PS600

Laboratory Experiment Inverted Pendulum

printed 16-October-2000

Assembly and Start-Up 1

Technical Data 2

State Controller and Practical Course Inverted Pendulum 3

Fuzzy Controller Inverted Pendulum 4

Program Operation 5

PC Plug-in Card DAC98 6

PS600 Software Inverted Pendulum 7

Assembly and Start-Up

Date: 19-June-2000

1 Safety Precautions 1-1

1.1 Dedicated Usage 1-1

1.2 Potential Danger and Danger Areas 1-1

1.3 General Risks on Disregarding the Safety Precautions 1-1

1.4 Maintenance under Safety Regulations 1-1

1.5 Safety Regulations for the Customer 1-1

1.6 Forbidden Modifications without any Authorization 1-2

2 Assembly and Start-Up 2-1

2.1 Unpacking 2-1

2.2 Setting up the System 2-2

2.2.1 The PS600 Mechanics 2-2

2.2.2 Actuator 2-2

2.3 Description of the PS600 Mechanical Setup 2-2

2.3.1 Description of the Pendulum Mechanics (Option 600-20) 2-2

2.3.2 Description of the Loading Bridge Mechanics (Option 600-30) 2-2

2.3.3 Description of the Tandem Pendulum Mechanics (Option 600-40) 2-3

2.4 Description of the PS600 Actuator 2-3

2.4.1 The Rear Panel 2-3

2.4.2 The Front Panel 2-4

2.4.2.1 SERVO MOTOR 2-4

2.4.2.2 SERVO BRIDGE (only with Option 600-30) 2-4

2.4.2.3 Mains Supply (POWER module) 2-5

2.4.2.4 SYSTEM STATUS 2-5

2.4.2.5 USER CONTROL 2-5

2.5 Connecting the System Components and Start Up 2-6

2.6 Manual Control 2-6

2.7 Control with External Controller 2-6

2.8 PC Control 2-6

2.9 Locating errors 2-7

1 Safety Precautions

1.1 Dedicated Usage

The laboratory setup PS600 is to be used:

- for demonstration purposes,
- for students practical courses in control education,
- as a base model for research.

Because the laboratory setup PS600 may be modified by mechanical extensions optionally delivered by amira the responsibility for the specific application is passed over to the user.

The laboratory setup PS600 is to be used only in those areas where persons cannot be endangered during operating the setup.

1.2 Potential Danger and Danger Areas

The cart must not be touched during operation because of bruising danger. Using the laboratory setup PS600 as an inverted pendulum, tandem pendulum or a loading bridge a protection area is required with the dimensions so that no person or object may be touched by moving parts of the mechanics.

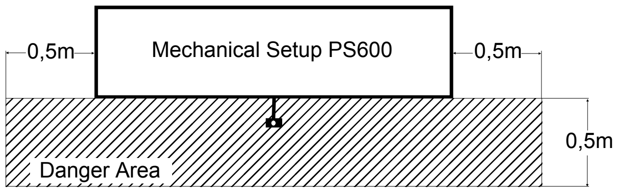


Figure 1.1: Definition of the danger area of the PS600

Danger areas are indicated at the setup by a yellow rectangle containing an exclamation mark.

1.3 General Risks on Disregarding the Safety Precautions

The product is designed according to the state of the art and it is operationally reliable. But the product may cause risks when it is modified or operated by unauthorized or uneducated persons or it is used in a specific application which causes risks.

Each person engaged to modify, operate, maintain or repair the PS600 must have read and understand these instructions especially the chapter "Safety Precautions".

1.4 Maintenance under Safety Regulations

The laboratory setup PS600 must only be modified by authorized and educated persons. Replacing expendable parts or spare parts must only be carried-out after consulting our service engineers. Otherwise only our service engineers are authorized for maintenance.

1.5 Safety Regulations for the Customer

Superior persons have to familiarize themselves with the complete chapter "Safety Precautions" and with the required operations of the laboratory setup PS600.

Superior persons have to take care that persons operating the setup have read, understand and follow the chapter "Safety Precautions" and the documentation of the corresponding operations.

The laboratory setup PS600 must be operated only under perfect conditions of the setup.

Each operation reducing the operational reliability of the laboratory setup PS600 has to be omitted.

Before operating the laboratory setup PS600 the chapters "Safety Precautions" and "Assembly and Start-Up" have to be read carefully.

The operating persons are obliged to check regularly the laboratory setup PS600 for externally visible damages and defects. When any detected changes (including the operation mode) reduce the operational reliability please contact us immediately. Components and accessories are especially designed for this product. Only original parts have to be used when replacing expendable parts or spare parts.

It is to be noticed particularly that original parts or accessories not purchased from us are neither proofed nor released by us in addition. Installing and/or using such products may change preassigned features of the setup and with this reduce the active or passive operational reliability.

In case of damages due to the usage of original parts and accessories not purchased from us any warranty of the manufacturer is excluded.

Safety devices must not be removed or deactivated in principle.

Protective parts must not be made ineffective or by-passed.

1.6 Forbidden Modifications without any Authorization

Neither the construction nor the safety design of the laboratory setup PS600 is to be modified without our agreement. Any unauthorized modification in this sense causes the exclusion of our warranty.

Only the mechanical extension kits, optionally offered by us, may be installed or removed. The installation procedure is described in the chapter "Assembly and Start-Up".

2 Assembly and Start-Up

2.1 Unpacking

After the PS600 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 PS600 consists of:

- The PS600 mechanics consisting of a DC motor, an incremental encoder, a guiding bar and a cart assembled completely inside a protecting Plexiglas cover. The two leads to connect the PS600 mechanics to the actuator are fixed to the mechanics.
- 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 mains supply lead.
- The detailed documentation of the hardware and software.

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

- Option 600-02, an external PC with keyboard, monitor, PC plug-in card DAC98 with a lead to connect to the actuator.
- Option 600-03, a PC plug-in card DAC98 with a lead to connect to the actuator.
- Option 600-12, a 3,5" floppy disk containing the executable controller program, corresponding documentation including a students practical course for the position control plant.
- Option 600-13, the Option 600-12 is extended by a fuzzy controller. The 3,5" floppy disk contains the executable fuzzy controller program.
- Option 600-15, a 3,5" floppy disk containing the C source files of the programs from the options 600-12 and 600-13 with additional library functions for fuzzy operations and graphic output.
- Option 600-20, an extension kit pendulum mechanics for the position control plant.
- Option 600-22, an executable digital controller program for students practical courses and the documentation for the inverted pendulum.
- Option 600-23, the Option 600-22 is extended by a fuzzy controller. The 3,5" floppy disk contains the executable fuzzy controller program.
- Option 600-25, a 3,5" floppy disk containing the C source files of the programs from the options 600-22 and 600-23 with additional library functions for fuzzy operations and graphic output.
- Option 600-30, an extension kit loading bridge mechanics for the position control plant.
- Option 600-32, an executable digital controller program for students practical courses and the documentation for the loading bridge.
- Option 600-33, the Option 600-32 is extended by a fuzzy controller. The 3,5" floppy disk contains the executable fuzzy controller program.
- Option 600-35, a 3,5" floppy disk containing the C source files of the programs from the options 600-32 and 600-33 with additional library functions for fuzzy operations and graphic output.
- Option 600-40, an extension kit tandem pendulum mechanics for the position control plant.
- Option 600-42, an executable digital controller program and the documentation for the tandem

pendulum.

- Option 600-43, the Option 600-42 is extended by a fuzzy controller. The 3,5" floppy disk contains the executable fuzzy controller program.

- Option 600-45, a 3,5" floppy disk containing the C source files of the programs from the options 600-42 and 600-43 with additional library functions for fuzzy operations and graphic output.

2.2 Setting up the System

2.2.1 The PS600 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.

The system must be placed on a solid surface supporting the weight. The mechanics must not project over the floor space. Please regard the safety hints of chapter 1.

2.2.2 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).

2.3 Description of the PS600 Mechanical Setup

Aluminium profiles form the platform and the framework of the laboratory setup which is covered by sheets of Plexiglas. The cart is driven along a guiding bar with an approximate length of 1,5m by a DC-motor, a clutch, a toothwheel and a toothbelt. Two proximity switches are mounted near to the guiding bar indicating the left and the right limit position of the cart. The position of the cart is measured by an incremental encoder. This sensor and the drive are mounted at the left side of the system. A dummy plug is connected to the small connector casing in the middle of the system when operating as a position control plant.

2.3.1 Description of the Pendulum Mechanics (Option 600-20)

The pendulum mechanics is premounted on a solid plate. The pendulum is joined to the pendulum axis rotating between two bearing blocks mounted on top of the plate. An incremental encoder is joined to the free shaft of the pendulum axis to measure the angle of the pendulum. All the electrical connections are contained in a plastic chain which is fixed to the pendulum mechanics. A connector is located at the free end of the plastic chain.

The following describes the conversion of the system position control plant to the system inverted pendulum. This conversion is carried out only by using the opening at the front side of the mechanics. Remove the dummy plug from the connector casing in the middle of the system. Then mount the pendulum mechanics on top of the cart of the PS600 using the four knurled screws. The screws fit through according holes in the plate of the pendulum mechanics and are fastened on the cart. Now the connector of the plastic chain is plugged in the connector casing and the chain itself is fixed with another knurled screw. To this end the system is ready for operation.

2.3.2 Description of the Loading Bridge Mechanics (Option 600-30)

The mechanics of the loading bridge is premounted completely on top of a solid plate. A rope running around two deflection rollers connects the load to the winch. The length of the rope is measured by an incremental encoder. Another incremental encoder measures the angle of the rope to the vertical. All the electrical connections are contained in a plastic chain which is fixed to the loading bridge mechanics. A connector is located at the free end of the plastic chain.

The following describes the conversion of the system position control plant to the system loading bridge. This conversion is carried out only by using the opening at the front side of the mechanics. Remove the dummy plug from the connector casing in the middle of the system. Then mount the loading bridge mechanics on top of the cart of the PS600 using the four knurled screws. The screws fit through according holes in the plate of the loading bridge mechanics and are fastened on the cart. Now the connector of the plastic chain is plugged in the connector casing and the chain itself is fixed with another knurled screw. To this end the system is ready for operation.

2.3.3 Description of the Tandem Pendulum Mechanics (Option 600-40)

The tandem pendulum mechanics is premounted on a solid plate. A short and a long pendulum are joined to the corresponding pendulum axis rotating between two bearing blocks mounted on top of the plate. An incremental encoder is joined to each shaft of the pendulum axis to measure the angles of the pendulums. All the electrical connections are contained in a plastic chain which is fixed to the pendulum mechanics. A connector is located at the free end of the plastic chain.

The following describes the conversion of the system position control plant to the system tandem pendulum. This conversion is carried out only by using the opening at the front side of the mechanics. Remove the dummy plug from the connector casing in the middle of the system. Then mount the tandem pendulum mechanics on top of the cart of the PS600 using the four knurled screws. The screws fit through according holes in the plate of the tandem pendulum mechanics and are fastened on the cart. Now the connector of the plastic chain is plugged in the connector casing and the chain itself is fixed with another knurled screw.

Important:
The long pendulum has to be fixed to the inner position, the short pendulum to the outer position.

To this end the system is ready for operation.

2.4 Description of the PS600 Actuator

2.4.1 The Rear Panel

Figure 2.1 displays the components located on the rear panel. The mains input unit is located on the right side. It contains a fuse holder with one fuse, the mains inlet and the power switch. The following connectors are located on the rear panel:

- the 30-polar and 4-polar connector for the mechanical system
- the 50-polar socket to connect the actuator to the PC plug-in card.

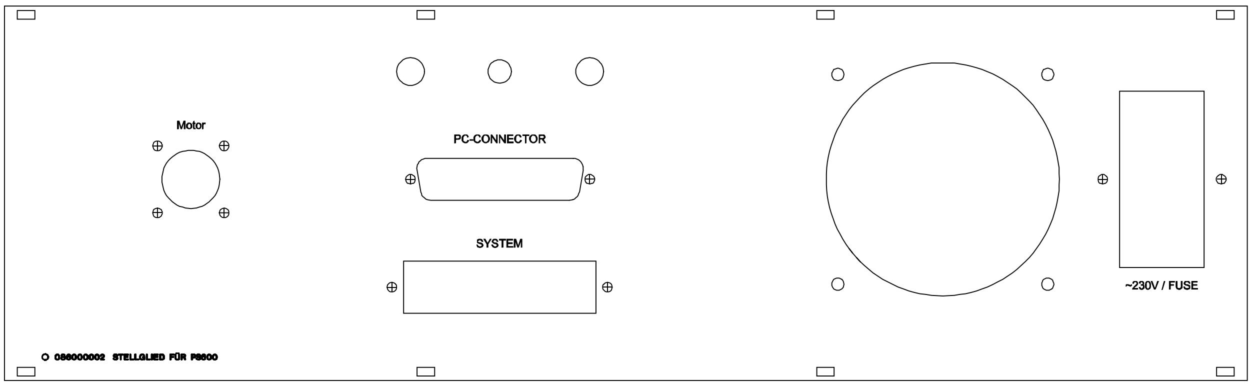


Figure 2.1 : Rear panel with denotations

2.4.2 The Front Panel

Figure 2.2 displays the components located on the front panel.

2.4.2.1 SERVO MOTOR

The servo amplifier functioning as a current controller for the motor is located on the left side of the front panel. 8 LEDs indicate its operating modes:

- On (green): The supply voltage is active.
- Ready (green): All required operating conditions are satisfied.
- I-Max ready (green): The servo amplifier is ready to provide the maximum current which is higher than the rated (nominal) current.
- Limit (red): The drive is working at the preadjusted nominal current limit. (in case of dynamical acceleration manoeuvres this indicator might light up briefly even in case no external momentum is needed.)
- Tacho (red): The tacho signal is disturbed (e.g. lead disruption or disturbance signals on the tacho input). Additionally to this LED, the disengagement LED lights up, because the controller is disengaged internally.
- Temperature (red): The internal temperature supervision of the output stage mounted on the cooling flange has responded. Additionally to this LED, the disengagement LED lights up, because the controller is disengaged internally.
- Disengagement (red): The release signal is not active, the actuator is disengaged, i.e. the output stage is without current. Additionally to the cases described previously, this is also possible in case the actuator is in the reset mode, or one of the two direction depending release circuits is open and the setpoint input requests at the same time the corresponding direction of rotation.
- Load (red): The upper limit of the internal intermediate circuit voltage has been exceeded.

2.4.2.2 SERVO BRIDGE (only with Option 600-30)

SERVO BRIDGE is the servo amplifier module for the loading bridge. It serves for changing the length of the rope by driving the winch accordingly. Its operation mode is indicated by two LEDs:

- Limit (red): The rope is winded up or off completely.
- Ready (green): All required operating conditions are satisfied.

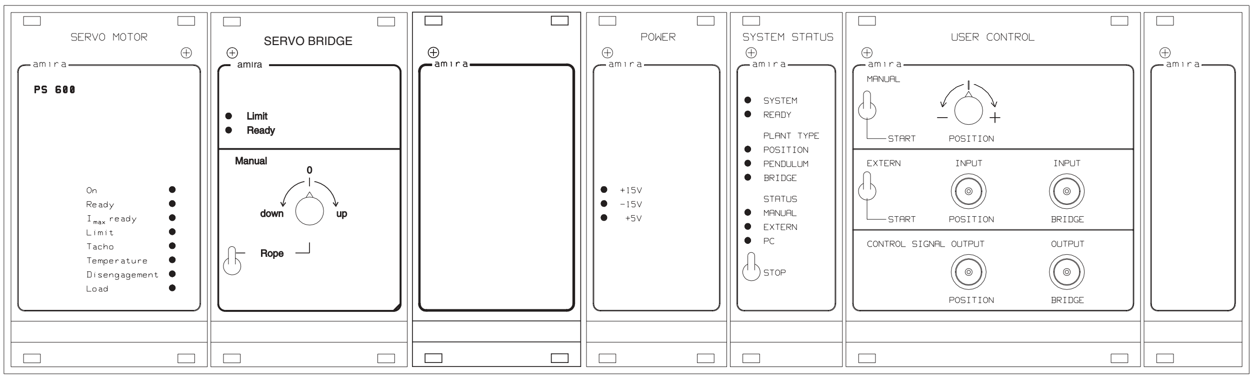


Figure 2.2 : Front panel with denotations

The Servo-Bridge-Module may be operated manually using the following components.

- Key Automatic/Manual: The nominal position of this key is "Automatic", meaning that the force for the rope winch is activated by a digital controller. The force is adjusted by a potentiometer when the key is hold in the position "Manual".

- Potentiometer: As long as the key is hold in the position "Manual", the potentiometer can be used to adjust the force for the rope winch.

2.4.2.3 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.

2.4.2.4 SYSTEM STATUS

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

Plant Type:

- Position (green): Lights up when the system "position control plant" was detected.

- Pendulum (green): Lights up when the system "inverted pendulum" or "tandem pendulum" was detected.

- Bridge (green): Lights up when the system "loading bridge" was detected.

Status:

- Manual (red): Manual control is enabled

- PC (red): PC control is enabled

- Extern (red): An external input is enabled

- Stop: Using this key will disconnect the input signals of the servo amplifiers.

2.4.2.5 USER CONTROL

This module allows a direct control of the servo amplifier either by using a potentiometer or by connecting an external controller.

- Potentiometer MANUAL: Allows for manual adjusting the control signal (force) for the cart.

- 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 INPUT POSITION: Input for setpoint of the motor current driving the cart (range ±10V, 1V=2.25N).

- BNC socket INPUT BRIDGE: Input for setpoint of the motor current driving the rope winch of the loading bridge (range ±10V). This input has no meaning for other systems.

- Key EXTERN START: Connects the control signal provided at the socket CONTROL SIGNAL 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.

- BNC socket OUTPUT POSITION: This measurement output is used to supervise the control signal of the motor current driving the cart.

- BNC-socket OUTPUT BRIDGE: This measurement output is used to supervise the control signal of the motor current driving the rope winch.

2.5 Connecting the System Components and Start Up

Before setting up the system, please check whether your mains supply is identically to the mains supply indicated on the type plate (230V, 50/60Hz resp. 110V, 50/60Hz). Connect the actuator to the mains supply and switch on the actuator. The LED's +15V, -15V and +5V 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 LEDs mentioned above the LEDs "System" and "Ready" should light up. After pressing the "Stop" key, all LEDs except for the LED "System" of the module "SYSTEM STATUS" should light up. The connected type of plant has to be indicated correctly by the LEDs "Position", "Pendulum" or "Bridge".
The "Ready" LEDs of the servo modules have to light up. The system is now ready for operation.

2.6 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 cart. The control signal may be measured at the terminal "CONTROL SIGNAL" (1V = 2,25N). 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.

When the Option 600-30 is installed in the system, the force for the rope winch of the loading bridge may be controlled manually in addition. To do this switch the key from the module "SERVO BRIDGE" to the position "Manual" and adjust the force for the rope winch by means of the potentiometer located below the key.

2.7 Control with External Controller

The signals for the determination of the measured value of the cart position are provided at the 50-pol. connector located on the rear of the actuator (see Technical Data). When additional options are installed the same is true for the pendulum angles (Option 600-20, Option 600-40) or for the rope angle and the rope length (Option 600-30). The cart position like the other signals is measured by a quadrature incremental encoder. Any external controller has to be able to process such signals. One increment corresponds to 43,95μm. The output of an external controller is to be connected to the terminal "INPUT POSITION" (range ±10V, 2.25 N/V). The control signal is connected to the servo amplifier of the cart drive 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 "SYSTEM STATUS".

2.8 PC Control

Controlling the system by a PC is described in the chapter "Operating Instructions" (only available with Option 600-12, 600-22, 600-32 or 600-42). 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 may be started immediately.

If you use your own PC controller please think of the release of the output stage. The output stage release is a safety function so that in case of program failure or exceeding of the max. system signals like max. pendulum angle (has to be programmed) the motor stops immediately.

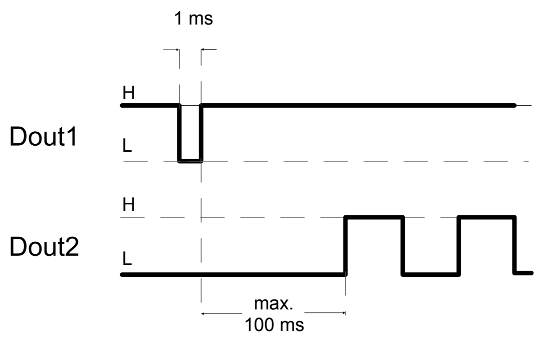


Figure 2.3 : Signals for the output stage release

You need two digital signals for the output stage release. Dout1 (pin 35 of the 50-pol. connector) gets first a high-level with pulse to low, duration about 1 ms. When Dout1 has completed its pulse the digital output Dout2 (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. 2.3).

2.9 Locating errors

First try to eliminate problems with the help of the following table. In case you cannot solve problems with your PS600 yourself, please contact us.

Problem	Possible cause
The LEDs do not light up	Check the connection to the mains supply Check the fuse for the mains supply (rear panel)
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
Neither "Position", "Pendulum" nor "Bridge" light up	Check the lead connection between mechanical system and actuator. Check the lead connection between the plastic chain and the connector casing. A dummy plug has to be connected when the system operates as a position control plant.
Selected controller cannot be enabled	Check whether another controller, is enabled (LEDs on "POWER SERVO" module) Check LED "Ready" lights up
LEDs on SERVO MOTOR module do not light up	Please contact us.

Technical Data

Date: 21-March-2002

1 Technical Data 1-1

1.1 PS600-Mechanics, Drive, Sensors . . . 1-1

1.1.1 Dimensions and Weight of the PS600 Mechanics . . . 1-1

1.1.2 The Drive . . . 1-1

1.1.3 PS600 Sensors . . . 1-1

1.1.4 Extension Kit Pendulum Mechanics (Opt. 600-20) . . . 1-2

1.1.5 Extension Kit Loading Bridge (Opt. 600-30) . . . 1-2

1.1.6 Extension Kit Tandem Pendulum Mechanics (Opt. 600-40) . . . 1-4

1.2 Actuator . . . 1-4

1.2.1 SERVO MOTOR Module . . . 1-5

1.2.2 SERVO BRIDGE Module . . . 1-5

1.2.3 POWER Module . . . 1-5

1.2.4 USER CONTROL . . . 1-5

1.2.5 Rear Panel Connections . . . 1-6

1 Technical Data

1.1 PS600-Mechanics, Drive, Sensors

1.1.1 Dimensions and Weight of the PS600 Mechanics

Dimensions and Weight	Value	Unit
Length	1880	mm
Depth	440	mm
Height	290	mm
Weight	42	kg

1.1.2 The Drive

Principle: 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 DC-motor	Value	Unit
Rated voltage	42	V
Rated speed	1600	Rpm
Rated current	5	A
Rated power	160	W
Degree of protection according to DIN 40050	IP 54	
Armature weight	1.4	kg
Motor weight	4.1	kg
Direction of rotation	reversible	
Distance variation of the cart with resp. to one rotation of the motor	180	mm
Moment of inertia	0.00037	kgm^2
Rated torque	0.955	Nm
Starting torque	6	Nm
Max. continuous torque during stillstand	1.1	Nm
Friction torque	0.1	Nm
Speed regulation per moment	2.7	$^1/_{min} * ^1/_{Ncm}$
Mechanical time constant	10.5	ms
Connected resistance	1.28	Ω
Armature inductance	2.8	mH
Armature resistance	1.08	Ω
Voltage constant	22.2	mV / $^1/_{min}$
Torque constant	0.212	Nm / A
Starting current	32	A
Max. peak current	35	A
Electrical time constant	2.2	ms
Max. temperature	40	°C
Degree of isolation according to VDE 0530	F	
Thermal time constant	40	min
Increase of temperature without cooling	ca. 2	K / W

1.1.3 PS600 Sensors

Proximity switch Type K58BÖ/1	Value	Unit
Power supply	15	V
Current consumption without load	11	mA
Switch type	break contact	

Incremental Encoder Type RI58-D	Value	Unit
Mode of attachment Shaft diameter: Incr. encoder for position	hollow shaft internal-bearings 12	 mm
Max. speed	10000	$^1/_{min}$
Torque	<1	Ncm
Moment of inertia Mass	0.02 0.17	kgcm^2 kg
Resolution	1024 22755	lines/ rotation incr./m
Output interface Power supply	RS 422 5	 V
Impulse channels Impulse wave shape Phase shift A/B Index signal N,/N	A,/A B,/B N,/N rectangle 90 1	 ° 1/Rotation

1.1.4 Extension Kit Pendulum Mechanics (Opt. 600-20)

Incremental Encoder Type RI58-D	Value	Unit
Mode of attachment Shaft diameter:	hollow shaft internal-bearings 10	 mm
Max. speed	10000	$^1/_{min}$
Torque	<1	Ncm
Moment of inertia Mass	0.02 0.17	kgcm^2 kg
Resolution	4096 45,5	lines/ rotation incr./°
Output interface Power supply	RS 422 5	 V
Impulse channels Impulse wave shape Phase shift A/B Index signal N,/N	A,/A B,/B N,/N rectangle 90 1	 ° 1/Rotation

Dimensions and Weights	Value	Unit
Width of base plate	145	mm
Overall height (without Pendulum)	70	mm
Overall Depth	240	mm
Total Weight	2	kg
Pendulum length	500	mm
Weight of pendulum rod	100	g
Weight of pendulum	210	g

1.1.5 Extension Kit Loading Bridge (Opt. 600-30)

Incremental Encoder Type RI58-D	Value	Unit
Mode of attachment Shaft diameter: Incr. encoder for angle	hollow shaft internal-bearings 12	 mm
Max. speed	10000	$^1/_{min}$

Incremental Encoder Type RI58-D	Value	Unit
Torque	<1	Ncm
Moment of inertia Mass	0.035 0.17	$kgcm^2$ kg
Resolution	4096 45,5	lines/rot. incr./°
Output interface Power supply	RS 422 5	 V
Impulse channels Impulse wave shape Phase shift A/B Index signal N,/N	A,/A B,/B N,/N rectangle 90 1	 ° 1/Rotation

Incremental Encoder Type M250	Value	Unit
Structural shape Shaft diameter:	"Kit"- structural shape, self-locating 6	 mm
Max. speed	6000	$^1/_{min}$
Moment of inertia Mass	50 0.07	gcm^2 kg
Resolution	250 50	lines/rot. incr./cm
Output interface Power supply	TTL 5	 V
Impulse channels Impulse wave shape Phase shift A/B Index signal N	A B N rectangle 90 1	 ° 1/Rotation

Dimensions and Weights	Value	Unit
Width of base plate	145	mm
Overall height	120	mm
Overall Depth	460	mm
Total Weight	4,5	kg
Zero point below pivot	17	cm
Changeable rope length	50	cm
Weight of base load	210	g

Proximity switch topside Type NJ0,8-5GM27-E2	Value	Unit
Power supply	15	V
Current consumption without load	11	mA
Switch type	make contact	

Limit switch end of rope Type DB2	Value	Unit
Switch type	break contact	

Motor Permanent excited DC-motor with worm gear pair	Value	Unit
Rated voltage	24	V
Rated power	32	W
Rated speed	125	Rpm
Gear reduction	2:26	
Weight	1,1	kg

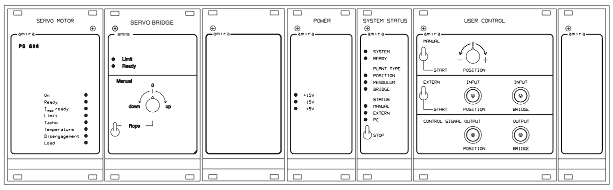
Winch	Value	Unit
Circumference	20	cm
Max. number of turns	8	

1.1.6 Extension Kit Tandem Pendulum Mechanics (Opt. 600-40)

Incremental Encoder Type RI58-D	Value	Unit
Mode of attachment Shaft diameter:	hollow shaft internal-bearings 10 and 12	 mm
Max. speed	10000	$^1/_{min}$
Torque	<1	Ncm
Moment of inertia Mass	about 0.035 0.17	kgcm^2 kg
Resolution	4096 45,5	lines/ rotation incr./°
Output interface Power supply	RS 422 5	 V
Impulse channels Impulse wave shape Phase shift A/B Index signal N,/N	A,/A B,/B N,/N rectangle 90 1	 ° 1/Rotation

Dimensions and Weights	Value	Unit
Width of base plate	145	mm
Overall height (without Pendulums)	75	mm
Overall Depth	290	mm
Total Weight	2,5	kg
Pendulum length 1	500	mm
Pendulum length 2	100	mm
Weight of pend. rod 1	100	g
Weight of pend. rod 2	20	g
Weight of pendulum 1	210	g
Weight of pendulum 2	210	g

1.2 Actuator



Dimensions and weight	Value	Unit
Length	470	mm
Depth	320	mm
Height	156	mm
Weight	10	kg

Mains supply	Value	Unit
Input voltage	230	V
Frequency	50/60	Hz
Power consumption max.	400	W
Fuses S1 and S2	2	A T

1.2.1 SERVO MOTOR Module

Inputs	Value	Unit
Supply voltage Fuse	42 20	V~ A
Limit position right Limit position left (break contact against "COM" disables servo)	break contact break contact	
Control signal	-10... +10	V
Output stage release (break contact against "COM" enables servo)	break contact	

Outputs	Value	Unit
Motor terminal voltage	± 42	V
Motor terminal current	8	A
Dyn. peak current	24	A
Rated power	340	W

1.2.2 SERVO BRIDGE Module

Inputs	Value	Unit
Supply voltages Current consumption	±35 2	V A
Control signal	-10... +10	V

Output	Value	Unit
Motor terminal voltage	± 30	V

1.2.3 POWER Module

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

1.2.4 USER CONTROL

Inputs (BNC-sockets)	Value	Unit
Input Position	± 10 0.8	V A/V
Input Bridge	± 10 0.2	V A/V

Outputs (BNC-sockets)	Value	Unit
Output Position	± 10 0.8	V A/V
Output Bridge	± 10, 0.2	V A/V

1.2.5 Rear Panel Connections

PC-Connector
a: Position control plant
b: Inverted pendulum (Opt. 600-20)
c: Loading bridge (Opt. 600-30)
d: Tandem pendulum (Opt. 600-40)

Pin-No.	Pin-Den.	Reservation	a	b	c	d
1	INC0 CHA	Position channel A	x	x	x	x
2	INC0 CHB	Position channel B	x	x	x	x
3	INC1 CHA	Pendulum angle / Rope angle channel A		x	x	x
4	INC1 CHB	Pendulum angle / Rope angle channel B		x	x	x
5	INC2 CHA	Rope length channel A / Short pendulum			x	x
6	INC2 CHB	Rope length channel B / short pendulum			x	x
7	INC3 CHA	n.c.				
8	INC3 CHB	n.c.				
9	DIN0	System identification Bit 0	x	x	x	x
10	DIN1	System identification Bit 1	x	x	x	x
11	DIN2	Key Manual	x	x	x	x
12	DIN3	n.c.				
13	OUT1	n.c.				
14	OUT2	n.c.				
15	AGND	AGND	x	x	x	x
16	n.c.	n.c.				
17	n.c.	n.c.				
18	INC0 /CHA	Position channel A (inverted)	x	x	x	x
19	INC0 /CHB	Position channel B (inverted)	x	x	x	x
20	INC1 /CHA	Pendulum angle / Rope angle channel A (inverted)		x	x	x
21	INC1 /CHB	Pendulum angle / Rope angle channel B (inverted)		x	x	x
22	INC2 /CHA	Rope length / Short pendulum channel A (inverted)			x	x
23	INC2 /CHB	Rope length / Short Pendulum channel B (inverted)			x	x
24	INC3 /CHA	n.c.				
25	INC3 /CHB	n.c.				
26	DIN4	PC-READY	x	x	x	x
27	DIN5	Right limit switch	x	x	x	x
28	DIN6	Left limit switch	x	x	x	x
29	DIN7	Limit switch rope			x	
30	n.c.	n.c.				
31	AGND	AGND	x	x	x	x
32	n.c.	n.c.				
33	n.c.	n.c.				
34	Dout0	n.c.				
35	Dout1	Release circuit pulse	x	x	x	x
36	Dout2	Release circuit rectangle	x	x	x	x
37	Dout3	n.c.				

PC-Connector
a: Position control plant
b: Inverted pendulum (Opt. 600-20)
c: Loading bridge (Opt. 600-30)
d: Tandem pendulum (Opt. 600-40)

Pin-No.	Pin-Den.	Reservation	a	b	c	d
38	Dout4	n.c.				
39	Dout5	n.c.				
40	Dout6	n.c.				
41	Dout7	n.c.				
42	DGND	DGND	x	x	x	x
43	DGND	DGND	x	x	x	x
44	IN1	n.c.				
45	IN2	n.c.				
46	IN3	n.c.				
47	AOUT0	Control signal for SERVO MOTOR	x	x	x	x
48	AOUT1	Control signal for SERVO BRIDGE			x	
49	n.c.	n.c.				
50	n.c.	n.c.				

System Connector (30-pol.)
a: Position control plant
b: Inverted pendulum (Opt. 600-20)
c: Loading bridge (Opt. 600-30)
d: Tandem pendulum (Opt. 600-40)

Pin-No.	Reservation	a	b	c	d
a1	Pendulum angle / Rope angle channel B (inverted)		x	x	
a2	Pendulum angle / Rope angle channel B		x	x	
a3	Pendulum angle / Rope angle channel A (inverted)		x	x	
a4	Pendulum angle / Rope angle channel A		x	x	
a5	Position channel B (inverted)	x	x	x	x
a6	Position channel B	x	x	x	x
a7	Position channel A (inverted)	x	x	x	x
a8	Position channel A	x	x	x	x
a9	Shield	x	x	x	x
a0	PE	x	x	x	x
b1	DGND	x	x	x	x
b2	+5VD	x	x	x	x
b3	+15V	x	x	x	x
b4	Lower limit switch rope			x	
b5	Rope length / Short pendulum channel B (inverted)			x	x
b6	Rope length / Short pendulum channel B			x	x
b7	Rope length / Short pendulum channel A (inverted)			x	x
b8	Rope length / Short pendulum channel A			x	x
b9	Motor loading bridge minus			x	
b0	n.c.				
c1	Upper limit switch rope			x	

System Connector (30-pol.)
a: Position control plant
b: Inverted pendulum (Opt. 600-20)
c: Loading bridge (Opt. 600-30)
d: Tandem pendulum (Opt. 600-40)

Pin-No.	Reservation	a	b	c	d
c2	n.c.				
c3	Left limit switch	x	x	x	x
c4	Right limit switch	x	x	x	x
c5	System identification Bit 1	x	x	x	x
c6	System identification Bit 0	x	x	x	x
c7	n.c.				
c8	Shield motor position control	x	x	x	x
c9	Motor loading bridge plus			x	
c0	n.c.				

