The function **GetParameter** at first copies all the current matrices of the state controller as well as the name of the "fuzzy-controller" file to the global structure *par* (type ServiceParameter) then it copies this structure to the destination structure pointed to by *lpData*. The last copy procedure is carried-out only, when the size of the source structure *par* is equal to the size of the destination structure.

Return Is equal to TRUE when the size of the source structure is less than or equal to the size of the destination structure, else return is equal to FALSE.

GetData

BOOL CALLBACK **GetData**(WORD *wSize*, LPSTR *lpData*)

Parameters *wSize* is the size (in bytes) of the data structure pointed to by *lpData*.

lpData is a pointer to a data structure of type ServiceData.

Description The function **GetData** at first copies the content of the measurement-vector *scopebuf* to the global structure *dat* (type ServiceData). Then *dat* is copied to the data structure pointed to by *lpData* only when the size of the source structure is less than or equal to the size of the destination structure. The controller state is set to 1 if the flag *stopped* is set.

Return Is equal to TRUE when the size of the source structure is less than or equal to the size of the destination structure, else return is equal to FALSE.

LockMemory

BOOL CALLBACK **LockMemory**(BOOL *bStart*, HDRVR *hDrv*)

Parameters *bStart* is a flag with the meaning:
=TRUE, code and data memory of the BWSERV16.DLL will be locked,
=FALSE, code and data memory of the BWSERV16.DLL will be unlocked.

hDrv is a handle for the IO-adapter card driver (is not used here).

Description The function **LockMemory** controls the lock status of the code and data memory of the complete BWSERV16.DLL. With *bStart*=TRUE this memory is locked. With *bStart*=FALSE this memory will be unlocked again.

Attention: This function is to be called only by the TIMER16.DLL !

Return Is equal to TRUE, when the handle of the code memory of the BWSERV16.DLL is valid, else return is equal to FALSE.

SetDriverHandle

BOOL CALLBACK **SetDriverHandle**(HDRVR *hDrv*)

Parameters *hDrv* is a handle for the IO-adapter card driver.

Description The function **SetDriverHandle** sets the internal handle for the IO-adapter card driver equal to the actual parameter.

Attention: This function may only be called by the TIMER16.DLL!

Return Always equal to 0.

ReadFuzzy

BOOL CALLBACK **ReadFuzzy**(void)

Description The function **ReadFuzzy** reads all the fuzzy description files and generates the accompanying fuzzy objects. The names of the fuzzy description files are read from the "fuzzy-controller" file, the name of which is taken from the global parameter structure *par*. The current controller type in *par* is set equal to FUZZYCONTROLLER only, when a fuzzy controller was active previously and the new fuzzy object generation was error-free. In case of a previously active fuzzy controller but errors occurred during the fuzzy object generation the controller type is set equal to NOCONTROLLER. In any other case the controller type remains as it has been before this function was called.

Return Is equal to TRUE when the new fuzzy object generation was successful, else FALSE is returned.

ReadStatePar

BOOL CALLBACK **ReadStatePar**(char* *name*)

Parameters *name* is a pointer to the name of a file from which the matrices of a state controller are to be read.

Description The function **ReadStatePar** reads all the matrices of a state controller from the file with the given name *name*. The current controller type in *par* is set equal to STATECONTROLLER only, when a state controller was active previously and the loading procedure was successful. In case of a previously active state controller but errors occurred during loading the matrices the controller type is set equal to NOCONTROLLER. In any other case the controller type remains as it has been before this function was called.

Return Is equal to TRUE, when reloading the matrices of the state controller was successful, else FALSE is returned.

WriteStatePar

BOOL CALLBACK **WriteStatePar**(char* *name*)

Parameters *name* is a pointer to the name of a file to which the matrices of a state controller are to be written.

Description The function **WriteStatePar** writes all the matrices of the state controller to a file with the given name *name*. The current controller type in *par* is set equal to STATECONTROLLER only, when a state controller was active previously and the writing procedure was successful. In case of a previously active state controller but errors occurred during writing the matrices the controller type is set equal to NOCONTROLLER. In any other case the controller type remains as it has been before this function was called.

Return Is equal to TRUE, when writing the matrices of the state controller was successful, else FALSE is returned.

IsDemo

int CALLBACK IsDemo(void)

Description The function **IsDemo** returns a 1 only when the BWSERV16.DLL is a DEMO version (generated with the macro __SIMULATION__ , instead of the IO-adapter card a mathematical model is accessed). Otherwise the function returns 0.

Return Is equal to 1 in case of a DEMO version, else equal to 0.

CalibrateSen

int CALLBACK **CalibrateSen**(int *mode*)

Parameters *mode* defines the calibration data:
=0, zero-position of the ball, zero-angle of the beam,
=1, left-most ball position,
=2, right-most ball position.

Description The function **CalibrateSen** determines the calibration data for the incremental encoder signal to measure the beam angle as well as the camera signal to measure the ball position depending on the parameter *mode*. With *mode*=0 the current incremental encoder signal is taken as the zero-angle of the beam and the current camera signal is taken as the zero-position of the ball. When the zero-position is outside of a range from 300 to 800 and error is returned. The camera signal for the left-most ball position is taken with *mode*=1. Hereby the beam is inclined automatically by a proportional controller and then the ball position is measured after a delay time of about 6 seconds. A measured ball position above a value of 550 result in an error return. A similar procedure is carried-out with *mode*=2 for the right-most ball position in a valid range from 550 to 1250.

Return Error state of the calibration procedure:
=0, no error,
=-1, invalid value for *mode*,
=-2, system detection failed,
=-3, PC control disabled,
=-4, ball position outside of expected range,
=-5, limit switches for maximum beam angle.

RawSensor

int CALLBACK **RawSensor**(int *mode*)

Parameters *mode* defines the return value:
=0, camera signal,
=1, incremental encoder signal.

Description The function **RawSensor** returns the sensor data (raw data) of the camera signal (*mode*=0) or the incremental encoder signal (*mode*=1).

Return Camera signal or incremental encoder signal.

MeasureStart

int CALLBACK **MeasureStart**(double *time*, double *trigger*, double *prestore*, int *tchannel*, int *slope*)

Parameters *time* is the total measuring time (in sec).

trigger is the trigger level for the trigger channel.

prestore is the time before the trigger condition is reached (in sec).

tchannel is the number of the trigger channel.

slobe is a flag for the direction of the trigger condition.

float *taint* is the sampling period of the service routine.

Description The function **MeasureStart** calls the function *scope*.**StartMeasure** to start a measuring. In advance the controller settings are copied to the global structure *measctrlstatus*.

See also STOREBUF::**StartMeasure**

The functions

MeasureLevel

double CALLBACK **MeasureLevel**(void)

MeasureStatus

int CALLBACK **MeasureStatus**(void)

Description call directly the corresponding functions *scope*.**GetBufferLevel**, *scope*.**GetStatus** of the class **STOREBUF**.

See also STOREBUF::**GetBufferLevel**, STOREBUF::**GetStatus**

OpenFuzzy3D

int CALLBACK **OpenFuzzy3D**(char* *filename*)

Parameters *filename* is a pointer to the name of a fuzzy description file.

Description The function **OpenFuzzy3D** opens the fuzzy description file with the name *filename* and generates the accompanying fuzzy object (*fuzzy3d* serving for the output of its 3-dimensional characteristic). the output file for state and error messages is FUZZY3D.OUT.

Return Error state:
=0, no error,
=-1, invalid pointer to fuzzy object,
=-2, syntax error in the fuzzy description file,
=-3, errors during fuzzy object generation.

CloseFuzzy3D

int CALLBACK **CloseFuzzy3D**(void)

Description The function **CloseFuzzy3D** removes an existing fuzzy object (*fuzzy3d* serving for the output of its 3-dimensional characteristic) from the memory and resets its pointer to NULL.

Return Always 0.

InfoFuzzy3D

Fuzzy3DInfo* CALLBACK **InfoFuzzy3D**(void)

Description The function **InfoFuzzy3D** returns the structure of type Fuzzy3DInfo belonging to an existing fuzzy object (*fuzzy3d* serving for the output of its 3-dimensional characteristic).

Return Structure of type Fuzzy3DInfo of an existing fuzzy object.

CalcFuzzy3D

double CALLBACK **CalcFuzzy3D**(double *x*, double *y*)

Parameters	*x* is the value of the first input variable of a fuzzy object. *y* is the value of the second input variable of a fuzzy object.
Description	The function **CalcFuzzy3D** calculates the output variable (index *fuzzy3dinfo.idz*) of a fuzzy object (*fuzzy3d*) with the two input variables *x* and *y* (indexes *fuzzy3dinfo.idx*, *fuzzy3dinfo.idy*). The return value is the output variable, when the fuzzy object exists and the indexes are within a valid range.
Return	Output variable of an existing fuzzy object with two given values for its input variables or 0.0.

1.5.2 The Class BW502STA in the BWSERV16.DLL

The class **BW502STA** provides functions to calculate the state controller and to determine missing state variables as well as disturbance signals.

Public Data:

enum *Observer*	{ NONE=0, CONSTANT, ACTIVE };

Private Data:

double *t*	sampling period
double *v*	pre-filter
double *ft[4]*	state feedback vector
double *x[4]*	state vector
double *z[3]*	current observer state vector
double *zold[3]*	previous observer state vector
double *lbd[4]*	L-matrix of the state observer
double *abd[4]*	A-matrix of the state observer
double *fbd[4]*	F-matrix of the state observer
double *bbd[4]*	B-vector of the state observer
double *dcon*	parameter of the constant friction compensation
double *dabd*	A-matrix of the disturbance observer
double *dbbd*	B-matrix of the disturbance observer
double *dfbd[4]*	F-matrix of the disturbance observer
double *dlbd[4]*	L-matrix of the disturbance observer
double *Position_alt*	measured ball position from the previous sampling period
double *Winkel_alt*	measured beam angle from the previous sampling period
int *start*	flag: reset of the observer
Observer *state*	mode of the state observer
Observer *dist*	mode of the disturbance observer
char *filename[MAXPATH]*	current name of the parameter file
int *errors*	error counter for file loading

BW500STA::BW500STA()

void **BW500STA** (void)

Description The constructor of the class **BW500STA** initializes values for the internal error, the sampling period, all the matrices of the state controller as well as for the constant friction compensation. The name of the current parameter file is set to DEFAULT.STA. When this file could be read successfully all the matrices of the state controller are set accordingly. The flag for resetting the observer is set.

BW500STA::Calc

void **Calc**(double *w*, double *Position*, double *Winkel*)

Parameters *w* is the setpoint of the ball position.

Position is the measured value of the ball position.

w is the measured value of the beam angle.

Description The function **Calc** is the main function of this class. It carries-out the calculation of the state controller. When the flag *start* is set, the initial states of the observer and the controller are reset. The missing state variables ball speed and beam angle speed are determined either by means of an observer or are calculated using difference quotients or are reset to 0.0 depending on the mode *state*. After limiting all the state variables the control signal is calculated by means of the state feedback vector. To compensate the effect of the ball friction an additional control signal is either calculated by means of a disturbance observer or taken as a positive or negative constant signal with respect to the current ball position or reset to 0.0 depending on the mode *dist* of the disturbance observer. After calculating the two observers for the missing state variables and the friction compensation the state variables as well as the control signals are stored in the measurement-vector *scopebuf*. The function returns the difference of the control signal and the additional control signal (for friction compensation).

Return The control signal with friction compensation of the state controller.

BW500STA::SetTa

void **SetTa**(float *ta*)

Parameters *ta* is the value of the real sampling period of the controller.

Description The function **SetTa** sets the internal sampling period of the state controller equal to the given value *ta*. This value has to be the same as the sampling period for the controller set by the TIMER16.DLL. Otherwise the calculation of the difference quotients fails.

BW500STA::Reset

void **Reset**(void)

Description The function **Reset** sets the flag *start* to reset the initial values of the disturbance observer and of the state controller.

BW500STA::Load

int **Load**(char* *name*)

Parameters *name* is a pointer to the name of a file from which the matrices of the state controller are to be read.

Description The function **Load** copies the given name to *filename* and tries to open the corresponding file. When the file cannot be opened, an error message is presented, the internal error *errors* is set to 1 and the function returns ERROR (-1) immediately. Otherwise all the parameters are read from the file and stored in the corresponding matrices of the state controller. The file must match a predefined format and the sequence of the parameters separated by comment lines with a closing "]" character. The internal error *errors* is reset to 0.

Return A value of 0 with a successful read operation from the file, else -1.

BW500STA::Save

int **Save**(char* *name*)

Parameters *name* is a pointer to the name of a file to which the matrices of the state controller are to be written.

Description The function **Save** copies the given name to *filename* and tries to open the corresponding file. When the file cannot be opened, an error message is presented, the internal error *errors* is set to 1 and the function returns ERROR (-1) immediately. Otherwise all the matrices of the state controller are written to the file with additional comment lines. The internal error *errors* is reset to 0.

Return A value of 0 with a successful write operation to the file, else -1.

BW500STA::SetStateObserver

void **SetStateObserver**(Observer *o*)

Parameters *o* is the new mode of the state observer.

Description The function **SetStateObserver** sets the mode of the state controller to determine the missing state variables ball speed as well as beam angle speed. These variables are either determined by means of a reduced-order state observer (ACTIVE) or calculated by difference quotients (CONSTANT) or reset to 0.0 (NONE).

BW500STA::SetDistObserver

void **SetDistObserver**(Observer *o*)

Parameters *o* is the new mode of the disturbance observer.

Description The function **SetDistObserver** sets the mode of the disturbance observer to determine the additional control signal to compensate the ball friction. This additional control signal is either determined by means of a disturbance observer (ACTIVE) or set to a positive or negative constant with respect to the current ball position (CONSTANT) or reset to 0.0 (NONE).

BW500STA::geterrors

int **geterrors**(void)

Description The function **geterrors** returns the value of the internal error *errors*. This variable is set during file accesses (see also **Load, Save**).

Return The value of the internal error *errors*.

BW500STA::GetV

double* **GetV**(void)

Description The function **GetV** returns a pointer to the parameter of the pre-filter.

Return A pointer to the parameter of the pre-filter.

BW500STA::GetFt

double* **GetFt**(void)

Description The function **GetFt** returns a pointer to the state feedback vector *ft*.

Return A pointer to the state feedback vector *ft*.

BW500STA::GetLBD

double* **GetLBD**(void)

Description The function **GetLBD** returns a pointer to the *L*-matrix of the reduced-order state observer.

Return A pointer to the *L*-matrix of the reduced-order state observer.

BW500STA::GetABD

double* **GetABD**(void)

Description The function **GetABD** returns a pointer to the *A*-matrix of the reduced-order state observer.

Return A pointer to the *A*-matrix of the reduced-order state observer.

BW500STA::GetFBD

double* **GetFBD**(void)

Description The function **GetFBD** returns a pointer to the *F*-matrix of the reduced-order state observer.

Return A pointer to the *F*-matrix of the reduced-order state observer.

BW500STA::GetBBD

double* **GetBBD**(void)

Description The function **GetBBD** returns a pointer to the *B*-matrix of the reduced-order state observer.

Return A pointer to the *B*-matrix of the reduced-order state observer.

BW500STA::GetDCON

double* **GetDCON**(void)

Description The function **GetDCON** returns a pointer to the parameter of the constant friction compensation.

Return A pointer to the value of the constant friction compensation.

BW500STA::GetDLBD

double* **GetDLBD**(void)

Description The function **GetDLBD** returns a pointer to the *L*-matrix of the disturbance observer (compensation of ball friction).

Return A pointer to the *L*-matrix of the disturbance observer.

BW500STA::GetDABD

double* **GetDABD**(void)

Description The function **GetDABD** returns a pointer to the *A*-matrix of the disturbance observer (compensation of ball friction).

Return A pointer to the *A*-matrix of the disturbance observer.

BW500STA::GetDFBD

double* **GetDFBD**(void)

Description The function **GetDFBD** returns a pointer to the *F*-matrix of the disturbance observer (compensation of ball friction).

Return A pointer to the *F*-matrix of the disturbance observer.

BW500STA::GetDBBD

double* **GetDBBD**(void)

Description The function **GetDBBD** returns a pointer to the *B*-matrix of the disturbance observer (compensation of ball friction).

Return A pointer to the *B*-matrix of the disturbance observer.

1.5.3 The Class BW502FUZ in the BWSERV16.DLL

The class **BW502FUZ** provides functions to apply a fuzzy controller. It serves as an interface to the fuzzy algorithms contained in **Fuzzy.lib**.

Public Data:

enum Observer	{NONE=0, CONSTANT, ACTIVE};

Private Data:

int *errors*	error counter for file access
Fuzzy **PosController*	fuzzy position controller
Fuzzy **AngController*	fuzzy beam angle controller
Fuzzy **PosObserver*	fuzzy position observer
Fuzzy **AngObserver*	fuzzy beam angle observer
Observer *posobserver*	mode of the fuzzy position observer
Observer *angobserver*	mode of the fuzzy beam angle observer
double *x[4]*	state vector
double *lastp*	control signal from previous sampling period
char *cname[MAXPATH]*	current "fuzzy-controller" file name
char *fname[4][MAXPATH]*	names of the fuzzy description files

BW502FUZ::BW502FUZ()

void **BW502FUZ** (void)

Description The constructor of the class **BW502FUZ** assigns NULL to all the pointers to fuzzy objects, reads the "fuzzy-controller" file DEFAULT.FBW (see also **SelectFuzzyFile**) and generates the accompanying fuzzy objects. The modes of the fuzzy position observer as well as of the fuzzy beam angle observer are set to ACTIVE, meaning that the additional control signals to compensate the effects of friction are determined by fuzzy observers. The control signal from the previous sampling period is reset.

BW502FUZ::~BW502FUZ()

void ~**BW502FUZ** (void)

Description The destructor of the class **BW502FUZ** removes all the fuzzy objects from the memory.

BW502FUZ::Calc

void **Calc**(double *w*, double *Position*, double *Winkel*)

Parameters *w* is the setpoint of the ball position.

Position is the measured value of the ball position.

w is the measured value of the beam angle position.

Description The function **Calc** is the main function of this class to calculate the fuzzy controller. When the internal error *errors* is unequal to zero, the function returns 0.0 immediately. The missing state variables ball speed and beam angle speed are calculated by difference quotients. After limiting all the state variables and storing the setpoint as well as the measured values to the measurement-vector *scopebuf* the control signal is calculated by means of a cascaded fuzzy controller/observer.

When the fuzzy position observer is activated an additional signal to compensate the ball friction is calculated by the fuzzy object with the two input signals beam angle and ball speed. Otherwise the additional signal is reset to 0.0. This additional signal and the output signal of the fuzzy object with the input signals position control error and ball speed produce the setpoint signal for the beam angle controller in the lower cascade.

The fuzzy object of the beam angle controller is driven by the two input signals, angle control error and the angular velocity of the beam. Its output signal is the control signal for the plant. When the fuzzy beam angle observer is activated a second additional signal to compensate the beam friction is calculated by the fuzzy object with the two input signals beam angular velocity and control signal from the previous sampling period. Otherwise the second additional signal is reset to 0.0. The two additional signals as well as the original control signal are stored to the measurement-vector *scopebuf*. The function returns the sum of the original control signal and the second additional signal.

Return The control signal with friction compensation of the fuzzy controller.

BW502FUZ::SelectFuzzyFile

double **SelectFuzzyFile**(char* *name*)

Parameters *name* is a pointer to the name of a "fuzzy-controller" file from which the names of the fuzzy description files and then the contents of these files are to be read.

Description The function **SelectFuzzyFile** at first searches for the extension "FBW" in the given file name. When this extension is missing, the internal error *errors* is set to 1 and the function returns -1.0 immediately. Otherwise the given name is stored to *cname* and after replacing its extension by "OUT" is taken as the output log file. If the file with the given name cannot be opened an error message is presented, the internal error *errors* is set to 1 and the function returns -1.0 immediately. Now in a sequence possibly existing fuzzy objects are deleted and fuzzy description files are read with a syntax check. In case of no errors the accompanying fuzzy objects are generated. If the generation was successful the mean calculation time for all of the fuzzy objects is determined and returned.

Return The mean calculation time of all fuzzy objects or -1.0 in case of an error.

BW502FUZ::Save

void **Save**(char* *name*)

Parameters *name* is a pointer to the name of a "fuzzy-controller" file to which the names of the fuzzy description files are to be written.

Description The function **Save** copies the given name to *cname*, when the internal error *errors* is not set. If the file with the given name can be opened the contents of *fnames*, that means the names of the currently used fuzzy description files are written to this file.

BW502FUZ::SetAngObserver

void **SetAngObserver**(Observer *o*)

Parameters *o* is the new mode of the fuzzy beam angle observer.

Description The function **SetAngObserver** determines the mode of the fuzzy beam angle observer to calculated the additional signal for the compensation of the beam friction. The additional signal is determined either by means of the fuzzy beam angle observer (ACTIVE) or is reset to 0.0 (NONE).

BW502FUZ::SetPosObserver

void **SetPosObserver**(Observer *o*)

Parameters *o* is the new mode of the fuzzy position observer.

Description The function **SetPosObserver** determines the mode of the fuzzy position observer to calculated the additional signal for the compensation of the ball friction. The additional signal is determined either by means of the fuzzy position angle observer (ACTIVE) or is reset to 0.0 (NONE).

BW502FUZ::getname

char* **getname**(void)

Description The function **getname** returns the name of the current "fuzzy-controller" file, meaning the content of *cname*.

Return A pointer to the name of the current "fuzzy-controller" file from *cname*.

BW502FUZ::getfname

char* **getfname**(int *i*)

Parameters *i* is the index of the fuzzy description file.

Description The function **getfname** returns the name of the current fuzzy description file with the index *i*, meaning the content of *fname[i]*.

Return A pointer to the name of the current fuzzy description file with the index *i* from *fname[i]*.

BW502FUZ::geterrors

int **geterrors**(void)

Description The function **geterrors** returns the value of the internal error *errors*. This variable is set during file accesses (see also **Load, Save**).

Return The value of the internal error *errors*.

1.5.4 The Class STOREBUF in the BWSERV16.DLL

The instance of the class **STOREBUF** realizes the function of data buffering. The data buffer created dynamically looks like a matrix with a maximum of assignable rows, where each row contains an adjustable number of components (i.e. float values from measurements). The storage in the data buffer is performed row by row, where each row is represented by a data vector, which was filled by another routine from an upper level. In this case it is the interrupt service routine which fills the data vector, i.e. with the setpoint value, measurements and control signals, in every sampling period. An element function (**StartMeasure**) of **STOREBUF** starts and controls the storage (**WriteValue**) of this data vector in the data buffer. With respect to the measuring time at first those sampling periods are determined in which storage is to be performed (number of store operations * sampling periods = measuring time). Where the number of store operations is calculated at first such that it is always less than the maximum number of measurement vectors (= number of rows of the memory matrix). At the end of the measuring time the store operation is terminated in case no additional trigger conditions are set. In case of an activated trigger condition, a signal crosses a given value with a selected direction, the store operation is continued until the end of the measuring time after the trigger condition was met. In case the signal does not meet the trigger condition, the store operation is performed endless in a ring until the user interactively terminates this operation. In addition a time before the trigger condition (prestore time) is adjustable in which storage in the data buffer is performed. The time after the trigger condition is met is then the measuring time reduced by the prestore time. The mentioned data vector will be named measurement-vector in the following.

Private Data:

float *taplt* is the sampling period of the interrupt service routine.

int *trigger_channel* is the channel (index) of the measurement-vector used for triggering.

int *startmessung* flag for starting new measuring.

int *gomessung* flag for measuring is started.

int *storedelay* is the number of sampling periods in between the storage of values.

int *storedelayi* is the counter for *storedelay*.

int *MaxVectors* is the maximum number of storable measurement-vectors.

int *anzahl* is the number of stored measurement-vectors.

int *anzahli* is the counter for *anzahl*.

int *stopmeasureindex* is the index for normal end of measuring.

int *triggerindex* is the trigger index..

int *prestoreoffset* is the number of stored measurement-vectors previous to the trigger.

int *nchannel* is the number of float values in the measurement-vector.

int *outchannel* is the channel (index) of the component of the measurement-vector, which is to be read (for output).

int *bufindex* is an internal index for the next storage location in the data buffer.

int *trigged* flag for trigger condition is met.

int *stored_values* number of measurement-vector storages since the start of the measuring.

float *trigger_value* trigger float value.

float **fptr* is a pointer to the start address of the dynamic data buffer.

float **sourceptr* is a pointer to the measurement-vector.

float **inptr* is a pointer to the actual data buffer location.

int *aktiv* flag for status of the dynamic data buffer.

int *status* flag for storage control.

STOREBUF::ResetBufIndex

void **ResetBufIndex**(void)

Description The private element function **ResetBufIndex** sets *bufindex* to 0 and *inptr* equal to *fptr* meaning that the start conditions for the data buffer are set.

STOREBUF::STOREBUF

STOREBUF(int *nchannel*, float **indata*, int *maxvectors*)

Parameters int *nchannel* is the number of float values of the external measurement-vector.

float **indata* is the pointer to the start address of the measurement-vector.

int *maxvectors* is the maximum number of measurement-vectors.

Description The constructor of this class initializes flags (*gomessung*, *startmessung*, *aktiv* = FALSE) to control the storage as well as a pointer to the measurement vector (*sourceptr* = *indata*. The maximum number of the measurement-vectors is set (*MaxVectors* = *maxvectors*) where the minimum value is limited to 1.

STOREBUF::~STOREBUF()

void ~**STOREBUF**(void)

Description The destructor of this class frees the dynamically allocated memory *fptr* in case it was created.

STOREBUF::StartMeasure

void **StartMeasure**(float *meastime*, float *triggervalue*, float *prestoretime*, int *triggerdir*, float *taint*)

Parameters *triggerchannel* is the number of the trigger channel.

meastime is the measuring time in seconds.

triggervalue is the trigger level of the trigger channel.

prestoretime is the time of storage previous to the trigger (in sec).

triggerdir is the flag for direction (below/above) of the trigger condition.

taint is the sampling period of the interrupt service routine.

Description The function **StartMeasure** initializes a new storage operation. To a maximum of *maxvectors* measurement-vectors are stored. In case the adjusted measuring time *meastime* is longer than *maxvectors* * *taint* (sampling period) the number of interrupt executions without data storage is calculated. The arguments of this function are all the parameters required for the storage.

STOREBUF::WriteValue

void **WriteValue**(void)

Description The function **WriteValue** stores *nchannel* float values from the array *indata* (measurement-vector) to the current address of the dynamically allocated array.

STOREBUF::SetOutChan

void **SetOutChan**(int *in*)

Parameters int *in* references the component of the measurement vector which is to be read (output).

Description The inline function **SetOutChannel** sets the channel number (index in the measurement-vector) of the signal which is to be returned by the function **ReadValue**.

STOREBUF::ReadValue

float **ReadValue**(void)

Description The function **ReadValue** returns the value of the next storage location belonging to the channel selected by **SetOutChannel**.

Return Value (float) read from measurement-vector.

STOREBUF::GetBufLen

int **GetBufLen**(void)

Description The function **GetBufLen** interrupts a current storage operation and returns the number of stored measurement-vectors.

Return Number (int) of stored measurement-vectors.

STOREBUF::GetBufTa

float **GetBufTa**(void)

Description The inline function **GetBufTa** returns the time between storage, which was calculated with respect to the measuring time and the sampling period.

Return Time (float) between storage depending on measuring time and sampling period.

STOREBUF::GetStatus

int **GetStatus**(void)

Description The function **GetStatus** returns the status of the store operation.

Return Status (int) of store operation

0	not initialized
1	storage before trigger
2	waiting for trigger condition
4	storage operation
5	storage complete
6	storage interrupted

STOREBUF::GetBufferLevel

double **GetBufferLevel**(void)

Description The function **GetBufferLevel** returns the percentage of the former measurement time with respect to the given trigger condition (= filling ratio or level of the data buffer). The return value will stay at 0% until the valid trigger condition is reached even when *prestoretime* is unequal to zero. That means the return value will start with an initial value of *prestoretime / meastime* in % at the time of a valid trigger condition.

Return The percentage of the filling ratio (double) of the data buffer.

1.5.5 The Class AFBUF in the BWSERV16.DLL

An instance of the class **AFBUF** is an object that creates dynamically a data array for an assignable number of float values. Data can be stored in this array and can be read afterwards when the data array is filled completely. The array is handled like a ring buffer.

Private data:

float ***fptr* is the pointer to the start of the dynamically created data array.

float **inptr* is the pointer to the current storage location ready to store a value (input).

float **outptr* is the pointer to the current storage location ready to read a value (output).

int *aktiv* flag: dynamic memory is initialized.

int *filled* flag: data array is filled.

int *abuflen* is the number of float values in the data array.

int *inbufindex* is the index of the current input position.

int *outbufindex* is the index for the current output position.

AFBUF::AFBUF()

void **AFBUF**(void)

Description The constructor of this class resets the flag *aktiv*, which indicates a dynamically created data array.

AFBUF::~AFBUF()

void ~**AFBUF**(void)

Description The destructor of this class frees the initialized data memory in case it was created dynamically.

AFBUF::NewFBuf

int **NewFBuf**(int *anzahl*)

Parameters *anzahl* is the size of the data array in float values.

Description The function **NewFBuf** initializes a data array with *anzahl* float values. A value of 1 is returned after a successful initialization, otherwise 0 is returned.

Return Status (int) of data array:
= 0, data array is not initialized,
= 1, data array is initialized.

AFBUF::ReadFBuf

float **ReadFBuf**(void)

Description The function **ReadFBuf** returns the float value of the next storage location of the dynamically created data array in case this array was filled previously.

Return Value (float) from data array.

AFBUF::WriteFBuf

int **WriteFBuf**(float *fvalue*)

Parameters float *fvalue* is the value, which is to be stored.

Description The function **WriteFBuf** stores the float value to the storage location.

Return Number (int) of stored values (=0 in case no data array initialized).

1.5.6 The Class TWOBUFFER in the BWSERV16.DLL

The class **TWOBUFFER** handles two instances of the class **AFBUF**. One instance (write-instance) can be used to store data while the other is used to read out data (read-instance). In case the data array of the write-instance is filled it is handled as a read-instance in the following. This condition guarantees that the interrupt service routine has always access to valid data.

Private objects:

AFBUF *Buf1* is an instance of the class **AFBUF**

AFBUF *Buf2* is an instance of the class **AFBUF**

Private data:

int *readbuffer* flag: data array is ready for read operation.

int *buffer1* flag: 0 = *Buf1* write,
1 = *Buf1* read.

int *buffer2* flag: 0 = *Buf2* write,
1 = *Buf2* read.

int *newbuffer* flag: 0 = not a new output buffer,
1 = *Buf1* is a new output buffer,
2 = *Buf2* is a new output buffer.

int *buf1len* length of the data array of the instance *Buf1*

int *buf1leni* index of the instance *Buf1*.

int *buf2len* length of the data array of the instance *Buf2*.

int *buf2leni* index of the instance *Buf2*.

int *inbufindex* index for input data array.

int *repw1* number of repeated values in *Buf1*

int *repw1i* index of repeated values in *Buf1*

int *repw2* number of repeated values in *Buf2*

int *repw2i* index of repeated values in *Buf2*

int *repb1* number of data array outputs of the instance *Buf1*.

int *repb1i* index of the array outputs of the instance *Buf1*.

int *repb2* number of data array outputs of the instance *Buf2*.

int *repb2i* index of the array outputs of the instance *Buf2*.