System Connector (4-pol.)
a: Position control plant
b: Inverted pendulum (Opt. 600-20)
c: Loading bridge (Opt. 600-30)
d: Tandem pendulum (Opt. 600-40)

Pin-No.	Reservation	a	b	c	d
A	PE	x	x	x	x
B	Motor position control minus	x	x	x	x
C	Motor position control plus	x	x	x	x
D	n.c.				

Subject to change without further notice (Date: 21-March-2002)

State Control and Practical Instructions

Inverted Pendulum

Date: 24-May-1996

1 Contents and Objectives of the Experiment 1-1

2 Theoretical Foundations 2-1

2.1 State Equations of Linear Sampled Data Systems . . . 2-1

2.2 Sampled Data Control with Feedback of the state vector . . . 2-3

2.3 The Luenberger State Observer . . . 2-5

3 The System "Inverted Pendulum" 3-1

3.1 Mathematical Model of the Inverted Pendulum . . . 3-1

3.2 Description of the Linearized System in the State Space . . . 3-2

3.3 Parameters of the Mathematical "Inverted Pendulum" Model . . . 3-3

3.4 Control and Disturbance Signal Observation on the "Inverted Pendulum" . . . 3-5

3.5 Design of the Luenberger Identity Observer . . . 3-8

4 Preparations for the Laboratory Experiment 4-1

5 Set-Up of the Experiment 5-1

6 Carrying-Out the Experiment 6-1

7 Experiment Elaboration 7-1

8 References 8-1

9 Solutions 9-1

9.1 Solutions of the Preparation Problems from Chapter 4 . . . 9-1

9.2 Solutions to Chapter 6 (Carrying-Out the Experiment) 9-5

9.3 Solutions to Chapter 7 (Experiment Elaboration) 9-21

1 Contents and Objectives of the Experiment

By means of state observers it is possible to reconstruct the state vector of observable systems using a mathematical model which is excited by the input and output signals.

Possible fields of application for state observers are for example instrument fault detection algorithms based on software redundancy, the realization of state space controllers without measurement of the state vector, or the estimation of unmeasurable disturbance signals in physical systems.

First of all, the linear system description is introduced in this experiment. Subsequently, the design and calculation of the Luenberger identity observer via the pole placement method is explained. To put the observer to a practical test, the algorithm was implemented on a PC and applied for purposes of state observation on the laboratory model "inverted pendulum".

By assignment of different observer poles, the dynamical behaviour of the observer is varied. The resulting effects of these variations on the observer estimates is measured. By comparison of the state estimates with the actual states in the physical system, the respective estimation errors can be determined and evaluated. The results demonstrate, which prerequisites for a meaningful state observation must be fulfilled for the example "inverted pendulum".

2 Theoretical Foundations

2.1 State Equations of Linear Sampled Data Systems

Time invariant physical systems can generally be described by n-th order nonlinear differential equations.

If the differential equation is linearized about a preassigned operating point and the differentials are defined as the new state vector,

$$\underline{x} = \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \tag{2.1}$$

one obtains with the input vector

$$\underline{u} = \begin{bmatrix} u_1 \\ . \\ . \\ . \\ u_p \end{bmatrix} \tag{2.2}$$

and the output vector

$$\underline{y} = \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_q \end{bmatrix} \tag{2.3}$$

a linear system of first order differential equations describing the state equation

$$\dot{\underline{x}}\,(t) = \underline{A}\,\underline{x}\,(t) \;+\; \underline{B}\,\underline{u}\,(t) \tag{2.4}$$

and the output equation

$$\underline{y}\,(t) = \underline{C}\,\underline{x}\,(t) \;+\; \underline{D}\,\underline{u}\,(t) \tag{2.5}$$

The integer n is the order of the system, p denotes the number of inputs and q the number of output signals. The expression

$$\underline{A} = \begin{bmatrix} a_{11}\; a_{12} \cdots a_{1n} \\ a_{21}\; a_{22} \cdots a_{2n} \\ \cdot \quad \cdot \quad \cdots \quad \cdot \\ a_{n1}\; a_{n2} \cdots a_{nn} \end{bmatrix} \tag{2.6}$$

is called system matrix and

$$\underline{B} = \begin{bmatrix} b_{11}\; b_{12} \cdots b_{1p} \\ b_{21}\; b_{22} \cdots b_{2p} \\ \cdot \quad \cdot \quad \cdots \quad \cdot \\ b_{n1}\; b_{n2} \cdots b_{np} \end{bmatrix} \tag{2.7}$$

denotes the input matrix of the system. The matrix

$$\underline{C} = \begin{bmatrix} c_{11}\; c_{12} \cdots c_{1n} \\ c_{21}\; c_{22} \cdots c_{2n} \\ \cdot \quad \cdot \quad \cdots \quad \cdot \\ c_{q1}\; c_{q2} \cdots c_{qn} \end{bmatrix} \tag{2.8}$$

is called output matrix and

$$\underline{D} = \begin{bmatrix} d_{11}\; d_{12} \cdots d_{1p} \\ d_{21}\; d_{22} \cdots d_{2p} \\ \cdot \quad \cdot \quad \cdots \quad \cdot \\ d_{q1}\; d_{q2} \cdots d_{qp} \end{bmatrix} \tag{2.9}$$

is the feed through matrix of the system. Methods for the linearization of differential equations or nonlinear control systems as well as for the formulation of the state equations can be found for instance in /1/ page 76 as well as in other textbooks on control theory. According to /1/, the solution of the state equation Eq.(2.4) can be stated in vector form

$$\underline{x}\,(t) = \underline{\Phi}\,(t - t_o)\,\underline{x}\,(t_o) + \int_{t_o}^{t} \underline{\Phi}\,(t - \tau)\,\underline{B}\,\underline{u}\,(\tau)\,d\tau \tag{2.10}$$

where the matrix

$$\underline{\Phi}\,(t) = e^{\underline{A}\,t} = \underline{I} + \underline{A}\,t + \underline{A}^2 \frac{t^2}{2\,!} + \underline{A}^3 \frac{t^3}{3\,!} + \dots$$

is called fundamental or transition matrix. The states of sampled data systems can be computed by integration over the sampling period T with Eq. (2.10). In case the input signal is a piecewise constant time function

$$\underline{u}\,(t) = \underline{u}\,(k\,T) \qquad k\,T \le t \le (k+1)\,T \qquad (2.11)$$

one obtains for the state vector at time (k+1)T the equation /1/ page 330

$$\underline{x}\,(\,(k+1)\,T) = \underline{A}_D\,(T)\,\underline{x}\,(k\,T) + \underline{B}_D\,(T)\,\underline{u}\,(k\,T) \qquad (2.12)$$

where

$$\underline{A}_D\,(T) = e^{\underline{A}\,T} = \underline{I} + \underline{A}\,T + \underline{A}^2 \frac{T^2}{2\,!} + \underline{A}^3 \frac{T^3}{3\,!} + \dots \qquad (2.13)$$

and

$$\underline{B}_D\,(T) = \int_0^T \underline{A}_D\,(\tau)\,\underline{B}\,d\tau \qquad (2.14)$$

Accordingly, the output equation of the sampled data system becomes

$$\underline{y}\,(k\,T) = \underline{C}\,\underline{x}\,(k\,T) + \underline{D}\,\underline{u}\,(k\,T) \qquad (2.15)$$

Eq. (2.12) is a recursive formula which allows to compute the state variables at time (k+1)T using the input and state variables at time kT. To this end, the matrices $\underline{A}_D\,(T)$ and $\underline{B}_D\,(T)$ must be computed only once, because they are independent of the input signals. In figure 2.1 the block diagram of the sampled data system according to Eq.(2.12) and Eq.(2.15) is displayed.

In case 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

$$\det\,(z\,\underline{I} - \underline{A}_D) = 0 \qquad (2.16)$$

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

$$z = e^{T\,s} \qquad (2.17)$$

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

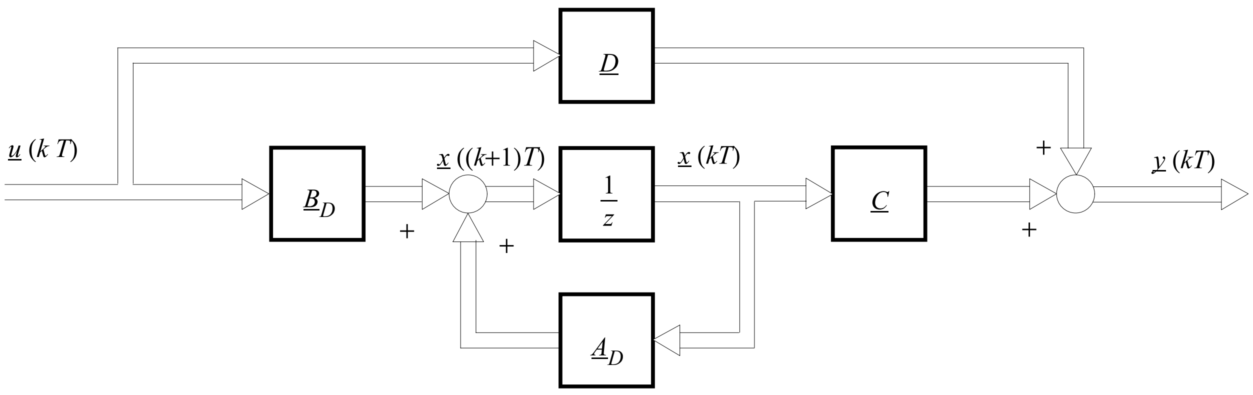


Figure 2.1 : Block diagram of the sampled data system

2.2 Sampled Data Control with Feedback of the state vector

In case a system description according to Eq.(2.12) and Eq.(2.15) is given, the system can be controlled by feedback of the state vector to the system input. If the system is controllable (definition in /2/ page 175 or /3/ chapter 4.7), the poles of the open loop system are shifted by a suitably selected feedback matrix $\underline{F}$ to stable locations. A prefilter which is described by the matrix $\underline{V}$, ensures in case p=q that in the steady state the output vector $\underline{y}$ is identical to the setpoint vector $\underline{w}$. The general structure of a sampled data control system is depicted in figure 2.2.

According to figure 2.2, the input signal $\underline{u}$ of the control system is now computed by the setpoint signal $\underline{w}$ and the state quantity $\underline{F}\,\underline{x}$ which is fed back

$$\underline{u}\,(k\,T) = \underline{V}\,\underline{w}\,(k\,T) - \underline{F}\,\underline{x}\,(k\,T) \quad . \tag{2.18}$$

This relation, substituted in Eq.(2.12) and Eq.(2.13), yields the system description of the closed loop control system

$$\underline{x}\,(\,(k+1)\,T) = (\underline{A}_D - \underline{B}_D\,\underline{F})\,\underline{x}\,(k\,T) + \underline{B}_D\,\underline{V}\,\underline{w}\,(k\,T) \tag{2.19}$$

$$\underline{y}\,(k\,T) = (\underline{C} - \underline{D}\,\underline{F})\,\underline{x}\,(k\,T) + \underline{D}\,\underline{V}\,\underline{w}\,(k\,T) \tag{2.20}$$

By comparison with the equations (2.12) and (2.15) the following relation between the open and the closed loop control systems can be derived:

$$\underline{A}_D \quad \to \quad \underline{A}_D - \underline{B}_D\,\underline{F} \tag{2.21}$$

$$\underline{B}_D \quad \to \quad \underline{B}_D\,\underline{V} \tag{2.22}$$

$$\underline{C} \quad \to \quad \underline{C} - \underline{D}\,\underline{F} \tag{2.23}$$

$$\underline{D} \quad \to \quad \underline{D}\,\underline{V} \tag{2.24}$$

According to Eq.(2.16), the stability behaviour can be evaluated by considering the eigenvalues of the new system matrix which can be computed from the characteristic equation

$$\det\,(\,z\,\underline{I} - (\underline{A}_D - \underline{B}_D\,\underline{F}\,)) = 0 \tag{2.25}$$

For the computation of the feedback matrix it is assumed here that the system has only one input signal ($\underline{u} = u$). In

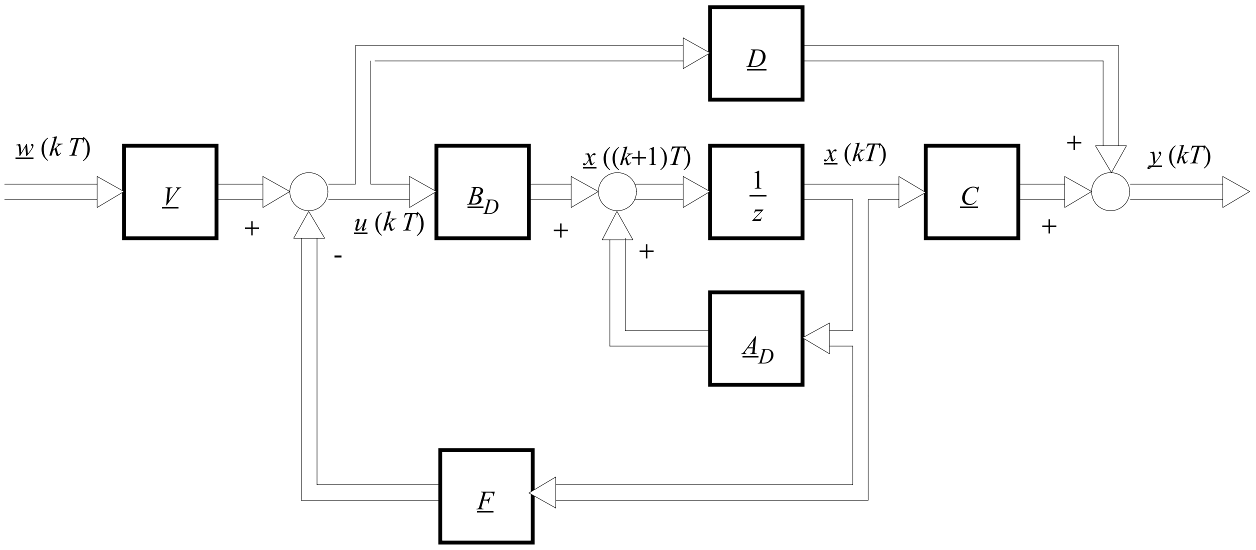


Figure 2.2 : Sampled data system with state feedback

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. If the characteristic polynomial

$$P(z) = z^n + p_{n-1} z^{n-1} + p_{n-2} z^{n-2} + \ldots + p_0 \tag{2.26}$$

of the closed loop control system is preassigned, the feedback vector can be determined by means of computation of

$$\det(z\underline{I} - (\underline{A}_D - \underline{b}_D \underline{f}^T)) \tag{2.27}$$

and a subsequent comparison with the coefficients of Eq.(2.26). In case the system description is given in form of a rational Z-transfer function

$$G(z) = \frac{b_0 + b_1 z + b_2 z^2 + \ldots + b_{n-1} z^{n-1}}{a_0 + a_1 z + a_2 z^2 + \ldots + a_n z^n}$$

it can be stated in the controller canonical form /3/ chapter 5.3.7

$$\underline{A}_D = \underline{A}_R = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & \cdot & & \cdot \\ -a_0 & -a_1 & -a_2 & -a_3 & \cdots & -a_{n-1} \end{bmatrix} \tag{2.28}$$

for the system matrix and

$$\underline{b}_D = \underline{b}_R = \begin{bmatrix} 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \\ 1 \end{bmatrix} \quad . \tag{2.29}$$

for the input matrix. This form allows to compute the feedback vector in a particularly simple fashion. A comparison of the coefficients of Eq.(2.27) and Eq.(2.26) yields the result

$$\underline{f}^T = [p_o - a_o, p_1 - a_1, p_2 - a_2, \ldots, p_{n-1} - a_{n-1}] \tag{2.30}$$

If the system is described in an arbitrary form, it is always possible to find a regular transformation under which the system can be written in the controller canonical form, provided the system is controllable. This computation is also performed in /3/ chapter 10.1.2. The assignment of the characteristic polynomial yielded for the feedback matrix according to Eq. (2.26)

$$\underline{f}^T = \underline{p}_o \underline{q}_s^T + \ldots + \underline{p}_{n-1} \underline{q}_s^T \underline{A}_D^{n-1} + \underline{q}_s^T \underline{A}_D^n \quad . \tag{2.31}$$

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

$$\underline{Q}_s^{-1} = [\underline{b}_D, \underline{A}_D \underline{b}_D, \ldots, \underline{A}_D^{n-1} \underline{b}_D]^{-1} \tag{2.32}$$

The prefilter can be determined by means of the final value theorem of the Z-transform. For the following explanations it is assumed that the feed through matrix $\underline{D}$ is identical to zero. Transforming the equations (2.19) and (2.20) into the Z-domain yields:

$$z\underline{X}(z) = (\underline{A}_D - \underline{B}_D \underline{F})\underline{X}(z) + \underline{B}_D \underline{V}\,\underline{W}(z) \tag{2.33}$$

$$\underline{Y}(z) = \underline{C}\,\underline{X}(z) \quad . \tag{2.34}$$

Solving Eq. (2.33) for $\underline{X}(z)$

$$\underline{X}(z) = (z\underline{I} - \underline{A}_D + \underline{B}_D \underline{F})^{-1} \underline{B}_D \underline{V}\,\underline{W}(z) \quad . \tag{2.35}$$

and substituting the result into Eq.(2.32) results in

$$\underline{Y}(z) = \underline{C}(z\underline{I} - \underline{A}_D \pm \underline{B}_D \underline{F})^{-1} \underline{B}_D \underline{V}\,\underline{W}(z) \, . \tag{2.36}$$

In this equation the term

$$\underline{G}(z) = \underline{C}(z\underline{I} - \underline{A}_D \pm \underline{B}_D \underline{F})^{-1} \underline{B}_D \underline{V} \tag{2.37}$$

denotes the Z-transfer function of the system. In the steady state the output vector can be computed by the final value theorem of the Z-transform to

$$\begin{aligned} \underline{y}(+\infty) &= \lim_{z\to 1} [\,(z-1)\,\underline{G}(z)\,\underline{W}(z)\,] \\ &= \underline{G}(1) \lim_{z\to 1} [\,(z-1)\,\underline{W}(z)\,] \\ &= \underline{G}(1)\,\underline{w}(+\infty) \qquad (2.38) \end{aligned}$$

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

$$\underline{G}(1) = \underline{I} \quad . \qquad (2.39)$$

This condition yields together with Eq.(2.37) the desired relation for the prefilter $\underline{V}$

$$\underline{V} = [\,\underline{C}\,(\underline{I} - \underline{A}_D + \underline{B}_D\,\underline{F})^{-1}\,\underline{B}_D\,]^{-1} \quad . \qquad (2.40)$$

This formula is particularly suitable for a computation of the prefilter by means of a digital computer. In case of single input-single output systems however, the prefilter can often be determined easier directly from the block diagram of the system.

2.3 The Luenberger State Observer

In chapter 2.2 the design of state feedback controllers for digital systems has been introduced. 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 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. for systems for which an unique

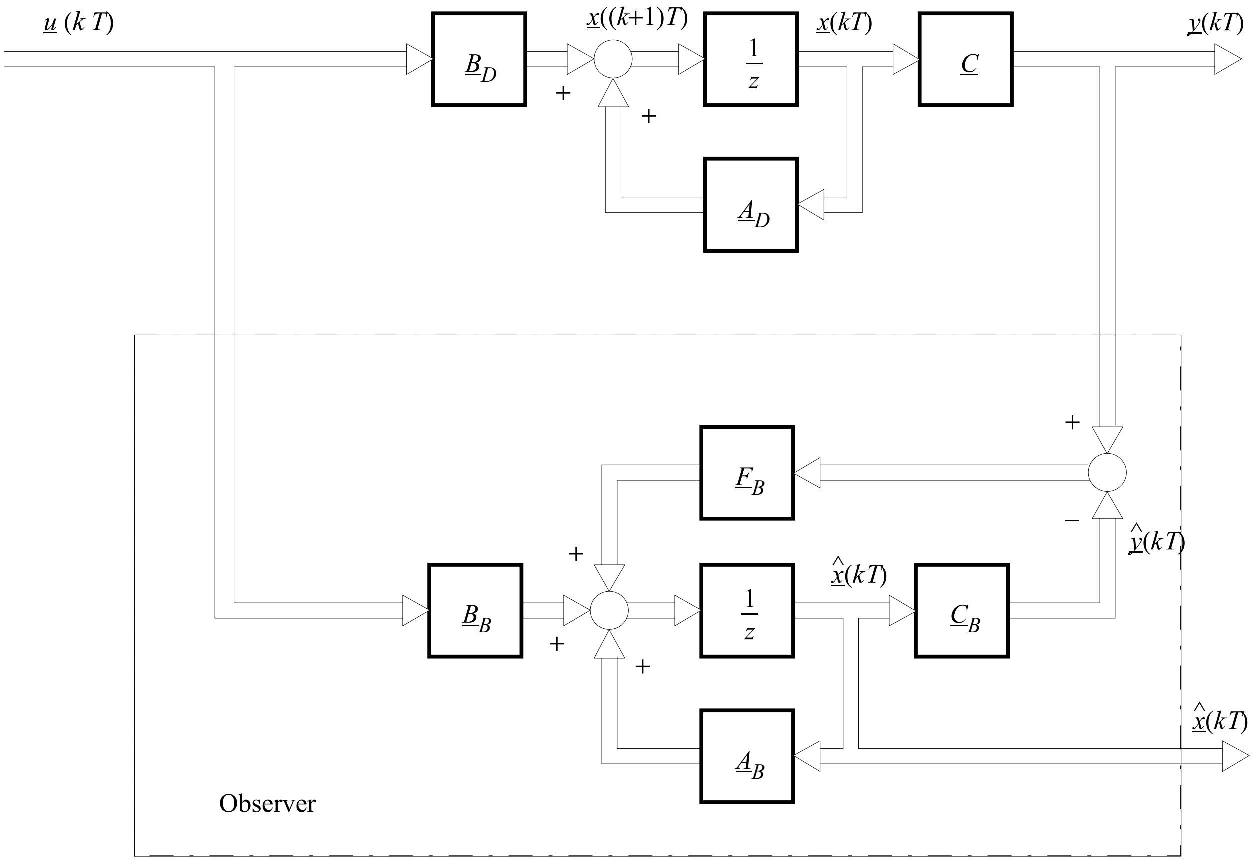


Figure 2.3 : Block diagram of the Luenberger state observer

relationship between the states and the output signals exists. For an exact definition under which conditions a system is observable we refer for example to /2/ page 175.

The Luenberger state observer assumes the measurement of the input and output signals of the system to be observed and reconstructs the state vector using these measurements. The interconnection of the overall system consisting of the system under control and the state observer is displayed in figure 2.3.

First of all, a general state space description with the matrices $\underline{A}_B$, $\underline{B}_B$, $\underline{C}_B$ and $\underline{F}_B$ is assumed for the observer. The state space equation of the observer

$$\hat{\underline{x}}\,(\,(k+1)\,T) = \underline{A}_B\,\hat{\underline{x}}\,(k\,T) + \underline{B}_B\,\underline{u}\,(k\,T) + \underline{F}_B\,(\,\underline{y}\,(k\,T) - \hat{\underline{y}}\,(k\,T)) \tag{2.41}$$

can be derived from the block diagram. The output equation of the observer is given by

$$\hat{\underline{y}}\,(k\,T) = \underline{C}_B\,\hat{\underline{x}}\,(k\,T) \quad . \tag{2.42}$$

The aim of the observer design is to determine the matrices $\underline{A}_B$, $\underline{B}_B$, $\underline{C}_B$ and $\underline{F}_B$ such that the state vector $\hat{\underline{x}}\,(k\,T)$ converges asymptotically to $\underline{x}\,(k\,T)$ for all initial conditions and an arbitrary input function $\underline{u}\,(k\,T)$. If the estimation error

$$\underline{e}\,(k\,T) = \underline{x}\,(k\,T) - \hat{\underline{x}}\,(k\,T) \tag{2.43}$$

is introduced, the requirement stated above can be written as

$$\lim_{k\to\infty} \underline{e}\,(k\,T) \equiv 0 \tag{2.44}$$

A computation of the estimation error by substracting Eq.(2.41) from Eq.(2.12) yields

$$\underline{e}\,(\,(k+1)\,T) = \underline{A}_D\,\underline{x}\,(k\,T) + \underline{B}_D\,\underline{u}\,(k\,T) - \underline{A}_B\,\hat{\underline{x}}\,(k\,T) - \underline{F}_B\,(\,\underline{y}\,(k\,T) - \hat{\underline{y}}\,(k\,T)\,) - \underline{B}_B\,\underline{u}\,(k\,T) \tag{2.45}$$

Using $\underline{y}\,(k\,T) = \underline{C}\,\underline{x}\,(k\,T)$ yields moreover:

$$\underline{e}\,(\,(k+1)\,T) = \underline{A}_D\,\underline{x}\,(k\,T) - \underline{A}_B\,\hat{\underline{x}}\,(k\,T) + (\underline{B}_D - \underline{B}_B)\,\underline{u}\,(k\,T) - \underline{F}_B\,(\underline{C}\,\underline{x}\,(k\,T) - \underline{C}_B\,\hat{\underline{x}}\,(k\,T)) \tag{2.46}$$

The selection of the observer matrices according to

$$\underline{A}_B = \underline{A}_D \tag{2.47}$$

$$\underline{B}_B = \underline{B}_D$$

$$\underline{C}_B = \underline{C}$$

results in the following state equation for the estimation error:

$$\underline{e}\,(\,(k+1)\,T) = (\underline{A}_D - \underline{F}_B\,\underline{C})\,\underline{e}\,(k\,T) \tag{2.48}$$

The estimation error converges to zero if and only if the eigenvalues of the system matrix in Eq.(2.48) are located inside the unit circle of the z-plane. The eigenvalues of the observer can again be computed by the characteristic equation

$$\det\,(\,z\,\underline{I} - (\underline{A}_D - \underline{F}_B\,\underline{C}\,)) = 0 \tag{2.49}$$

The observer matrix must be selected such that the observer possesses the preassigned eigenvalues. In the following explanations a solution algorithm for the computation of the observer matrix in case of single input-single output systems (q = p = 1) is stated. The problem is reduced to the computation of the feedback matrix in chapter 2.2. With $\underline{C} = \underline{c}^T$ (row vector) und $\underline{F}_B = \underline{f}_B$ (column vector) Eq.(2.49) becomes then

$$\det\,(\,z\,\underline{I} - (\underline{A}_D - \underline{f}_B\,\underline{c}^T\,)) = 0 \tag{2.50}$$

If the characteristic polynomial is preassigned according to

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

the n components of the observer vector $\underline{f}_B$ can be determined uniquely. If Eq.(2.49) is written in the form

$$\begin{aligned} &\det\,(\,z\,\underline{I} - (\underline{A}_D - \underline{f}_B\,\underline{c}^T\,)\,)^T \\ &= \det\,(\,(z\,\underline{I})^T - (\underline{A}_D - \underline{f}_B\,\underline{c}^T\,)^T\,) \\ &= \det\,(\,z\,\underline{I} - (\underline{A}_D^T - \underline{f}_B^T\,\underline{c}\,)\,) \quad . \qquad (2.51) \end{aligned}$$

a comparison of Eq.(2.51) with Eq.(2.27) yields the following relations

$$\underline{A}_D \;\rightarrow\; \underline{A}_D^T$$

$$\underline{b}_D \;\rightarrow\; \underline{c}$$

$$\underline{f}^T \;\rightarrow\; \underline{f}_B^T$$

With this result, the observer matrix is computed in analogy to Eq.(2.31) by

$$\begin{aligned} \underline{f}_B &= (\,\underline{p}_o\,\underline{q}_B^T + \dots + \underline{p}_{n-1}\,\underline{q}_B^T\,(\,\underline{A}_D^{n-1}\,)^T + \underline{q}_B^T\,(\,\underline{A}_D^n)^T\,)^T \\ &= (\,\underline{p}_o\,\underline{q}_B + \dots + \underline{p}_{n-1}\,\underline{A}_D^{n-1}\,\underline{q}_B + \underline{A}_D^n\,\underline{q}_B\,) \quad (2.52) \end{aligned}$$

Here the term $\underline{q}_B$ denotes the last column of the inverse observability matrix

$$\underline{Q}_B^{-1} = \begin{bmatrix} \underline{c}^T \\ \underline{c}^T\,\underline{A}_D \\ \cdot \\ \cdot \\ \cdot \\ \underline{c}^T\,\underline{A}_D^{n-1} \end{bmatrix}^{-1} \quad . \qquad (2.53)$$

For systems with more than one output signal the observability matrix cannot be determined uniquely any-more. This opens the possibility to define and fulfill, besides the preassigned observer poles, additional conditions.

3 The System "Inverted Pendulum"

The system "inverted pendulum" consists of a cart which can be moved along a metal guiding bar. An aluminium rod with a cylindrical weight is fixed to the cart by an axis. The cart is connected by a transmission belt to a drive wheel. The wheel is driven by a current controlled direct current motor which delivers a torque proportional to the acting control voltage u_S such that the cart is accelerated.

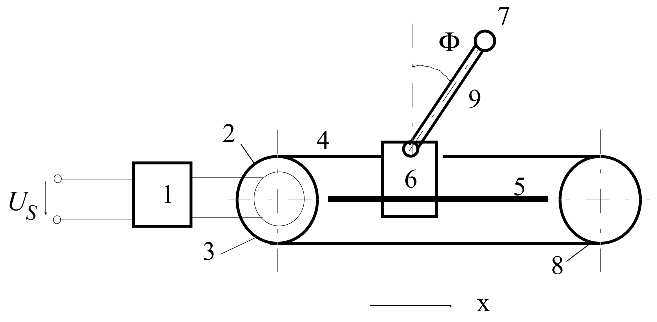


Figure 3.1 : Principle scheme of the model "inverted pendulum"

1 Servo amplifier	6 Cart
2 Motor	7 Pendulum weight
3 Drive wheel	8 Guide roll
4 Transmission belt	9 Pendulum rod
5 Metal guiding bar	

The following quantities are measured at the system "pendulum":

1. the position of the cart by means of an incremental encoder which is fixed to the driving shaft of the motor.
2. the angle of the pendulum rod by means of an incremental encoder which is fixed to the pivot of the pendulum.

3.1 Mathematical Model of the Inverted Pendulum

In the following, a mathematical model for the system "inverted pendulum" is to be derived. For the following explanations we refer to /4/. In figure 3.2 the overall system is divided into the two systems "cart" and "pendulum". The acting forces are also shown.

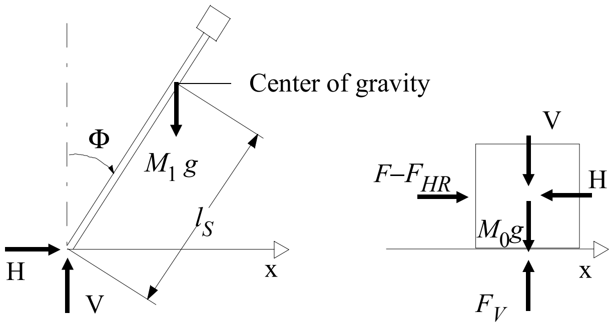


Figure 3.2 : Free body diagram of the pendulum and cart

If the mass of the pendulum is called M_1 and r denotes the position of the cart, the following force acts horizontally on the bottom point of the pendulum

$$H = M_1 \frac{\partial^2}{\partial t^2} (r + l_s \sin \Phi) \tag{3.1}$$

This force is due to the accelaration of the center of gravity. The vertical component of the force can be computed to

$$V = M_1 \frac{\partial^2}{\partial t^2} (l_s \cos \Phi) + M_1 g \quad . \tag{3.2}$$

The angular momentum conservation law yields for the rotary motion of the rod about the center of gravity:

$$\Theta_S \frac{\partial^2 \Phi}{\partial t^2} = V l_S \sin \Phi - H l_S \cos \Phi - C \frac{\partial \Phi}{\partial t} \quad . \tag{3.3}$$

where Θ_S denotes the inertia moment of the pendulum rod with respect to the center of gravity. C is the friction constant of the pendulum. For the system cart, the equation of motion can be written as

$$M_0 \frac{\partial^2 r}{\partial t^2} = F - H - F_r \frac{\partial r}{\partial t} \tag{3.4}$$

where the mass of the cart is denoted by M_0. The velocity proportional friction constant is called F_r. The force acting via the transmission belt is represented by F. A differentiation of the trigonometric functions in Eq.(3.1) and Eq.(3.2) yields

$$H = M_1 \, (\ddot{r} + l_S \ddot{\Phi} \cos \Phi - l_S (\dot{\Phi})^2 \sin \Phi) \tag{3.5}$$

and

$$V = -M_1 \, l_S (\ddot{\Phi} \sin \Phi + (\dot{\Phi})^2 \cos \Phi) + M_1 \, g \tag{3.6}$$

By substitution of Eq.(3.5) and Eq.(3.6) into Eq.(3.3) and Eq.(3.4) the quantities V and H are eliminated. After some straight forward manipulations, one obtains the nonlinear differential equations

$$\Theta \, \ddot{\Phi} + C \, \dot{\Phi} - M_1 \, l_S \, g \sin \Phi + M_1 \, l_S \, \ddot{r} \cos \Phi = 0 \tag{3.7}$$

and

$$M \, \ddot{r} + F_r \, \dot{r} + M_1 \, l_S (\ddot{\Phi} \cos \Phi - (\dot{\Phi})^2 \sin \Phi) = F \quad . \tag{3.8}$$

The Eq.(3.7) and Eq.(3.8) describe the mathematical model of the pendulum in form of a system of coupled differential equations. The following abbreviations have been used

$$\Theta = \Theta_S + M_1 \, l_S^2 \tag{3.9}$$

$$M = M_0 + M_1 \tag{3.10}$$

These equations are the basis for the derivation of a mathematical control model of the inverted pendulum.

3.2 Description of the Linearized System in the State Space

To the end of describing the system "pendulum" according to Eq.(2.1) and Eq.(2.2) a linarization about a suitable operating point must be performed. The first step is the introduction of a state vector

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} r \\ \dot{r} \\ \Phi \\ \dot{\Phi} \end{bmatrix} \tag{3.11}$$

and an input signal

$$u = F \tag{3.12}$$

Transforming Eq.(3.5) and Eq.(3.6) into the form

$$\dot{\underline{x}} = \underline{f} \, (\underline{x}, \underline{u}) \quad . \tag{3.13}$$

yields:

$$\dot{x}_1 = f_1 \, (\underline{x}, F) = x_2 \tag{3.14}$$

$$\dot{x}_3 = f_3 \, (\underline{x}, F) = x_4 \tag{3.15}$$

$$\begin{aligned} \dot{x}_2 = f_2 \, (\underline{x}, F) = \beta \, (x_3) \, (& a_{23} \sin x_3 \cos x_3 + a_{22} \, x_2 \\ & + a_{24} \cos x_3 \; x_4 + a_{25} \sin x_3 \; (x_4)^2 + b_2 \, F \,) \end{aligned} \tag{3.16}$$

$$\begin{aligned} \dot{x}_4 = f_4 \, (\underline{x}, F) = \beta \, (x_3) \, (& a_{43} \sin x_3 + a_{42} \cos x_3 \; x_2 \\ & + a_{44} \, x_4 + a_{45} \cos x_3 \; \sin x_3 \; (x_4)^2 \\ & + b_4 \cos x_3 \; F) \end{aligned} \tag{3.17}$$

where the abbreviations

$$\beta(x_3) = \left(1 + \frac{N^2}{N_{01}^2}\sin^2 x_3\right)^{-1} \tag{3.18}$$

$$N = M_1\, l_S \tag{3.19}$$

$$N_{01}^2 = \Theta\, M - N^2 \tag{3.20}$$

have been used. The coefficients are given by:

$$a_{23} = -\frac{N^2 g}{N_{01}^2} \quad a_{22} = -\frac{\Theta F_r}{N_{01}^2} \quad a_{24} = \frac{N C}{N_{01}^2} \quad a_{25} = \frac{\Theta N}{N_{01}^2}$$
$$a_{43} = \frac{M N g}{N_{01}^2} \quad a_{42} = \frac{N F_r}{N_{01}^2} \quad a_{44} = -\frac{M C}{N_{01}^2} \quad a_{45} = -\frac{N^2}{N_{01}^2}$$
$$b_2 = \frac{\Theta}{N_{01}^2} \quad b_4 = -\frac{N}{N_{01}^2}$$

With the second step, the operating point, about which the linearization is to be performed, is defined. It is given by the condition

$$\underline{x}_A = \begin{bmatrix} x_1 = k_1 \\ x_2 = 0 \\ x_3 = 0 \\ x_4 = 0 \end{bmatrix} \quad . \tag{3.21}$$

The equations (3.14) to (3.17) are developed into a Taylor series which is terminated after the first element.

$$\Delta\dot{\underline{x}} \approx \left.\frac{\partial \underline{f}}{\partial \underline{x}}\right|_{\substack{\underline{x}=\underline{x}_A \\ F=0}} \cdot \Delta\underline{x} + \left.\frac{\partial \underline{f}}{\partial F}\right|_{\substack{\underline{x}=\underline{x}_A \\ F=0}} \cdot \Delta F \tag{3.22}$$

The computation of the differential quotient yields for the linearized system matrix

$$\left.\frac{\partial \underline{f}}{\partial \underline{x}}\right|_{\substack{\underline{x}=\underline{x}_A \\ F=0}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & 0 & 1 \\ 0 & a_{42} & a_{43} & a_{44} \end{bmatrix} := \underline{A}_A \tag{3.23}$$

and for the input matrix

$$\left.\frac{\partial \underline{f}}{\partial F}\right|_{\substack{\underline{x}=\underline{x}_A \\ F=0}} = \begin{bmatrix} 0 \\ b_2 \\ 0 \\ b_4 \end{bmatrix} := \underline{b}_A \tag{3.24}$$

The deviations from the operating point are introduced as the new state variables and input signals ($\Delta\,\underline{x} = \underline{x}$, $\Delta\,F = F$). The state equation can then be written as

$$\dot{\underline{x}} = \underline{A}_A\, \underline{x} + \underline{b}_A\, F \quad . \tag{3.25}$$

This equation constitutes the linear state space description of the "inverted pendulum". Thus, the prerequisites for the application of the controller and observer design techniques which have been introduced in chapter "Theoretical Foundations" are satisfied.

3.3 Parameters of the Mathematical "Inverted Pendulum" Model

The parameters of the "inverted pendulum" have been determined by means of extensive physical experiments. The numerical values are stated in table 3.1.

The following parameters are generated by measurements at one realized pendulum. These parameters are means. Thus the technical data of your system could vary from the below mentioned parameters, in particular the coefficient of the friction.

Constant	Numerical value	Unit
M_0	4,0	Kg
M_1	0,36	Kg
M	4,36	Kg
l_S	0,451	m
Θ	0,08433	kgm^2
N	0,1624	Kgm
N_{01}^2	0,3413	Kg^{2}m^2
N^2/N_{01}^2	0,07723	-
F_r	10,0	Kg/s
C	0,00145	Kgm^2/s

Table 3.1 : Physical quantities

Using these data, the numerical values of the system matrix (compare Eq.(3.23))

$$\underline{A}_A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -2{,}47\ {}^{1}/_{s} & -0{,}757\ {}^{m}/_{s^2} & 6{,}8\cdot 10^{-4}\ {}^{m}/_{s} \\ 0 & 0 & 0 & 1 \\ 0 & 4{,}7569\ {}^{1}/_{(ms)} & 20{,}346\ {}^{1}/_{s^2} & -0.0185\ {}^{1}/_{s} \end{bmatrix} \tag{3.26}$$

and the input matrix (compare Eq.(3.24))

$$\underline{b}_A = \begin{bmatrix} 0 \\ 0.247\ {}^{m}/_{(s^2 N)} \\ 0 \\ -0.475\ {}^{1}/_{(s^2 N)} \end{bmatrix} \tag{3.27}$$

can be computed directly.

The matrices of the sampled data system can be computed using the matrices $\underline{A}_n$ and $\underline{b}_n$ by means of Eq. (2.13) and Eq.(2.14), where a sampling period of T=0.03 sec has been selected. One obtains:

$$\underline{A}_D = \begin{bmatrix} 1 & 2.89\cdot 10^{-2} & -3.33\cdot 10^{-4} & -3.04\cdot 10^{-6} \\ 0 & 0.928 & -2.19\cdot 10^{-2} & -3.13\cdot 10^{-4} \\ 0 & 2.09\cdot 10^{-3} & 1.009 & 3.01\cdot 10^{-5} \\ 0 & 0.138 & 0.610 & 1.008 \end{bmatrix} \tag{3.28}$$

$$\underline{b}_D = \begin{bmatrix} 1.08\cdot 10^{-4} \\ 7.14\cdot 10^{-3} \\ -2.09\cdot 10^{-4} \\ -1.38\cdot 10^{-2} \end{bmatrix} \tag{3.29}$$

The state variables x_1 and x_3 are defined as the output signals. This yields for the output matrix

$$\underline{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.30}$$

With these equations, a linear state space description for the pendulum has been found. It was not possible to consider nonlinear effects directly in this framework. They are to be reduced or compensated by means of external measures. The most significant nonlinearities acting on the cart are the dry friction (Coulomb friction)

$$F_C = -|F_C|\ sign\ (\dot{r}\ (t)) \tag{3.31}$$

and the static friction

$$F_{HR} = \begin{cases} -\mu\ F_n & f\ddot{u}r\ \dot{r} = 0 \\ 0 & f\ddot{u}r\ \dot{r} \neq 0 \end{cases} \tag{3.32}$$

with the static friction coefficient μ and the normal force F_n. In this case, a compensation can be achieved by superimposing an additional input voltage

$$F_{S0} = F_{HR} + F_c \tag{3.33}$$

(see figure 3.3).

The quantity F_{S0} can either be determined directly at the time the system is built, or it can be estimated "ON-LINE" by means of a disturbance signal observer. Both alternatives are implemented on the "inverted pendulum" as well as on the system "cart". In figure 3.3, the state space description from Eq.(3.25) is shown in form of a block diagram.

This mathematical model is valid as long as the following conditions are satisfied:

1. a limitation of the control force F_S. The linearity of the servo amplifier is guaranteed only for a control force satisfying
   $|F_S| \leq 20\ N$

2. a limitation of the guiding bar. It is:
   $|r| \leq 0.5\ m$

3. a limitation of the angle Φ, because the approximation for small angles
   $|\Phi| \leq 10°$
   is valid.

3.4 Control and Disturbance Signal Observation on the "Inverted Pendulum"

Because the system "PS600 Inverted Pendulum" is to be controlled by a digital controller (PC) all the following calculations have to carried out in the z plane.

In the chapter "Mathematical Model of the Inverted Pendulum" it has already been pointed out that only two

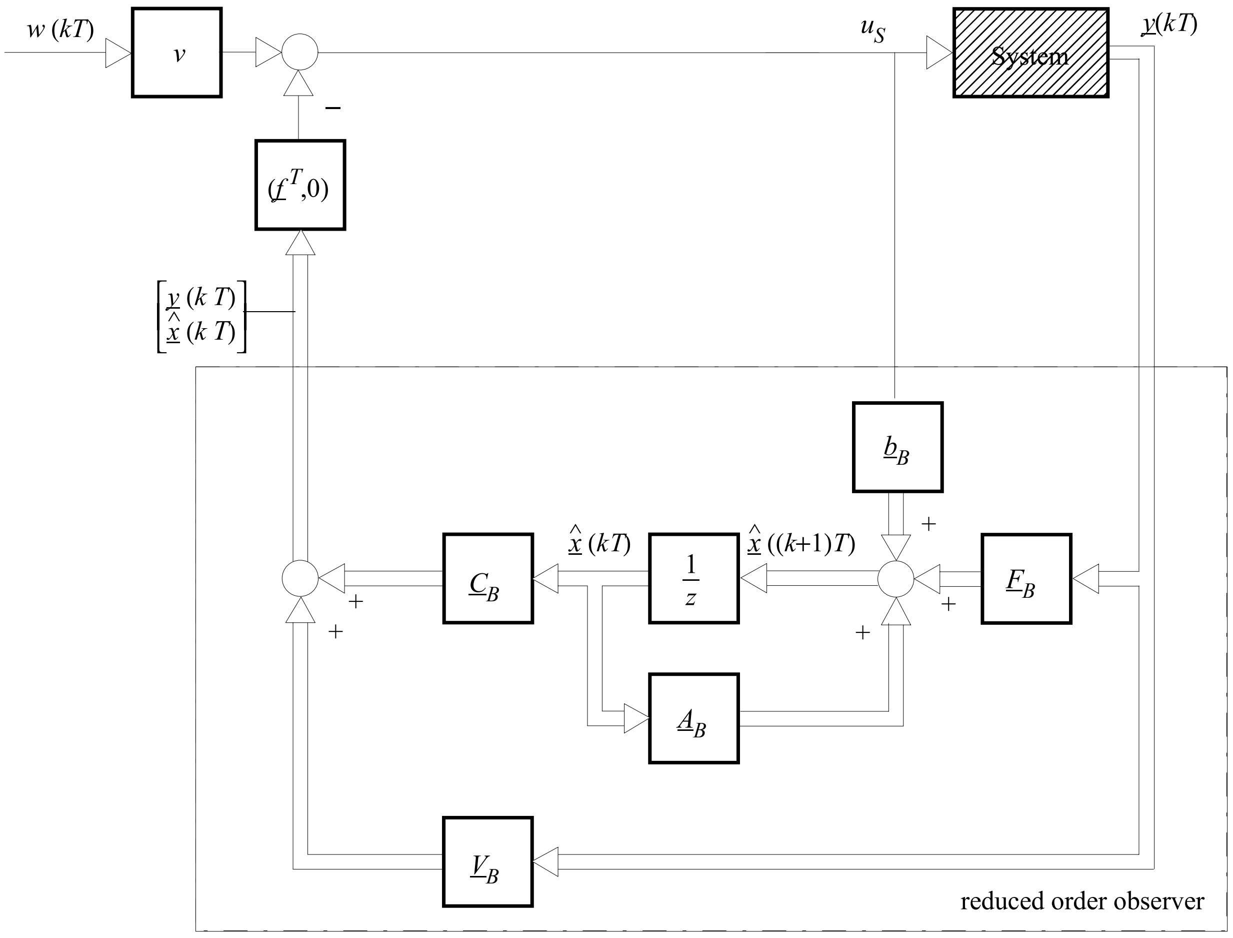


Figure 3.3 : Block diagram of the controlled pendulum with state feedback and reduced order observer

of altogether four state variables are measured at the system "PS600". In the following, the design of a reduced order observer for the estimation of the missing state variables as well as of the disturbance signal F_{S0} is introduced. The reduced order observer is distinguished from the identity observer in the sense that it does not estimate output signals which are already being measured. Thus the reduced order observer has only the order n-q.

According to figure 3.3, the equation for the reduced order observer is given by

$$\underline{\hat{x}}\,(\,(k+1)\,T) = \underline{A}_B\,\underline{\hat{x}}\,(k\,T) + \underline{F}_B\,\underline{y}\,(k\,T) + \underline{b}_B\,u_S\,(k\,T) \tag{3.34}$$

$$\begin{bmatrix} \underline{y}\,(k\,T) \\ \underline{\hat{x}}\,(k\,T) \end{bmatrix} = \underline{C}_B\,\underline{\hat{x}}\,(k\,T) + \underline{V}_B\,\underline{y}\,(k\,T) \tag{3.35}$$

For purposes of computing the unknown matrices, the system in subdivided (compare Eq. 3.36) with respect to the known and the unknown state variables, that means it is seperated into the two subsystems stated in Eq. (3.37) and Eq.(3.38). The 4x4 system matrix can be divided into the 4 submatrices $\underline{A}_{11}, \underline{A}_{12}, \underline{A}_{21}, \underline{A}_{22}$. The vectors $\underline{b}_1$ and $\underline{b}_2$ are build by the input vector.

$$\begin{bmatrix} \underline{y}\,((k+1)\,T) \\ \underline{\hat{x}}_B\,((k+1)\,T) \end{bmatrix} = \begin{bmatrix} \underline{A}_{11} & \underline{A}_{12} \\ \underline{A}_{21} & \underline{A}_{22} \end{bmatrix} \begin{bmatrix} \underline{y}\,(k\,T) \\ \underline{\hat{x}}_B\,(k\,T) \end{bmatrix} + \begin{bmatrix} \underline{b}_1 \\ \underline{b}_2 \end{bmatrix} u_S\,(k\,T) \tag{3.36}$$

where

$$\underline{\hat{x}}_B\,(k\,T) = \begin{bmatrix} \hat{x}_2\,(k\,T) \\ \hat{x}_4\,(k\,T) \end{bmatrix}$$

The measurable state variables (x_1, x_3) of this system are identical to the output signals and are described by the vector $\underline{y}$.

$$\underline{y}\,((k+1)\,T) = \underline{A}_{11}\,\underline{y}\,(k\,T) + \underline{A}_{12}\,\underline{\hat{x}}_B\,(k\,T) + \underline{b}_1\,u_S\,(k\,T) \tag{3.37}$$

$$\underline{\hat{x}}_B\,((k+1)\,T) = \underline{A}_{21}\,\underline{y}\,(k\,T) + \underline{A}_{22}\,\underline{\hat{x}}_B\,(k\,T) + \underline{b}_2\,u_S\,(k\,T) \tag{3.38}$$

Since the system Eq.(3.37) is known, the expression $\underline{A}_{12}\,\underline{\hat{x}}_B\,(k\,T)$ enters the system Eq.(3.38) as a measured quantity. If an reduced order observer is designed for this system, it is described by Eq.(3.39).

$$\begin{aligned} \underline{\hat{x}}_B\,((k+1)\,T) &= (\underline{A}_{22} - \underline{L}\,\underline{A}_{12})\,\underline{\hat{x}}_B\,(k\,T) \\ &\quad + \underline{L}\,(\underline{y}\,((k+1)\,T) - \underline{A}_{11}\,\underline{y}\,(k\,T) - \underline{b}_1\,u_S\,(k\,T)) \\ &\quad + \underline{A}_{21}\,\underline{y}\,(k\,T) + \underline{b}_2\,u_S\,(k\,T) \end{aligned} \tag{3.39}$$

Via the change of variables

$$\underline{z}\,(k\,T) = \underline{\hat{x}}_B\,(k\,T) - \underline{L}\,\underline{y}\,(k\,T) \tag{3.40}$$

the observer equation

$$\underline{z}\,((k+1)\,T) = \underline{A}_B\,\underline{z}\,(k\,T) + \underline{F}_B\,\underline{y}\,(k\,T) + \underline{b}_B\,u_S\,(k\,T) \tag{3.41}$$

Using

$$\underline{L} = \begin{bmatrix} 30{,}392 & -7{,}33 \cdot 10^{-3} \\ 2{,}465 & 31{,}872 \end{bmatrix}$$

$$\underline{A}_B = \underline{A}_{22} - \underline{L}\,\underline{A}_{12} = \begin{bmatrix} 0{,}0498 & 0 \\ 0 & 0{,}0498 \end{bmatrix}$$

$$\underline{F}_B = \underline{A}_B\,\underline{L} + \underline{A}_{21} - \underline{L}\,\underline{A}_{11} = \begin{bmatrix} -28{,}878 & -4{,}81 \cdot 10^{-3} \\ -2{,}342 & -29{,}96 \end{bmatrix}$$

$$\underline{b}_B = \underline{b}_2 - \underline{L}\,\underline{b}_1 = \begin{bmatrix} 3{,}84 \cdot 10^{-3} \\ -7{,}39 \cdot 10^{-3} \end{bmatrix}$$

is obtained. The matrix $\underline{L}$ has been selected, such that the poles of the observer are located at $z_{1,2} = -0{,}0498$. The remaining matrices eventually result to

$$\underline{V}_B = \begin{bmatrix} 1 & 0 \\ l_{11} & l_{12} \\ 0 & 1 \\ l_{21} & l_{22} \end{bmatrix}, \quad \underline{C}_B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

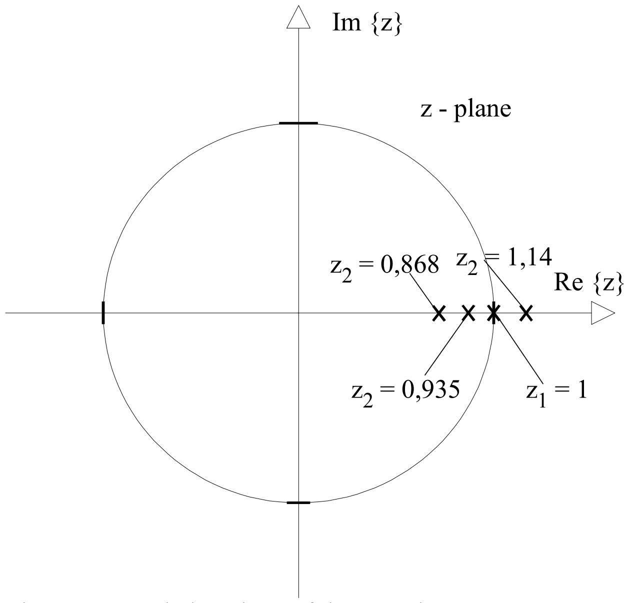


Figure 3.4 : Pole locations of the open loop system

Mathematical techniques for the design of reduced order observers are stated in /3/ chapter 10.2.2 and /1/ chapter 12.5.3. Before the feedback matrix is finally computed, the poles of the open loop system will be calculated. They can be determined by

$$\det(s\,\underline{I} - \underline{A}_D) = 0 \tag{3.42}$$

This results in the configuration which is displayed in figure 3.4:

It can be seen that one has an unstable system which can be stabilized by means of the state feedback shown in Figure 3.3. Via assignment of the eigenvalues of the closed loop system to

$z_{1,2} = 0{,}88692$
$z_{3,4} = 0{,}86719$

the feedback matrix is computed by Eq.(2.31) as

$$\underline{f}^T = (-61{,}83;\ -68{,}0;\ -278{,}3;\ -63{,}25)$$

The scalar prefilter becomes (compare Eq.(2.40))

$$v = \underline{f}[0] = -61{,}83$$

This completely determines the algorithm for the control of the system. The state vector $\underline{x}$ is assumed to be known in the following. This fact is represented by Figure 3.3.

To estimate the additional disturbance signal F_{S0}, a model augmentation is required which describes the properties of the disturbance. In the mathematical model (figure 3.5) the quantity F_{S0} has been assumed to be constant. Thus it can be interpreted as a solution of the differential equation

$$\dot{x}_5 = 0 \tag{3.43}$$

This signal acts via the control vector $\underline{b}_A$ on the variable $\dot{x}$, such that the augmented system can be written in the form

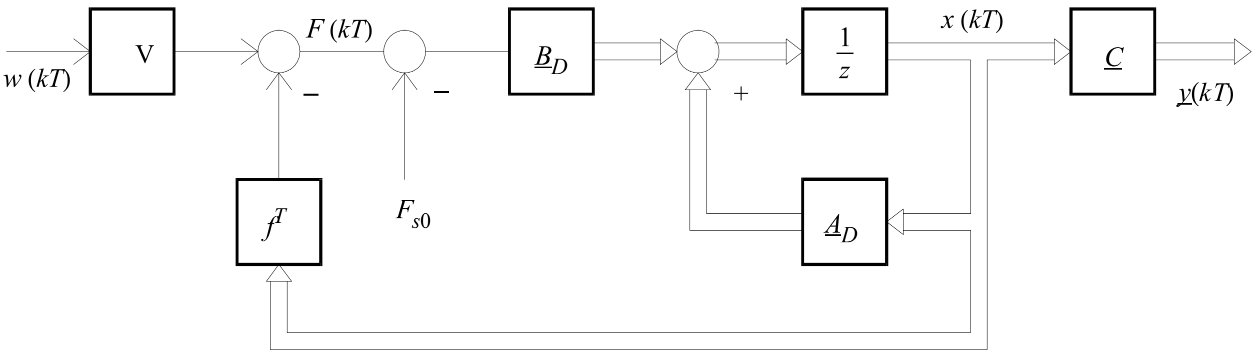


Figure 3.5: Closed control loop

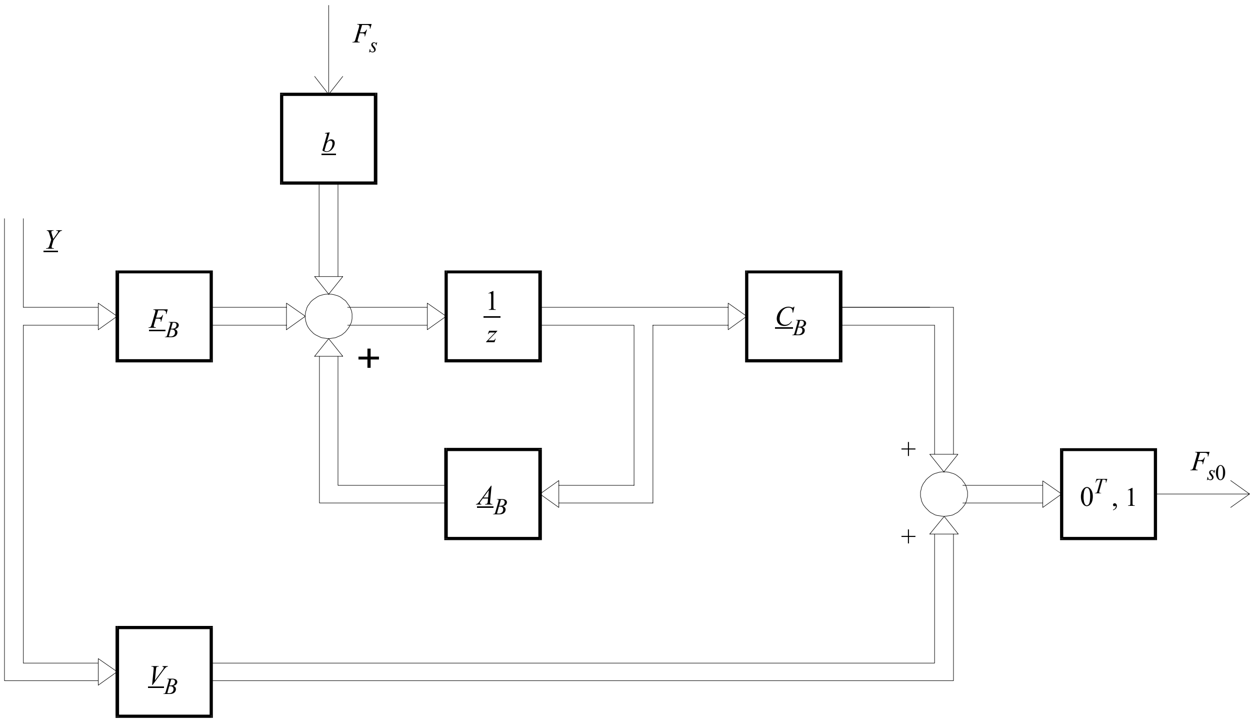


Figure 3.6: Observer for the disturbance signal Us0

$$\begin{bmatrix} \dot{\underline{x}} \\ \dot{x}_5 \end{bmatrix} = \begin{bmatrix} \underline{A}_A & \underline{b}_A \\ \underline{0}^T & 0 \end{bmatrix} \begin{bmatrix} \underline{x} \\ x_5 \end{bmatrix} + \begin{bmatrix} \underline{b}_A \\ 0 \end{bmatrix} F \qquad (3.44)$$

An observer, employing all the measured signals as well as the estimated state variables is now designed for the augmented system (see figure 3.6). The design of this disturbance observer is carried out similar to the design of the reduced order observer as described above. The vector y now contains the state variables (x_1,x_2,x_3,x_4) whereas the vector x_B is given by the state variable x_5, which is to be estimated.

The resulting matrices are given by:

$\underline{L}_B = [\,0;\ 0;\ 0;\ -68{,}889\,]$

$\underline{A}_B = [\,0{,}0497\,]$

$\underline{F}_B = [\,0;\ 9{,}5;\ 42{,}05;\ 66{,}05\,]$

$\underline{b}_B = [\,-0{,}95\,]$

$$\underline{V}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ l_{B1} & l_{B2} & l_{B3} & l_{B4} \end{bmatrix}$$

$\underline{C}_B = [\,1\,]$

3.5 Design of the Luenberger Identity Observer

The system matrices, as given in chapter 2.3, can be used directly for the Luenberger identity observer design (compare Eq.(2.47)). To this end, an output signal y from which the system is completely observable, is to be provided for the observer.

For this application example the position coordinate $y_1 = r$ has been selected as a suitable output signal. This yields for the output matrix

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

The yet unknown feedback matrix can be computed by Eq.(2.52) via assignment of the observer eigenvalues. The results are listet in table 3.2 (T = 0.03 sec).

s_i $[1/s]$	$z_i = e^{T s}$	$\underline{f}_{B\,i}^{T}$
- 3	0.9139	0.29, 1.51, -13.71, -61.11
- 6	0.8353	0.6, 4.64, -44.72, -199.28
- 10	0.7408	0.98, 11.24, -128.24, -591.15
- 12	0.6977	1.15, 15.21, -189.45, -902.92

Table 3.2 : Numerical values of the observer matrix $\underline{f}_B$ for the pendulum

4 Preparations for the Laboratory Experiment

4.1 State the differential equation for a cart of mass M_0 which is accelerated by a force F(t). Consider the velocity dependent friction ! Friction coefficient F_R .

4.2 Draw the block diagram of the system with *F(t)* as the input and *x(t)* as the output signal. State the transfer function *G(s) = X(s)/F(s)* of the system and sketch the step response of the first order system ($v = \dot{x} = f(t)$). How can the time constant τ be calculated ?

4.3 On the experimental set-up which is used here, only the position is measured by a sensor. The velocity of the cart is calculated from this measurement mathematically. Proceed from the blockdiagram shown in Figure 4.1 and state the equivalent state space description (system matrix $\underline{A}$ and input vector $\underline{b}$). The input signal of the system is the control force F_S .

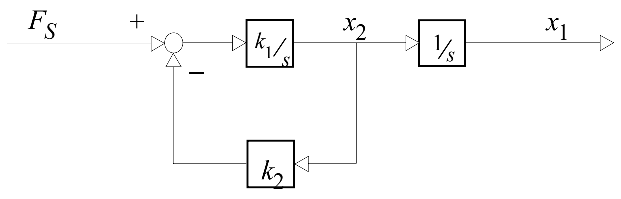


Figure 4.1 : Block diagram of the system "cart"

4.4 For the design of the Luenberger observer, a mathematical model of the system under control is needed. Use the state space description which has been derived in point 4.3 and determine the system matrix $\underline{A}_D$ as well as the input vector $\underline{b}_D$ of the sampled data system ! (sampling period T)

4.5 The cart is to be controlled via state feedback. Determine first of all the eigenvalues of the open loop system and determine subsequently the feedback vector $\underline{f}$, such that the poles of the closed loop sampled data control system are located at z_1, z_2.

4.6 A Luenberger state observer for the system "cart" is to be designed. Compute the observer vector $\underline{f}_B$, such that the poles of the observer are located at z_1, z_2 ! Hint: The result of problem 4.5 can be used for the computation !
$\det((\underline{A}\,\underline{B})^T) = \det(\underline{A}^T\underline{B}^T)$

5 Set-Up of the Experiment

The experimental set-up consists of the following devices:

1. Laboratory model "inverted pendulum" and "cart", respectively

2. Actuator

3. PC with PC plug-in card, monitor, keyboard and mouse

4. Plotter or printer

ref. 1

The model of the "inverted pendulum" has already been introduced in chapter 3.1. By removing the pendulum rod the system "cart" is created. The laboratory model is connected to the servo amplifier and the PC via a lead.

ref. 2

Inside the actuator are the output stage which drives the direct current motor. The control signal is measureable at a jack on the frontpanel.

ref. 3

The algorithms for the control and the observation of the respective system are implemented on the computer (PC). Additionally, the subroutines needed to plot the measured results and to determine the step responses of the system "cart" can be called. All parameters can be entered or varied interactively.

ref. 4

Besides the screen itself a plotter (HP-compatible) or a matrix printer (Epson-compatible) serve as output units. The plotter is connected to the serial interface (RS 232) of the computer, the matrix printer to the parallel interface.

6 Carrying-Out the Experiment

During this experiment a state observation algorithm using a Luenberger identity observer is realized for the system "cart" and the system "inverted pendulum". The parameters of the system "cart" can be determined via the recorded step response. Subsequently, the control algorithm as well as the state observation algorithm for various pole assignments are computed and tested practically on the systems "cart" and "inverted pendulum".

For the realization of the single assignments, the various menu items must be called according to the manual and the required inputs must be entered into the computer. First of all, start the system as it has been described in the manual.

6.1 Record the step response of the system "cart" using the two different control voltages $F = 9\,N$, $F = 10\,N$ or two variable values (ask your tutor !). (Menu item "Identify Cart")

6.2 Using the step responses, verify that the system "cart" is a nonlinear system !
What are the major reasons for the nonlinearities?

6.3 The design methods which have been introduced in chapter 2 are applicable only for linear systems. For this reason, an additional compensation voltage will be superimposed on the input, in order to compensate for the nonlinear portion. First of all, the model of the cart must be extended, such that the measured step response coincides with our model. To this end, an additional disturbance signal u_{S0} is superimposed on the input of the model displayed in Figure 4.1. The extended model is shown in Figure 6.1. Proceed from this model and determine the integration constant k_1, the friction constant k_2 as well as the quantity F_0, using the step responses. (Note the sign of the state variable x_2 !)
It is $K_3 = {}^{n_{11}}\!/\!_{n_{33}} = -1.95$.

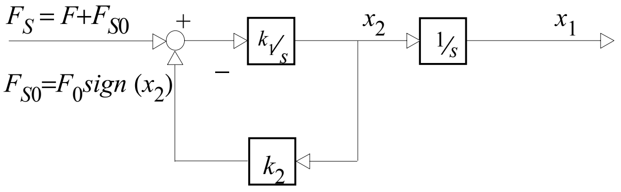


Figure 6.1 : Extended model of the cart

6.4 Using the formulas from the preparation problems 4.3 and 4.4, compute the input and the system matrix of the sampled data system! (sampling period T=0.03 s). Select the menu item "Parameters" and enter the numerical values into the computer.

6.5 Using the result of the preparation problem 4.5, compute the feedback matrix, such that the eigenvalues of the closed loop system are located at $z_{1,2} = 0{,}914$! Enter these numerical values also into the computer !
Test your design by starting the control algorithm and controlling the cart using different setpoints! Use the constant disturbance compensation !

6.6 Next, compute the observer matrix for observer poles located at a.) $z_{1,2} = 0{,}835$ and b.) $z_{1,2} = 0{,}5488$ according to the corresponding preparation problem 4.6.

6.7 For a setpoint step of $\Delta\, w = 30\, cm$ (Menu item 8) record for both observer poles from problem 6.6 the following quantities:

1. measured and estimated position

2. measured and estimated velocity

3. transient of the disturbance signal F_{S0}

4. transient of the control voltage (comp. as well as uncomp.)

6.8 In order to improve the control performance, the observer principle has been employed for the estimation of the compensating voltage at the system "cart". Using the disturbance compensation via observer, record the curves as described for point 6.7.

6.9 Another system for the test of the identity observer is the system "inverted pendulum". Model descriptions for the controller and the observer design have been introduced in chapter 3. The parameters which have been stated there are already stored in the computer. Fasten the pendulum rod together with the pendulum weight to the cart. Move the cart manually to the center of the guiding bar and keep the pendulum vertically in the upright position. Now, you start the control algorithm for the "inverted pendulum" using the observer based disturbance compensation. Select the observer poles from a.) $z_{1,2,3,4} = 0{,}835$ (to do this load the file PZ0835.STA) and b.) $z_{1,2,3,4} = 0{,}5488$ (to do this load the file PZ05488.STA) and plot the following quantities for a setpoint step of $\Delta w = 30\ cm$:

1. measured and estimated position

2. measured and estimated velocity

3. measured and estimated angle of the pendulum rod

4. transient of the disturbance signal F_{S0}

5. transient of the control voltage (comp. as well as uncompensated)

7 Experiment Elaboration

7.1 Compare and evaluate the state observations for the system "cart" using the plots which have been recorded in 6.7.
What is the effect of the relatively large steady state deviations of speed and position from the corresponding measured state variables in case the constant disturbance compensation has been used?