TWOBUFFER::TWOBUFFER()

void **TWOBUFFER**(void)

Description The constructor of this class initializes flags and counters as follows:

readbuffer = FALSE, buffer cannot be read,
buf1len = 1, length of the buffer *Buf1*,
buf2len = 1, length of the buffer *Buf2*,
buffer1 = 1, buffer *Buf1* for read operation,
buffer2 = 0, buffer *Buf2* for write operation,
newbuffer = 0, no buffer for read or write operation available.

TWOBUFFER::New2Buffer

void **New2Buffer**(int *anzahl* , int *repeatwert*, int *repeatbuf*)

Parameters int *anzahl* is the number of float values of the new array.

int *repeatwert* defines how often a value is to be repeated during a read operation by **Read2Buffer**.

int *repeatbuf* defines how often the array is to be sent to the output.

Description The function **New2Buffer** creates data arrays dynamically with *anzahl* float values. With *buffer1* = 0 *Buf1* is created and with *buffer2* = 0 *Buf2* is created.

TWOBUFFER::Write2Buffer

int **Write2Buffer**(float *wert*)

Parameters float *wert* is the value, which is to be stored.

Description The function **Write2Buffer** writes the argument value to the data array. In case the end of the array is reached, the array is used as a source for the function **Read2Buffer**.

Return Total number (int) of stored (written) values.

TWOBUFFER::Read2Buffer

float **Read2Buffer**(void)

Description The function **Read2Buffer** returns the values of the read-array handling like a ring. In case the argument *repeatbuf* of the function **New2Buffer** was equal to *x*, the array is read *x* times. After *x* read operations zero is returned. In case *repeatbuf* is equal to 0, the read operation is cyclic.

Return Value (float), which is read from the array.

1.5.7 The Class Signal in the BWSERV16.DLL

An instance of the class **SIGNAL** is an object to create a data array, which represents a given signal shape in case it is read out with constant time intervals. To do this an instance of the class **TWOBUFFER** is used. Adjustable signal shapes are rectangle, triangle, sawtooth and sine. In addition the amplitude, an offset and the time period is adjustable.

Private Data:

float *abtastzeit* sampling period to read out values.
float *stuetzst* number of base points of a signal period.
float *signaloffset* offset of the signal.
float *signalamplitude* amplitude of the signal.
float *minrange* minimum available return value.
float *maxrange* maximum available return value.

Private objects:

TWOBUFFER *sign* is an instance of the class **TWOBUFFER**

SIGNAL::SIGNAL( )

void **SIGNAL**(void)

Description The constructor of this class initializes the variables *abtastzeit*, *minrange* and *maxrange*.

SIGNAL::InitTime

float **InitTime**(float *settime*)

Parameters float *settime* is the sampling period of the read routine (in sec.)

Description The function **InitTime** sets the sampling time, which is used to read out the values from the interrupt routine, equal to the given controller sampling period.

Return Adjusted sampling period (float) in seconds.

SIGNAL::MakeSignal

int **MakeSignal**(int *form*, float *offset*, float *ampl* ,float *periode* , int *repeatbuf*)

Parameters

int *form* is the signal shape indicator
konstform (constant) 0
rectform (rectangle) 1
triform (triangle) 2
saegeform (sawtooth) 3
sinusform (sine) 4

float *offset* offset value of the signal.

float *ampl* amplitude of the signal.

float *periode* period of the signal (in sec).

int *repeatbuf* defines how often the signal is to be read out (0 = continuously).

Description

The function **MakeSignal** The function **MakeSignal** generates a data array with a maximum of 1024 float values, in which the values of the selected signal shape are stored. The signal shape is adjusted by the argument *form*. The absolute value of the signal f(t) is given by the sum *offset* + *amplitude* * f(t). In case the number of base points determined by the division *periode* / sampling period is greater than 1024 the number of base points is halved and the repeat value *repw1* or *repw2* is doubled until the number is less than 1024.

After the generation of a data array it is assigned as a source to the function **ReadNextValue** (see class **TWOBUFFER**).

Return

Error status:
=0, no error
=1, illegal signal shape.

SIGNAL::ReadNextValue

float **ReadNextValue**(void)

Description

The function **ReadNextValue** reads the data from the assigned array. The value is internally limited to the range *minrange* to *maxrange*. It is called by the interrupt service routine. Due to the locking mechanism in **TWOBUFFER**, new signal shapes can be created even in case the active interrupt outputs another one.

Return

Value (float) read from the data array.

SIGNAL::SetRange

void **SetRange**(float *min*, float *max*)

Parameters float *min* is the minimum return value of the function **ReadNextValue**.

float *max* is the maximum return value of the function **ReadNextValue**.

Description The function **SetRange** adjusts the range of the base points forming the signal, i.e. the minimum and maximum values returned by the function **ReadNextValue**.

SIGNAL::WriteBuffer

void **WriteBuffer**(float *value*)

Parameters float *value* is the value which has to be stored.

Description The private element function **WriteBuffer** writes the argument *value* to the data array of the instance **TWOBUFFER**.

SIGNAL::Stuetzstellen

int **Stuetzstellen**(float *Periodenzeit*, int *form*)

Parameters float *Periodenzeit* is the time period of the signal.

int *form* is the indicator for the adjusted signal shape.

Description The private element function **Stuetzstellen** calculates the number of base points and with this the length of the data arrays of the instance **TWOBUFFER** depending on the time period and the signal shape. The number of the base points is determined by the division *Periodenzeit* / sampling period. In case of a constant signal shape the minimum number of base points is 1.

Return Calculated number (int) of base points.

1.5.8 The DLL Interface PLOT

Included in the BWSERV16.DLL, the functions of the file PLOT.CPP provide the interfaces for graphic output of measured data and for displaying information about the contents of documentation files (*.PLD).

Global Data:

HWND *handlelist*[100] is an array to store the handles of plot windows.

PROJECT *project* is a structure with data for the project identification.

CTRLSTATUS *measctrlstatus* is a structure containing the controller state, controller parameters as well as the measuring time at the time a measuring is started.

CTRLSTATUS *ctrlstatus* is a structure containing the controller state, controller parameters as well as the measuring time at the time a controller is started.

DATASTRUCT *datastruct* is structure containing the number of measurement-vectors, the number of its components as well as the sampling period of a measuring.

char *FileName*[60] is a string containing the name of a documentation file (*.PLD).

double ***ppData* is a pointer to a buffer containing measurements loaded from a documentation file (*.PLD).

int *NumberOfCurvesInChannel* is the number of curves of a plot depending on the "plot channels" (= selected groups of components of the measurement vector).

int *ChannelToScope* is the relation between curves (index) of the measurement buffer *scope* or the pointer ***ppData* and the "plot channels" (= selected groups of components of the measurement vector).

char **ScopeNames* contains the curve descriptions (strings) for the linestyle table of the plot.

char **TitleNames* contains the drawing titles for different "plot channels".

char **YAxisNames* contains the description of the Y-axis for different "plot channels".

char **XAxisName* contains the description of the X-axis for different "plot channels".

ReadPlot

int CALLBACK **ReadPlot**(char ***lpfName*)

Parameters **lpfName* is a pointer to the name of a documentation file, from which measurements are to be read.

Description The function **ReadPlot** reads the structures *project*, *ctrlstatus* and *datastruct* as well as the measurements from the documentation file with the given name *lpfName* and stores the measurements to a new global data array pointed to by ***ppData*. Up to 59 characters of the file name *lpfName* are copied to the global file name *FileName*.

Return The state of the file access:
=0, measurements read successfully,
=-1, file with the given name could not be opened,
=-2, the PROJECT structure from the file contains a wrong project number.

WritePlot

int CALLBACK **WritePlot**(char ***lpfName*)

Parameters **lpfName* is a pointer to the name of a documentation file, to which measurements are to be written.

Description The function **WritePlot** writes the global structures *project*, *measctrlstatus*, the local structure DATASTRUCT *mydatastruct* as well as the content of the global measurement buffer *scope* to a documentation file with the given name *lpfName*. The local structure *mydatastruct* contains the number of measurement-vectors, the number of its components as well as the sampling period of a measuring.

Return The state of the file access:
=0, measurements written successfully,
=-1, file with the given name could not be created.

Plot

int CALLBACK **Plot**(int *command*, int *channel*)

Parameters

command defines the data source:
=1, data from the global measurement buffer *scope*,
=2, data from the global array ***ppData*

channel defines the curves related to "plot channels":
=0, Measured position with setpoint [m] (2 curves),
=1, Measured position [m] (1 curve),
=2, Measured angle [rad] (1 curve),
=3, Controller output [N] (1 curve),
=4, Measured ball speed [m/s] (1 curve),
=5, Measured angle speed [rad/s] (1 curve),
=6, Beam friction comp. [N] (1 curve),
=7, Ball friction comp. (Fuz.)[N] (1 curve)

Description

The function **Plot** represents the curves specified by *channel* with accompanying descriptions in a graphic inside a plot window. The data sources are the global measurement buffer *scope* or the global array ***ppData* depending on the parameter *command.*

Return

The state of the graphic output:
=0, successful graphic output of measured curves,
=-1, invalid values for *command*,
=-2, invalid values for *channel*,
=-3, length of the global array ***ppData* is 0,
=-4, length of the global measurement buffer *scope* is 0.

See also

CreateSimplePlotWindow, SetCurveMode, AddAxisPlotWindow, AddXData, AddPlotTitle, ShowPlotWindow.

GetPlot

int CALLBACK **GetPlot**(int *start*, char **lpzName*)

Parameters *start* is a flag indicating the first plot window.

**lpzName* is a pointer to the title of the plot window, the Windows handle of which was found.

Description The function **GetPlot** determines the Windows handle of an existing plot window referenced by a local index *index*. The Windows handle is copied to the global list *handlelist* and the index is incremented. If the Windows handle is unequal to 0 up to 60 characters of the title of the corresponding plot window are copied to *lpzName*. With *start*=TRUE the Windows handle of the plot window with index=0 is determined.

Return The state of the handle determination:
=0, handle = 0, plot window with current index could not be found,
=1, handle determined for current index, title copied.

See also **GetValidPlotHandle**.

PrintPlot

int CALLBACK **PrintPlot**(int *idx*, HDC *dcPrint*, int *iyOffset*)

Parameters *idx* is the index for the global list of handles referencing existing plot windows.

dcPrint is the device context of the output device.

iyOffset is the beginning of the printout in vertical direction as a distance in [mm] from the upper margin of a page.

Description The function **PrintPlot** prints the content of the plot window with the Windows handle from the global list *handlelist*[*idx*] to the device with the device context *dcPrint*. The printout has a width of 180 mm and a height of 140 mm. It is located at the left margin with a distance of *iyOffset* mm from the upper margin of a (i.e. DIN A4) page.

Return Is always equal to 0.

See also **PrintPlotWindow**.

GetPldInfo

int CALLBACK **GetPldInfo**(int &*controller*, char ***s*, int &*n*, int &*c*, double &*d*)

Parameters &*controller* is a reference to the controller structure (state controller, fuzzy controller, none).

***s* is a (double) pointer to the string containing date and time of the measuring.

&*n* is a reference to the number of samples of each measured signal (curve).

&*c* is a reference to the number of measured signals.

&*d* is a reference to the sampling period of the measuring.

Description The function **GetPldInfo** reads selected elements of the structures *ctrlstatus* as well as *datastruct* and stores these elements to the mentioned parameter references. It is assumed that the structures were filled previously with data from a loaded documentation file (*.PLD).

Return Is the result:
=0, the structure elements have been copied,
=-1, the global data array ***ppData* does not exist, length = 0.

See also **ReadPlot**.

2 Driver Functions for BW500

2.1 The Class DICDRV

The class **DICDRV** provides the interface between the BW500 controller program and the driver functions of the PC plug-in card. The class **WDAC98** containing the driver functions is the basic class of **DICDRV**. In addition this class contains the mathematical model of the ball and beam system when it is compiled with '#define __SIMULATION__' (see file BWDEFINE.H). With this all program functions except for the calibration can be carried out for a simulated ball and beam system. The PC plug-in card is no longer required in this case. This program version will be called 'DEMO-Version' in the following.

Basic Class:

WDAC98 driver functions of the PC adapter card (file WDAC98.CPP)

The files DICDRV.H, DICDRV.CPP contain the class **DICDRV** with the functions:

DICDRV(void)

~**DICDRV**(){}

double **ReadWinkel**(void)

double **ReadPosition**(void)

void **SetKraft**(double *n*)

int **ReadCamera**(void)

double **CorrectPos**(double *p*, double *w*)

void **getxcenter**(void)

void **getwcenter**(void)

int **eichok**(void)

int **CheckSystem**(void)

int **CheckFree**(void)

int **LeftSwitch**(void)

int **RightSwitch**(void)

virtual void **StartInterrupt**(void)

virtual void **TriggerEndstufe**(void)

void **LinModell**(void)

Public Data:

double *xeich* is the zero-position of the ball in [mm].

short *xcenter* is the camera signal for the zero-position of the ball.

short *IncOffset* is the incremental encoder signal for the zero-angle of the beam.

short *cam* is the current camera signal.

short *ang* is the current incremental encoder signal.

double *angle* is the beam angle of the mathematical model.

double *position* is the ball position of the mathematical model.

double *kraft* is the driving force for the beam of the model.

DICDRV::DICDRV

DICDRV(void)

Description The constructor of the class **DICDRV** initializes an object of the class **WDAC98** and sets initial values for the calibration data *IncOffset, xeich, xcenter*. The two analog outputs are reset to 0 and the camera is enabled. When the file DEFAULT.CAL exists, the calibration data are read from this file. With the DEMO-version initial values are set for the ball position and the beam angle.

DICDRV::~DICDRV

~DICDRV(void)

Description The destructor of the class **DICDRV** resets the real control signal to 0 and writes the current calibration data to the file DEFAULT.CAL.

DICDRV::ReadWinkel

double **ReadWinkel**(void)

Description The function **ReadWinkel** reads the incremental encoder signal to measure the beam angle, stores this value to *ang* and returns its conversion to an angle in [rad].

With the DEMO-version the beam angle *angle* calculated by the mathematical model is returned.

Return The beam angle in [rad].

DICDRV::ReadPosition

double **ReadPosition**(void)

Description The function **ReadPosition** reads the camera signal (**ReadCamera**) to measure the ball position, converts this value to a ball position in [m] with respect to the zero-position in the middle of the beam and returns the converted value. With *xeich* greater than 400.0 the return value is always 0.0.

With the DEMO-version the ball position *position* calculated by the mathematical model is returned.

Return The ball position in [m].

DICDRV::SetKraft

void **SetKraft**(double *n*)

Parameters *n* is the driving force for the beam in [N].

Description The function **SetKraft** calculates the control signal in [Volt] required for the given driving force *n* in [N] for beam drive. This control signal is limited to +/-3V and transferred to the D/A-converter.

The D/A-conversion is omitted for the DEMO-version. The given driving force *n* in [N] is assigned to the variable *kraft* and the mathematical model of the ball and beam system is calculated (**LinModell**).

DICDRV::ReadCamera

int **ReadCamera**(void)

Description The function **ReadCamera** controls the camera such that at first the upper 4 bits and after a delay time the lower 8 bits of the camera signal are readable. Then the camera is enabled for another picture measurement. The complete camera signal is stored in *cam* and returned.

Return The camera signal (12 bits).

DICDRV::CorrectPos

double **CorrectPos**(double *p*, double *w*)

Parameters *p* is the measured ball position in [m].

w is the measured beam angle in [rad].