7.2 How are the observer poles to be selected in order to guarantee a good state estimation ?
How is the state estimation affected by the observer poles which are located in the z plane nearer to the origin (respectively further to the left in the s plane) than the poles of the control system?

7.3 What measures could decrease the steady state deviation of the control algorithm for the system "cart" in case the constant disturbance compensation is used?

7.4 Mutually compare the measurement results and the state estimation for the "inverted pendulum" for various observer poles! What is the reason for the poor quality of the estimate of the pendulum angle?

7.5 State measures, allowing for a better estimation of the angle! Which physical quantities can be changed on the system "inverted pendulum" in order to obtain better results?

8 References

/1/ : O. Föllinger :
Regelungstechnik, 2. Edition, published by: Elitera - Verlag, Berlin 1978

/2/ : H. Unbehauen :
Regelungstechnik II, 4. Edition, published by: Vieweg - Verlag, Braunschweig 1987

/3/ : J. Ackermann :
Abtastregelung, published by: Springer - Verlag, Berlin 1972

/4/ : Shozo Mori, Hiroyoshi Nishihara and Kattsuhisa Furuta, Control of unstable mechanical system, Control of pendulum, Int. J. Control, 1976, Vol. 23, No. 5, pp. 673 - 692

9 Solutions

9.1 Solutions of the Preparation Problems from Chapter 4

ref. 4.1 :

In order to derive the differential equation, the equation of motion governing the cart is formulated according to figure 9.1.

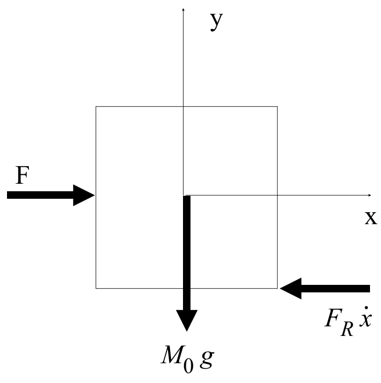


Figure 9.1 : Free-body diagram of the cart

In the free-body diagram *F* denotes the force acting on the cart. F_R is the friction constant of the cart. According to figure 9.1 it is:

$$M_0 \ddot{x} = \sum_{i=0}^{n} F_i = F - F_R \dot{x} \tag{9.1}$$

or solved for *F*:

$$\ddot{x} + \frac{F_R}{M_0} \dot{x} = \frac{F}{M_0} \tag{9.2}$$

ref. 4.2 :

Eq. (9.2) is a second order differential equation for the position of the cart. The time constant τ is given by:

$$\tau = \frac{M_0}{F_R} \tag{9.3}$$

Applying the Laplace transform to the differential equation allows to compute the transfer function *G(s)* of the system:

$$s^2 X(s) + \frac{F_R}{M_0} s X(s) = \frac{F(s)}{M_0} \tag{9.4}$$

The transfer function can be stated as

$$G(s) = \frac{X(s)}{F(s)} = \frac{\frac{1}{M_0}}{s\,(s + \frac{F_R}{M_0}\,)} \tag{9.5}$$

which yields a block diagram as displayed in figure 9.2. It is a series connection of a first order lag with an integrator.

Since integrators, amplifiers and summing junctions are linear, the overall system in figure 9.2 is linear, too.

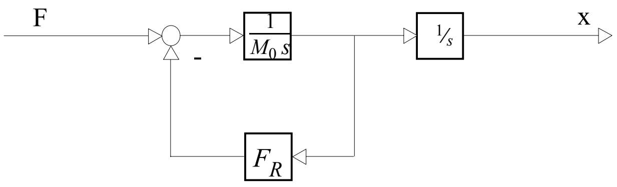


Figure 9.2 : Block diagram of the system "cart"

ref. 4.3 :

It has been shown already that the ideal system cart results from the series connection of a first order lag with an integrator. For the normalized system one therefore starts off with the block diagram shown in figure 9.3, where the constants K_1, K_2 and the voltage F_0 for the compensation of the static friction are yet to be determined.

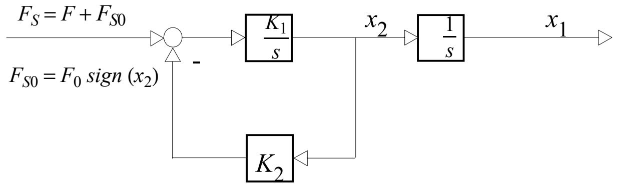


Figure 9.3 : Block diagram of the standardized system

The transfer function $G_2\,(s)$ of the normalized system in figure 9.3 is given by:

$$G_2\,(s) = \frac{X_2\,(s)}{F_s\,(s)} = \frac{K_1}{(s + K_1\,K_2)} \tag{9.6}$$

and for the state space equation one obtains

$$\dot{\underline{x}} = \begin{bmatrix} 0 & K_3 \\ 0 & -K_1\,K_2 \end{bmatrix} \underline{x} + \begin{bmatrix} 0 \\ K_1 \end{bmatrix} u \tag{9.7}$$

where $F_S = F + F_{S0}$.

ref. 4.4 :

In order to compute the sampled data system, the fundamental matrix $\underline{A}_D\,(T)$ is needed. According to Eq.(2.13) it is computed as

$$\underline{A}_D\,(T) = e^{\underline{A}\,T} = \underline{I} + \underline{A}\,T + \underline{A}^2\,\frac{T^2}{2\,!} + \underline{A}^3\,\frac{T^3}{3\,!} + \ldots$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & K_3 \\ 0 & -K_1\,K_2 \end{bmatrix} T + \begin{bmatrix} 0 & -K_1\,K_2\,K_3 \\ 0 & (-K_1\,K_2)^2 \end{bmatrix} \frac{T^2}{2\,!}$$

$$+ \begin{bmatrix} 0 & (-K_1\,K_2)^2\,K_3 \\ 0 & (-K_1\,K_2)^3 \end{bmatrix} \frac{T^3}{3\,!} + \ldots$$

$$= \begin{bmatrix} 1 & -\frac{K_3}{K_1\,K_2}\left(e^{-K_1\,K_2\,T} - 1\right) \\ 0 & e^{-K_1\,K_2\,T} \end{bmatrix} \tag{9.8}$$

According to Eq.(2.14), the matrix $\underline{b}_D$ is computed as:

$$\underline{b}_D\,(T) = \int_0^T \underline{A}_D\,(\tau)\,\underline{b}\,d\tau$$

$$= \int_0^T \begin{bmatrix} 1 & -\frac{K_3}{K_1\,K_2}\left(e^{-K_1\,K_2\,\tau} - 1\right) \\ 0 & e^{-K_1\,K_2\,\tau} \end{bmatrix} \begin{bmatrix} 0 \\ K_1 \end{bmatrix} \partial\tau$$

$$= \int_0^T \begin{bmatrix} -\frac{K_3}{K_2}\left(e^{-K_1\,K_2\,\tau} - 1\right) \\ K_1\,e^{-K_1\,K_2\,\tau} \end{bmatrix} \partial\tau$$

$$= \begin{bmatrix} -\frac{K_3}{K_2}\left(-\frac{1}{K_1\,K_2}\,e^{-K_1\,K_2\,\tau} - \tau\right)\Big|_0^T \\ -\frac{1}{K_2}\,e^{-K_1\,K_2\,\tau}\Big|_0^T \end{bmatrix}$$

$$= \begin{bmatrix} \frac{K_3}{K_2}\left(\frac{1}{K_1\,K_2}\,e^{-K_1\,K_2\,T} + T - \frac{1}{K_1\,K_2}\right) \\ -\frac{1}{K_2}\left(e^{-K_1\,K_2\,T} - 1\right) \end{bmatrix} \tag{9.9}$$

Substituting the numerical values from the control system for K_1, K_2 and K_3, and assigning a sampling period of T=0.03s results in the system matrix

$$\underline{A}_D = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & -0.057137 \\ 0 & 0.9535 \end{bmatrix} \qquad (9.10)$$

The input matrix can be stated as

$$\underline{b}_D = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 0.0089128 \\ -0.30227 \end{bmatrix} \qquad (9.11)$$

accordingly.

ref. 4.5 :

The eigenvalues of the continuous time system result from the characteristic equation

$$\det(s\,\underline{I} - \underline{A}) = \det\left(\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 0 & -1/\tau \end{bmatrix}\right)$$

$$= s\,(s + 1/\tau) \equiv 0 \qquad (9.12)$$

This yields

$s_1 = 0$ and $s_2 = -1/\tau$

and

$z_1 = 1$ and $z_2 = e^{-T/\tau}$

respectively.

The system is unstable, because one eigenvalue is located on the imaginary axis in the s-plane, respectively the unit circle |z|=1 in the z-plane. In the following, a state feedback for the sampled data system is computed. Assigning the poles of the closed loop sampled data system to (z_1, z_2), yields the characteristic polynomial

$$P(z) = (z - z_1)(z - z_2)$$
$$= z^2 - (z_1 + z_2)\,z + z_1 z_2 \qquad (9.13)$$

According to Eq.(2.16), the eigenvalues of the closed loop sampled data system can be computed with $\underline{F} = \underline{f}^T$ by

$$\det(z\,\underline{I} - \underline{A}_D + \underline{b}_D \underline{f}^T)$$

$$= \det\left(\begin{bmatrix} z & 0 \\ 0 & z \end{bmatrix} - \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}[f_1, f_2]\right)$$

$$= \det\left(\begin{bmatrix} z - a_{11} + b_1 f_1 & -a_{12} + b_1 f_2 \\ -a_{21} + b_2 f_1 & z - a_{22} + b_2 f_2 \end{bmatrix}\right)$$

$$= z^2 + (-a_{22} + b_2 f_2 - a_{11} + b_1 f_1)\,z$$

$$-(-a_{21} + b_2 f_1)(-a_{12} + b_1 f_2)$$

$$+(-a_{11} + b_1 f_1)(-a_{22} + b_2 f_2) \equiv 0 \qquad (9.14)$$

By comparing the coefficients of this polynomial with P(z), a linear system of equations for the unknowns f_1 and f_2 can be formulated:

$$(-a_{22} + b_2 f_2 - a_{11} + b_1 f_1) = -(z_1 + z_2) \qquad (9.15)$$

$$(-a_{21} + b_2 f_1)(-a_{12} + b_1 f_2)$$
$$+(-a_{11} + b_1 f_1)(-a_{22} + b_2 f_2) = z_1 z_2 \qquad (9.16)$$

Solving the equation 9.15 for f_1 yields

$$f_1 = \frac{-z_1 - z_2 + a_{22} - b_2 f_2 + a_{11}}{b_1} \qquad (9.17)$$

Substituting this into the equation 9.16, yields for f_2 :

$$f_2 = \frac{-a_{21}a_{12} + a_{11}a_{22} - z_1 z_2 + \dfrac{(a_{11} + a_{22} - z_1 - z_2)(b_2 a_{12} - b_1 a_{22})}{b_1}}{-b_1 a_{21} + b_2 a_{11} + \dfrac{b_2}{b_1}\left(b_2 a_{12} - a_{22} b_1\right)} \qquad (9.18)$$

Substituting the numerical values for the matrices $\underline{A}_D$ and $\underline{b}_D$ and selecting the eigenvalues according to

$s_{1,2}$ bzw. $z_{1,2} = e^{T s_{1,2}}$

yields

$$\underline{f}^T = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

In order to achieve exactness in the steady state ($x_1 = w$), the prefilter

$$\underline{V} = [\ \underline{C}\,(\underline{I} - \underline{A}_D + \underline{B}_D\,\underline{F})^{-1}\,\underline{B}_D\,]^{-1} \qquad (9.19)$$

can be computed according to Eq.(2.40). In this case, the prefilter can be determined easier directly from the block diagram of the closed loop control system (Figure 9.4).

Under stationary conditions it can be stated:

1. $F\,(k\,T) = 0$

2. $x_2\,(k\,T) = 0$

Thus, the block diagram yields:

$$v\,w\,(kT) = f_1\,x_1\,(kT) \qquad (9.20)$$

Solving for v and setting $x_1\,(kT) = w\,(kT)$ results in:

$$v = \frac{f_1\,x_1\,(kT)}{w\,(kT)} = f_1 \qquad (9.21)$$

This completes the controller design for the system "cart". It has been designed such that there will be no steady state deviation of the cart position from the desired value.

ref. 4.6 :

For the state estimation only the position is measured. Thus, the output matrix is given by:

$$\underline{C} = \underline{c}^{\,T} = [c_{1,}\ c_2] = [1,\ 0]$$

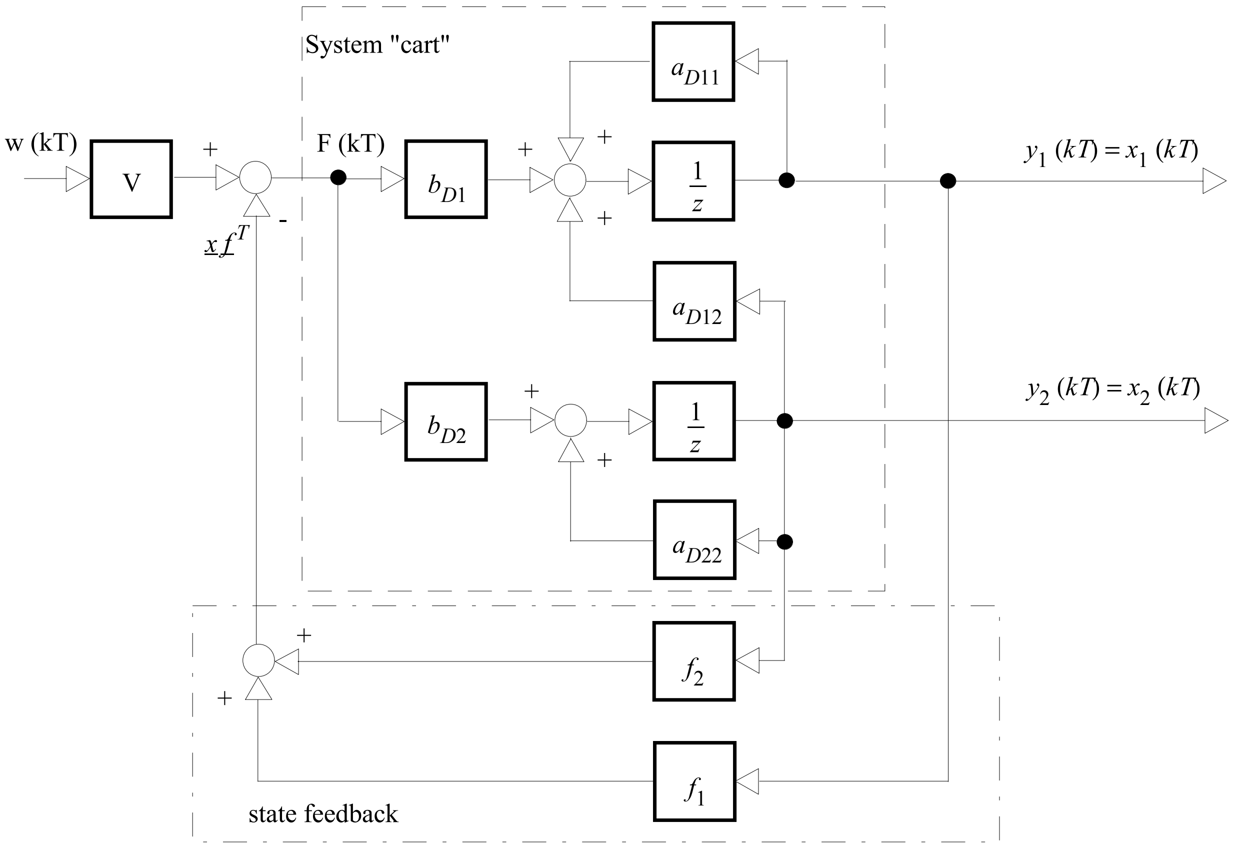


Figure 9.4 : Block diagram of the system "cart" with state feedback (sampled data system)

Using the relations

$$\underline{A}_D \rightarrow \underline{A}_D^T$$

$$\underline{b}_D \rightarrow \underline{c}$$

$$\underline{f}^T \rightarrow \underline{f}_B^T$$

which have been derived in chapter 2.3, the result from 4.5 can be employed for the computation of the observer matrix. This results in

$$f_{B1} = \frac{-z_1 - z_2 + a_{22} - c_2 f_{B2} + a_{11}}{c_1} \qquad (9.22)$$

and

$$f_{B2} = \frac{-a_{21}a_{12}+a_{11}a_{22}-z_1z_2+\frac{(a_{11}+a_{22}-z_1-z_2)(c_2a_{12}-c_1a_{22})}{c_1}}{-c_1a_{21}+c_2a_{11}+\frac{c_2}{c_1}\left(c_2a_{12}-a_{22}c_1\right)} \qquad (9.23)$$

With the numerical values for $\underline{A}_D$ and $\underline{b}_D$ and the poles z_i one obtains the observer matrix $\underline{f}_{B_i}$.

$$f_{B2} = \frac{-a_{21}a_{12}+a_{11}a_{22}-z_1z_2+\frac{(a_{11}+a_{22}-z_1-z_2)(c_2a_{12}-c_1a_{22})}{c_1}}{-c_1a_{21}+c_2a_{11}+\frac{c_2}{c_1}\left(c_2a_{12}-a_{22}c_1\right)} \qquad (9.24)$$

9.2 Solutions to Chapter 6 (Carrying-Out the Experiment)

The following diagrams are based on one specific pendulum. These diagrams are only qualitative examples and need not agree on your measuring results, in particular the step response.

ref. 6.1

Since the available pendulum has a small friction constant, the experiment has been performed with F_1 = 10 N and F_2 = 12 N.

According to the step responses in figure 9.5 the following numerical values have been determined:

$\tau = 0.4$ s

$F_1 = 10$ N	$x_{21\infty} = 1\ m/s$
$F_2 = 12$ N	$x_{22\infty} = 1{,}4\ m/s$

ref. 6.2

Definition:

A system $Y = \varphi\,(u)$ is called linear, if it satisfies the superposition and amplification principle. I.e. the relation

$$\varphi(c_1\, u_1 + c_2\, u_2) = c_1\, \varphi(u_1) + c_2\, \varphi(u_2)$$

with arbitrary constants u_1, u_2, c_1 and c_2 must hold.

An evaluation of this definition using the numerical values from figure 9.5 yields:

$F_{S2} = c_1\, F_{S1}$, mit $c_1 = 1.2$

$Y\,(F_{S1\infty}) = x_{21\infty} = 1\ m/s$

$Y\,(F_{S2\infty}) = x_{22\infty} = 1.4\ m/s \;\neq\; c_1\, x_{21\infty} = 1.2\ m/s$

The last equation shows that for F_s=12N there is a difference of approximately 0.2N between the linear approximation and the actual steady state value. This is a violation of the amplification principle, thus proving the nonlinearity of the system.

These nonlinearities are mainly due to the effects of the static friction. They can be described by an additional voltage F_{S0}, which is superimposed on the input of the system.

ref. 6.3

For the steady state value one obtains:

$$x_2(t \to \infty) = \lim_{s \to 0} [s\, G_2(s)\, U(s)]$$

$$= G_2(0)\, u(t \to \infty)$$

or evaluated:

$$x_{2\infty} = \frac{1}{K_2} F_\infty \text{ mit } F_\infty = F_S + F_{S0} \quad .$$

For the superimposed disturbance voltage one has: $F_{S0} = F_0\, sign\,(x_2)$.

$$x_{21\infty} = \frac{1}{K_2}(F_{S1\infty} + F_0)$$

and

$$x_{22\infty} = \frac{1}{K_2}(F_{S2\infty} + F_0) \quad .$$

Solving these equations for F_0, yields with the numerical values from 6.1:

$$F_0 = \frac{-F_{S2\infty}\, x_{21\infty} + F_{S1\infty}\, x_{22\infty}}{x_{21\infty} - x_{22\infty}} = -5\ N$$

$$K_2 = \frac{F_{S2\infty} + F_0}{x_{21\infty}} = 10\ Ns/m$$

$$\tau = \frac{1}{K_2 K_1} \sim 0.4\, s$$

$$K_1 = \frac{1}{\tau K_2} = 0.25\ {}^{m}\!/\!{}_{N\,s}$$

In conjunction with the result of the preparation problem 4.3, the determined parameters yield a complete description of the analog system "cart":

$$\dot{\underline{x}} = \begin{bmatrix} 0 & 1\ {}^1\!/\!{}_s \\ 0 & -2.5\ {}^1\!/\!{}_s \end{bmatrix} \underline{x} + \begin{bmatrix} 0 \\ 0.25\ {}^{m}\!/\!{}_{Ns^2} \end{bmatrix} F$$

ref. 6.4

Using the equations from the preparation problems 4.3 and 4.4, the system description determined in 6.3 is transformed into a sampled data system. One obtains:

$$\underline{A}_D = \begin{bmatrix} 1 & 0.0289 \\ 0 & 0.9277 \end{bmatrix}$$

$$\underline{b}_D = \begin{bmatrix} 1.09 \cdot 10^{-4} \\ 7.23 \cdot 10^{-3} \end{bmatrix}$$

ref. 6.5

Using the equation derived in the preparation problem 4.5 and the numerical values which have been determined up to this point, the feedback matrix is computed as

$$\underline{f}^T = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 34.174 \\ 13.304 \end{bmatrix}$$

ref. 6.6

$s_i\ [{}^1\!/\!{}_s]$	$z_i = e^{T s_i}$	$\underline{f}B_i$
- 6	0.9139	0.2572, 0.29587
- 20	0.5488	0.8312, 4.9680

Table 9.1 : Numerical values of the observer matrix for the system "cart"

The equation determined in the preparation problem 4.6 for the computation of the observer matrix yields for the poles $s_{12} = -6$ and $s_{12} = -20$ the following results:

ref. 6.7

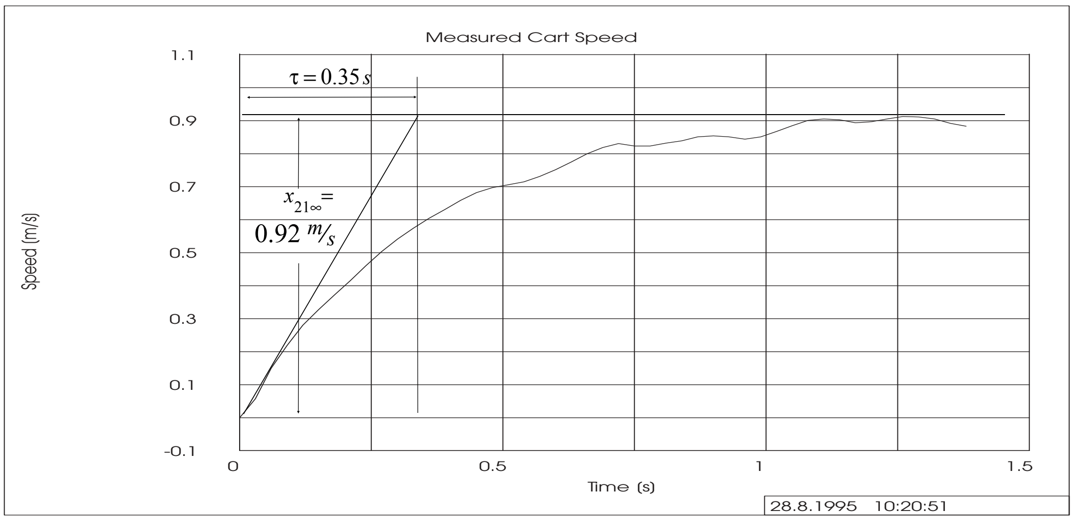
refer to figures 9.6 to 9.9

ref. 6.8

refer to figures 9.10 to 9.13

ref. 6.9

refer to figures 9.14 to 9.18



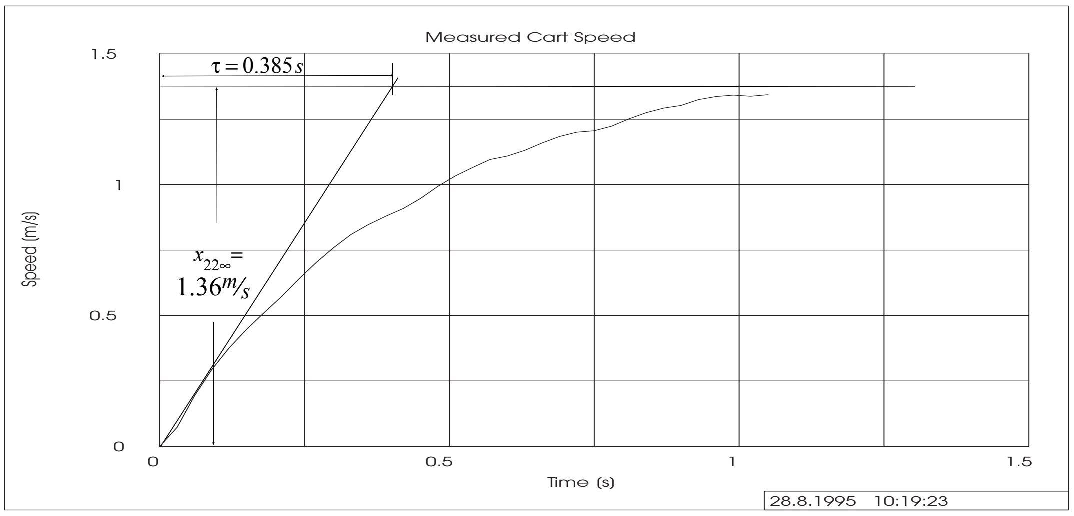
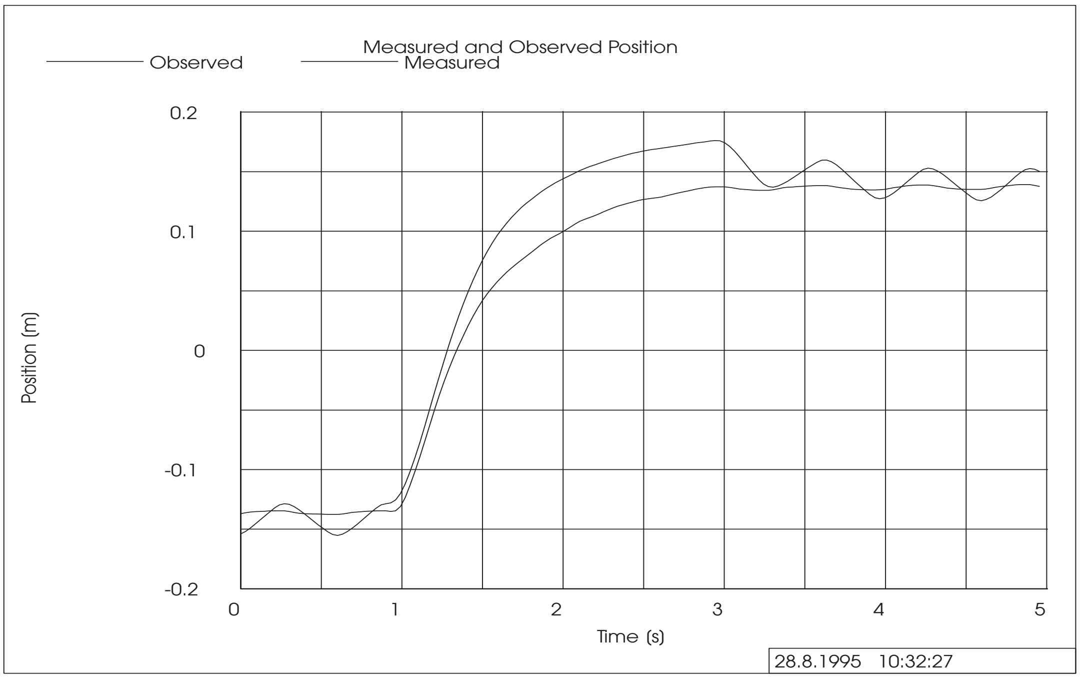


Figure 9.5 : Step response of the system "cart"

top : $F_S = 10N$

bottom : $F_S = 12N$



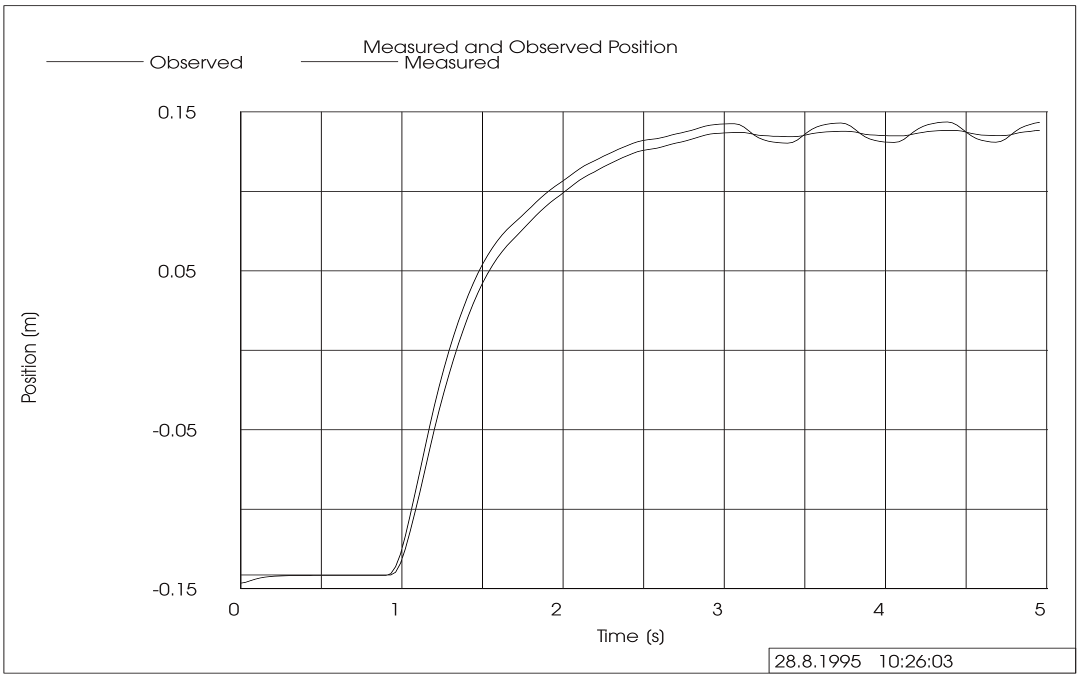
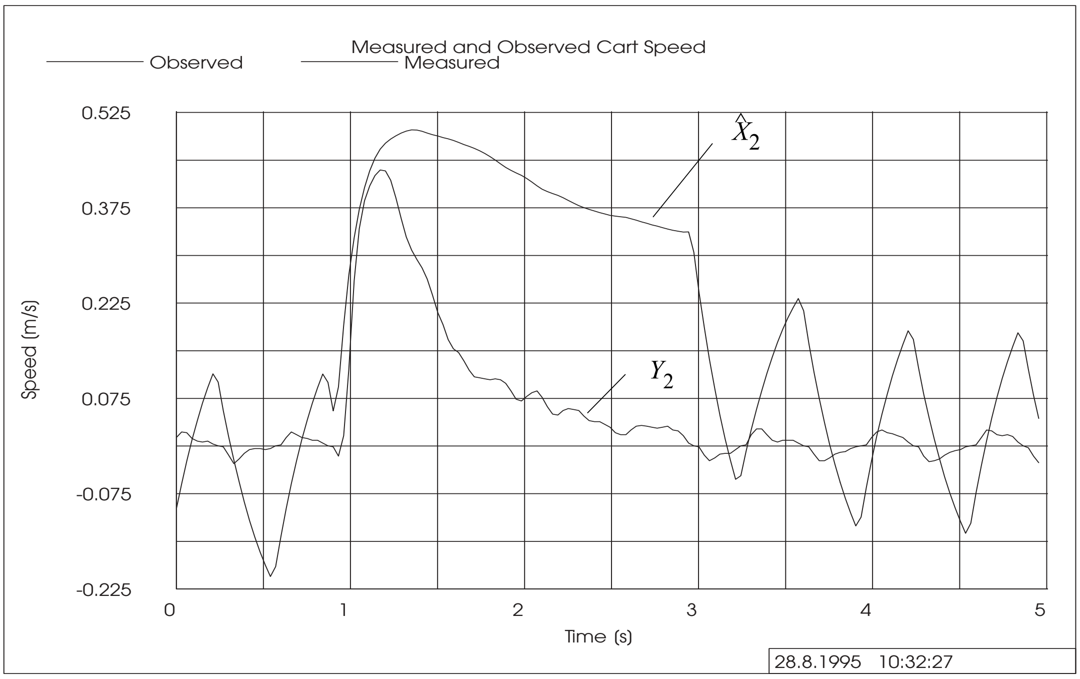


Figure 9.6 : Observation of the position for the system
"cart"
Compensation: constant
top: Observer poles at: s=-6 1/s
bottom: Observer poles at s=-20 1/s



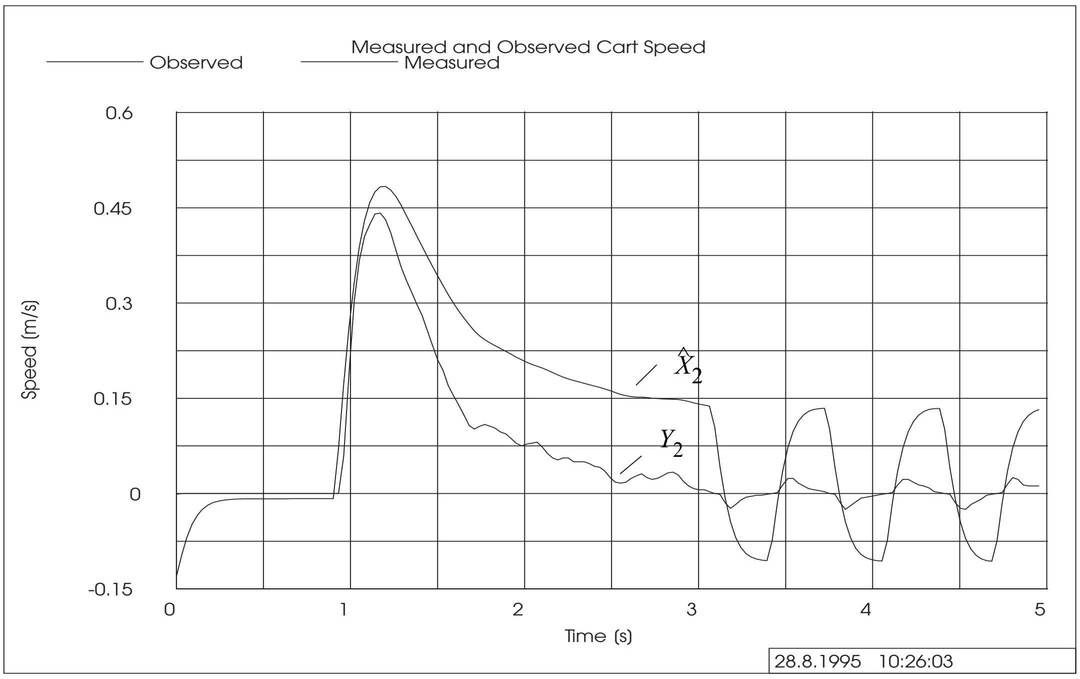


Figure 9.7 : Observation of the speed for the system "cart"

Compensation: constant

top: Observer poles at: s=-6 1/s

bottom: Observer poles at s=-20 1/s

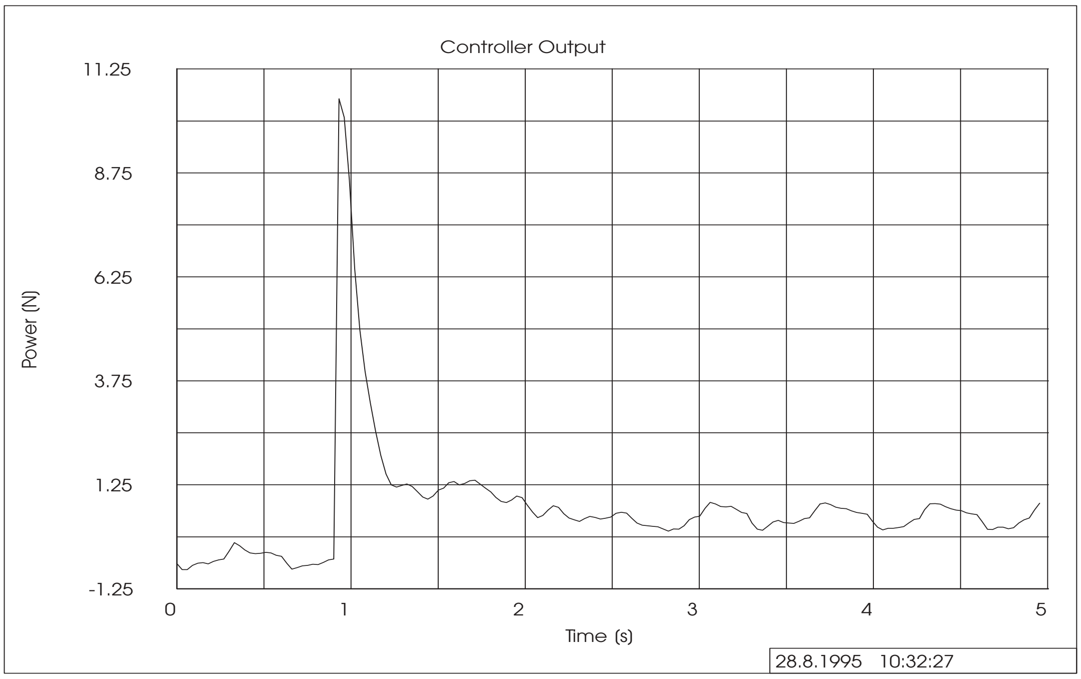


Figure 9.8 : Transient of the control voltage for the system "cart" (uncompensated)

Compensation: constant,

the Luenberger observer poles have no influence on this quantity

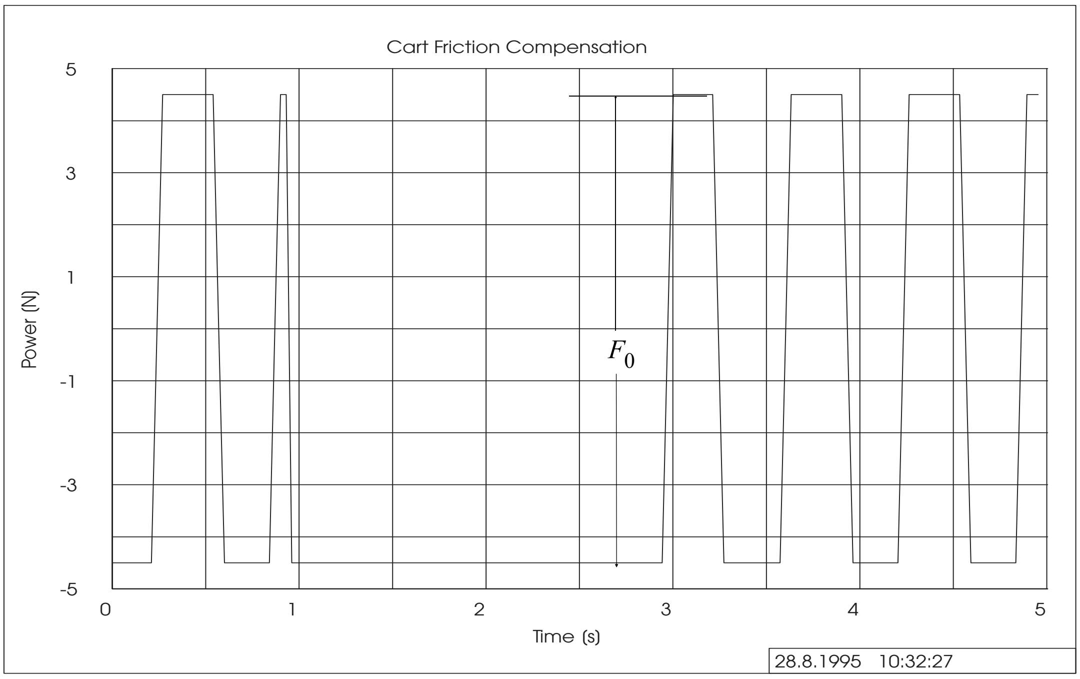
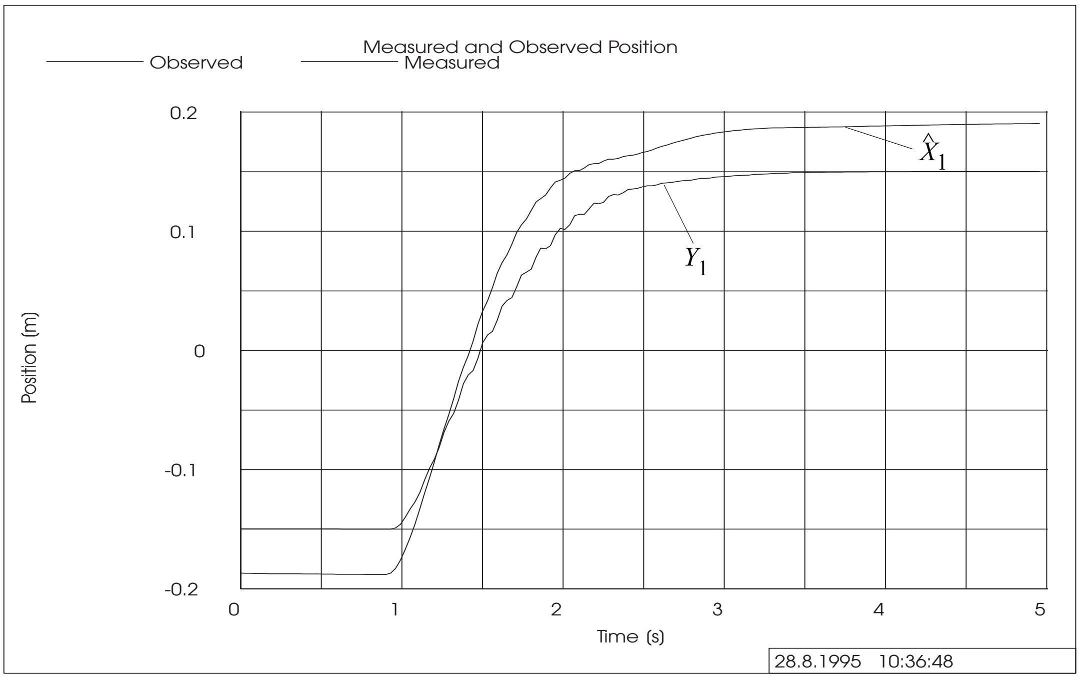


Figure 9.9 : Transient of the compensation voltage for the system "cart"

Compensation: constant,

the Luenberger observer poles have no influence on this quantity



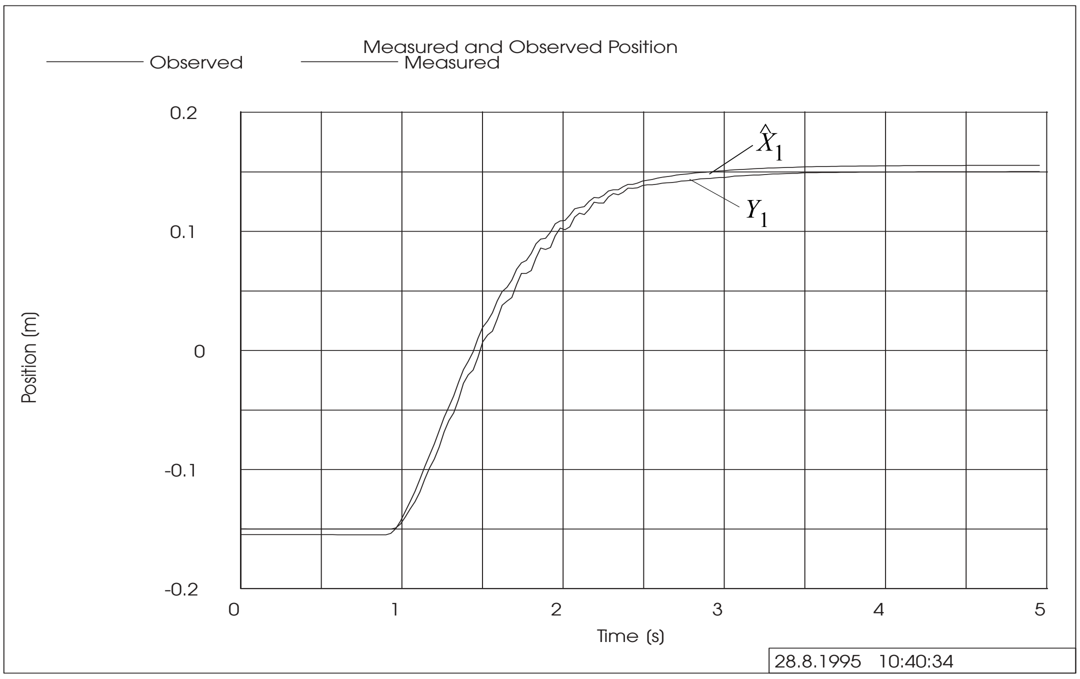
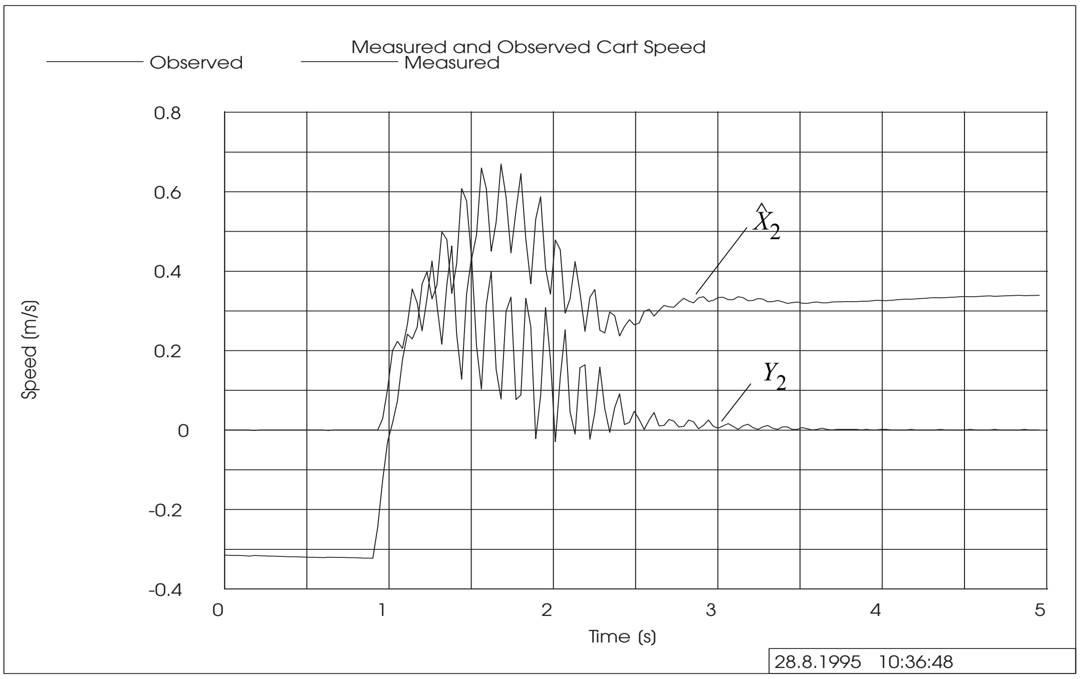


Figure 9.10 : Observation of the position for the system "cart"

Compensation: Observer

top: Observer poles at: s=-6 1/s

bottom: Observer poles at s=-20 1/s



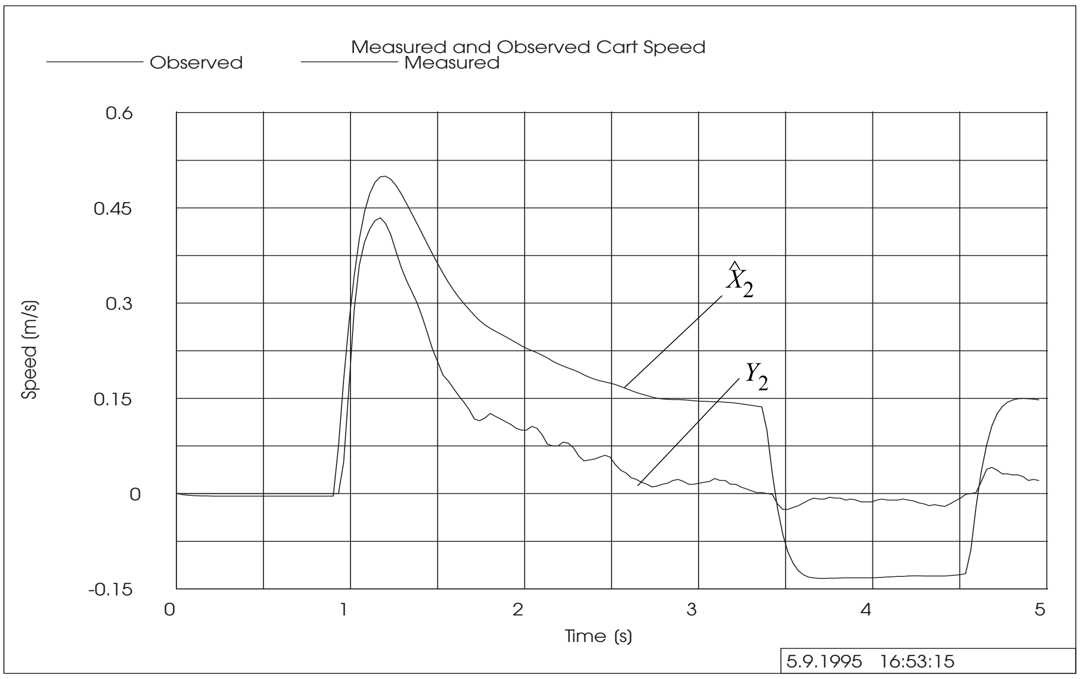


Figure 9.11 : Observation of the speed for the system "cart"

Compensation: Observer

top: Observer poles at: s=-6 1/s

bottom: Observer poles at s=-20 1/s

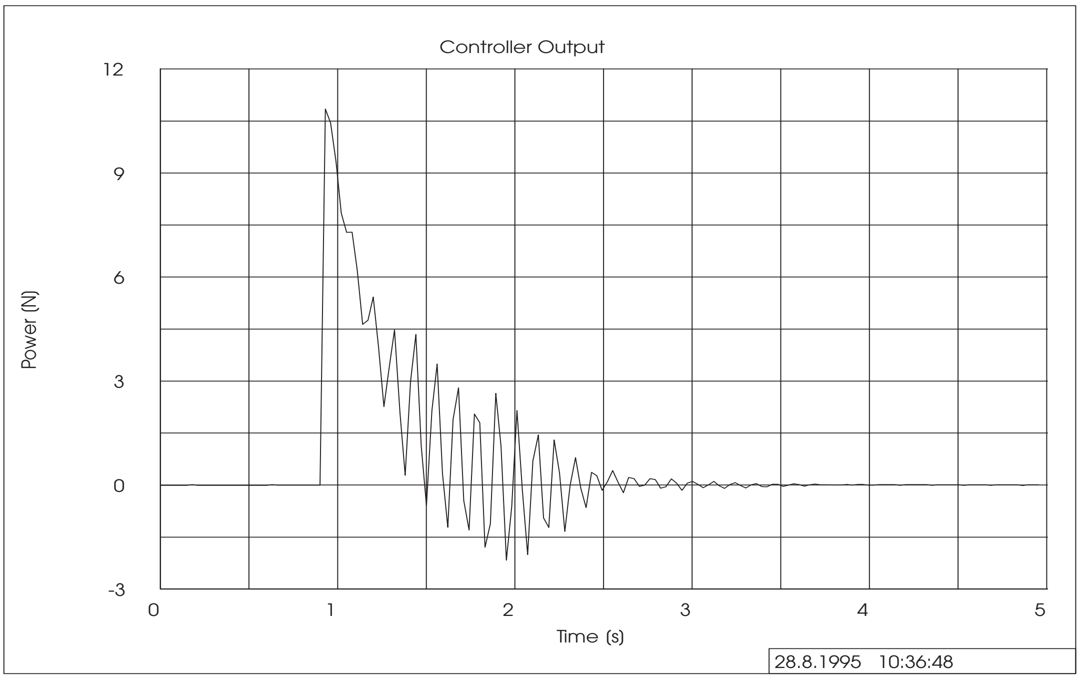


Figure 9.12 : Transient of the control voltage for the system "cart" (uncompensated)

Compensation: Observer

the Luenberger observer poles have no influence on this quantity

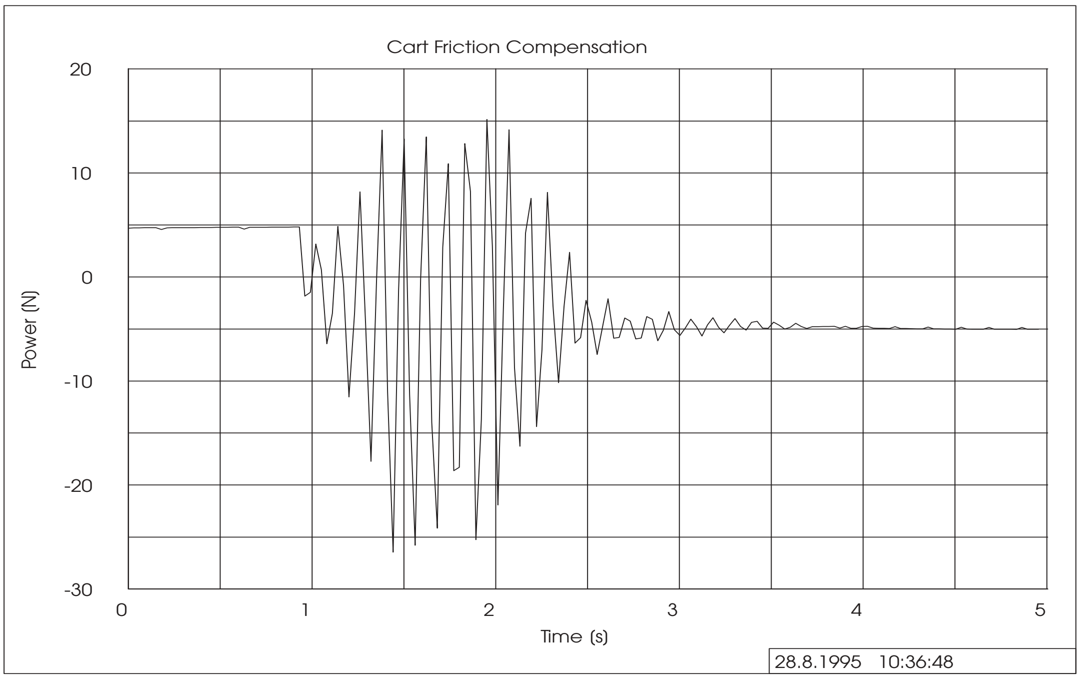
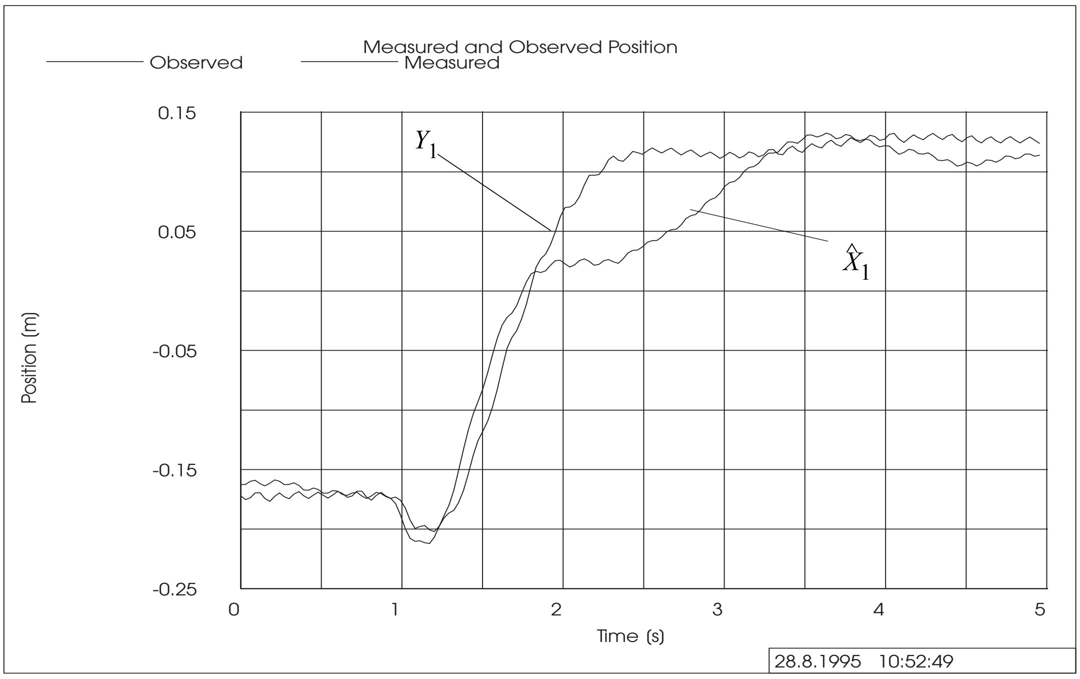


Figure 9:13 : Transient of the compensation voltage for system "cart"

Compensation: Observer,

the Luenberger observer poles have no influence on this quantity



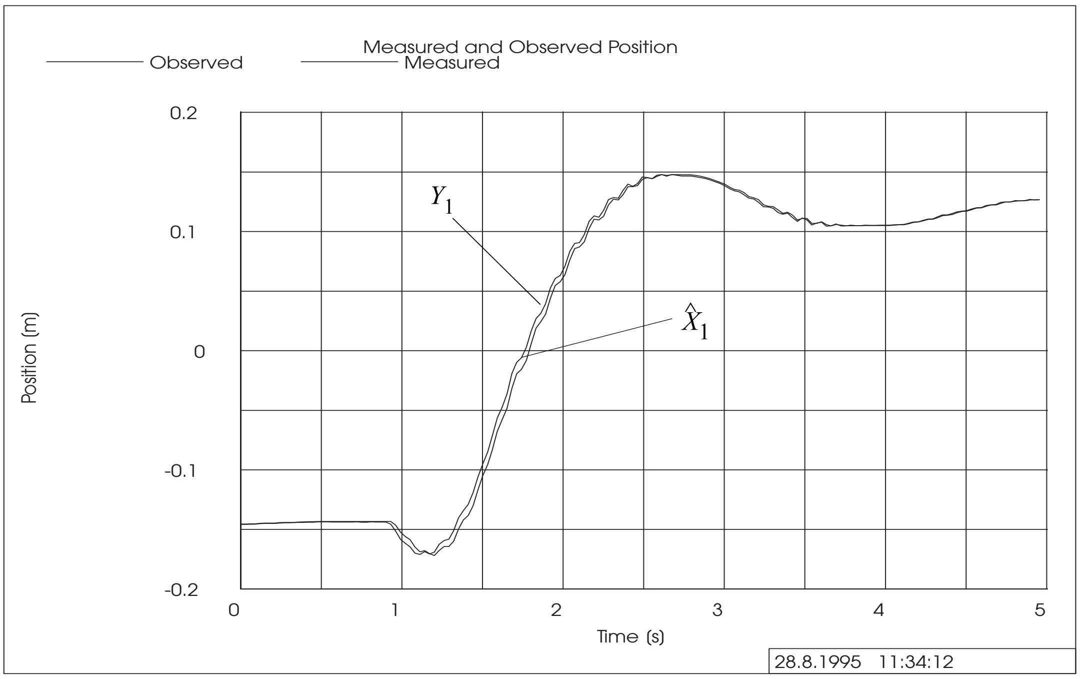
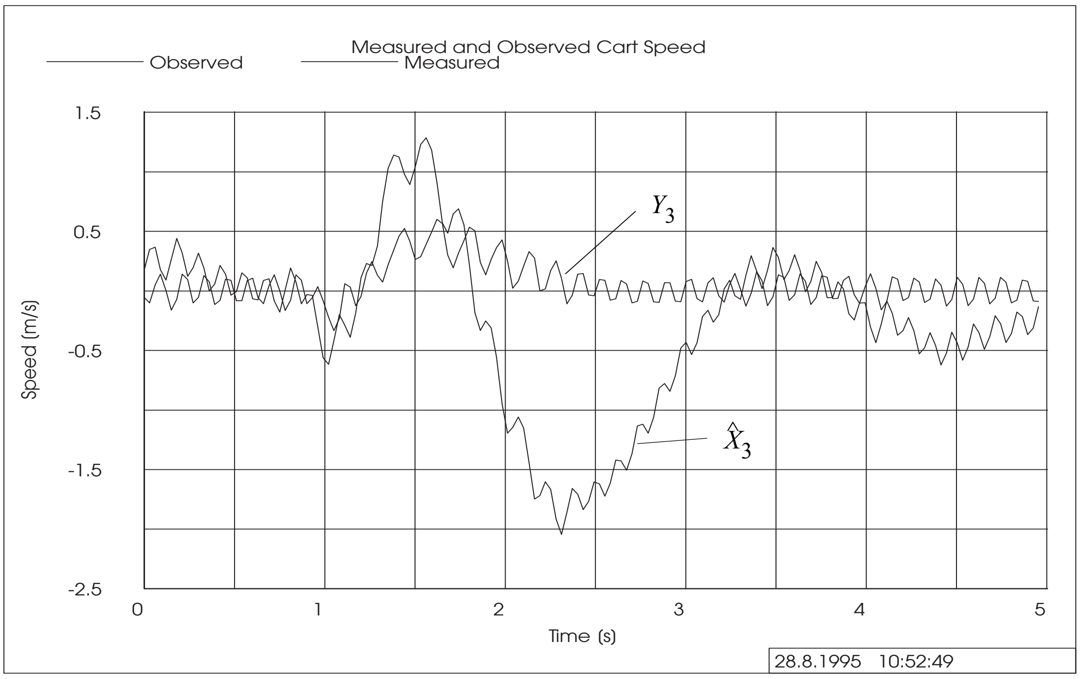


Figure 9.14 : Observation of the position for the system
"inverted pendulum"
top: Observer poles at: s=-6 1/s
bottom: Observer poles at s=-20 1/s



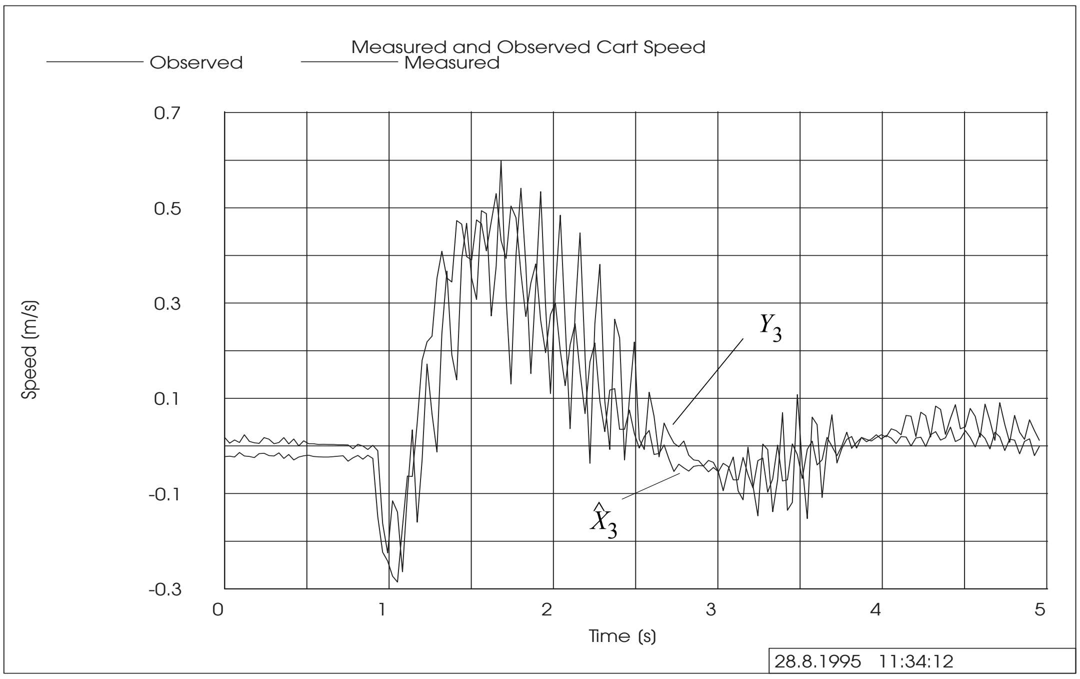
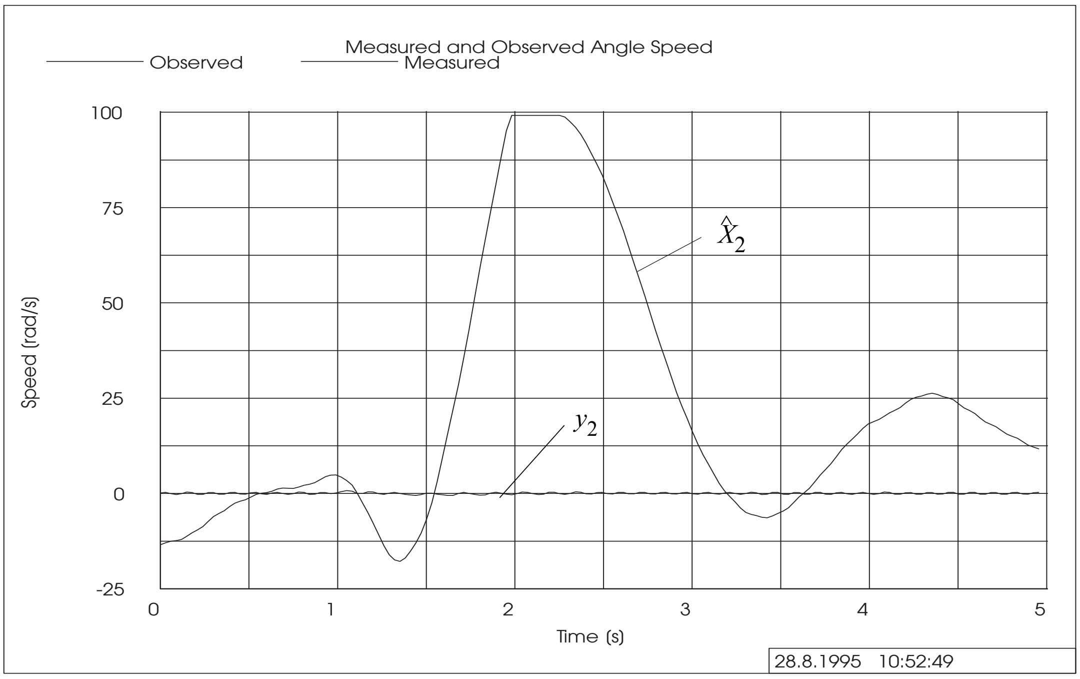


Figure 9.15 : Observation of the speed for the system "inverted pendulum"

top: Observer poles at: s=-6 1/s

bottom: Observer poles at s=-20 1/s



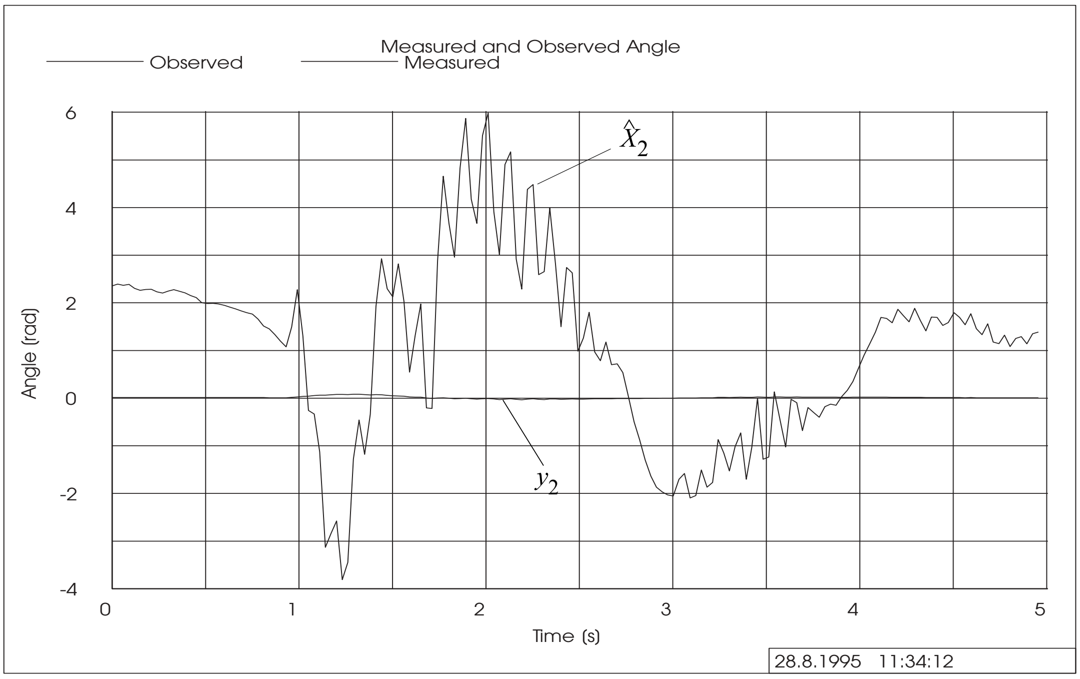


Figure 9.16 : Observation of the angle for the system
"inverted pendulum"
top: Observer poles at: s=-6 1/s 1/s
bottom: Observer poles at s=-20 1/s