Description The function **CorrectPos** corrects the parallactic error of the position measurement with respect to a beam angle unequal to zero. A filter procedure is applied on the ball position in addition. When the current ball position differs too much from the previous measurement (runaway), this measurement is taken as a valid ball position as long as this case does not happen the 10th time (continuous error). The corrected and filtered ball position is returned.

Return The corrected and filtered ball position in [m].

DICDRV::getxcenter

void **getxcenter**(void)

Description The function **getxcenter** resets the control signal for the beam drive to 0 and takes the current camera signal (**ReadCamera**) as a valid value for the zero-position *xcenter* of the ball.

DICDRV::getwcenter

void **getwcenter**(void)

Description The function **getwcenter** takes the current incremental encoder signal (**ReadDDM**) as a valid value for the zero-angle *IncOffset* of the beam.

DICDRV::eichok

int **eichok**(void)

Description The function **eichok** checks if the current calibration data for the position measurement are inside of expected ranges. When *xeich* is not inside the range 500.0 to 900.0, its value is set to 700.0 and when *xcenter* is outside of the range 300 to 800, its value is set to 500. If one of the conditions failed the return value is 0, else it is 1.

Return A value of 1 for valid values of the calibration data, else 0.

DICDRV::CheckSystem

int **CheckSystem**(void)

Description	The function **CheckSystem** checks if the two limit switches (StopLeft, StopRight) indicating a maximum beam angle are not active at the same time (camera is connected) and if the system state (LEDReady) is equal to 0. When one of the conditions fails (system lead not connected or defect?) the return value is 0, else it is 1.
Return	A value of 1 for a positive system check, else 0.

DICDRV::CheckFree

int **CheckFree**(void)

Description	The function **CheckFree** returns a value of 1, when the control by the PC (PCREADY) is enabled, else it returns 0.
Return	Enable state of the PC control (0/1).

DICDRV::LeftSwitch

int **LeftSwitch**(void)

Description	The function **LeftSwitch** returns a value of zero, when the left limit switch for a maximum beam angle is activated, else it returns 1.
Return	A value of 0 for an activated left limit switch, else 1.

DICDRV::RightSwitch

int **RightSwitch**(void)

Description	The function **RightSwitch** returns a value of zero, when the right limit switch for a maximum beam angle is activated, else it returns 1.
Return	A value of 0 for an activated right limit switch, else 1.

DICDRV::StartInterrupt

virtual void **StartInterrupt**(void)

Description The function **StartInterrupt** activates the output stage release by sending a trigger pulse and starting a rectangle signal. Any interrupt is left unchanged.

DICDRV::TriggerEndstufe

virtual void **TriggerEndstufe**(void)

Description The function **TriggerEndstufe** toggles the level of the rectangle signal for the output stage release.

DICDRV::LinModell

virtual void **LinModell**(void)

Description The function **LinModell** calculates the linearized state space model of the ball and beam system with the DEMO-version.

2.2 The Class WDAC98

The class **WDAC98** realizes the interface between the class DICDRV and the driver functions (DIC24.DRV, DAC98.DRV) of the PC adapter card. Calling the DRV-functions is carried out by "SendMessage"-functions using commands and parameters as described with the driver software (see also IODRVCMD.H).

The files WDAC98.CPP and WDAC98.H contain the class **WDAC98** with the functions:

double **ReadAnalogVolt**(int *channel*)

void **WriteAnalogVolt**(int *channel*, double *val*)

int **ReadDigital**(int *channel*)

void **WriteDigital**(int *channel*, int *value*)

unsigned int **GetCounter**(void)

unsigned long **GetTimer**(void)

unsigned int **ReadDDM**(int *channel*)

void **ResetDDM**(int *channel*)

void **ReadAllDDM**(unsigned int &*cnt0*, unsigned int &*cnt1*, unsigned int &*cnt2*)

void **ResetAllDDM**(void)

WDAC98::ReadAnalogVolt

double **ReadAnalogVolt**(int *channel*)

Parameters *channel* is the number of the analog input channel, which is to be read.

Description The function **ReadAnalogVolt** reads the analog input channel specified by *channel* and returns the corresponding voltage value. The value is in the range from -10.0 to +10.0 with the assumed unit [Volt].

Return: The input voltage of the analog channel in the range from -10.0 to +10.0.

WDAC98::WriteAnalogVolt

void **WriteAnalogVolt**(int *channel*, double *val*)

Parameters *channel* is the number of the analog output channel, to which a value is to be written.

val is the value for the analog output.

Description The function **WriteAnalogVolt** writes the value *val* in the range from -10.0 to +10.0 (with the assumed unit [Volt]) as an analog voltage to the specified analog output channel. Values outside of the mentioned range are limited internally.

WDAC98::ReadDigital

int **ReadDigital**(int *channel*)

Parameters *channel* is the number of the digital input channel, which is to be read.

Description The function **ReadDigital** reads the state (0 or 1) of the specified digital input channel and returns this value.

Return: The state (0 or 1) of the specified digital input.

WDAC98::WriteDgital

void **WriteDgital**(int *channel*, int *val*)

Parameters *channel* is the number of the digital output channel, to which a value is to be written.

value is the new state of the digital output.

Description The function **WriteDgital** writes the value *val* (0 or 1) to the specified digital output channel and with this sets its state.

WDAC98::GetCounter

unsigned int **GetCounter**(void)

Description The function **GetCounter** returns the content of 16-bit-counter register.

Return: The content of the 16-bit-counter register.

WDAC98::GetTimer

unsigned long **GetTimer**(void)

Description The function **GetTimer** returns the content of the 32-bit-timer register.

Return: The content of the 32-bit-timer register.

WDAC98::ReadDDM

unsigned int **ReadDDM**(int *channel*)

Parameters *channel* is the number of the DDM device, which is to be read.

Description The function **ReadDDM** returns the content of the counter register of the specified DDM device (incremental encoder).

Return: The content of the specified DDM counter register.

WDAC98::ResetDDM

unsigned int **ResetDDM**(int *channel*)

Parameters *channel* is the number of the DDM device, which is to be reset.

Description The function **ResetDDM** resets the content of the counter register of the specified DDM device (incremental encoder).

WDAC98::ReadAllDDM

void **ReadAllDDM**(unsigned int &*cnt0*, unsigned int &*cnt1*, unsigned int &*cnt2*)

Parameters &*cnt0* is a reference to the content of the counter register of the DDM device No. 0, which is to be read.

&*cnt1* is a reference to the content of the counter register of the DDM device No. 1, which is to be read.

&*cnt2* is a reference to the content of the counter register of the DDM device No. 2, which is to be read.

Description The function **ReadAllDDM** should read the contents of the counter registers of the DDM devices 0, 1 and 2 at the same time and return the values by references.

This function is still not realized but reserved for future applications.

WDAC98::ResetAllDDM

void **ResetAllDDM**(void)

Description The function **ResetAllDDM** resets the contents of the counter register of all DDM devices (incremental encoders) at the same time.

3 The Fuzzy Library

3.1 Introduction to the Structure of the Fuzzy Library

The fuzzy library **Fuzzy.lib** is constructed with a strict hierarchical structure. Since an object oriented programming language supports this, the library was programmed using the programming language "C++".

As described in "Backgrounds of the Fuzzy Controller", the fuzzy set is the lowest level of this hierarchy. A separate class with the name **FuzzySet** was defined for the fuzzy set. Since nearly all run time operations use the class **FuzzySet**, the design was carried out with respect to the optimization of the run time and a definition range as wide as possible. As these aspects compete with each other, a compromise had to be found between run time and flexibility. The number representation "double" was chosen, because this is supported directly by the arithmetic coprocessors. But it is recommended to use a 486DX computer, which has a good performance even with this number representation. Without an arithmetic coprocessor the fuzzy library can only be applied to slow processes or off-line calculations, e.g. of a lookup table. The number representation "double" provides a nearly unlimited definition range for the fuzzy sets. The definition range of a fuzzy set should be between 0 and 1 to achieve a good overall view, but the correct function of the library does not require this range. The class **FuzzySet** represents a fuzzy set by a polygonal line. This polygonal line is stored in form of a corresponding number of x/y values. An object of type **FuzzySet** could hold theoretically up to 32768 of those values. But this number will never be reached, since the available memory is limited. The next level in the hierarchy of the fuzzy library is represented by the linguistic variable. This variable is included in the class **FuzzyVar**. A linguistic variable combines a group of fuzzy sets which have the same definition range. The purpose of the class **FuzzyVar** is to prepare and handle its data elements of the type **FuzzySet**. The x/y values of the objects of type **FuzzySet** of the class **FuzzyVar** are expanded automatically so that all of the fuzzy sets contain the same number of x/y values (normalizing of the sets). The x co-ordinates of the x/y values are identical. With respect to the run time it is therefore meaningful to use as few x/y values as possible (usually 3-5 are sufficient) and to use the same x co-ordinates in the x/y values of sets which are grouped to one linguistic variable.

The fuzzy rules are built by object types of the class **FuzzyRule**. The class **FuzzyRule** combines the input and output linguistic variables together with their sets with respect to the syntax of a fuzzy rule. A separate object has to be generated for every rule. The class **FuzzyRule** includes functions to interpret the rules.

The class **Fuzzy** holds the top of the hierarchy of the fuzzy library. Functions of this class are able to read a fuzzy description file, to detect syntax errors and to a certain extent logical errors, to generate an executable rule base by means the above mentioned classes and to compute the mean run time for this base. The class **Fuzzy** is the only one of the mentioned classes of the library, which the user calls in his program. An object of the type **Fuzzy** is a complete rule base, which is configured by a fuzzy description file. It is problem-free to handle multiple objects of type **Fuzzy**, which are stored together in the memory. After reading the fuzzy description file it is recommended to check by means of corresponding functions if errors occurred during the read and interpret operations. In case of no error a fuzzy control base can then be generated. A built-in function for computing the mean run time of the controller in one sampling period should be called before using this control base. The controller run time strongly depends on the rule base, the number of linguistic variables and the used computer system. The control base should only be called from an interrupt service routine in case the run time is about 50% shorter than the sampling period (time between two interrupts).

3.2 Description of the Classes

3.2.1 General

The library **Fuzzy.lib** at hand is compiled using the Borland C++ compiler version 4.2. The compiler switches code optimization for the 386 processor as well as 16 bit, large memory model, were set.

3.2.2 Overview of the Classes

class **FuzzySet**

FuzzySet(char **name*, int *p*)
FuzzySet(char **name*, int *p*, double **x*)
FuzzySet(char **name*, int *p*, double **x*, double **y*)
FuzzySet(const FuzzySet& org)
~FuzzySet()
char ***getname**(void)
void **cleary**(void)
double ***getxvector**(void)
double ***getyvector**(void)
int **getstuetzen**(void)
void **insert**(double *x*, double *y*)
void **normalize**(int *p*, double **x*)
double **coa**(void)
double **crisp**(double *x*)
void **conclude**(FuzzySet **a*, double *weigh*t)
FuzzySet& operator = (const FuzzySet& *org*)
FuzzySet& operator *= (double *factor*)
FuzzySet& operator += (const FuzzySet& *org*)
FuzzySet& operator << (ostream& *o*, const FuzzySet& *s*)
void **tout**(void)

class **FuzzyVar**

FuzzyVar(char **name*, int *c*, int *m*)
~FuzzyVar()
char ***getname**(void)
char ***getsetname**(int *i*)
int **getsetcount**(void)
void **add**(int *index*, FuzzySet **set*)
void **check**(void)
void **norm**(void)
int **getmode**(void)
double **getmaxx**(void)
double **getminx**(void)
double **get**(int *SetNo*)
void **set**(int *SetNo*, double *weight*)

void **clear**(void)
double **out**(void)
double **getval**(void)
void **setval**(double *v*)
void **tout**(void)
int **vsort**(int *c*, double **x*)
FuzzyVar& operator << (ostream& *o*, const FuzzyVar& *v*)

class **FuzzyRule**

FuzzyRule(char **name*, int *i*, int *o*)
~FuzzyRule()
char ***getname**(void)
void **addIn**(FuzzyVar **inv*, int *set*, int *op*)
void **addOut**(FuzzyVar **outv*, int *set*)
int **Do**(void)
void **tout**(void)
FuzzyRule& operator << (ostream& *o*, const FuzzyRule& *r*)

class **Fuzzy**

Fuzzy()
Fuzzy(char *, ostream& *eout* = cout)
~Fuzzy()
int **read**(char **n* = NULL, ostream& *eout* = cout)
int **write**(char **n* = NULL)
void **generate**(void)
void **calc**(double *, double *)
int **getinputcount**(void)
int **getoutputcount**(void)
int **geterrors**(void)
int **getrulecatch**(int *i*)
double **speed**(long *count* = 1000)
char ***getname**(void)
void **tout**(void)
friend ostream& operator<<(ostream&, const Fuzzy&)
friend istream& operator>>(istream&, Fuzzy&)
int **parser**(istream&, ostream&)
void **calcsetup**(void)
char ***gettoken**(istream&, int *mode*=0)
int **defvar**(istream&, ostream&)
int **defset**(VarDes*, istream&, ostream&)
int **defrule**(istream&, ostream&)
int **deflabel**(istream&, ostream&)
char ***getlabel**(char *)
void **killstructures**(void)
void **killfuzzybase**(void)
void **killlabel**(void)
void **out**(ostream&) const

3.2.3 References of the Classes, their Data and Element Functions

3.2.3.1 The Class FuzzySet

The class **FuzzySet** is a digital representation of a fuzzy set. The class is designed as a data element of the class **FuzzyVar** which is a representation of a fuzzy linguistic variable.

Basic Class:

none

Public Data:

none

Public Element Functions

FuzzySet::FuzzySet

FuzzySet(char **name*, int *p*)

Parameters char **name* is a pointer to the name of the fuzzy set.
int *p* is the number of x/y values for which memory is to be allocated.

Description The function is a constructor for an empty fuzzy set, but with a defined memory allocation for the given number of x/y values.

FuzzySet::FuzzySet

FuzzySet(char **name*, int *p*, double **x* #following lines)

Parameters char **name* is a pointer to the name of the fuzzy set.
int *p* is the number of x/y values for which memory is to be allocated.
double **x* is a vector of *p* double numbers, which represent the x values of the *p* x/y values.

Description The function is a constructor for a fuzzy set with a defined X vector. The elements of the Y vector are set to 0.

FuzzySet::FuzzySet

FuzzySet(char **name*, int *p*, double **x*, #following linesdouble **y*)

Parameters char **name* is a pointer to the name of the fuzzy set.
int *p* is the number of x/y values for which memory is to be allocated.
double **x* is a vector of *p* double numbers, which represent the x values of the *p* x/y values.
double **y* is a vector of *p* double numbers, which represent the y values of the *p* x/y values.

Description The function is a constructor for a fuzzy set with defined X and Y vectors.

FuzzySet::FuzzySet

FuzzySet(const FuzzySet& *org*)

Parameters const FuzzySet& *org* is a reference to the fuzzy set, which is to be copied.

Description The function is a copy constructor.

FuzzySet::~FuzzySet

~**FuzzySet**()

Description The function is the destructor.

FuzzySet::getname

char ***getname**(void)

Description The function **getname** returns a pointer to the name of the fuzzy set.

Return The pointer (char *) to the name of the fuzzy set.

FuzzySet::cleary

void **cleary**(void)

Description The function **cleary** sets all elements of the Y vector to 0.

FuzzySet::getxvector

double ***getxvector**(void)

Description The function **getxvector** returns a pointer to the data array of the X vector.

Return The pointer (double *) to the X vector.

FuzzySet::getyvector

double ***getyvector**(void)

Description The function **getxvector** returns a pointer to the data array of the Y vector.

Return The pointer (double *) to the Y vector.

FuzzySet::getstuetzen

int **getstuetzen**(void)

Description The function **getstuetzen** returns the number of the x/y values.

Return The number (int) of the x/y values in the set.

FuzzySet::insert

void **insert**(double *x*, double *y*)

Parameters double *x* is the x value to be inserted.

double *y* is the y value to be inserted.

Description The function **insert** inserts a x/y value in the fuzzy set in case the new value is not redundant.

FuzzySet::normalize

void **normalize**(int *p*, double **x*)

Parameters int *p* is the new number of x/y values.

double **x* is the new x vector with *p* co-ordinates.

Description The function **normalize** normalizes the fuzzy set such that it contains *p* x/y values with the x co-ordinates from the given *x* vector. The fuzzy set will not loose information only in case its old x co-ordinates are a subset of the new x co-ordinates.

FuzzySet::coa

double **coa**(void)

Description The function **coa** calculates a modified centre of area of the fuzzy set.

Return The value (double) of the centre of area point.

FuzzySet::crisp

double **crisp**(double *x*)

Parameters double *x* is the x value for which the crisp value is to be calculated.

Description The function **crisp** calculates the y crisp value belonging to the given *x* value.

Return The calculated crisp value (double).

FuzzySet::conclude

void **conclude**(FuzzySet **a*, double *weight*)

Parameters FuzzySet **a* is the fuzzy set overlay.

double *weight* is the weighting coefficient.

Description The function **conclude** overlays the set with the given fuzzy set **a* evaluated by the weighting coefficient *weight* (Maximum/Product method).

FuzzySet::tout

void **tout**(void)

Description The function **tout** provides online-debugging. Its output is a representation of the set in readable text on the screen. This function is still available only to guarantee compatibility with older version of the fuzzy library. Please use instead the operator <<.

Private Data:

char **SetName*	is a pointer to the name of the fuzzy set.
double **xval*	is a pointer to the x vector.
double **yval*	is a pointer to the y vector.
int *size*is	the reserved number of x/y values.
int *ss*	is the actual number of the x/y values.

Private Element Functions:

none

Operators:

FuzzySet::=

FuzzySet& operator = (const FuzzySet& *org*)

Parameters const FuzzySet& *org* is a reference to the set, which is to be copied.

Description The assignment operator only operates in case the sets are of the same size. The actual set will be a copy of the set given by its reference.

Return FuzzySet& is a reference to the copied set.

FuzzySet::*=

FuzzySet& operator *= (double *factor*)

Parameters double *factor* is the scaling factor.

Description The scaling operator is used to weight the set according to the Maximum/ Product method. The format of the weighting coefficient is double.

Return FuzzySet& is a reference to the set.

FuzzySet::+=

FuzzySet& operator += (const FuzzySet& *org*)

Parameters FuzzySet& *org* is a reference to the set, which is to be added.

Description The summation operator only operates in case the two sets have the same size. The set given by its reference is added to the actual set.

Return FuzzySet& is a reference to the sum of the sets.

FuzzySet::<<

FuzzySet& operator<< (ostream& *o*, const FuzzySet& *s*)

Parameters ostream& *o* is a reference to the output stream.
FuzzySet& *s* is a reference to the fuzzy set, which is to be
written to the output stream.

Description The operator writes the state of the fuzzy set to the given stream using readable text format.

Return A reference to the output stream.

3.2.3.2 The class FuzzyVar

The class **FuzzyVar** is the digital representation of a fuzzy linguistic variable. The class is used as a data element of the class **Fuzzy**.

Basic Classes:

none

Public Data:

none

Public Element Functions:

FuzzyVar::FuzzyVar

FuzzyVar(char **name*, int *c*, int *m*)

Parameters char **name* is the name of the linguistic variable.
int *c* is the number of fuzzy sets, which can be inserted.
int *m* is the operation mode of the variables.

Description The constructor of a linguistic variable requires 3 parameters, the variable name, the number of sets and the mode. The mode defines the direction of the variable i.e. input (bit0 = 0) or output (bit0 = 1).

FuzzyVar::~FuzzyVar

~**FuzzyVar**()

Description The destructor not only erases the data defined by the constructor but also all the fuzzy sets, which were assigned to the linguistic variable by the function **add**. Therefore a fuzzy set can only be assigned to one fuzzy variable.

FuzzyVar::getname

char ***getname**(void)

Description	The function returns a pointer to the name of the linguistic variable.
Return	The pointer (char *) to the variable name.

FuzzyVar::getsetname

char ***getsetname**(int *i*)

Parameters	int *i* is the index of the fuzzy set.
Description	The function returns a pointer to the name of a fuzzy set of the variable. The index of the set is given by the parameter *i*.
Return	The pointer (char *) to the name of the fuzzy set.

FuzzyVar::getsetcount

int **getsetcount**(void)

Description	The function returns the number of the fuzzy sets assigned to this variable.
Return	The number (int) of sets assigned to the variable.

FuzzyVar::add

void **add**(int *index*, FuzzySet **set*)

Parameters	int *index* is the index of the fuzzy set. FuzzySet **set* is the pointer to the fuzzy set, which will be inserted.
Description	The function **add** assigns the fuzzy set, referenced by its pointer (**set*), to the linguistic variable. The position of the set assigned to the variable is defined by the value index. The calling function has to take care about the index. The fuzzy sets have to be created dynamically since they will be deleted by the destructor of the linguistic variable. Calling the function **add** will transfer the handling of the fuzzy set completely to the class **FuzzyVar**.

FuzzyVar::check

void **check**(void)

Description The function **check** checks the logic structure of the linguistic variable. This function is not implemented at the moment. It is intended for a future expansion of the class.

FuzzyVar::norm

void **norm**(void)

Description The function **norm** normalizes the linguistic variable. That means every set of the variable has the same number of x/y values at the same x co-ordinates. This is required for calculations with the fuzzy sets.

FuzzyVar::getmode

int **getmode**(void)

Description The function **getmode** returns the operation mode of the variable i.e. its direction input (bit0= 0) or output (bit0 = 1).

Return The operation mode (int) of the variable.

FuzzyVar::getmaxx

double **getmaxx**(void)

Description The function **getmaxx** determines the maximum X value of the definition range of the normalized fuzzy variable.

Return The maximum value (double) of the x vector of the variable.

FuzzyVar::getminx

double **getminx**(void)

Description The function **getminx** determines the minimum X value of the definition range of the normalized fuzzy variable.

Return The minimum value (double) of the x vector of the variable.

FuzzyVar::get

double **get**(int *SetNo*)

Parameters int *SetNo* is the index of the fuzzy set.

Description The function **get** returns the crisp value of the set referenced by its index *SetNo*. The input value of the set is identical to the input value of the variable (see also function **setval**).

Return The determined crisp value (double).

FuzzyVar::set

void **set**(int *SetNo*, double *weight*)

Parameters int *SetNo* is the index of the fuzzy set, which is to be overlaid.

double *weight* is the weighting factor.

Description The function **set** overlays the output set, referenced by its index *SetNo*, of the variable. The overlay is weighted by the given weighting coefficient. This function is applicable only to linguistic variables generated as output variables.

FuzzyVar::clear

void **clear**(void)

Description The function **clear** erases the output set of the variable. This is required at the beginning of every sampling period (control period), but not for every rule. This function is only applicable to output variables (see also the constructor).

FuzzyVar::out

double **out**(void)

Description The function **out** returns the centre of area of the output set of the variable. This function is only applicable to output variables (see also the constructor).

Return The centre of area (double) of the output set of the variable.

FuzzyVar::getval

double **getval**(void)

Description The function **getval** returns the current input value of the variable.

Return The input value (double) of the variable.

FuzzyVar::setval

void **setval**(double *v*)

Description The function **tout** provides online-debugging. Its output is a representation of the linguistic variable in readable text on the screen. This function is still available only to guarantee compatibility with older version of the fuzzy library. Please use instead the operator <<.

Private Data:

char **VarName*	is the pointer to the name of the linguistic variable.
int *SetCount*	is the number of fuzzy sets assigned to this variable.
FuzzySet ***d*	is the pointer to the array of fuzzy sets.
double **normx*	is the pointer to the normalized x vector.
double *value*	is the input value of the variable.
int *mode*	is the operation mode of the variable: Bit 0: =0, input variable = 1, output variable Bit 1-15 reserved for future expansions.

Private Element Functions:

FuzzyVar::vsort

int **vsort**(int *c*, double **x*)

Parameters	int *c* is the number of elements in the x vector. double **x* is the given x vector.
Description	The help function **vsort** sorts the *c* elements of the given *x* vector. The vector is sorted with ascending order, double elements are deleted. The new number of elements is returned. This function is used in case of the normalization of the linguistic variable.
Return	The new number (int) of elements in the x vector.

Operators:

FuzzyVar::<<

FuzzyVar& operator<< (ostream& *o*, const FuzzyVar& *v*)

Parameters	ostream& *o* is a reference to the output stream. FuzzyVar& *v* is a reference to the fuzzy variable, which is to be written to the output stream.
Description	The operator writes the state of the fuzzy variable to the given stream using readable text format.
Return	A reference to the output stream.

3.2.3.3 The class FuzzyRule

The class **FuzzyRule** is the digital representation of a fuzzy rule. The class is used as a data element of the class **Fuzzy**.

Basic Classes:

none

Public Data:

none

Public Element Functions:

FuzzyRule::FuzzyRule

FuzzyRule(char **name*, int *i*, int *o*)

Parameters char **name* is the pointer to the name of the rule.
int *i* is the number of input combinations.
int *o* is the number of output combinations.

Description The constructor generates a fuzzy rule. Three parameters are required, the name of the rule (**name*), the number of input combinations (*i*) and the number of output combinations (*o*).

FuzzyRule::~FuzzyRule

~**FuzzyRule**()

Description The destructor erases the memory allocated by the constructor.

FuzzyRule::getname

char ***getname**(void)

Description The function **getname** returns the pointer to the name of the rule.

Return The pointer (char *) to the name of the rule.

FuzzyRule::addIn

void **addIn**(FuzzyVar **inv*, int *set*, int *op*)

Parameters FuzzyVar **inv* is the referenced variable of an input combination.

int *set* is the set index of an input combination.

int *op* is the operator of an input combination.

Description The function **addIn** adds an input combination to the fuzzy rule. Therefore the pointer to an input fuzzy variable, the set index and the combination operator has to be given.

FuzzyRule::addOut

void **addOut**(FuzzyVar **outv*, int *set*)

Parameters FuzzyVar **inv* is the referenced variable of an output combination.

int *set* is the set index of an output combination.

int *op* is the operator of an output combination.

Description The function **addOut** adds an output combination to the fuzzy rule. Therefore the pointer to an output fuzzy variable, the set index and the combination operator have to be given.

FuzzyRule::Do

int **Do**(void)

Description The function **Do** interprets a fuzzy rule. A value has to be assigned to the input variables previously. The defuzzification of the output sets of the output variables is not performed since this is only meaningful in case all the rules are interpreted. The operators of the input use the MIN/MAX (and/or) method. The interference is carried out using the Maximum/Product method.

Return Status (int) is 0 in case the rule is not applicable.

FuzzyRule::tout

void **tout**(void)

Description The function **tout** provides online-debugging. Its output is a representation of the fuzzy rule in readable text on the screen. This function is still available only to guarantee compatibility with older version of the fuzzy library. Please use instead the operator <<.

Private Data:

char **RuleName*	is the pointer to the name of the rule.
int *incount*	is the number of input combinations of the rule.
FuzzyVar ***invars*	is the pointer to an array of pointers to input variables.
int **inset*	is the pointer to an array of index of the sets.
int **operators*	is the pointer to an array of operators.
int *outcount*	is the number of output combinations of the rule.
FuzzyVar ***outvars*	is the pointer to an array of pointers to output variables.
int **outset*	is the pointer to an array of index of the sets.
int *iidx, oidx*	are index variables.

Private Element Functions:

none

Operators:

FuzzyRule::<<

FuzzyRule& operator << (ostream& *o*, const FuzzyRule& *r*)

Parameters ostream& *o* is a reference to the output stream.

FuzzyVar& *r* is a reference to the fuzzy rule, which is to be written to the output stream.

Description The operator writes the state of the fuzzy rule to the given stream using readable text format.

Return A reference to the output stream.

3.2.3.4 The Class Fuzzy

The class **Fuzzy** is the digital representation of a fuzzy controller. The class provides methods to handle, to read/write, to interpret and to execute fuzzy control bases. The class **Fuzzy** has no basic class but it requires data elements which are objects of the following classes:

FuzzySet,

FuzzyVar,

FuzzyRule,

Basic Classes:

none

Public Data:

none

Public Element Functions:

Fuzzy::Fuzzy

Fuzzy()

Description The constructor prepares the fuzzy control base. All of the pointers are initialized and a mechanism to supervise the ’new’ operator is installed. The control base can be used only after a call to the functions **parser** and **generate**.

Fuzzy::Fuzzy

Fuzzy(char **name*, ostream& *eout* = cout)

Parameters char **name* is the name of the fuzzy description file.

ostream& *eout* is the reference to an output stream, which is to be used for status/error messages (default: cout).

Description Alternatively to the a. m. standard constructor the control base can be generated using a data file name. In this case the function read is executed besides the operations of the standard constructor. The function **read** reads the file referenced by its name (**name*) and prints out status/error messages to the given stream (*eout*). Attention: The constructor does not return any error information. Therefore it is strongly required to test the error status by using the function geterrors before the program is continued.

Fuzzy::~Fuzzy

~**Fuzzy**()

Description The destructor has the task to free the memory, which was allocated by this object. To do this the operator uses several help functions (see **killstructures**, **killbase**, **killlabel**).

Fuzzy::read

int **read**(char **name* = NULL, ostream& *eout* = cout #following lines)

Parameters char **name* is the name of the fuzzy description file (default: NULL)

ostream& *eout* is the reference to an output stream, which is to be used for status/error messages (default: cout).

Description The function **read** opens the fuzzy description file referenced by its file name (**name*) and interprets its data using the function **parser**. Status and error messages are sent to the given stream (*eout*).

Return int, is the number of errors occurred.

Fuzzy::write

int **write**(char **n* = NULL)

Parameters char **n* is the name of the fuzzy description file to be created (default: NULL)

Description The function **write** creates a fuzzy description file on the mass storage depending on the structure of the fuzzy control base stored in the memory of the computer. This file is readable later on by the function **read**. Its name is the given file name (**name*).

Return The error status (int) (=0, no error).

Fuzzy::generate

void **generate**(void)

Description The function **generate** creates the fuzzy control base using the tree of structures generated by the function **parser**. Doing this objects of type **FuzzySet**, **FuzzyVar** and **FuzzyRule** are created. Existing rule bases are deleted previously (be careful in case of online calls).

Fuzzy::calc

void **calc**(double **in*, double **out*)

Parameters double **in* is the vector with the values of the input variables.

double **out* is the vector with the values of the output variables.

Description The function **calc** executes the controller function. An array of input values (format: double) is referenced by its pointer (**in*). The pointer (**out*) points to an array, which is to be used to store the output values (format: double). A sufficient size of the arrays has to be regarded. The array sizes of the control base are known from the fuzzy description file. The order of the array items is according to the order of their definitions in the description file.

Fuzzy::getinputcount

int **getinputcount**(void)

Description The function **getinputcount** returns the number of input variables.

Return The number (int) of input variables.

Fuzzy::getoutputcount

int **getoutputcount**(void)

Description The function **getoutputcount** returns the number of output variables.

Return The number (int) of output variables.

Fuzzy::geterrors

int **geterrors**(void)

Description The function **geterrors** returns the number of errors occurred during the last call to the function **parser**.

Return The number (int) of errors occurred.

Fuzzy::getrulecatch

int **getrulecatch**(int *i*)

Parameters int *i* is the index of the rule.

Description The function **getrulecatch** detects whether the rule *i* was activated during the last controller execution. The parameter *i* is the index of the rule inside the rule base. In case of an illegal rule index the value -1 is returned.