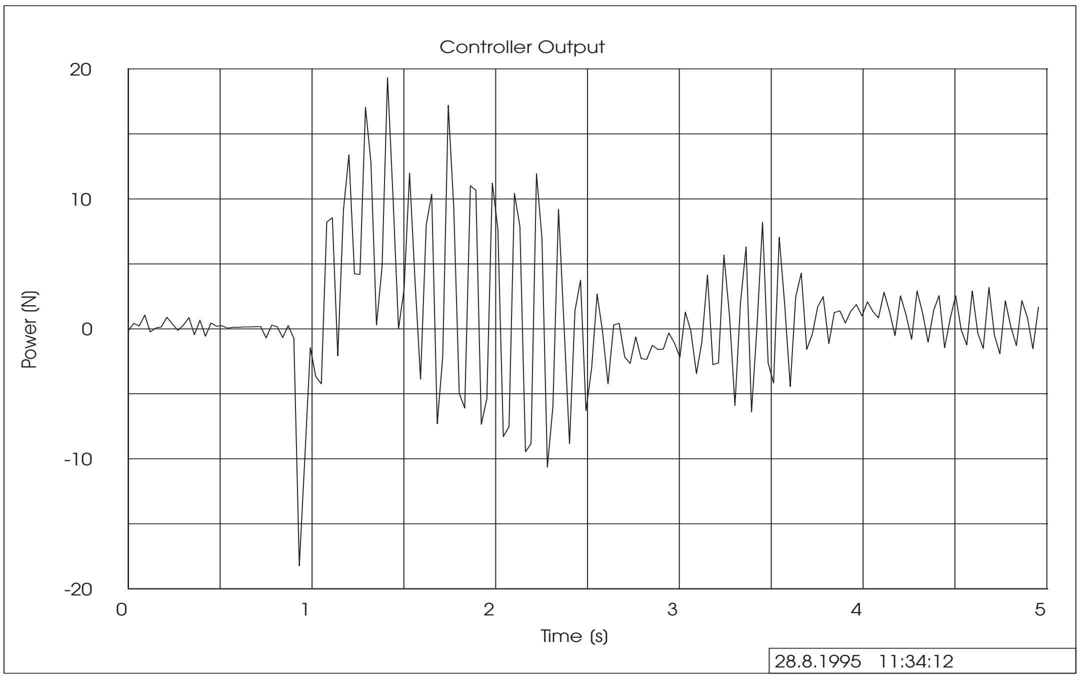


Figure 9.17 : Transient of the control voltage for the
system "inverted pendulum" (uncompensated)
the Luenberger observer poles have no influence on this quantity
top: Observer poles at: s=-6 1/s
bottom: Observer poles at s=-20 1/s

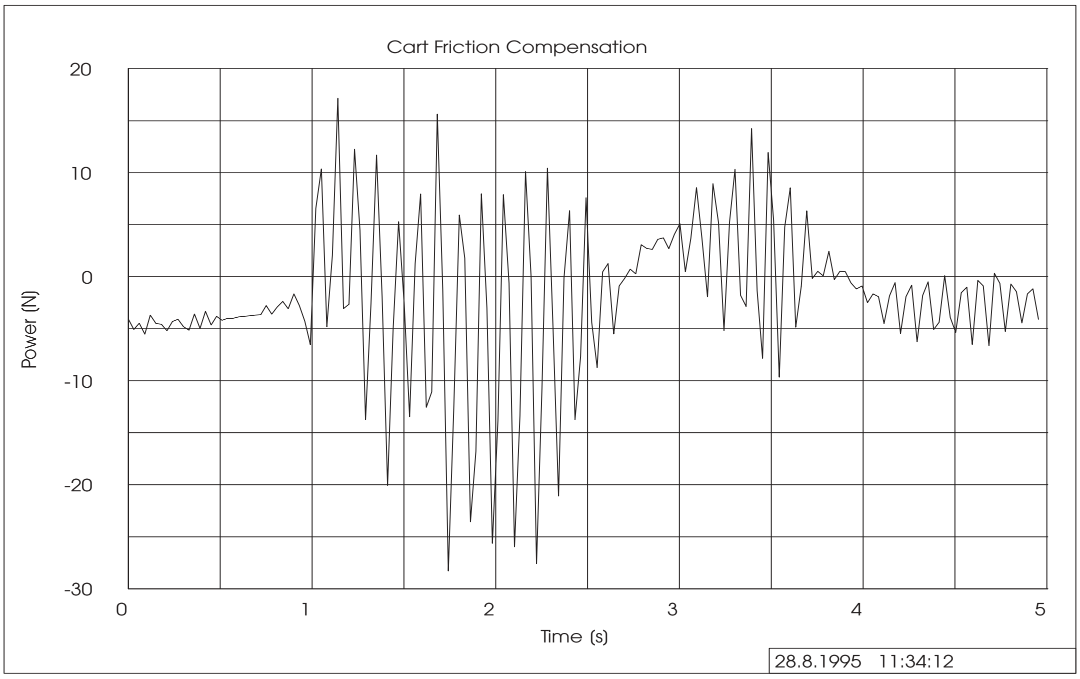


Figure 9.18 : Transient of the compensation voltage for
system "inverted pendulum"
the Luenberger observer poles have no influence on this quantity
top: Observer poles at: s=-6 1/s
bottom: Observer poles at s=-20 1/s

9.3 Solutions to Chapter 7 (Experiment Elaboration)

ref. 7.1 :

In the figures 9.6 to 9.9 the state feedback for the "cart" has been performed using a constant disturbance compensation. To this end the observer poles $s_{1,2} = -6\ 1/s$ and $s_{1,2} = -20\ 1/s$ (i.e. $z_{1,2} = 0.835$ and $z_{1,2} = 0.5488$) have been selected. The measurements for the position and the velocity oscillate in the steady state. (see figures 9.9 and 9.6, 9.7).

Figure 9.6 shows the time evolution of the state quantity x_1 (position). One sees that the quality of the position estimates with constant compensation is very poor. The reason is the insufficient compensation of the static friction. For t > 3s for example, the position of the cart does not change even though the input voltage differs from zero. On can see that this error is compensated better in the bottom half of figure 9.6 than in the top half. In the bottom half of the figure, the observer poles are located farther to the left in the s-plane. This results in larger numerical values for the observer matrix. Thus, compared with the input voltage, the output signal y_1 has a higher impact on the observer states.

In figure 9.7 the time evolution of the velocity is displayed. The errors of the position estimates cause large steady state errors. During the time period t <1.5s, i.e. for a moving cart, the observer provides satisfying estimates. The reason is that in case the control voltages is high, the insufficient compensation of the static friction is not as significant as for small voltages.

Figure 9.8 and 9.8 show the time evolution of the disturbance and the control voltages using constant compensation. One sees that the quantity F_{s0} depends only on the direction in which the cart moves.

ref. 7.2 :

The constant compensation voltage can be increased which results in an oscillation (limit cycle) of the cart around the steady state value. Furthermore compensation voltages which are to small result in steady state errors.

Due to poles located farther to the left in the s-plane, not only the observer but also an observer dependend control algorithm is stationary.

ref. 7.3 :

An observer must at least have the dynamics of the system under observation. Otherwise, the observer is unable to follow the system. Therefore, the observer poles must always be located to the left of the poles of the control system in the complex s-plane (resp. nearer to the origin in the z-plane).

Observer poles which are located too far to the left, are on the other hand distinguished by strong oscillations.

ref. 7.4 :

The Luenberger identity observer is connected in parallel to the controller of the "inverted pendulum". Based on the measured position, it provides estimates for the remaining states.

In figure 9.14 the measured and the estimated position are displayed in case different observer poles are selected. It can be seen that the estimation error for the position of the cart is relatively small in both diagrams. Especially for a pole location of s=-20 1/s, the observer reacts fast enough to provide estimates which are identical to the measured values. However, the good results are due to the fact that the position itself is an input signal of the observer.

In figure 9.15 the time evolution of the velocity is shown for the same period of time. Here, the correspondence between the estimated and the measured value can also be characterized as good. The observer in the bottom half of figure 9.15 yields better estimates than the observer in the top half of figure 9.15. Altogether, the good results can be explained by the fact that the velocity and the position are closely related in a system theoretical sense.

A different situation appears in case the angle is estimated. Since the angle is only weakly observable from the position, the corresponding large numerical value of the observer matrix amplifies the always present measurement noise to a multiple of the signal.

ref. 7.5 :

Using for example the bottom half of figure 9.16, an approximation of the angle can be gained by means of filtering the signal with a lowpass.

Better results will be obtained by using for instance a larger pendulum weight or a longer pendulum rod. It can be seen in the block diagram that the angle (x_2) acts on the acceleration of the cart via the matrix element a_{23}. Thus, it affects the velocity and the position measurements. Since this element is computed according to Eq.(3.23) by

$$a_{23} = \frac{M_1^2\, l_s^2\, g}{(\Theta_s + M_0\, l_s)\, M_1 + M_0\, l_s}$$

the measures stated above cause a stronger coupling between the velocity of the cart and the angle of the pendulum rod.

Fuzzy Controller

Inverted Pendulum

Date: 24-May-1996

1 Backgrounds of the Fuzzy Controller 1-1

1.1 The Fuzzy Set 1-1

1.2 The Linguistic Variable 1-2

1.3 The Fuzzification 1-3

1.4 The Rule and the Rule Base 1-3

1.5 The Inference Operation 1-4

1.5.1 The Aggregation 1-4

1.5.2 The Implication 1-4

1.5.3 The Conclusion 1-4

1.5.4 The Inference 1-5

1.6 The Defuzzification 1-5

1.7 Remarks 1-5

2 Realization of the Fuzzy Controller 2-1

1 Backgrounds of the Fuzzy Controller

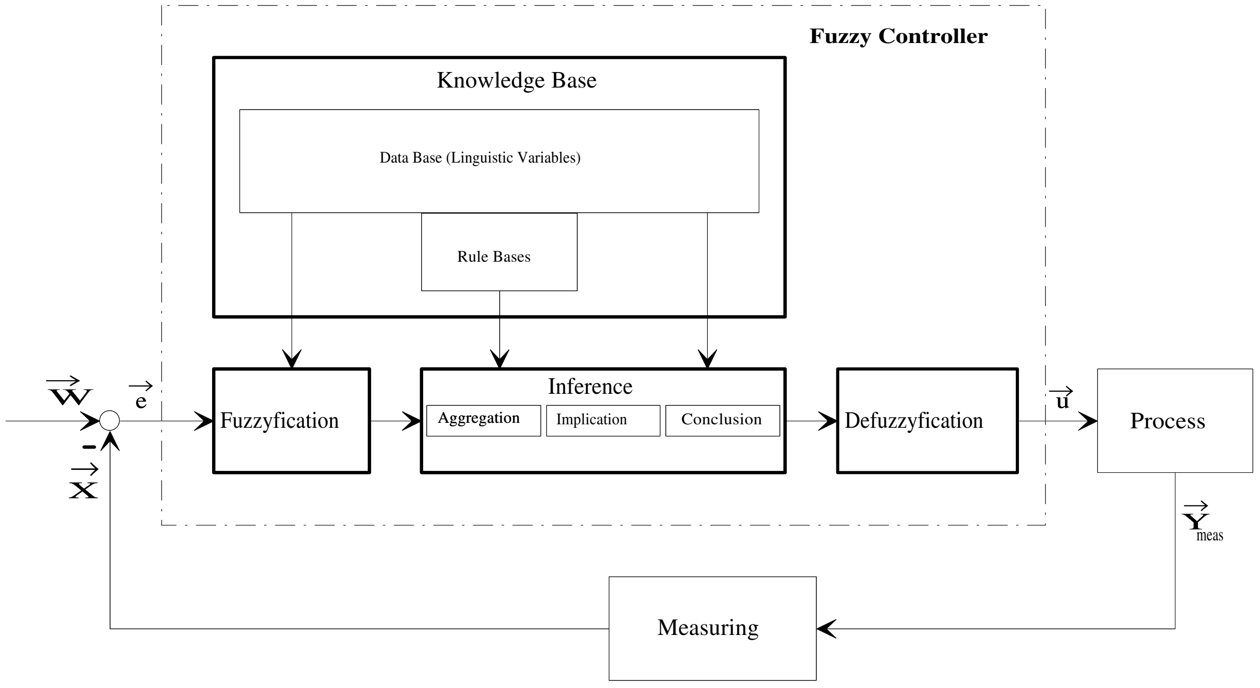


Figure 1.1: Components of the fuzzy Controller

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.

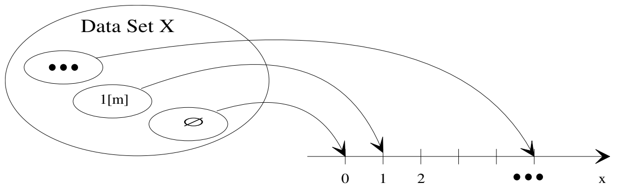


Figure 1.2: Mapping of the data set X to the real numbers

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.

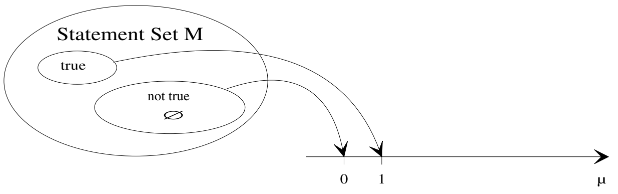


Figure 1.3: Mapping of the statement set M to the real numbers

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 $0 \le x \le 1$.

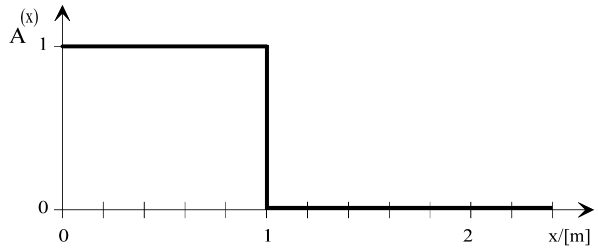


Figure 1.4: Example for a distinct set

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.

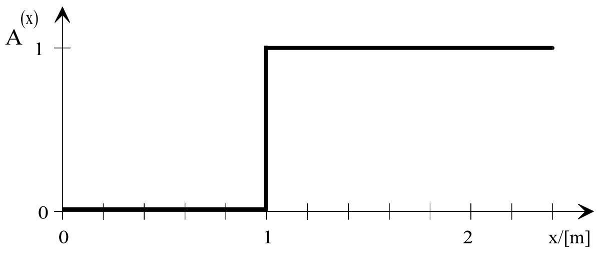


Figure 1.5: Example for the complement of the distinct set (see above)

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:

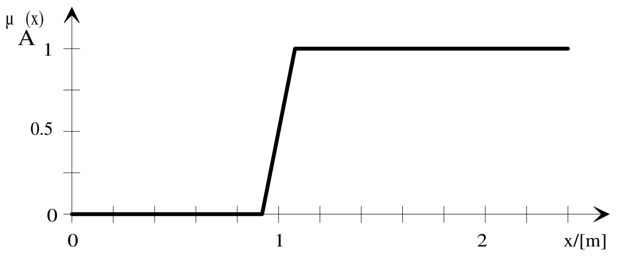


Figure 1.6: Example for a fuzzy set

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

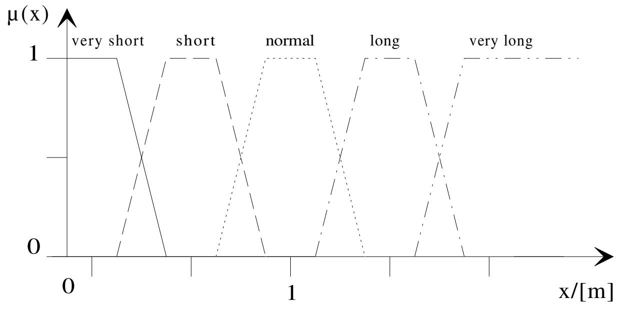


Figure 1.7: Example for a linguistic variable

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*

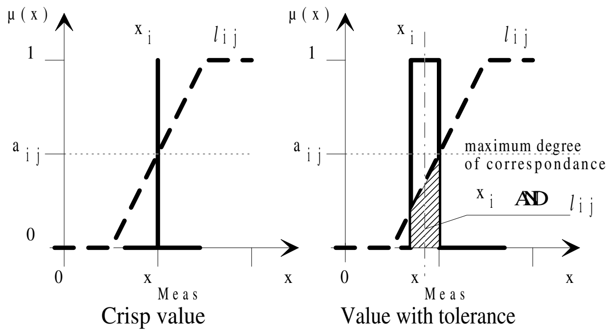


Figure 1.8: Fuzzification

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 $\mu_{(x)}$.

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 $a'_{gr} \quad \Leftrightarrow \quad a_{gr} \to a'_{gr} \le \mu_{\max}$

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:

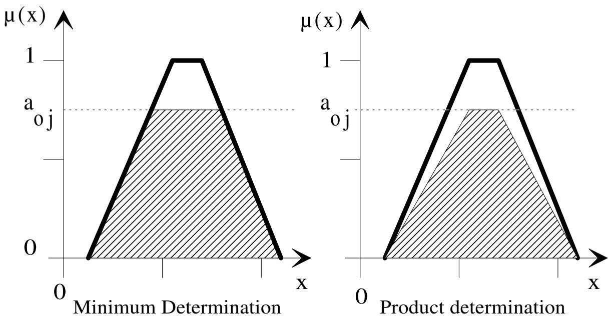


Figure 1.9: Min- and product method

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.

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:

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

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.

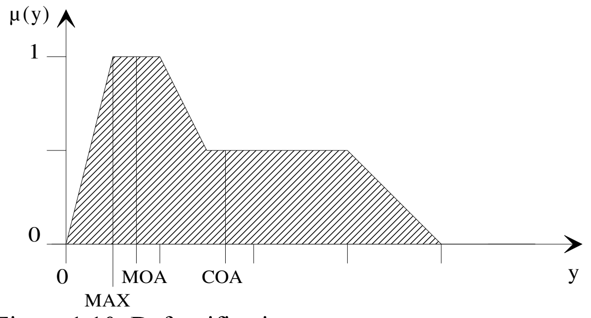


Figure 1.10: Defuzzification

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

$$\cup$$

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

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

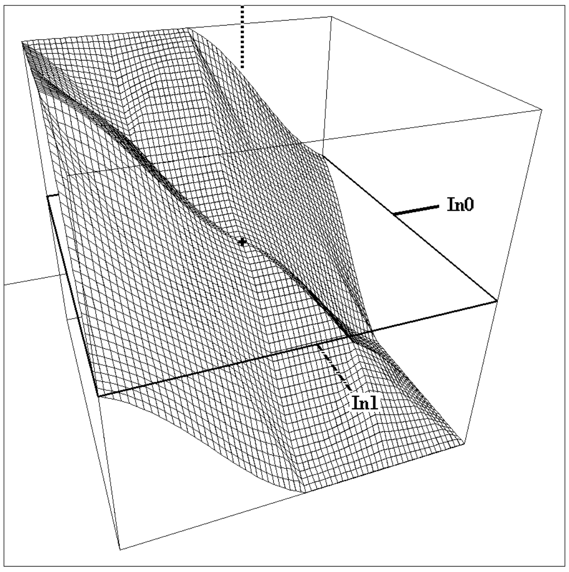
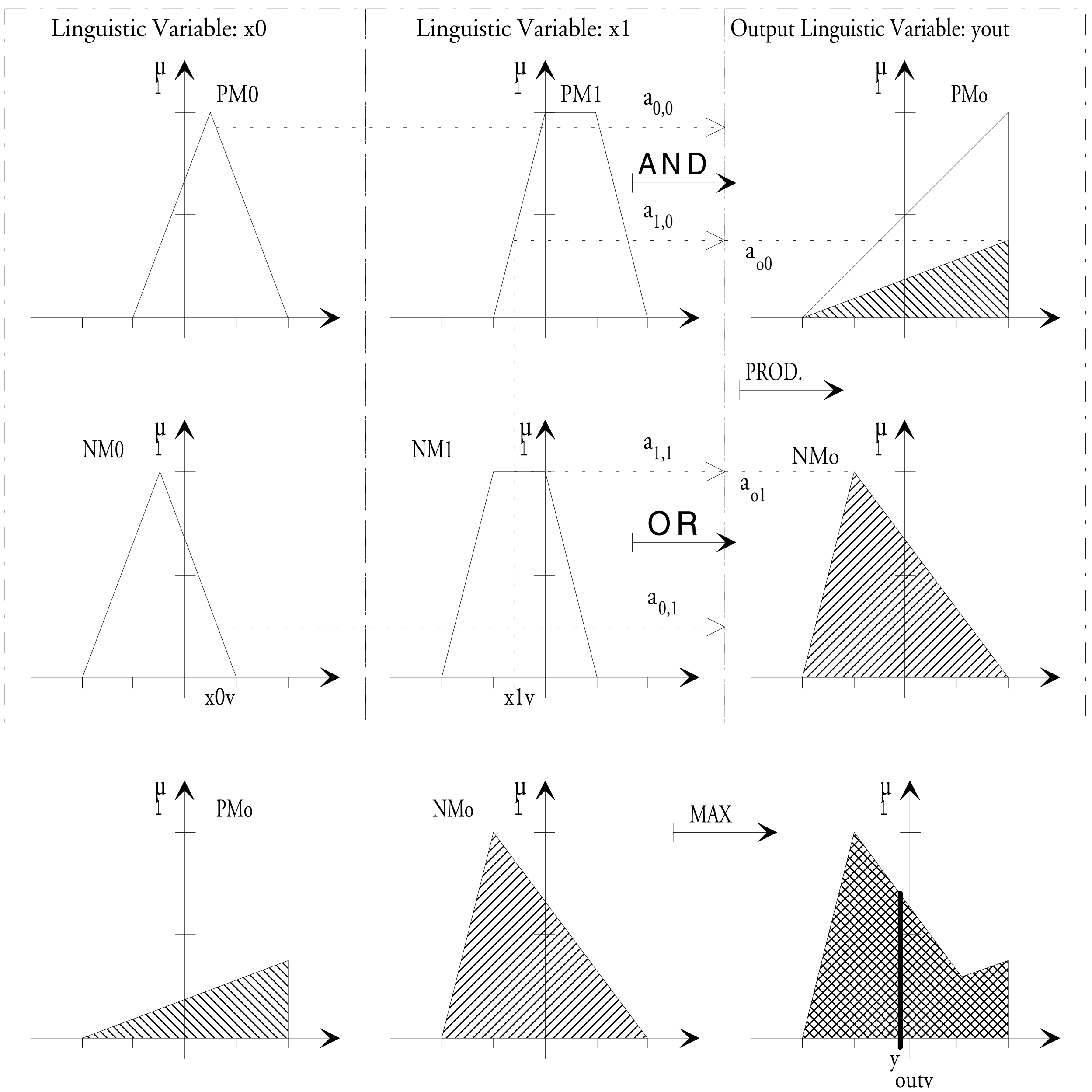


Figure 1.11: Example for a characteristic control area



```
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 pendulum. The setpoint for this controller is provided by the outer position controller. The difference of the pendulum angle setpoint and its measured value as well as the pendulum angular velocity are the required input signals of the angle controller. Its output signal controls directly the force acting on the cart. The difference of the position setpoint and its measured value as well as the cart speed are the required input signals of the cart position controller. The angle setpoint of the pendulum 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 pendulum friction and the cart friction which are estimated by two additional fuzzy blocks operating like fuzzy observers.

To estimate the cart friction the corresponding fuzzy observer requires the current control signal for the force and the speed of the cart as input signals. The fuzzy observer provides an offset value which is added to the control signal to compensate the friction effects of the

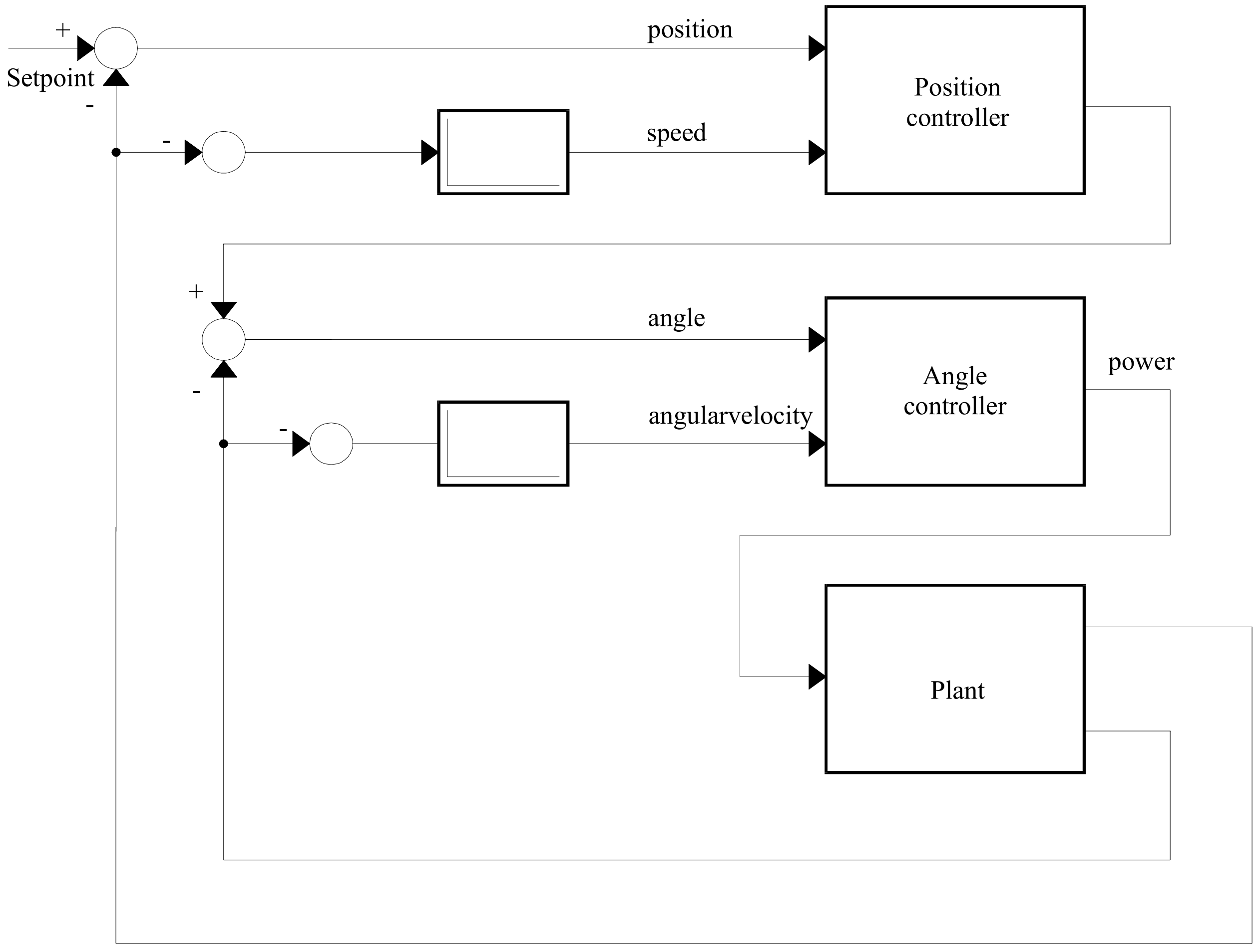


Figure 2.1: The reduced structure of the Fuzzy controller for the model "Inverted Pendulum"

cart. The fuzzy observer for the pendulum friction operates in a similar way. The input signals speed of the cart and angular velocity of the pendulum are used to determine an offset value. This offset value is added to the angle setpoint of the pendulum.

Difference quotients are used to determine the missing signals cart speed and angular velocity of the pendulum out of the measured signals cart position and pendulum 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 "Program Operation". 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

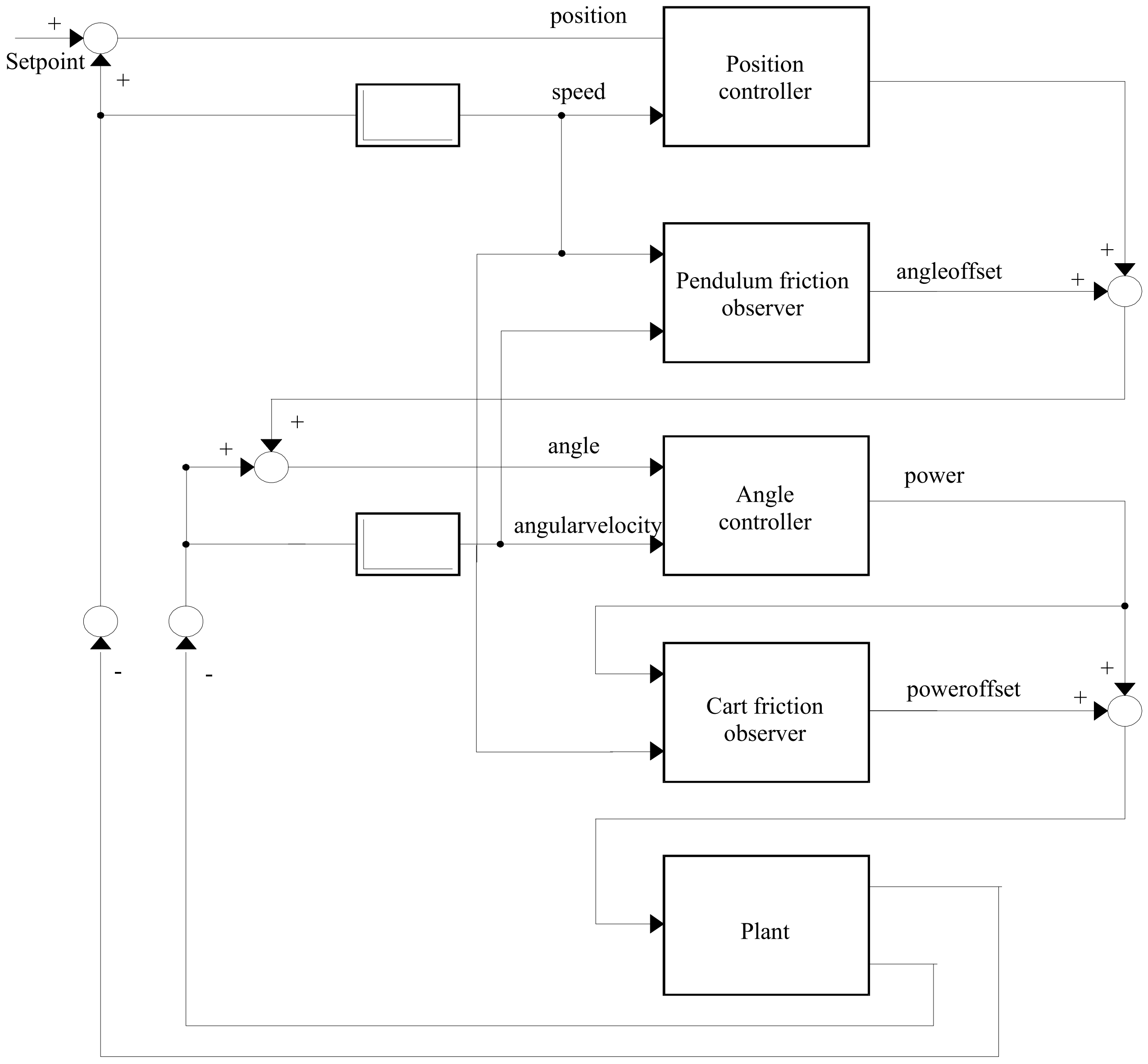


Figure 2.2: The structure of the Fuzzy controller for the model "Inverted Pendulum" with disturbance compensation

variable names, their ranges and units.

Name:	Unit	Minimum Value	Maximum Value
position	m	-1.5	1.5
speed	m/s	-4.0	4.0
angle	rad	-6.29	6.29
angleoffset	rad	-6.29	6.29
angular velocity	rad/s	-4.0	4.0
power	N	-20	20
poweroffset	N	-20	20

The values above represent limits set by the software which does not mean that the real values will reach these limits.

The following four fuzzy description files realize in example the fuzzy control of the system "Inverted Pendulum" according to figure 2.2:

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

```
/* variable definitions */
var input angle
 set bignegativ -0.04  1   0.0 0                endset
 set negativ   -0.04 0     -0.02 1   0 0        endset
 set zero      -0.02 0     0 1       0.02 0     endset
 set positiv   0 0         0.02 1    0.04 0     endset
 set bigpositiv 0.0 0      0.04 1               endset
endvar

var input anglespeed
 set bignegativ -0.5 1     0 0                  endset
 set negativ    -0.5 0     -0.15 1   0 0        endset
 set zero      -0.07 0     0 1       0.07 0     endset
 set positiv    0 0        0.15 1    0.5 0      endset
 set bigpositiv 0 0        0.5 1                endset
endvar

var output power
 set bignegativ -20.5 0    -20.0 1   -19.5 0    endset
 set midnegativ -16.5 0    -16.0 1   -15.5 0    endset
 set smallnegativ -8.5 0   -8.0 1    -7.5 0     endset
 set zero       -0.5 0     0 1       0.5 0      endset
 set smallpositiv 7.5 0    8.0 1     8.5 0      endset
 set midpositiv 15.5 0     16.0 1    16.5 0     endset
 set bigpositiv 19.5 0     20.0 1    20.5 0     endset
endvar