Return Status (int) is equal to 1 in case the rule was activated during the

Fuzzy::speed

double **speed**(long *count* = 1000)

Parameters long *count* is the number of test passes (default: 1000).

Description The function **speed** provides run time analysis (available only for DOS and Windows). It determines the definition range of the input variables, generates random input values belonging to this definition range and calculates the mean run time of the function calc. The number of passes through the function **calc**, which is to be used to determine the mean value, is given by the parameter *count*. The mean run time is returned in milli seconds. The function requires an executable rule base. The longest possible run time cannot be determined, since the run time depends on the number of the active rules and with that on the input values. Attention: Manipulations of the timer interrupt (i.e. for the sampling period) falsify the result.

Return The mean run time (double) of the rule base in milli seconds.

Fuzzy::getname

char ***getname**(void)

Description The function **getname** returns a pointer to the name of the rule base.

Return The pointer (char *) to the name of the rule base.

Fuzzy::tout

void **tout**(void)

Description The function **tout** provides online-debugging. Its output is a representation of the fuzzy rule base in readable text on the screen. This function is still available only to guarantee compatibility with older version of the fuzzy library. Please use instead the operator <<.

Private Data:

char **basename*	is the pointer to the name of the rule base.
int *errorcnt*	is the counter for the errors occurred during run time.
FuzzyVar ***vars*	is the pointer to an array of pointers to fuzzy variables.
int *varcount*	contains the number of fuzzy variables in the a. m. array.
FuzzyRule ***rules*	is the pointer to an array of pointers to fuzzy rules.
int *rulecount*	contains the number of fuzzy rules.
int **rulecatch*	is the pointer to an integer array (its size is equal to the number rules) containing information, whether the specified rule was active during the last execution pass (!=0) or inactive (==0).

Description structures in form of trees and chained lists for loading, saving and interpreting of fuzzy knowledge bases are described in the following. Each element of the fuzzy rule base is described by its own structure.

struct PointDes{	
PointDes **next*,	is a pointer to the next element.
double *x*,	is the X value of the base point (X/Y-values).
double *y*,	is the Y value of the base point.
}	is a description structure (off-line) for the base points of a fuzzy set.
struct SetDes{	
SetDes **next*,	is a pointer to the next set of the variable.
PointDes **first*,	is a pointer to the first base point.
int *pointcount*,	is the base point counter.
char **setname*	is a pointer to the name of the set.
}	is a description structure (off-line) for the fuzzy sets of a fuzzy variable.

struct VarDes{	
VarDes **next*,	is a pointer to the next variable.
SetDes **first*,	is a pointer to the first set of the variable.
int *setcount*,	is the set counter.
char **varname*,	is a pointer to the name of the variable.
int *mode*	is the operation mode of the variable.
}	is a description structure (off-line) for a fuzzy variable.
struct PraeDes{	
PraeDes **next*,	is a pointer to the next premise of the rule.
VarDes **var*,	is a pointer to the variable structure of the premise.
SetDes **set*,	is a pointer to the set structure of the a. m. variable.
int *op*	is the operator
}	is a description structure (off-line) for the premise of a fuzzy rule.
struct ConDes{	
ConDes **next*,	is a pointer to the next conclusion of the rule.
VarDes **var*,	is a pointer to the variable structure of the conclusion.
SetDes **set*,	is a pointer to the set structure of the a. m. variable.
}	is a description structure (off-line) for the conclusion of a fuzzy rule.
struct RulDes{	
RulDes **next*,	is a pointer to the next rule.
PraeDes **firstPrae*,	is a pointer to the first premise of the rule.
ConDes **firstCon*,	is a pointer to the first conclusion of the rule.
char **rulename*,	is a pointer to the name of the rule.
}	is a description structure (off-line) for a fuzzy rule.
struct label{	
label **next*,	is pointer to the next label structure.
char **name*,	is a pointer to the label name.
char **val*,	is a pointer to the label definition.
}	is a description structure (off-line) for a label definition.
VarDes **Varbase*	is the base address of the list of variables.
RulDes **Rulebase*	is the base address of the list of rules.
label **Labelbase*	is the base address of the list of labels.
int *incount*	is the number of inputs.
int *outcount*	is the number of outputs.
int **invars*	is a pointer to an index array for the input variables.
int **outvars*	is a pointer to an index array for the output variables.

Private Element Functions:

Fuzzy::parser

int **parser**(istream& *in*, ostream& *out*)

Parameters istream& *in* is the reference to the stream from which the fuzzy description file is read.

ostream& *out* is the reference to the stream to which error messages are written.

Description An input stream and an output stream are given to the function **parser**. The function reads characters from the input stream (*in*) and interprets it as a fuzzy description file for a fuzzy rule base. Status and error messages are sent to the output stream (*out*). To describe the rule base a tree structure containing chained lists is generated with respect to the description file for a fuzzy rule base. Syntax errors and to a certain extent logical errors are detected during the interpretation of the description file. The number of errors is returned by the function but it can be inquired alternatively by the function **geterrors**. In case a tree structure for describing the fuzzy rule base is existing before the function **parser** is called, this structure is deleted automatically. To interpret the fuzzy description file the function **parser** uses the following help functions:

gettoken()	to read 'words' (separated by spaces)
defvar()	to read and handle a variable definition
defrule()	to read and handle a rule definition
deflabel()	to read and handle a label definition

Return The number (int) of errors occurred.

Fuzzy::calcsetup

void **calcsetup**(void)

Description The function **calcsetup** prepares a complete rule base for calculation of its values. To do this index arrays of inputs and outputs are installed. This function is called automatically by the function **generate**.

Fuzzy::gettoken

char ***gettoken**(istream& *in*, int *mode*=0)

Parameters istream& *in* is the reference to the stream from which the token is to be read.

int *mode* is the operation mode, see above (default = 0).

Description The function **gettoken** reads a character string from the given input stream (*in*) and returns it. The character strings are separated by white spaces (i.e. spaces, tabs etc.). Comments starting with ’/*’ and ending with ’*/’ are ignored. Labels belonging to the list of labels are replaced automatically by their definition. The second parameter of the function is assignable to the legal values 0 or 1:

mode = 0, an arbitrary string (without spaces etc.) is read.

mode = 1, a numerical value (incl. dec. point etc.) is read,
in case of an error in the numerical input the first characters of the returned string is -1.

Return The pointer (char *) to the token read.

Fuzzy::defvar

int **defvar**(istream& *in*, ostream& *out*)

Parameters istream& *in* is the reference to the stream from which the description of the linguistic variable is read.

ostream& *out* is the reference to the stream to which the status and error messages are written.

Description The function **defvar** reads and handles a fuzzy linguistic variable. It is called as a help function by the function **parser**. The input stream (*in*) for reading and the output stream (*out*) to which error and status messages are written is given to the function. The function returns the number of errors which occurred during the definition of the linguistic variable.

Return The number (int) of errors occurred.

Fuzzy::defset

int **defset**(VarDes **v*, istream& *in* , ostream& *out*)

Parameters VarDes **v* is the pointer to the descriptive structure of the linguistic variable from the higher level.

istream& *in* is the reference to the stream from which the description of the fuzzy set is read.

ostream& *out* is the reference to the stream to which status and error messages are written.

Description The function **defset** reads and handles a fuzzy set. It is called as a help function by the function **defvar** which reads and handles a linguistic variable. The arguments given to the function are the descriptive structure of the linguistic variable from the higher level (**v*), the input stream (*in*) from which is read and the output stream (*out*) to which error and status messages are written. The function returns the number of errors which occurred during the definition of the fuzzy set.

Return The number (int) of errors occurred.

Fuzzy::defrule

int **defrule**(istream& *in*, ostream& *out*)

Parameters istream& *in* is the reference to the stream from which the description of the rule is read.

ostream& *out* is the reference to the stream to which status and error messages are written.

Description The function **defrule** reads and handles a fuzzy rule. It is called as a help function by the function **parser**. The arguments of the function are the input stream (*in*) from which is read and the output stream (*out*) to which error and status messages are written. The function returns the number of errors which occurred during the definition of the rule.

Return The number (int) of errors occurred.

Fuzzy::deflabel

int **deflabel**(istream& *in*, ostream& *out*)

Parameters istream& *in* is the reference to the stream from which the description of the label is read.

ostream& *out* is the reference to the stream to which status and error messages are written.

Description The function **deflabel** reads and handles the definition of a label. It is called as a help function from the function **parser**. The arguments of the function are the input stream (*in*) from which is read and the output stream (*eout*) to which error and status messages are written. The function returns the number of errors which occurred during the definition of the label. Attention: This function does not operate with **gettoken** (compare with numerical input)!

Return The number (int) of errors occurred.

Fuzzy::getlabel

char ***getlabel**(char **s*)

Parameters char **s* is the pointer to the name of the label to be searched.

Description The function **getlabel** searches the list of label definitions for the string referenced by **s*. In case this string is found in the list, a pointer to its definition in the list is returned. This pointer is NULL in the other case. This function is called as a help function from the function **gettoken**.

Return The pointer (char *) to the label found in the list (=NULL not found).

Fuzzy::killstructures

void **killstructures**(void)

Description The function **killstructures** erases the tree of structures which was used to generate the rule base. The rule base itself is left unchanged.

Fuzzy::killfuzzybase

void **killfuzzybase**(void)

Description The function **killfuzzybase** erases the rule base. Further calls to the function **calc** are no longer permitted.

Fuzzy::killlabel

void **killlabel**(void)

Description The function **killlabel** erases a list of definitions which were constructed according to '#define' assignments from the description file. The rule base and its description by a tree of structures are left unchanged.

Fuzzy::out

void **out**(ostream& *o*) const

Parameters ostream& *o* is the reference to the stream to which the fuzzy description file is written.

Description The function **out** writes the fuzzy rule base in readable text (format of fuzzy description file) to the referenced stream. The basic representation is the tree of structures of the fuzzy rule base stored in the memory of the computer. The function is called as an elementary function by the functions **write**, **tout** and the operator <<.

Operators:

Fuzzy::<<

friend ostream& operator<<(ostream& *o*, const Fuzzy& *f*)

Parameters ostream& *o* is the reference to the stream to which the fuzzy description file is to be written.

const Fuzzy& *f* is the object of type **Fuzzy**, which is to be written.

Description The operator << writes the fuzzy rule base in readable ASCII text to the referenced stream.

Return ostream& is the reference to the stream to which the fuzzy description

Fuzzy::>>

friend istream& operator>>(istream& *i*, Fuzzy& *f*)

Parameters istream& *i* is the reference to the stream from which the fuzzy description file is to be read.

const Fuzzy& *f* is the object of type **Fuzzy**, which is to be read.

Description The operator >> reads the ASCII text of the fuzzy rule base from the referenced stream. Information about syntax or logical errors are to be detected by the function **geterrors.**

Return ostream & is the reference to the stream from which the fuzzy description

3.3 Description of the File Formats

3.3.1 The Format of the Fuzzy Description File (*.FUZ)

The fuzzy description file with the extension FUZ is a file to configure a fuzzy controller. The file format is developed by the amira GmbH and is used by several products of the amira.

The fuzzy description file is used to configure a fuzzy object, which i.e. may operate as a fuzzy controller.

The fuzzy description file is a simple ASCII file, which can be edited by a text editor. The length of a line is limited to 255 characters. Single assignments are separated by spaces or tabulators.

It contains four types of elements, which are described in the following sections:

Comments [optional]

The file can include a comment in classical C-style ('/*' at the beginning and '*/' at the end) at every position except for the definition part of label. At least one space has to separate the comment string from the 'keywords' '/*' and '*/'.

The Definition of a Label [optional]

The definition of a label is limited to one line. It starts with the statement '#define'. The next statement contains the label name and the last statement contains the label definition. Thus a label can be defined as follows:

#define name This_is_the_definition_of_the_label_name

The Definition of Fuzzy Sets and Variables

The definition of fuzzy sets is only allowed within the definition of variables. It is ignored in the other case. The definition of a variable starts with the statement 'var'. The next statement can hold two different names, either 'input' in case an input variable is to be defined or 'output' in case an output variable is to be defined. The third statement of a variable definition is its name. Now the definition of the fuzzy set follows. It begins with the statement 'set' followed by the name of the fuzzy set. The name is followed by the x/y values as base points for a polygonal line. Similar to the statements the numbers are separated by spaces or tabulators. The definition of the fuzzy set ends with the statement 'endset'. The definition of a variable ends with the statement 'endvar' after all the

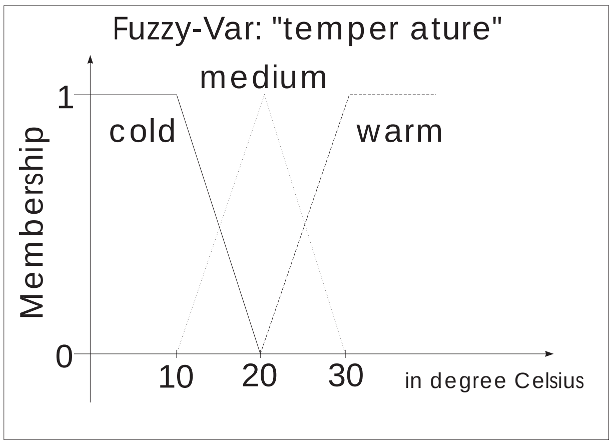


Figure 3.1: The fuzzy variable 'temperature'

fuzzy sets of the fuzzy variable are defined. Such a definition may look like the following:

```
var input temperature
set cold      10 1     20 0            endset
set medium    10 0     20 1     30 0   endset
set warm      20 0     30 1            endset
endvar
```

The Definition of Fuzzy Rules

The definition of a fuzzy rule is recognized from its first statement 'if'. The last statement of a fuzzy rule is named 'end'. The definition of a fuzzy rule contains two parts, the premise and the conclusion. Both parts are separated by the statement 'then'. The premise and the conclusion are built by a series of expressions which are combined by operators (further details are shown in the chapter of the theoretical backgrounds of a fuzzy controller). Permitted operators of the premise are 'and' (Min-Operator) and 'or' (Max-Operator) whereas the conclusion requires no operator to separate the expressions. An expression is the linkage of a fuzzy variable with one of its sets using the statement 'is'.

The formulation of a fuzzy rule requires that all the variables in use are defined previously since the fuzzy description file is interpreted only once from top to bottom. The syntax check of a fuzzy object tests whether the variables are defined, whether the used sets really belong to the variable and if the expressions are used correctly (input variables with the premise and output variables with the conclusion). A simple definition of a fuzzy rule may look like the following:

```
if temperature is cold then heating is high end
```

Table of the valid commands (keywords) and their explanation:

Command	Explanation
#define NAME TEXT	Defines a NAME, which is usable in the following statements and will be replaced by the definition TEXT automatically by the pre-processor.
/*	Begin of comment, ignored by the fuzzy controller kernel.
*/	End of comment.
var	Begin of linguistic variable definition. The statements "input" or "output" and the name of the variable must follow this keyword. Fuzzy sets are definable only in the following. The definition of the variable is terminated with the statement "endvar".
input	Defines the direction input for a variable.
output	Defines the direction output for a variable.
endvar	End of definition of a variable.
set	Begin of fuzzy set definition. A set name and a series of pairs of values must follow this keyword. The pairs of values are the base points of the set.
endset	End of set definition.
Command	Explanation

if	Begin of fuzzy rule definition. One or multiple premises separated by operators, the statement "then" and one or multiple conclusions must follow this keyword. The rule definition is terminated by the statement "end". A premise consists of a name of an input variable, the statement "is" and the name of the set belonging to this input variable. The conclusion is built in a similar way but the input variable is replaced by the output variable.
is	Separates variable and set in a premise or conclusion.
then	Separates the condition and the assignment part of a fuzzy rule.
and	Is the Minimum-Operator.
or	Is the Maximum-Operator.
end	End of rule definition.

Remark

The status and error messages which occur during the interpretation of the fuzzy description file are written to the file **ERROR.OUT** or appear on the screen.

3.3.2 The Format of the Error Output File ERROR.OUT

During loading and interpreting of a fuzzy description file status and possible error messages are written to the file ERROR.OUT. This file has the following format:

```
Fuzzy Parser Version 1.04 (07-DEC-94)

Fuzzy-Set <set_name> is already defined.
Fuzzy-Set <set_name> expects numerical value.
Unknown variable specification <string>.
Variable <var_name> is already defined.
Rule error, fuzzy variable <var_name> not found.
Rule error, fuzzy variable <var_name> is an output variable.
Rule error, fuzzy variable <var_name> is an input variable.
Rule syntax error, missing is.
Rule error, fuzzy set <set_name> is not member of <var_name>.
Rule syntax error, unknown Operator <string>.
<label_name> is already defined.
<n> Errors detected.
```

3.4 A Very Simple Example

3.4.1 The Fuzzy Description File of a Temperature Control

The simple temperature control of an electrical heating requires two variables, the temperature and the heating current. Here the temperature is the input variable and the heating current is the output variable. The description of a fuzzy controller for this system may look like the following:

```
/* A simple temperature control */
var input temperature      /* Temperature in centigrade degrees (Celsius) */
set cold      10 1         20 0                  endset
set medium    10 0         20 1          30 0    endset
set warm      20 0         30 1                  endset
endvar

var output heating_current /* Current in Ampere (0A - 4A) */
set small     0 1          2 0                   endset
set medium    0 0          2 1           4 0     endset
set high      2 0          4 1                   endset
endvar

/* now the rules follow: */

if temperature is cold then heating_current is high end
if temperature is medium then heating_current is medium end
if temperature is warm then heating_current is small end

/* End of File */
```

As can be seen from the file, the heating operates with a heating current between 0A and 4A. The rules shall control a temperature of 20°C.

As usual the file name of the fuzzy description file ends with the extension ’fuz’. In this example we choose the file name ’SIMPLE.FUZ’.

3.4.2 The C++ Sources of a Temperature Control

After the fuzzy description file was created according to 3.4.1, it has to be included in a C++ program. The following example briefly shows the solution:

```
/* Example Program: SIMPLE.CPP */

/* The program is to be adapted to your */
/* development environment. */
```

```
#include "iostream.h"
#include "ferror.h"
#include "fuzzy.h"

double readtemperature( void )
{
/* here is the code to read the temperature sensor */
return temperature;
}

void writeheatingcurrent( double heating_current )
{
/* here is the code to adjust the heating current */
}

void main( void )
{
Fuzzy f;                    // create the fuzzy object
double in[1];               // only one input variable
double out[1];                      // only one output variable
if(f.read("simple.fuz"))    // load the description file
        error();            // file
f.generate();               // create the rule base
while(1)                    // endless loop
{
        in[0] = readtemperature();  // read input value
        f.calc( in, out );          // execute controller
        writeheatingcurrent( out[0] ); //write output value
}
}
/* end of file */
```

With this the programming of a very simple fuzzy controller is terminated.

4 Functions of the PLOT16.DLL

List of the functions (all of type **far _pascal**) of the standard interface:

int **Version**(void),

HWND **CreateSimplePlotWindow**(HWND *parentHWnd*, WORD *NumberOfCurves*, WORD *NumberOfPoints*, double far** *data*)

void **ShowPlotWindow**(HWND *HWnd*, BOOL *bflag*)

void **ClosePlotWindow**(HWND *HWnd*)

void **UpdatePlotWindow**(HWND *HWnd*)

HWND **GetValidPlotHandle**(int *index*)

void **AddPlotTitle**(HWND **HWnd**, int *Position*, LPSTR *title*)

WORD **AddAxisPlotWindow**(HWND *HWnd*, WORD *AxisID*, LPSTR *title*, WORD *Position*, WORD *ScalingType*, double *ScalMin*, double *ScalDelta*, double *ScalMax*)

void **AddXData**(HWND *HWnd*, WORD *XCount*, double far* *XData*)

void **AddTimeData**(HWND *HWnd*, WORD *XCount*, double *StartTime*, double *SamplingPeriod*)

WORD **AddYData**(HWND *HWnd*, WORD *nYCount*, double far* *YData*)

void **SetAxisPosition**(HWND *HWnd*, WORD *AxisID*, WORD *Position*)

int **SetCurveMode**(HWND *HWnd*, WORD *idCurve*, LPSTR *title*, WORD *AxisId*, WORD *LineStyle*, DWORD *Colour* , WORD *MarkType*)

int **SetPlotMode**(HWND *HWnd*, WORD *TitlePosition*, DWORD *TitleColour*, LPSTR *Title*, WORD *WithLineStyleTable*, WORD *WithAxisFrame*, WORD *WithPlotFrame*, DWORD *FrameColour*, WORD *WithDate*, long *OldDate*, LPSTR *FontName*, int *MaxCharSize*)

void **PrintPlotWindow**(HWND *HWnd*, HDC *printerDC*, int *xBegin*, int *yBegin*, int *xWidth*, int *yHeight*, BOOL *scale*)

HWND **CreateEmptyPlotWindow**(HWND *parentHWnd*)

Table of the macros in use:

Macro	Value	Meaning
X_AXIS	1	reference AxisID for the X-axis
Y_AXIS	2	reference AxisID for the Y-axis
Y_AXIS	4	reference AxisID for the Y2-axis
AXE_BOTTOM	1	X-axis bottom to axis frame
AXE_LEFT	1	Y/Y2-axis left to axis frame
AXE_RIGHT	2	Y/Y2-axis right to axis frame
AXE_TOP	2	X-axis top to axis frame
AXE_MIDDLE	4	X/Y-axis in the middle of the axis frame
TITLETEXT_TOP	1	drawing title top position
TITELTEXT_BOTTOM	2	drawing title bottom position
TITELTEXT_APPEND	4	drawing title appended to the window title
LINEAR_SCALING	0	linear scaling of the min/max-values of an axis
LOG_SCALING	1	logarithmic scaling of the min/max-values of an axis
INTERN_SCALING	0	automatic internal scaling of the min/max-values of an axis
EXTERN_SCALING	2	external adjustment of the min/max/delta-scaling values of an axis
NO_MARK	0	without marking a Y-curve
CROSS	1	marking by a laying cross
TRIANG_UP	2	marking by a triangle top oriented
TRIANG_DOWN	3	marking by a triangle bottom oriented
QUAD	4	marking by a square
CIRCLE	5	marking by a circle

Version

int **Version**(void)

Description: The function **Version** returns the version number (at this time = **19** for the version 1.2 dated 01. April 1999) of this DLL.

Return The version number of this DLL.

CreateSimplePlotWindow

HWND far _pascal **CreateSimplePlotWindow** (HWND *parentHWnd,*
WORD *NumberOfCurves*, WORD *NumberOfPoints*, double far** *data*)

Parameters *parentHWnd* is the windows handle of the parent window.

NumberOfCurves is the number of curves in the plot object.

NumberOfPoints is the number of points of each curve in the plot object.

data is a pointer to the value matrix of the curves.

Description The function **CreateSimplePlotWindow** creates a window containing a standard plot object. This plot object contains the value matrix *data* consisting of *NumberOfCurves* Y-curves (rows of the value matrix) with *NumberOfPoints* points (columns of the value matrix) for each curve with respect to a common X-axis. The X-axis is interpreted as a time axis with *NumberOfPoints* steps to be drawn at the top of the axes frame including labels and a standard axis title. All Y-curves correspond to one common Y-axis to be drawn left to the axes frame including a standard axis title and labels determined by an automatic internal scaling. A grid net with dashed lines is added to the axes frame. A linestyle table is located in the upper part of the window containing a short piece of a straight line for each Y-curve with the accompanying attributes linestyle, colour and marking type followed by a short describing text ("Curve #xx"). Each curve is displayed with attributes according to the following table.

Curve No.:	Text	Linestyle	Colour	Marking Type
1	Curve # 1	PS_SOLID	BLACK	none
2	Curve # 2	PS_DASH	RED	cross
3	Curve # 3	PS_DOT	GREEN	triangle top
4	Curve # 4	PS_DASHDOT	BLUE	triangle bottom
5	Curve # 5	PS_DASHDOTDOT	MAGENTA	square
6	Curve # 6	PS_SOLID	CYAN	circle
7	Curve # 7	PS_DASH	YELLOW	none
8	Curve # 8	PS_DOT	GRAY	cross

The 5 different linestyles, 6 marking types and 8 colours are repeated serially. The curve handles (identifiers) are set automatically equal to the curve numbers. A standard drawing title will be added below the axes frame.

Return The Windows handle of the plot object window for a successful windows creation. Otherwise NULL is returned.

ShowPlotWindow

void far _pascal **ShowPlotWindow**(HWND *HWnd*, BOOL *bflag*);

Parameters *HWnd* is a Windows handle of a plot object window.

bflag is a flag to control the visibility of a plot object window (=TRUE - visible, else invisible).

Description The function **ShowPlotWindow** displays a previously created plot object window with the Windows handle *HWnd* when the flag *bflag* is set equal to TRUE. Otherwise the plot object window is hidden.

ClosePlotWindow

void far _pascal **ClosePlotWindow**(HWND *HWnd*)

Parameters *HWnd* is a Windows handle of a plot object window.

Description The function **ClosePlotWindow** closes a previously created plot object window with the Windows handle *HWnd* and removes all the corresponding objects from the memory.

UpdatePlotWindow

void far _pascal **UpdatePlotWindow**(HWND *HWnd*)

Parameters *HWnd* is a Windows handle of a plot object window.

Description The function **UpdatePlotWindow** updates the drawing of a previously created plot object window with the Windows handle *HWnd*.

GetValidPlotHandle

HWND far _pascal **GetValidPlotHandle**(int *index*)

Parameters *index* is an index to reference a plot object window.

Description The function **GetValidPlotHandle** determines the Windows handle *HWnd* of that plot object window which is referenced by the given *index*. Starting with an index of 0 the handle of each previously created plot object window is determinable. The function returns the value 0, when a plot object window with the given *index* does not exist.

Return The handle *HWnd* of the plot object window referenced by *index* if it exists else 0.

AddPlotTitle

void far _pascal **AddPlotTitle**(HWND *HWnd*, int *Position*, LPSTR *title*)

Parameters *HWnd* is a Windows handle of a plot object window.

Position is the position of the drawing title (TITLETEXT_TOP or
TITLETEXT_BOTTOM + possibly TITLETEXT_APPEND).

title is a pointer to the new drawing title with a maximum of 255 characters.

Description: The function **AddPlotTitle** inserts a new drawing title *title* at the position *Position* in a previously created plot object window with the Windows handle *HWnd*. The position is either the upper part of the drawing frame (TITLETEXT_TOP) or the lower part (TITLETEXT_BOTTOM). If the macro TITLETEXT_APPEND is defined in addition the *title* is appended also to the windows title. However the overall length of this windows title is limited to 79 characters. The drawing title must not exceed 255 characters. Line wrapping is carried-out automatically if necessary but the drawing title will be truncated if it exceeds a third of the drawing height.

AddAxisPlotWindow

WORD far _pascal **AddAxisPlotWindow**(HWND *HWnd*, WORD *AxisID*, LPSTR *title*, WORD *Position*, WORD *ScalingType*, double *ScalMin*, double *ScalDelta*, double *ScalMax*)

Parameters

HWnd is a Windows handle of a plot object window.

AxisID is a reference to the axis (X-axis = X_AXIS, Y_axis = Y_AXIS, second Y-axis = Y2_AXIS).

title is a pointer to the new axis title with a maximum of 255 characters.

Position is the position of the axis inside the axes frame:
X-axis at the bottom (AXE_BOTTOM), at the top (AXE_TOP) or in the middle (AXE_MIDDLE), a Y-axis left (AXE_LEFT), right (AXE_RIGHT) or in the middle (AXE_MIDDLE) of the frame.

ScalingType is the scaling mode for the new axis:
= LINEAR_SCALING | INTERN_SCALING - internal, linear
= LINEAR_SCALING | EXTERN_SCALING - external, l
= LOG_SCALING | INTERN_SCALING - internal, logarithmic
= LOG_SCALING | EXTERN_SCALING - external, logarithmic

ScalMin is the minimum external scaling value for the axis.

ScalDelta is the external scaling step for the axis.

ScalMax is the maximum external scaling value for the axis.

Description

The function **AddAxisPlotWindow** adds a new axis with the reference *AxisID* (X_AXIS, Y_AXIS or Y2_AXIS) to a previously created plot object window with the Windows handle *HWnd*. Any existing axis in this plot object with the same reference will be replaced by the new one. The axis title *title* , the position *Position* inside the axes frame (AXE_BOTTOM / AXE_LEFT, AXE_RIGHT / AXE_TOP or AXE_MIDDLE) as well as the scaling mode *ScalingType* (LOG_SCALING / LINEAR_SCALING and EXTERN_SCALING / INTERN_SCALING) are to be defined for the new axis. The scaling values *ScalMin*, *ScalDelta* and *ScalMax* are considered only when the macro EXTERN_SCALING is defined for the scaling mode. Otherwise the scaling values are determined automatically.

Return

The axis reference *AxisID* when the axis was created successfully, else 0.

AddXData:

void far _pascal **AddXData**(HWND *HWnd*, WORD *XCount*, double far* *XData*)

Parameters: *HWnd* is a Windows handle of a plot object window.

XCount is the number of points for the X-axis in the plot object.

**Xdata* is a pointer to the data of the X-axis in the plot object.

Description: The function **AddXData** adds new data *XData* with a number of *XCount* values for the X-axis to a previously created plot object window with the Windows handle *HWnd*. Any existing data of a X-axis in this plot object are replaced by the new data.

AddTimeData:

void far _pascal **AddTimeData**(HWND *HWnd*, WORD *XCount*, double *StartTime*, double *SamplingPeriod*)

Parameters: *HWnd* is a Windows handle of a plot object window.

XCount is the number of points (time values) for the X-axis in the plot object.

StartTime is the initial value for the time axis (=X-axis).

SamplingPeriod is the sampling period, the time distance between two successive values for the time axis (=X-axis).

Description: The function **AddTimeData** adds new data with a number of *XCount* time values for the X-axis to a previously created plot object window with the Windows handle *HWnd*. The time values start with *StartTime* and end with (*XCount* - 1) * *SamplingPeriod*. Any existing data of a X-axis in this plot object are replaced by the new data.

AddYData:

WORD far _pascal **AddYData**(HWND *HWnd*, WORD *nYCount*, double far* *YData*)

Parameters: *HWnd* is a Windows handle of a plot object window.

nYCount is the number of points for the Y-curve in the plot object.

**Ydata* is a pointer to the data for the Y-curve in the plot object.

Description: The function **AddYData** adds a new Y-curve given by the data *YData* with a number of *YCount* values to a previously created plot object window with the Windows handle *HWnd*. The function returns an automatically generated reference (handle) for the Y-curve when a valid plot object window exists. The standard values for the curve-attributes linestyle, colour, marking type and describing text ("Curve #xx") are set automatically as described with the function **CreateSimplePlotWindow** with respect to the returned reference value. In case no data are defined for a X-axis, a standard time axis from 1.0 to *nYCount**1.0 is generated in addition.

Return Is equal to the automatically generated reference (*idCurve)* of the added Y-curve, when the plot object window exists, else equal to 0.

See also **CreateSimplePlotWindow**.

SetCurveMode

int far _pascal **SetCurveMode**(HWND *HWnd*, WORD *idCurve*, LPSTR *title*, WORD *AxisId*, WORD *LineStyle*, DWORD *Colour*, WORD *MarkType*)

Parameters *HWnd* is a Windows handle of a plot object window.

idCurve is the reference (handle) of the Y-curve.

title is a pointer to the new describing text of the Y-curve used for the linestyle table with a maximum of 255 characters. The current describing text is retained when the length of this string is equal to 0.