/* rule definitions */
if angle is negativ and anglespeed is bignegativ then power is
bignegativ end
if angle is negativ and anglespeed is negativ then power is
midnegativ end
if angle is negativ and anglespeed is zero then power is
smallnegativ end
if angle is negativ and anglespeed is positiv then power is zero
end
if angle is negativ and anglespeed is bigpositiv then power is
smallpositiv end

if angle is bignegativ and anglespeed is bignegativ then power
is bignegativ end
if angle is bignegativ and anglespeed is negativ then power is
bignegativ end
if angle is bignegativ and anglespeed is zero then power is
midnegativ end
if angle is bignegativ and anglespeed is positiv then power is
smallnegativ end
if angle is bignegativ and anglespeed is bigpositiv then power
is zero end

if angle is zero and anglespeed is bignegativ then power is
midnegativ end
if angle is zero and anglespeed is negativ then power is
smallnegativ end
if angle is zero and anglespeed is zero then power is zero end
if angle is zero and anglespeed is positiv then power is
smallpositiv end
if angle is zero and anglespeed is bigpositiv then power is
midpositiv end

if angle is positiv and anglespeed is bignegativ then power is
smallnegativ end
if angle is positiv and anglespeed is negativ then power is zero
end
if angle is positiv and anglespeed is zero then power is
smallpositiv end
if angle is positiv and anglespeed is positiv then power is
midpositiv end
if angle is positiv and anglespeed is bigpositiv then power is
bigpositiv end

if angle is bigpositiv and anglespeed is bignegativ then power
is zero end
if angle is bigpositiv and anglespeed is negativ then power is
smallpositiv end
if angle is bigpositiv and anglespeed is zero then power is
midpositiv end
if angle is bigpositiv and anglespeed is positiv then power is
bigpositiv end
if angle is bigpositiv and anglespeed is bigpositiv then power is
bigpositiv end
```

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

```
/* variable definitions */
var output angle
 set bignegativ -0.075 0   -0.07 1    -0.065 0   endset
 set midnegativ -0.05 0    -0.045 1   -0.04 0    endset
 set smallnegativ -0.025 0 -0.02 1    -0.015 0   endset
 set zero        -0.005 0   0 1        0.005 0   endset
 set smallpositiv 0.015 0   0.02 1     0.025 0   endset
 set midpositiv 0.04 0      0.045 1    0.05 0    endset
 set bigpositiv 0.065 0     0.07 1     0.075 0   endset
endvar

var input position
 set bignegativ -0.2 1      -0.1 0               endset
 set negativ   -0.2 0       -0.1 1     0 0       endset
 set zero      -0.1 0       0 1        0.1 0     endset
 set positiv   0 0          0.1 1      0.2 0     endset
 set bigpositiv 0.1 0       0.2 1                endset
endvar

var input speed
 set bignegativ -0.15 1     -0.075 0             endset
 set negativ   -0.15 0      -0.075 1   0 0       endset
 set zero      -0.075 0     0 1        0.075 0   endset
 set positiv   0 0          0.075 1    0.15 0    endset
 set bigpositiv 0.075 0     0.15 1               endset
endvar

/* rule definitions */
if position is zero and speed is bignegativ then angle is
midpositiv end
if position is zero and speed is negativ then angle is
smallpositiv end
if position is zero and speed is zero then angle is zero end
if position is zero and speed is positiv then angle is
smallnegativ end
if position is zero and speed is bigpositiv then angle is
midnegativ end

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

if position is bignegativ and speed is bignegativ then angle is
midpositiv end
if position is bignegativ and speed is negativ then angle is
bigpositiv end
if position is bignegativ and speed is zero then angle is
midpositiv end
if position is bignegativ and speed is positiv then angle is
smallpositiv end
if position is bignegativ and speed is bigpositiv then angle is
midnegativ end

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

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

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

```
/* compensation of angular Friction is neglected
/* - WERROR.FUZ is replaced by NOP.FUZ
/* variable definitions */
var output o
 set zero    -1.0 0.0   0.0 1.0   1.0 0.0   endset
endvar

var input i1
 set zero    -1.0 0.5   0.0 1.0   1.0 0.5   endset
endvar

var input i2
 set zero    -1.0 0.5   0.0 1.0   1.0 0.5   endset
endvar

/* rule definitions */
if i1 is zero then o is zero end
if i2 is zero then o is zero end
```

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

```
/* variable definitions */

var input speed
 set zero     -0.05 0   0 1     0.05 0   endset
 set notzero  -0.1 1    0 0     0.1 1    endset
 set positiv  0 0       0.1 1            endset
 set negativ   -0.1 1       0 0                        endset
endvar

var input power
 set negativ   -5.0 1       -1.0 0                     endset
 set zero      -1.0 0       0 1          1.0 0         endset
 set positiv   1.0 0        5.0 1                      endset
endvar

var output poweroffset
 set negativ   -4.0 0       -3.0 1       -2.0 0        endset
 set zero      -0.1 0       0 1          0.1 0         endset
 set positiv   2.0 0        3.0 1        4.0 0         endset
endvar

/* rule definitions */
if speed is zero    then poweroffset is zero end
if speed is positiv then poweroffset is positiv end
if speed is negativ then poweroffset is negativ 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.

Remark: Due to the small mechanical friction of the pendulum, the corresponding friction estimator may not be used.

Program Operation

Inverted Pendulum

(WINDOWS Version)

Printed: 02. November 2000

1 Program Operation 1-1

1.1 Program Start 1-1

1.2 Sensor Calibration 1-2

1.3 Main Window 1-3

1.4 Menu File 1-5

1.5 Menu IO-Interface 1-6

1.6 Menu Edit 1-7

1.7 Menu Run 1-9

1.8 Menu View 1-11

1.9 Menu Help 1-13

1.10 Description of the File Formats 1-14

1.10.1 The Format of the Fuzzy Description File (*.FUZ) 1-14

1.10.2 Format of the Error Output File ERROR.OUT 1-15

1.10.3 Format of the Fuzzy Controller File for the Laboratory Experiment PS600 (*.FBW) 1-16

1.10.4 Format of the State Controller File for the Laboratory Experiment PS600 (*.STA) 1-16

1.10.5 The Format of the Documentation File *.PLD 1-17

1.11 The DEMO Version 1-18

1 Program Operation

The software package provides two versions to control the plant "Inverted Pendulum". The first version is a state controller as described in chapter "State Control Inverted Pendulum". Controller parameters are on-line adjustable. The second version to control the PS600 Inverted Pendulum system is a fuzzy controller as described in chapter "Fuzzy Controller Inverted Pendulum". Controller adjustments are storable to a hard disk and may be read at a later time. The setpoint for the pendulum 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

Operating the program is significantly simplified when a mouse is used. So ascertain that a mouse is installed at your computer and that the mouse driver is running before you start the controller program.

The correct execution of the program requires that besides **PENDW16.EXE** the following files are available in the actual directory:

BC450RTL.DLL
DAC98.DRV
DIC24.DRV
PLOT16.DLL
PS6PEND.HLP
PENDW16.INI
PENSRV16.DLL
TIMER16.DLL

together with the parameter files and fuzzy description files
PENDULUM.FBW
PENDULUM.STA
XCONTROL.FUZ
WCONTROL.FUZ
XERROR.FUZ
NOP.FUZ

The additional file
P_CART.STA

is required only when the system single "cart" is to be controlled (see menu ’Edit’, item ’Switch to system ...’). The single inverted pendulum may be controlled either by a state or by a fuzzy controller whereas the single cart is controllable only by a state controller.

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).

An additional driver DUMMY.DRV is required for the DEMO version of the program.

The help file **PS6PEND.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 **PENDW16.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.

PENDULUM.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.

PENDULUM.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 **PENDW16.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 'PS600 Inverted Pendulum Calibration Dialog' (see figure 1.2) will appear automatically on the screen.

This dialog allows for checking the system type of the mechanics as well as for calibrating the cart position and pendulum angle sensors (incremental encoders). 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 additional check box labelled 'View plots' allows for viewing at the controller output and the resulting cart position after calibrating the position sensor. The calibration steps are as follows:

Pressing the 'Start' button will at first start the timer for the sampling period of any controller and then check the connections between the mechanics and the actuator by reading two of its signals which identify the type of the mechanics. In this case the reading should be equal to the mask for an inverted pendulum system. After pressing the 'Start' button the background colour will turn to green until either valid readings 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 measure the zero angle of the pendulum pointing to the ground (non-inverted position). A possibly swinging pendulum should be damped until its amplitude is less than approximately 1 degree. Such a measurement is taken as a successful result, which is indicated by a white background colour, a check mark replacing the 'Start' button and an automatic jump to the next calibration step. If a valid measurement range could not be obtained within a time period of 120 s the dialog signals a false result, recognizable by a red background colour, a 'Retry' button replacing the 'Start button and an additional error message (See below for possible error

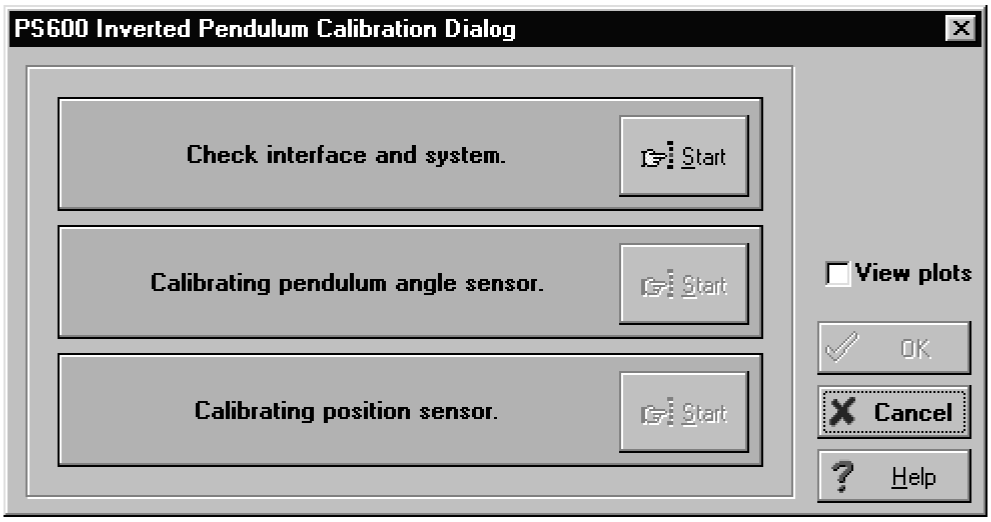

PS600 Inverted Pendulum Calibration Dialog
Check interface and system.
Start
Calibrating pendulum angle sensor.
Start
Calibrating position sensor.
Start
View plots
OK
Cancel
Help

Figure 1.2: The PS600 Calibration Dialog

messages). To ’repair’ the error the user has to stop the swinging of the pendulum as good as possible before pressing the ’Retry’ button.

The third calibration step tries to calibrate the position sensor by moving the cart at first to the right limit switch and then to the left limit switch and using the corresponding sensor readings to determine the sensor signal mapping to the zero position of the cart. After switching on the servo amplifier for the drive the procedure is carried out in a loop which is terminated by reaching the left limit switch or after a time-out of 60 s. The first case is taken as a valid result indicated by moving the cart near to the middle position, a white background colour, a check mark replacing the ’Start’ button and an enabled "Ok" button to terminate the complete dialog. When the check box labelled ’View plots’ was marked, two graphics are presented on the screen showing the controller output as well as the measured cart position during this calibration step. A false result is 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 ’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. (-2) when the system identification signal was missing,

Disengagement does not respond. (-3) when the output stage release circuit failed to enable the servo amplifier,

Time-out during waiting for small angle. (-4) when within a total time of 120 seconds the amplitudes of the pendulum angle did not remain in a range of about 1 for a period of 4 seconds.

Time-out during driving the cart. (-6) when the cart could not be moved to the right limit switch and then to the left limit switch within 60 seconds.

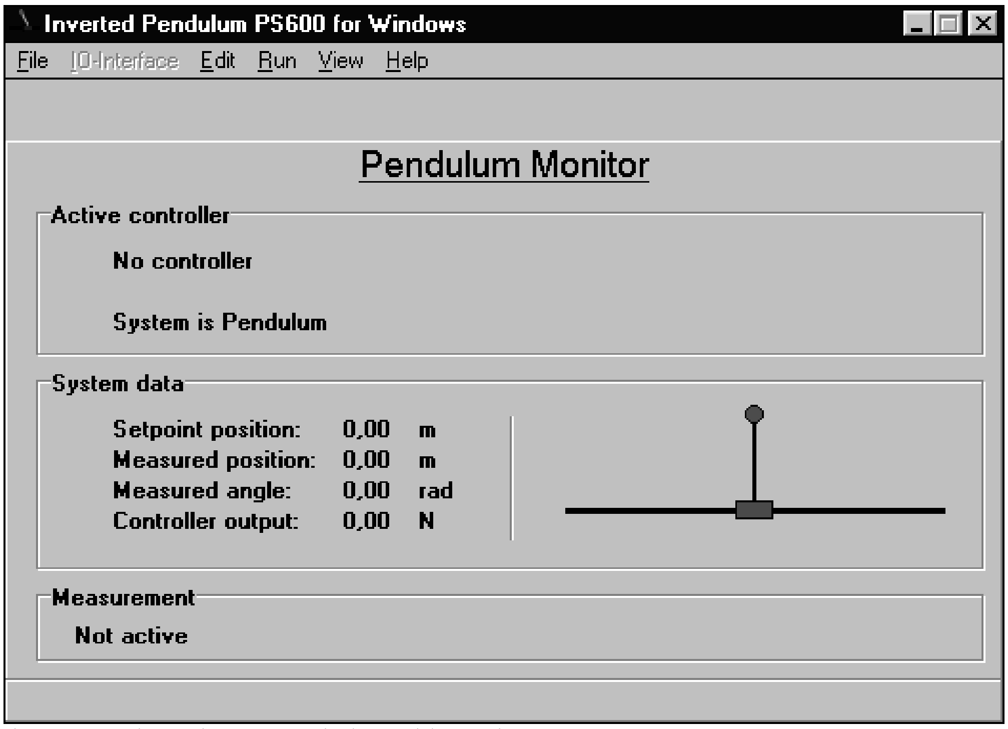
Cannot initialize IO-Interface. Select the correct card and address. (-12) when trying to open the driver for the given PC adapter card type and its address failed.

1.3 Main Window

Following a successful calibration the main window titled **Inverted Pendulum PS600 for Windows** appears on the screen as shown in figure 1.3. The first window row contains the main menu items. Its submenus are described in the following sections. The lower parts of the window **Inverted Pendulum Monitor** indicate the ’Active controller’, the ’System data’ as well as the state of the ’Measurement’.

The ’Active controller’ panel may display up to three lines, where the first line indicates the type of the current controller (No controller, State controller or Fuzzy controller). The second line may display the type of the disturbance (friction) compensation and for the state controller the method for obtaining the missing state variables (cart speed, angular velocity). The third line indicates the type of the controlled system (’Inverted Pendulum’, or ’Cart’). Selecting another controlled system means selecting another controller. In principle the hardware of the plant itself must not be changed but it is recommended to remove the pendulums when the system is changed from ’Inverted Pendulum’ to ’Cart’ because the swinging pendulums may disturb the controller performance of the controlled cart.

The ’System data’ panel contains the setpoint of the cart position as well as its measured value, the measured pendulum angle and the value of the controller output. The right half of this panel displays an animation picture of the currently selected system.


Inverted Pendulum PS600 for Windows
File IO-Interface Edit Run View Help

Pendulum Monitor

Active controller

No controller

System is Pendulum

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 PS600 window with monitor

The ’Measurement’ panel indicates the state of a measurement acquisition which is either ’Not active’ or its progress is visualized by a status bar.

1.4 Menu File

The pulldown menu **File** (see figure 1.4) provides functions for loading or saving of different files, to print plot windows as well as to terminate the program.

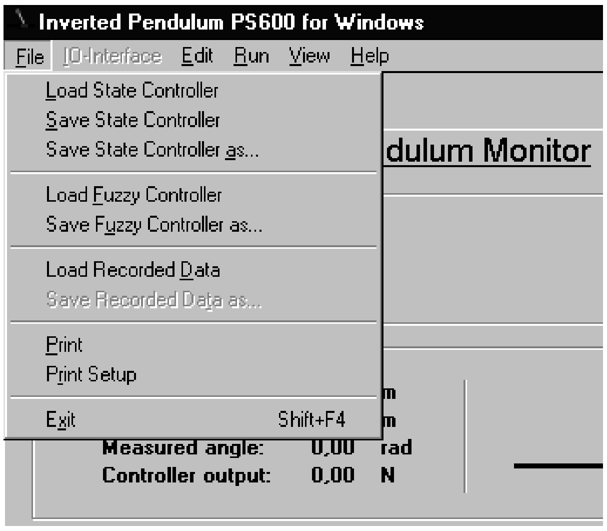


Figure 1.4: The sub menu ’File’

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 disk file. Destination file is the parameter file of the state controller, which was recently opened. Please notice that the file "PENDULUM.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 four file names of the fuzzy description files required to control the inverted pendulum system. 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 1.10.1 "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).

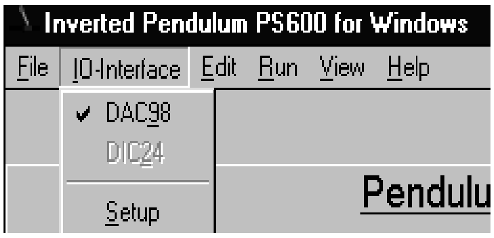
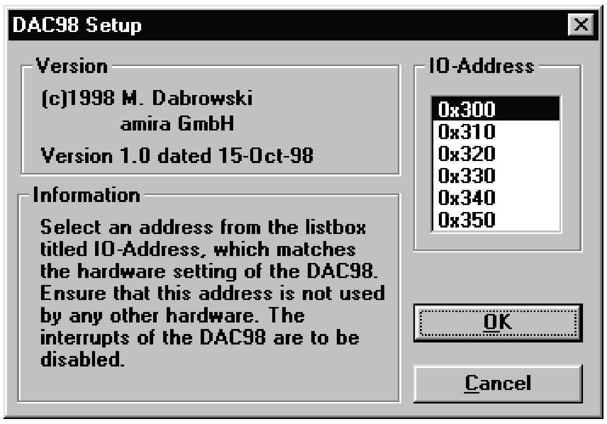


Figure 1.5a: The sub menu 'IO-Interface'

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 PENDW16.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 PENDW16.INI which is controlled by the program automatically. When this file is missing the default driver is always the DAC98.DRV.


DAC98 Setup

Version
(c)1998 M. Dabrowski
amira GmbH
Version 1.0 dated 15-Oct-98

Information
Select an address from the listbox titled IO-Address, which matches the hardware setting of the DAC98. Ensure that this address is not used by any other hardware. The interrupts of the DAC98 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 !

The selected address is stored automatically as a decimal number in a specific entry of the file SYSTEM.INI from the Windows directory. This entry may look like:

```
[DAC98]
Adress=768
```

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

Attention:
After activating one of the menu items mentioned above a new sensor calibration has to be carried-out. This is true even in case the settings were not changed. The reason for this is that the timer for the sampling period of any controller is stopped with each of these menu items.

1.6 Menu Edit

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

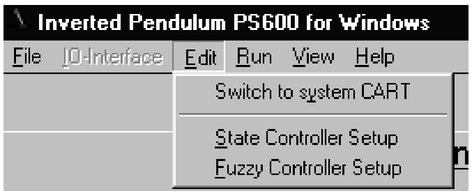
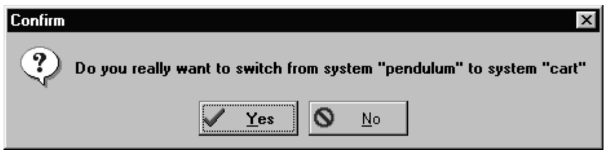

Inverted Pendulum PS600 for Windows
File IO-Interface Edit Run View Help
Switch to system CART
State Controller Setup
Fuzzy Controller Setup

Figure 1.6a: The sub menu 'Edit'

Switch to system CART: Selecting this menu item, which is enabled only for an inactive controller, will present a warning message (see figure 1.6b) 'Do you really want to switch from system "pendulum" to system "cart"'. A positive answer will change the control structure such that instead of the inverted pendulum only

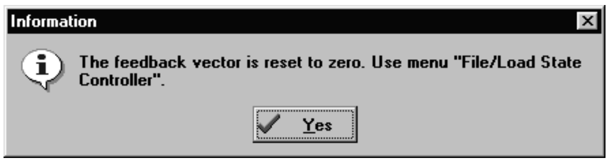


Confirm

Do you really want to switch from system "pendulum" to system "cart"

Figure 1.6b: The 'Select System Dialog'

the single cart is controllable. In that case an additional message (see figure 1.6c) informs the user that new parameters for the state controller are to be loaded by means of the menu item **File/Load State Controller**. Finally the original menu item is changed to **Switch to system PENDULUM** allowing for the corresponding controller change.



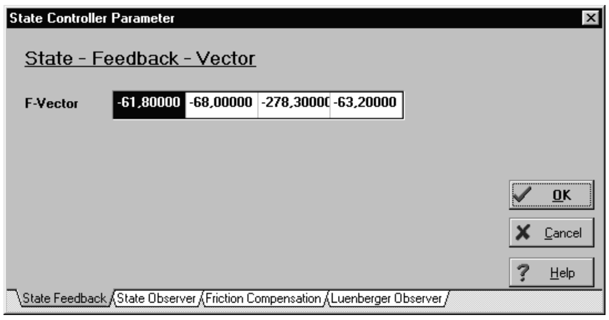
Information

The feedback vector is reset to zero. Use menu "File/Load State Controller".

Figure 1.6c: The message box for system change

Attention: It is recommended to remove the pendulum when the system is changed from "Inverted Pendulum" to "Cart" before starting the controller. Otherwise the uncontrolled pendulum will disturb the control behaviour. The fuzzy controller cannot be started for the system "Cart" but any fuzzy controller for the system "inverted Pendulum" remains in the memory without any changes.

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



State Controller Parameter

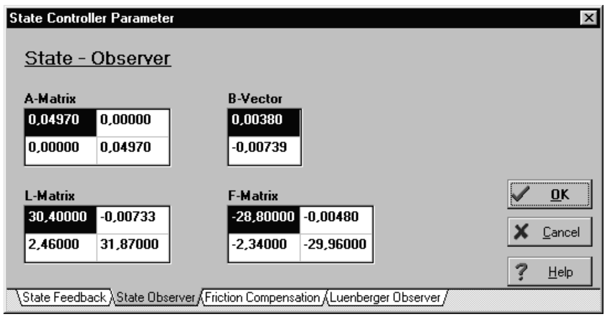
State - Feedback - Vector

F-Vector	-61,80000	-68,00000	-278,3000(	-63,20000

Figure 1.6d: The 'State Feedback' dialog

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

Selecting the second tab labelled **State Observer** allows for manipulating the A, L and F matrix or the B vector of the reduced-order observer to determine the missing state variables (see figure 1.6e).



State Controller Parameter

State - Observer

A-Matrix

0,04970	0,00000
0,00000	0,04970

B-Vector

0,00380
-0,00739

L-Matrix

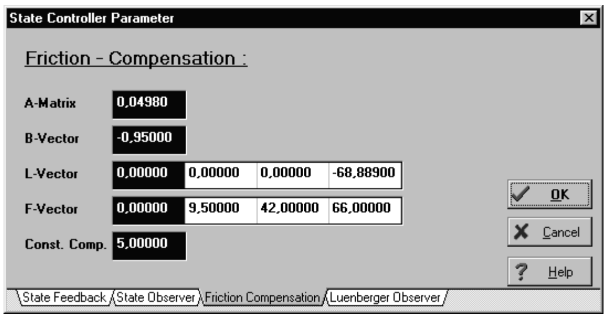
30,40000	-0,00733
2,46000	31,87000

F-Matrix

-28,80000	-0,00480
-2,34000	-29,96000

Figure 1.6e: The 'State Observer' dialog

The third tab provides the adjustment of the parameters (disturbance observer parameters or constant friction compensation parameter) used by the **Friction Compensation** (see figure 1.6f).



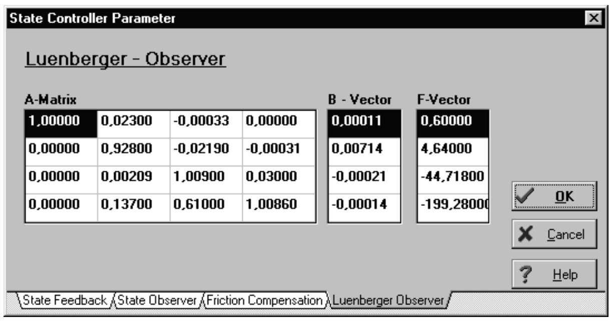
State Controller Parameter

Friction - Compensation :

A-Matrix	0,04980			
B-Vector	-0,95000			
L-Vector	0,00000	0,00000	0,00000	-68,88900
F-Vector	0,00000	9,50000	42,00000	66,00000
Const. Comp.	5,00000			

Figure 1.6f: The 'Friction Compensation' dialog

The fourth tab provides the adjustment of the **Luenberger Observer** matrices (see figure 1.6g).



State Controller Parameter

Luenberger - Observer

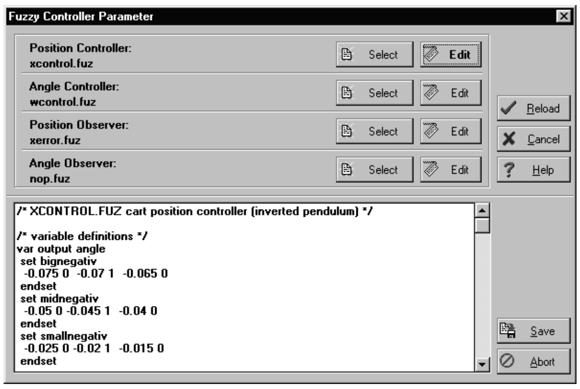
A-Matrix				B - Vector	F-Vector
1,00000	0,02300	-0,00033	0,00000	0,00011	0,60000
0,00000	0,92800	-0,02190	-0,00031	0,00714	4,64000
0,00000	0,00209	1,00900	0,03000	-0,00021	-44,71800
0,00000	0,13700	0,61000	1,00860	-0,00014	-199,2800

Figure 1.6g: The 'Luenberger Observer' dialog

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 Setup** 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 1.6h displays this dialog with an opened edit field for the fuzzy description file named XCONTROL.FUZ.



Fuzzy Controller Parameter

Position Controller:	xcontrol.fuz
Angle Controller:	wcontrol.fuz
Position Observer:	xerror.fuz
Angle Observer:	nop.fuz