AxisId is the assignment to the Y-axis (Y_AXIS) or Y2-axis (Y2_AXIS).

LineStyle is the linestyle of the Y-curve (see CreateSimplePlotWindow). The current linestyle is retained when this parameter is equal to 0xFFFF.

Colour is the (RGB-) colour of the Y-curve. The current colour is retained when this parameter is equal 0xFFFFFFFFL.

MarkType is the marking type of the Y-curve. The current marking type is retained when this parameter is equal 0xFFFF.

Description The function **SetCurveMode** changes the attributes of a Y-curve referenced by *idCurve* belonging to a previously created plot object window with the Windows handle *HWnd*. The describing text *title* for the linestyle table, the assignment *AxisId* to the Y- or Y2-axis, the linestyle *LineStyle*, the colour *Colour* as well as the marking type *MarkType* are assignable.

Remark: When a Y2-axis is not existing but a curve is assigned to this axis the linestyle table demonstrates this fact by displaying only the describing text for this curve without the short piece of a straight line. The number of characters in the describing text should be short with respect to the number of curves.

Return Is equal to 1, when the Y-curve with *idCurve* exists, else equal to 0.

SetPlotMode

int far _pascal **SetPlotMode**(HWND *HWnd*, WORD *TitlePosition*, DWORD *TitleColour*, LPSTR *Title*, WORD *WithLineStyleTable*, WORD *WithAxisFrame*, WORD *WithPlotFrame*, DWORD *FrameColour*, WORD *WithDate*, long *OldDate*, LPSTR *FontName*, int *MaxCharSize*)

Parameters *HWnd* is a Windows handle of a plot object window.

TitlePosition is the position of the drawing title (TITLETEXT_TOP or TITLETEXT_BOTTOM + possibly TITLETEXT_APPEND).

TitleColour is the (RGB-) colour for the drawing title. The current colour is retained when this parameter is equal 0xFFFFFFFFL.

Title is a pointer to the new drawing title with a maximum of 255 characters. The current drawing title text is retained when the length of this string is equal to 0.

WithLineStyleTable enables (=TRUE) or disables (=FALSE) the display mode of the linestyle table.

WithAxisFrame is a flag determining if a frame is to be drawn around the axes crossing (=TRUE) or not (=FALSE).

WithPlotFrame is a flag determining if a frame is to be drawn around the drawing (=TRUE) or not (=FALSE) only during output to a Windows Meta File or a raster device.

FrameColour is the (RGB-) colour for the axes frame. The current colour is retained when this parameter is equal 0xFFFFFFFFL.

WithDate is a parameter determining if no date (=FALSE), the current date (=NEW_DATE) or a given 'old' date (=OLD_DATE) is to be inserted in the upper left part of the drawing.

OldDate is the 'old' date, which is considered only when *WithDate* is set to OLD_DATE.

FontName is the font name of the character set used for all text outputs (titles, date, linestyle table, labels). If the length of this name is equal to 0, the default character set will be used.

MaxCharSize is the maximum character height for all text outputs used with a maximum window size. Reducing the plot window size will scale down the character height to a minimum of 12 pixels. If this parameter is equal to 0xFFFF, the default maximum character size will be used.

Description: The function **SetPlotMode** changes the general layout of a plot object window with the Windows handle *HWnd* previously created i.e. by **CreateSimplePlotWindow**.
As described with the function **AddPlotTitle** a new drawing title *title* is inserted at the position *Position* . The position is either the upper part of the drawing frame (TITLETEXT_TOP) or the lower part (TITLETEXT_BOTTOM). If the macro TITLETEXT_APPEND is defined in addition the *title* is appended also to the windows title. However the overall length of this windows title is limited to 79 characters. The drawing title must not exceed 255 characters. Line wrapping is carried-out automatically if necessary but the drawing title will be truncated if it exceeds a third of the drawing height. The drawing title is displayed using the colour *TitleColor* and the character set *FontName* with a maximum character height *MaxCharSize* (for a maximum size of the plot window). The character set as well as the maximum character height are also used for the other text outputs.
If the flag *WithLineStyleTable* is set to TRUE a linestyle table is inserted above the axes frame containing a short piece of a straight line for each Y-curve with the accompanying attributes linestyle, colour and marking type followed by a short describing text in the standard form "Curve #xx" or defined by the function **SetCurveMode**.
When the flag *WithAxisFrame* is set to TRUE, a frame is drawn around the axes crossing using the colour *FrameColour* only at those margins, which are not occupied by an axis.
When the flag *WithPlotFrame* is set to TRUE, an additional frame is drawn around the complete

drawing using the colour *FrameColour* only in case the plot window is output to a Windows Meta File or to a raster device.
The parameter *WithDate* determines the display mode of the date in the upper left part of the drawing. With *WithDate* set to FALSE the date output is missing. With *WithDate* set to NEW_DATE the current date (day, month, year and time during drawing the plot) is inserted while *WithDate* set to OLD_DATE will display the date given by the parameter *OldDate*.

Return Is equal to 1, when the plot window with the handle *HWnd* exists, else equal to 0.

See also **CreateSimplePlotWindow, AddPlotTitle, SetCurveMode**.

PrintPlotWindow

void far _pascal **PrintPlotWindow**(HWND *HWnd*, HDC *printerDC,* int *xBegin*, int *yBegin*, int *xWidth*, int *yHeight*, BOOL *scale*)

Parameters *HWnd* is a Windows handle of a plot object window.

printerDC is the device context of the output device.

xBegin is the left margin of the hardcopy (mm/Pixel)

yBegin is the upper margin of the hardcopy (mm/Pixel)

xWidth is the width of the hardcopy (mm/Pixel)

yHeight is the height of the hardcopy (mm/Pixel)

scale =TRUE, position and size of the hardcopy in [mm],
else position and size of the hardcopy in pixels.

Description: The function **PrintPlotWindow** generates an output (typically a hardcopy) of a previously created plot object window with the Windows handle *HWnd*. The output device is defined by its device context handle *printerDC*. The position and the size of the hardcopy are determined by the parameters *xBegin, yBegin, xWidth* and *yHeight*. These parameters are interpreted as [mm], when the parameter *scale* is set to TRUE. Otherwise these parameters are taken as pixel numbers.

CreateEmptyPlotWindow

HWND far _pascal **CreateEmptyPlotWindow**(HWND *parentHWnd*)

Parameters *parentHWnd* is the windows handle of the parent window.

Description: The function **CreateEmptyPlotWindow** creates a window with an 'empty' plot object. This plot object only contains the current date, an empty axes frame as well as a standard drawing title above this frame.

Return The Windows handle of the plot object window for a successful windows creation. Otherwise NULL is returned.

5 Interface Functions of the TIMER16.DLL

The TIMER16.DLL supports the cyclic call of specific functions of the "Service"-DLL which realizes a sampled data control with a constant sampling period.

LibMain

int **LibMain**(HINSTANCE , WORD, WORD, LPSTR)

Parameters All parameters will be left out of consideration.

Description The function **LibMain** loads the SERVICE.DLL and determines the addresses of the functions **DoService, SetParameters, GetData** and **LockMemory** included in this DLL. A value of 1 is returned only when the operations DLL loading as well as address determination have been successful. Otherwise a value of 0 is returned meaning that further accesses to the SERVICE.DLL are invalid.

Return Is equal to 1 in case of successful loading the SERVICE.DLL and correct address determination, else equal to 0.

StartTimer

int **StartTimer**(double *Time*)

Parameters Time is the sampling period in seconds (minimum 0.001 s).

Description The function **StartTimer** opens and initializes the PC adapter card driver with the name given by the global variable *szDriverName* (see also **SelectDriver**). The code and data memory of this DLL as well as that of the SERVICE.DLL is locked (no longer moveable because of the function start addresses). A multi-media timer is programmed according to the given sampling period. A value of 0 (TERR_OK) is returned only when all of the operations were carried-out successfully.

Return Error state :
TERR_OK (0) on successful operations,
TERR_RUNNING (1), when a timer is still running,
TERR_TOOFAST (2), when the selected sampling period is too small
TERR_DRV_LOAD_FAIL (5), when the card driver opening fails
TERR_MEM_LOCK_FAIL (6), when locking the memory of the
TIMER16.DLL and the SERVICE.DLL fails.

StopTimer

int **StopTimer**(void)

Description The function **StopTimer** stops the currently running multi-media timer, unlocks the memory of this DLL as well as of the SERVICE.DLL and closes the current adapter card driver.

Return Error state:
TERR_OK (0) on successful operations,
TERR_RUNNING (1), when no timer is running

GetMinMaxTime

GetMinMaxTime(DWORD *&min* , DWORD *&max*, BOOL *res*)

Parameters &min is a reference to the minimum sampling period in ms.

&max is a reference to the maximum sampling period in ms.

res is a flag to reset the minimum and maximum value of the sampling period.

Description The function **GetMinMaxTime** returns the minimum and maximum value of the real sampling period determined up to this time. With *res*=1 the minimum and maximum value are set to the nominal sampling period.

GetSimTime

float **GetSimTime**(void)

Description The function **GetSimTime** returns the (simulation) time passed since the last start of a multi-media timer. This value is calculated by the product of the nominal sampling period and the number of calls of the function **DoService**.

Return Time in seconds since the last call of **StartTimer**.

SetupDriver

int **SetupDriver**(void)

Description The function **SetupDriver** opens the PC adapter card driver with the name given by the global variable *szDriverName* (see also **SelectDriver**) and starts the dialog to adjust the base address only when no multi-media timer is running and in case no card driver is open. The driver is closed again at the end of the dialog. If opening or closing the driver or carrying-out the dialog fails corresponding messages will appear on the screen.

Return Error state :
TERR_OK (0) on successful operations,
TERR_RUNNING (1), when a timer is still running,
TERR_FAIL (99), when a driver is open or opening and closing the driver fails.

SelectDriver

int **SelectDriver**(LPSTR *lpDriverName*)

Parameters lpDriverName is the file name of a new driver for the PC adapter card.

Description The function **SelectDriver** copies the given file name to the global variable *szDriverName* which adjusts the driver for the PC adapter card only when no driver is open and no multi-media timer is running.

Return Error state :
TERR_OK (0) on successful operations,
TERR_RUNNING (1), when a timer is still running,
TERR_FAIL (99), when a driver is still open.

6 Windows Drivers for DAC98, DAC6214 and DIC24

The drivers are installable 16-Bit drivers applicable to 16- or 32-Bit programs with Windows 3.1 / 95 / 98. To exchange data with the drivers the following three 16-Bit API functions are used:

OpenDriver

HDRVR *hDriver* = **OpenDriver**(*szDriverName*, NULL, NULL)

Parameters *szDriverName* is the file name of the driver, valid names are "DAC98.DRV", "DAC6214.DRV" and "DIC24.DRV" (according to the PC adapter cards) possibly combined with complete path names.

Description The function **OpenDriver** initializes the driver and returns a handle for following accesses to this driver. If this function is called the first time the driver is loaded into the memory. Any further calls return another handle of an existing driver. The driver handle is valid only when the return value is unequal to NULL. In case the return value is equal to NULL, the function **OpenDriver** failed meaning that further driver accesses by the functions **SendDriverMessage** or **CloseDriver** are invalid. The parameter *szDriverName* of the function **OpenDriver** contains the DOS file name of the driver. The file name may include the disk name as well as the complete path names according to the 8.3 name convention but it must not exceed 80 characters. When only a single file name is used, the drivers location is expected in the standard search path of Windows. The other parameters are meaningless and should be equal to NULL.

Return Valid driver handle or NULL.

SendDriverMessage

LRESULT *result* = **SendDriverMessage**(*hDriver*, *DRV_USER*, *PARAMETER1*, *PARAMETER2*)

Parameters *hDriver* is a handle of the card driver.

DRV_USER is the flag indicating special commands.

PARAMETER1 is a special command and determines the affected channel number (see table below).

PARAMETER2 is the output value for special write commands.

Description The function **SendDriverMessage** transfers a command to the driver specified by the handle *hDriver*. The drivers for the adapter cards from **amira** expect the value *DRV_USER* for the second parameter (further commands can be found in the API documentation of **SendDriverMessage**). The third parameter *PARAMETER1* is of type ULONG specifying the command which is to be carried-out. The lower 8 bits of this parameter determine the channel (number) which is to be affected by the given command. The commands are valid for all of the three drivers. But the valid channel numbers depend on the actual hardware. The last parameter *PARAMETER2* is of type ULONG and is used with write commands. It contains the output value. The return value depends on the command. Commands and channel names are defined in the file "IODRVCMD.H".

Return Is equal to 0 in case of unsupported commands or special write commands. Otherwise it contains the result of special read commands.

Table of the supported standard API commands

Command	Return	Remark
DRV_LOAD	1	loads the standard base address from SYSTEM.INI
DRV_FREE	1	
DRV_OPEN	1	
DRV_CLOSE	1	
DRV_ENABLE	1	locks the memory range for this driver
DRV_DISABLE	1	unlocks the memory range for this driver
DRV_INSTALL	DRVCNF_OK	
DRV_REMOVE	0,	
DRV_QUERYCONFIGURE	1	
DRV_CONFIGURE	1	calls the dialog to adjust the base address
DRV_POWER	1	
DRV_EXITSESSION	0	
DRV_EXITAPPLICATION	0	

Table of the special commands with the flag DRV_USER:

PARAMETER1				Return
Command	Channel Number			
	DAC98	DAC6214	DIC24	
DRVCMD_INIT initializes the card and has to be the first command				0
DRVINFO_AREAD returns the number of analog inputs				8 for DAC98, 6 for DAC6214, 0 for DIC24
DRVINFO_AWRITE returns the number of analog outputs				2 for all cards
DRVINFO_DREAD returns the number of digital inputs				8 for DAC98, DIC24 4 for DAC6214
DRVINFO_DWRITE returns the number of digital outputs				8 for DAC98, DIC24 4 for DAC6214
DRVINFO_COUNT returns the number of counters and timers				5 for DAC98 1 for DAC6214 6 for DIC24
DRVCMD_AREAD reads an analog input	0-7	0-5	no inputs	16 bit value from -32768 to 32767 according to the input voltage range
DRVCMD_AWRITE writes to an analog output	0-1	0-1	0-1	0
DRVCMD_DREAD reads a single digital input or all inputs (ALL_CHANNELS)	0-7 or ALL_CHAN	0-3 or ALL_CHAN	0-7 or ALL_CHAN	state (0 or 1) of a single input or states binary coded (channel0==bit0)
DRVCMD_DWRITE writes to a single digital output or to all outputs (channel0==bit0)	0-7 or ALL_CHAN	0-3 or ALL_CHAN	0-7 or ALL_CHAN	0
DRVCMD_COUNT reads a counter / timer	DDM0 DDM1 DDM2 COUNTER TIMER	DDM0	DDM0 DDM1 DDM2 DDM3 COUNTER TIMER	counter- / timer-content as an unsigned 32-bit value
DRVCMD_RCOUNT resets a counter / timer (counter, timer to the value -1) or all DDM's (ALL_CHANNELS)	DDM0 DDM1 DDM2 COUNTER TIMER ALL_CHAN	DDM0	DDM0 DDM1 DDM2 DDM3 COUNTER TIMER ALL_CHAN	0
DRVCMD_SCOUNT presets a counter / timer to an initial value	COUNTER TIMER		COUNTER TIMER	0

CloseDriver

CloseDriver(*hDriver*, NULL, NULL)

Parameters *hDriver* is a handle of the card driver.

Description The function **CloseDriver** terminates the operation of the driver specified by the handle *hDriver*. The driver is removed from the memory when all of its handles are released by the function **CloseDriver**.