```
/* XCONTROL.FUZ cart position controller (inverted pendulum) */

/* variable definitions */
var output angle
 set bignegativ
  -0.075 0  -0.07 1  -0.065 0
 endset
 set midnegativ
  -0.05 0 -0.045 1  -0.04 0
 endset
 set smallnegativ
  -0.025 0 -0.02 1  -0.015 0
 endset
```

Figure 1.6h: The 'Fuzzy Controller Parameter' dialog

The edit field itself is closed either by the button 'Save' which stores the content of the edit field to the file or by 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 identify parts of the system, to calibrate sensors, to record measurements and to adjust the setpoint. Any controller may be started only when the accompanying parameter file could be loaded previously with success and when the previous sensor calibration could be carried-out with success. An active controller is indicated by a check mark left to the menu item.

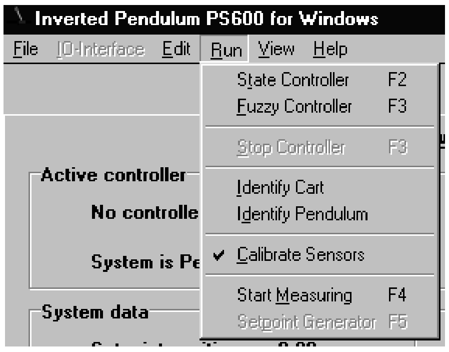
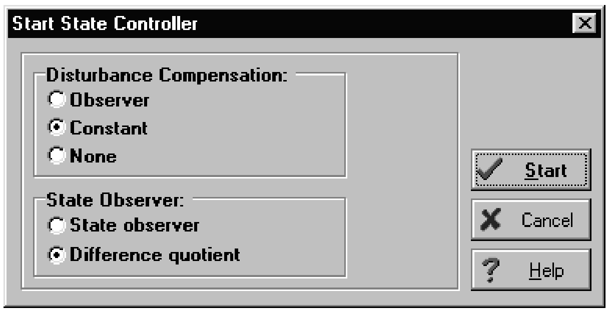


Figure 1.7a: The sub menu ’Run’

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.


Start State Controller

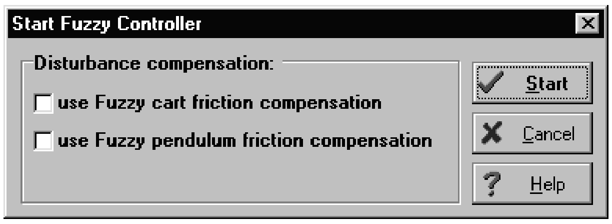
Disturbance Compensation: Observer [✓ Constant] None

State Observer: State observer [✓ Difference quotient]

Figure 1.7b: The ’Start State Controller’ dialog

When a controller was not running previously and when the pendulum is to be controlled the user is asked to move the pendulum into an upright position.

Fuzzy Controller: Starts the fuzzy controller defined by its fuzzy description files (*.FBW) only, when an inverted pendulum is to be controlled. A dialog window is opened


Start Fuzzy Controller

Disturbance compensation:

☐ use Fuzzy cart friction compensation

☐ use Fuzzy pendulum friction compensation

Figure 1.7c: The ’Start Fuzzy Controller’ dialog

automatically to configure the disturbance compensation (and the observers for the inverted pendulum) before the controller is active.

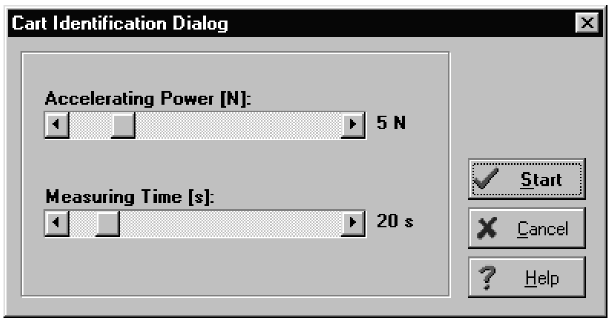
As for the state controller the user is asked to move the pendulum into an upright position if necessary.

Note: It is possible to switch directly from the state controller to the fuzzy controller and vice versa even for a running setpoint generator but the pendulum control is disturbed more or less depending on the computing power of the PC.

Stop Controller: Stops the selected controller and disables the menu item Setpoint Generator.

Note: Any controller may be switched off automatically when the servo amplifier is disabled, when the cart is moved to the right or left limit switch or when a pendulum angle exceeds a range of about 10 degrees.

Identify Cart: Records the step response of the system "Cart". For this purpose the desired actuating power (from 2 to 20 [N]) as well as the measuring time (from 5 to 150 [s]) are determined by means of a dialog (see figure 1.7d). The measurement is carried-out after pressing the ’Start’ button.


Cart Identification Dialog

Accelerating Power [N]: 5 N

Measuring Time [s]: 20 s

Figure 1.7d: The ’Cart Identification Dialog’

Attention: For cart movement the input actuating power should be greater than the static friction of the cart (typical values are 5 to 10 N as can be seen from the plots obtained from the calibration procedure).

The measurement is terminated if one of the limit switches will be actuated or the maximum measuring time will be exceeded. As a measuring result the cart position, in dependence on the time, is displayed on the screen.

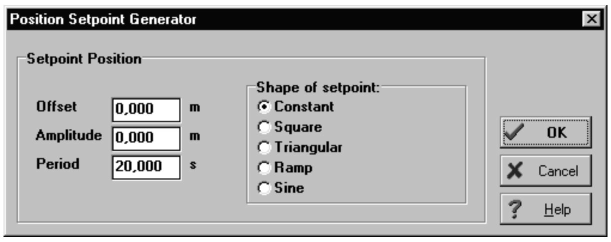
Identify Pendulum: Is used for the identification of the pendulum extension kit. For this purpose the pendulum mechanics is moved once. Afterwards the angle of the pendulum is measured within the indicated period. The actuating power as well as the measuring time are determined by means of a dialog as it was used to identify the cart (see figure 1.7d). At the end of the automatic measurement the resulting angle of the pendulum is displayed on the screen.

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.

Start Measuring opens a window to adjust the measuring time and to assign trigger conditions to start recording the measurements. Figure 1.7e shows this window. 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.

The adjustment of the setpoint, **Setpoint Generator**, for the cart position is handled by means of the dialog window shown in figure 1.7f.


Position Setpoint Generator

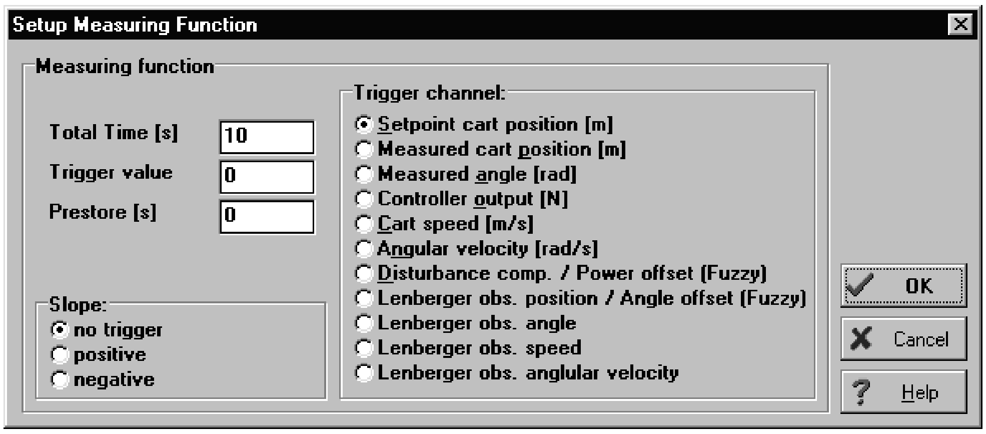
Setpoint Position

Offset 0,000 m
Amplitude 0,000 m
Period 20,000 s

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

Figure 1.7f: The 'Position Setpoint Generator' dialog

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.


Setup Measuring Function

Measuring function

Total Time [s] 10
Trigger value 0
Prestore [s] 0

Slope: [✓ no trigger] positive negative

Trigger channel: [✓ Setpoint cart position [m]] Measured cart position [m] Measured angle [rad] Controller output [N] Cart speed [m/s] Angular velocity [rad/s] Disturbance comp. / Power offset (Fuzzy) Lenberger obs. position / Angle offset (Fuzzy) Lenberger obs. angle Lenberger obs. speed Lenberger obs. anglular velocity

Figure 1.7e: The 'Setup Measuring Function' 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.

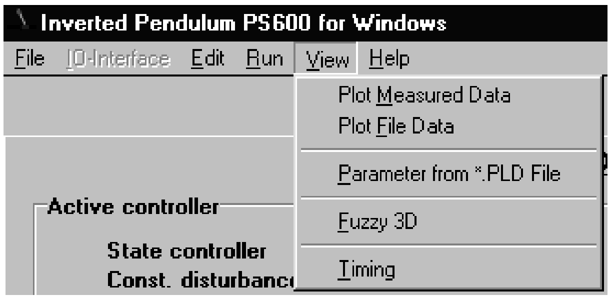


Figure 1.8a: The sub menu 'View'

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.

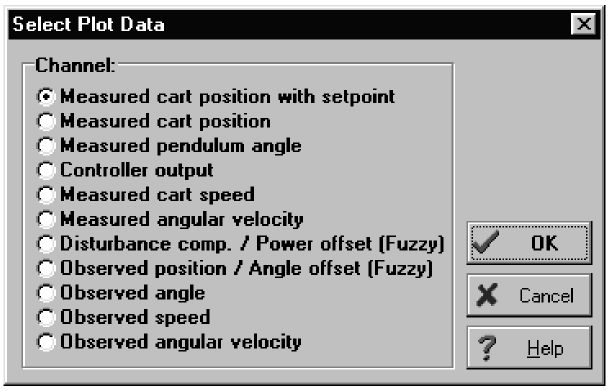

Select Plot Data
Channel:
Measured cart position with setpoint
Measured cart position
Measured pendulum angle
Controller output
Measured cart speed
Measured angular velocity
Disturbance comp. / Power offset (Fuzzy)
Observed position / Angle offset (Fuzzy)
Observed angle
Observed speed
Observed angular velocity
OK
Cancel
Help

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.

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.

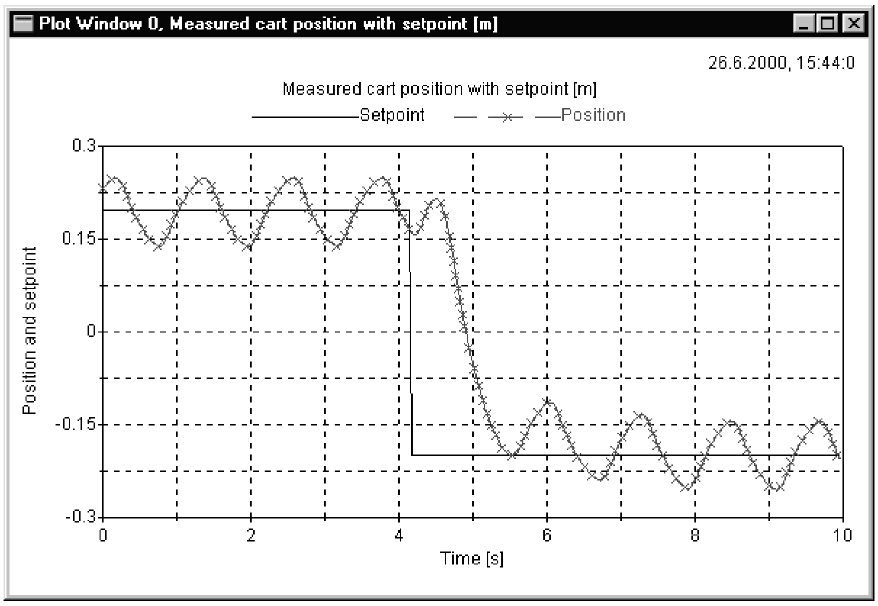


Figure 1.8c: Example of a 'Plot Window'

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).

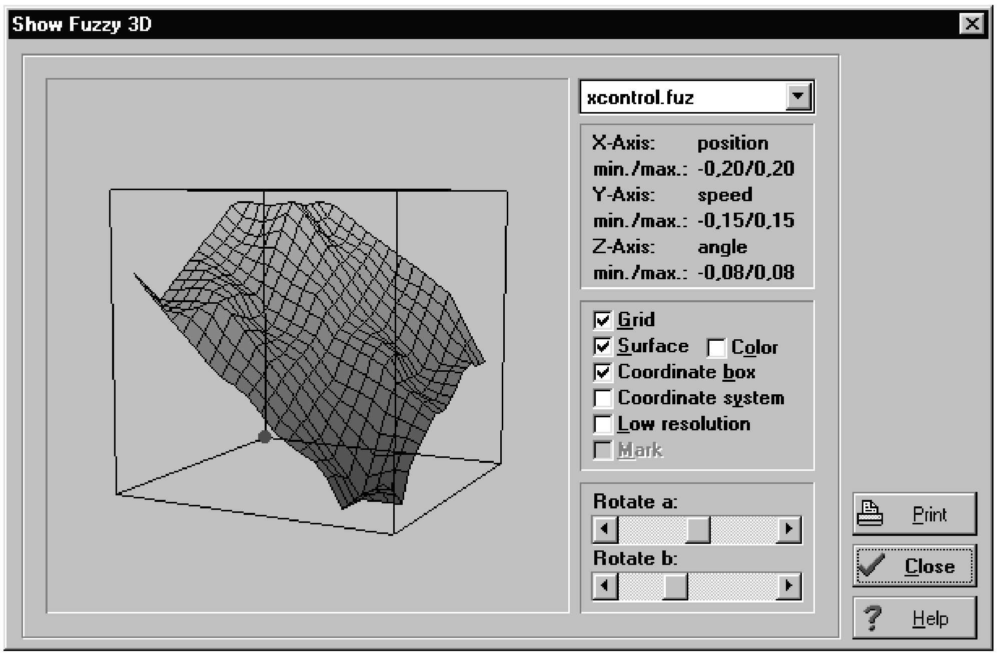
The margins of the cube are displayed only when 'Coordinate box' is marked. The same is valid for the three axes depending on the setting of 'Coordinate system'.

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 correspond 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'.



X-Axis: position
min./max.: -0,20/0,20
Y-Axis: speed
min./max.: -0,15/0,15
Z-Axis: angle
min./max.: -0,08/0,08

[✓] Grid
[✓] Surface [ ] Color
[✓] Coordinate box
[ ] Coordinate system
[ ] Low resolution
[ ] Mark

Rotate a:
Rotate b:

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

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 sampling period. While its minimum value is normally close to the nominal value of 15 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 mean time.

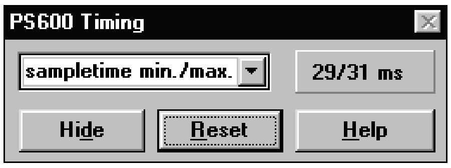


Figure 1.8e: The ’PS600 Timing’ dialog

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.

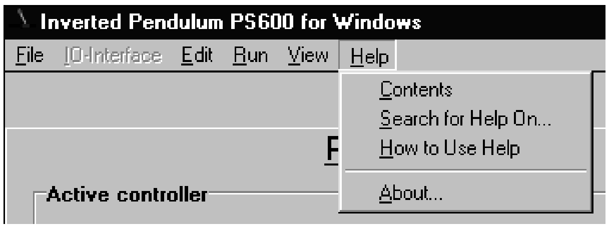
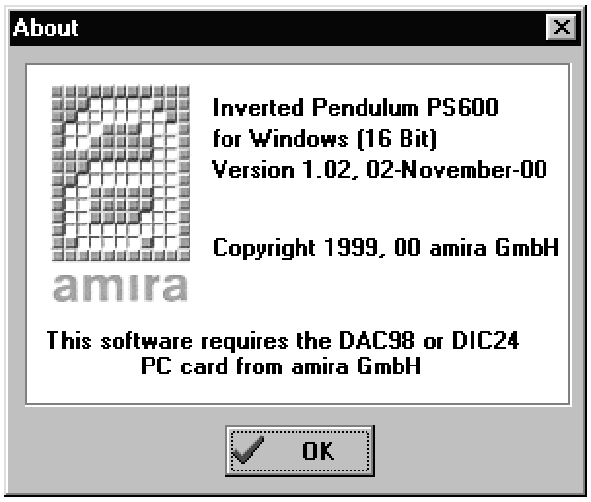


Figure 1.9a: The sub menu ’Help’

The menu item **Contents** displays the contents of the help file PS6PEND.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

Inverted Pendulum PS600
for Windows (16 Bit)
Version 1.02, 02-November-00

Copyright 1999, 00 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 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
```

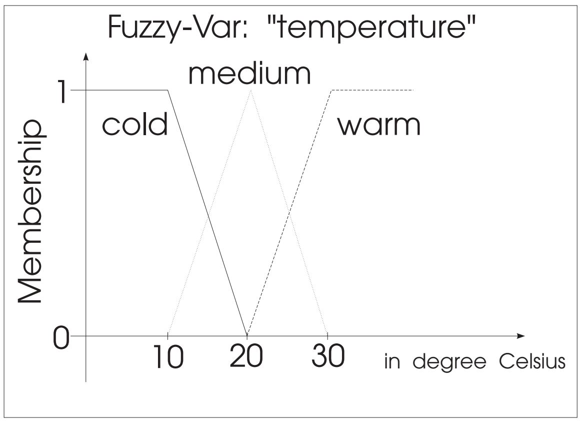


Figure 1.9: The fuzzy variable 'temperature'

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 preprocessor.
/*	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.

1.10.2 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>isalreadydefined.
<n>Errorsdetected.

1.10.3 Format of the Fuzzy Controller File for the Laboratory Experiment PS600 (*.FBW)

The fuzzy controller file PENDULUM.FBW for the PS600 Inverted Pendulum contains four file names of fuzzy description files required for the fuzzy controller. Each file name begins in a new line, comments or empty lines are not allowed. Please change this file only using corresponding functions of the PS600 controller software.

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

```
XCONTROL.FUZ
WCONTROL.FUZ
XERROR.FUZ
NOP.FUZ
```

1.10.4 Format of the State Controller File for the Laboratory Experiment PS600 (*.STA)

This file contains all parameters of the state controller as well as the corresponding observers for the laboratory experiment PS600. 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 PS600 controller software.

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

```
[Sampling Period]
0.03
[Feedback Vector]
-61.8 -68 -278.3 -63.2
[Observer Transformation Vector (l)]
30.4 -0.00733 2.46 31.87
[Observer System Matrix (a)]
0.0497 0 0 0.0497
[Observer Feedback Matrix (f)]
-28.8 -0.0048 -2.34 -29.96
[Observer Control Vector (b)]
0.0038 -0.00739
[Disturbance Observer Transformation Vector (l)]
0 0 0 -68.889
[Disturbance Observer System Matrix (a)]
0.0498
[Disturbance Observer Feedback Matrix (f)]
0 9.5 42 66
[Disturbance Observer Control Vector (b)]
-0.95
[Constant Disturbance Compensation]
5
[Luenberger Observer System Matrix (a)]
1 0.023 -0.000333 -3.04e-06 0 0.928 -0.0219 -0.000313
0 0.00209 1.009 3.01e-05 0 0.137 0.61 1.0086
[Luenberger Observer Feedback Matrix (f)]
0.6 4.64 -44.718 -199.28
[Luenberger Observer Control Vector (b)]
0.000108 0.00714 -0.000209 -0.0001379
```

```
[Settings]
2 2
```

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

```
[Sampling Period]
0.03
[Feedback Vector]
36.17 13.3 0 0
[Observer Transformation Vector (l)]
30.4 0 0 0
[Observer System Matrix (a)]
0.0497 0 0 0
[Observer Feedback Matrix (f)]
-28.8 0 0 0
[Observer Control Vector (b)]
0.00389 0
[Disturbance Observer Transformation Vector (l)]
0 131.5 0 0
[Disturbance Observer System Matrix (a)]
0.0498
[Disturbance Observer Feedback Matrix (f)]
0 -115.46 0 0
[Disturbance Observer Control Vector (b)]
-0.95
[Constant Disturbance Compensation]
5
[Luenberger Observer System Matrix (a)]
1 0.0289 0 0.9277 0 0 0 0 0 0 0 0 0 0 0 0
[Luenberger Observer Feedback Matrix (f)]
0.83 4.96 0 0
[Luenberger Observer Control Vector (b)]
0.000109 0.00722 0 0
[Settings]
2 2
```

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. (60 bytes)
The structure CTRLSTATUS. (96 bytes)
The structure DATASTRUCT. (8 bytes)
The data array with float values (4 bytes per value).

The size of the data array is defined in the structure DATASTRUCT. With the PS600 Inverted Pendulum the number of the stored channels is always 11 (the length of the measurement vector is 11, i.e. equal to 44 bytes). When the state controller was active during the measuring the vector contains the following signals (estimated values from Luenberger observer):

the position setpoint	in [m],
the measured position	in [m],
the measured angle	in [rad],
the control force	in [N],
the measured cart speed	in [m/s],
the angular velocity	in [rad/s],
the friction compensation	in [N],
the estimated position	in [m],
the estimated speed	in [m/s],
the estimated angle	in [rad],
the estimated angular velocity	in [rad/s].

When the fuzzy controller was active during the measuring the vector contains the following signals:

the position setpoint	in [m],
the measured position	in [m],
the measured angle	in [rad],
the control force	in [N],
the measured cart speed	in [m/s],
the angular velocity	in [rad/s],
the cart friction compensation	in [N],
the pendulum friction comp.	in [rad],
a dummy zero signal,	
a dummy zero signal,	
a dummy zero signal.	

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.

1.11 The DEMO Version

The demo version of the program **PENDW16.EXE** is indicated by the title "Inverted Pendulum Monitor (Demo-Version)" in the monitor window. It operates with a 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 select the IO-interface and to control the calibration 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.

PC Plug-In Card DAC98

Date: 09-February-1998

1 The PC Plug-in Card DAC98 (PCA902) 1-1

1.1 Introduction 1-1
1.2 Features 1-1
1.3 Specifications 1-1
1.4 Installation of the DAC98 1-1
1.4.1 Adjustment of the Base Address 1-1
1.4.2 Adjustment of the Interrupt Channel 1-2
1.4.3 The Operation Modes of the 16 Bit Timer/Counter 1-3
1.4.4 Adjustment of the Analog Output Signal Range 1-3
1.4.5 Pin-Reservations of the DAC98 1-3
1.4.6 Installation of the Card in the PC 1-5
1.5 Programming of the DAC98 1-5
1.5.1 The Registers of the DAC98 1-5
1.5.2 Configuration of the DAC98 1-6
1.5.3 Reading the Identification String 1-6
1.5.4 A/D Conversion 1-6
1.5.5 D/A Conversion 1-6
1.5.6 Programming the Timer Chip 8253 1-7
1.5.7 Reading the Digital Inputs 1-7
1.5.8 Setting the Digital Outputs 1-7
1.5.9 Internal Digital Functions 1-8
1.5.10 Reading an Incremental Encoder Input Channel 1-8
1.5.11 Interrupt / Clock Signal 1-8

2 DAC98 Adapter Card 2-1

2.1 Adapter Card EXPO-DAC98 (Opt. 902-01) 2-1

3 Operating Instructions for the Test Program 3-1

3.1 Installation 3-1

3.2 Program Start 3-1

3.3 Menu ’File’ 3-1

3.4 Menu ’IO-Interface’ 3-2

3.5 Menu ’Test’ 3-3

3.6 Menu ’Measure’ 3-4

3.7 Menu ’Help’ 3-4

4 Source Files of the DAC98 Driver 4-1

4.1 The Class DAC98 4-1

DAC98 4-2

GetAdress 4-2

SetAdress 4-3

GetInterrupt 4-3

SetInterrupt 4-4

Identifikation 4-4

Init 4-5

Exit 4-5

Setup 4-6

SetClock 4-6

GetClock 4-7

WriteDigital 4-7

WriteAllDigital 4-8

ReadDigital 4-8

ReadAllDigital 4-9

SetCounter 4-9

GetCounter 4-10

WaitCounter 4-10

TestCounterJMP 4-11

GateCounter 4-11
SetTimer 4-12
SetTimer 4-12
GetTimer 4-13
GateTimer 4-13
SetINT 4-14
GetINT 4-14
ResetDDM 4-15
ResetAllDDM 4-15
ReadDDM 4-16
ReadAllDDM 4-16
ReadAnalogInt 4-17
ReadAnalogVolt 4-17
WriteAnalogInt 4-18
WriteAnalogVolt 4-18
ReadDigitalInputs 4-19
4.2 The Class DIC 4-19
DIC 4-20
~DIC 4-20
GetDigitalOut 4-21
GetAnalogOut 4-21
GetDDMCounter 4-22
GetDDMTimer 4-22
GetDDMStatus 4-23
FilterINC 4-23
TimerDirINC 4-24
SetINT 4-24
SetTimer 4-25
GetTimer 4-25

GateTimer 4-26

4.3 The Class PCIO 4-26

PCIO 4-27

~PCIO 4-27

DigitalOutStatus 4-28

IsPCIO 4-28

ReadAnalogVoltMean 4-29

ResetHCTL 4-29

ReadHCTL 4-30

SetINT 4-30

5 Windows Drivers for DAC98, DAC6214 and DIC24 5-1

OpenDriver 5-1

SendDriverMessage 5-2

CloseDriver 5-4

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.

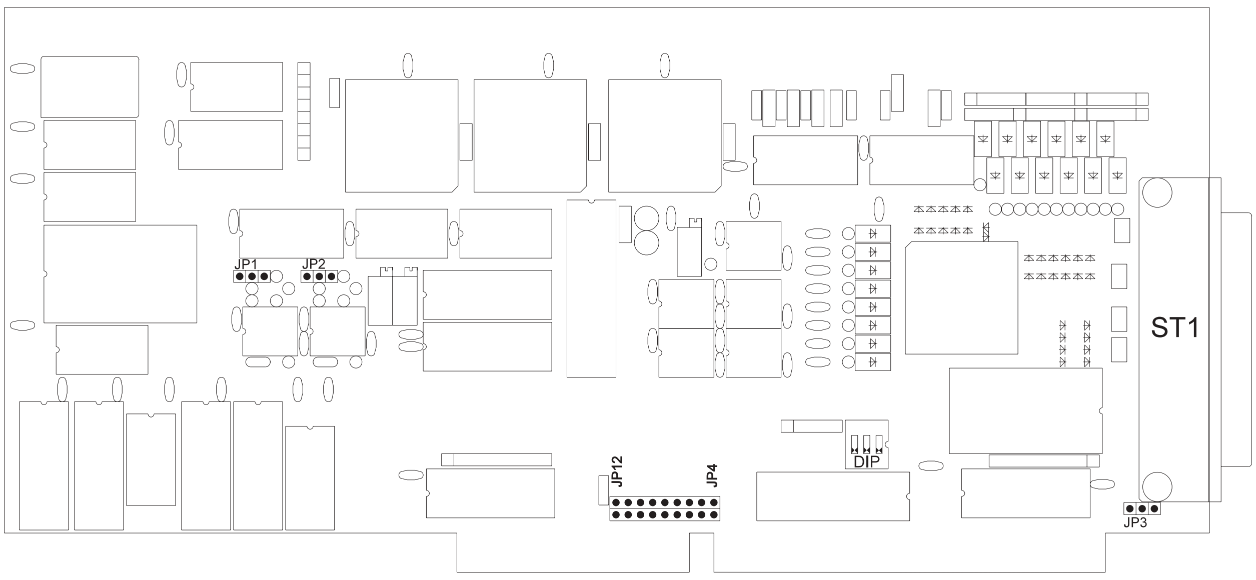
I/O Address (Hex)	3	2	1
300(*)	1	1	1
310	0	1	1
320	1	0	1
330	0	0	1
340	1	1	0
350	0	1	0
360	1	0	0
370	0	0	0

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

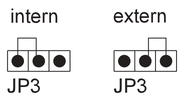
The PC hardware may be damaged when more than one jumper is installed or in case the selected interrupt channel is in use by another card.



Jumpers of the DAC98

1.4.3 The Operation Modes of the 16 Bit Timer/Counter

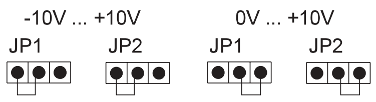
The 16 bit counter either counts external events or it counts the timer clock. The jumper JP3 adjusts the corresponding operation mode.



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.

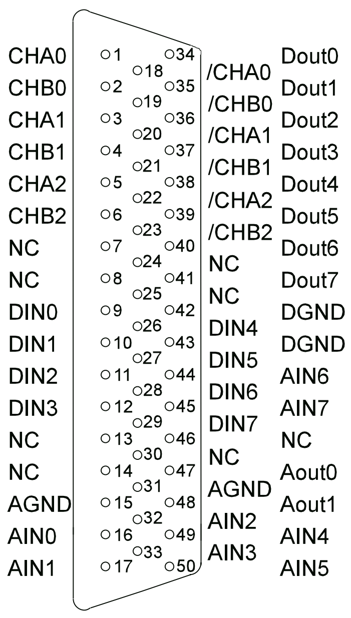


1.4.5 Pin-Reservations of the DAC98

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

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

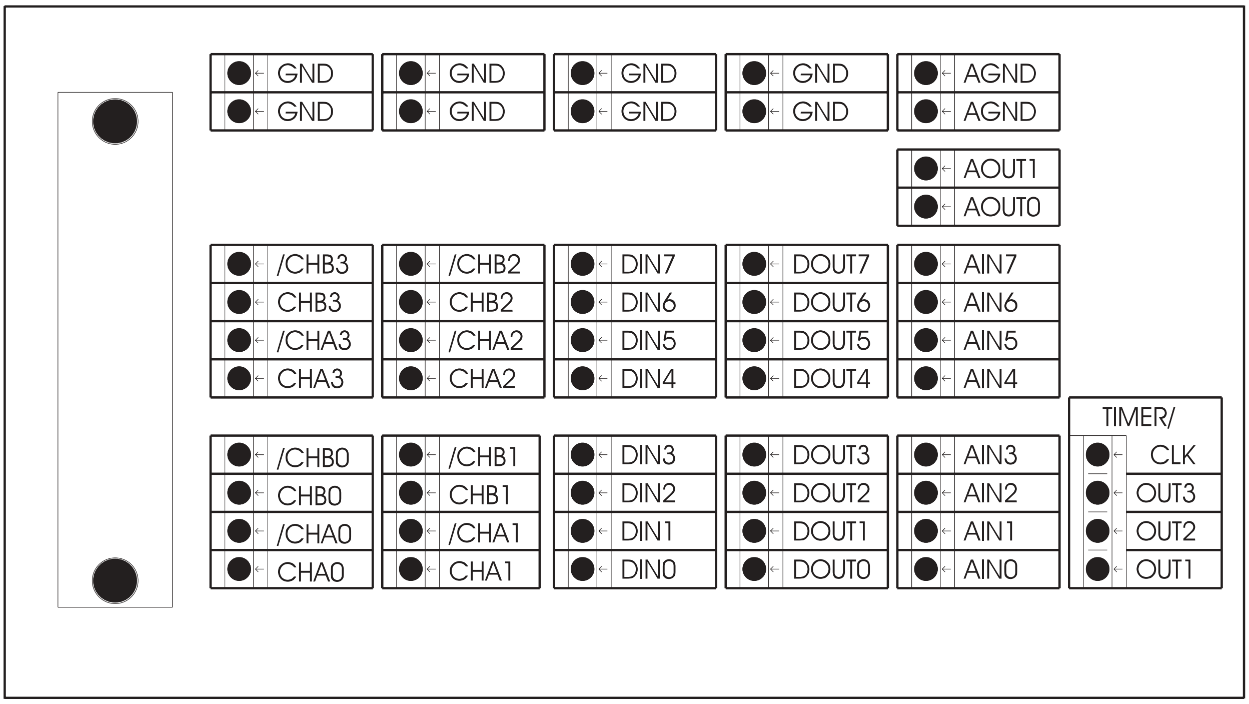
Note: To maintain any selected clock the corresponding bits have to be the same in following write operations to the interrupt / clock signal register.

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.

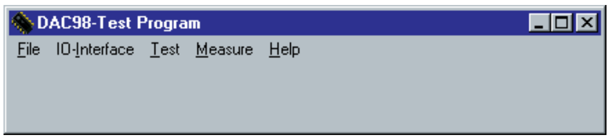


Figure 3.1: The main window of the DAC98 test software

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.

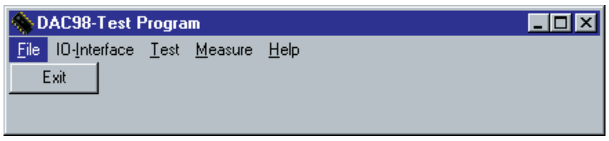
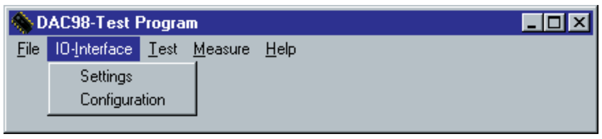


Figure 3.2: The menu ’File’

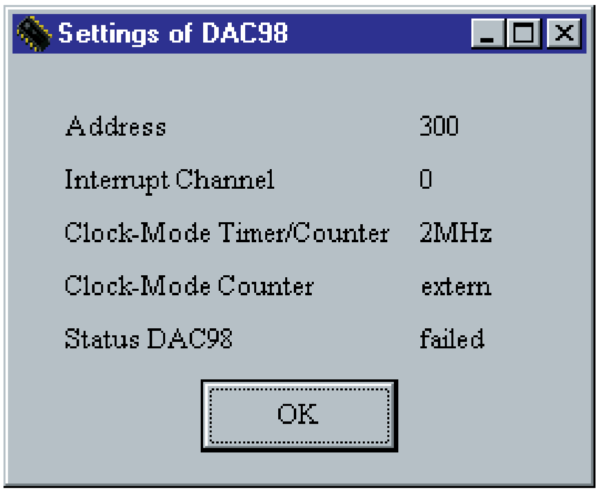
3.4 Menu ’IO-Interface’

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.


Settings of DAC98

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

Figure 3.4: The window ’Settings’

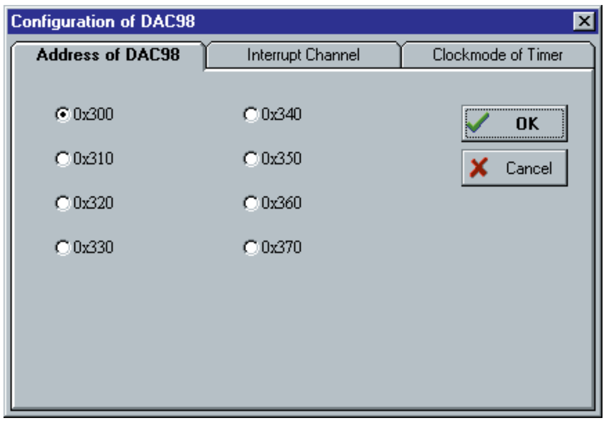
The function ’Configuration’ opens a window containing selectable dialogs as shown in figure 3.5.

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!


Configuration of DAC98

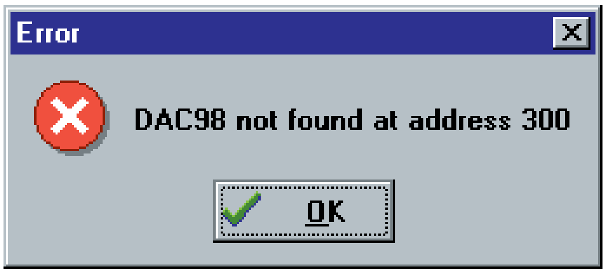
Address of DAC98 | Interrupt Channel | Clockmode of Timer

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


Error

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.

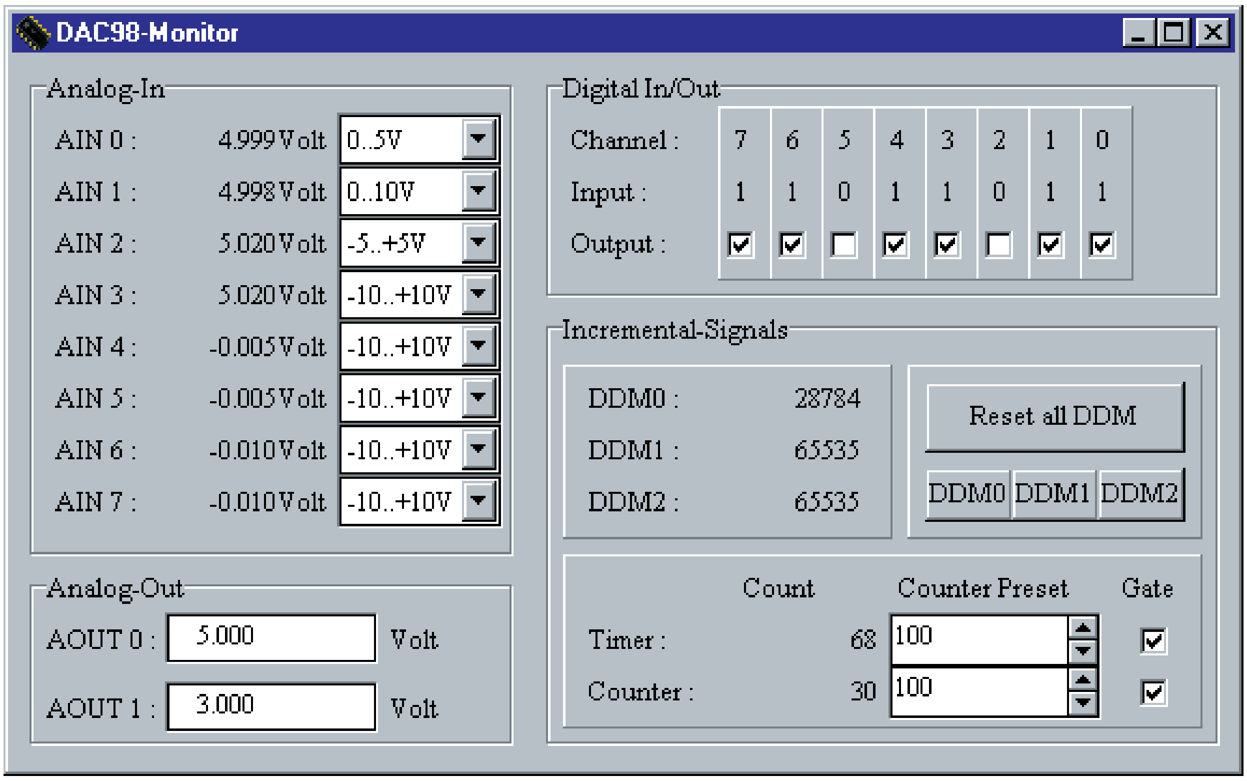
The input voltage range is selectable for each analog input left to the displayed measured value.

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

Reset all DDM

DDM0 DDM1 DDM2

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

Figure 3.7: The window ’Showall’

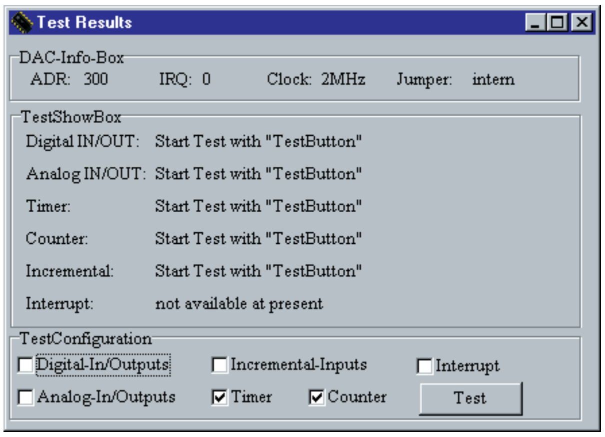
The menu item ’Test’ opens a window as shown in figure 3.8.

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.

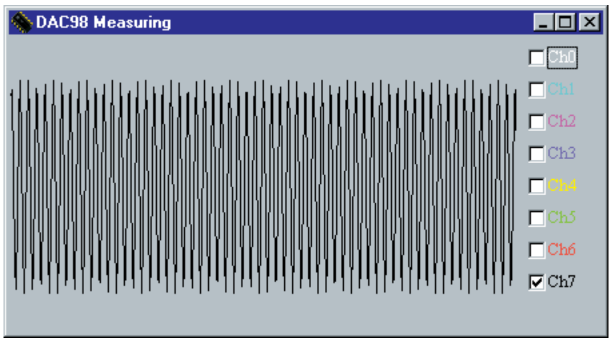
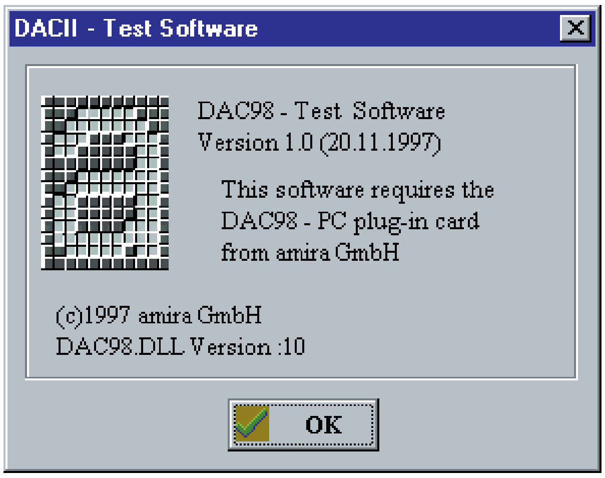


Figure 3.9: The window ’Measure’

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. Each driver may be opened only once meaning that only one PC adapter card may be handled by this driver. 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.

The address of the PC adapter card handled by this driver is read from a specific entry of the file SYSTEM.INI from the public Windows directory. When this entry is missing the default address 0x300 (=768 decimal) will be taken.

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 and stores it to SYSTEM.INI, i. e. [DAC98] Adress=768
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**.

PS600 Inverted Pendulum

Windows Software V1.0

Printed: 16. October 2000

1 Source Files of the PENDW16 Controller Program 1-1

1.1 General 1-1

1.2 Global Data and Functions 1-2

1.3 Dialogs and Windows of the Desktop 1-5

TMainForm.ShowHint 1-11

TMainForm.FormCreate 1-11

TMainForm.FormShow 1-11

TMainForm.FormClose 1-12

TMainForm.FormDestroy 1-12

TMainForm.FileMenuClick 1-12

TMainForm.OpenStateClick 1-12

TMainForm.SaveStateClick 1-13

TMainForm.SaveStateasClick 1-13

TMainForm.OpenFuzzyClick 1-13

TMainForm.SaveFuzzyasClick 1-14

TMainForm.LoadPlotData1Click 1-14

TMainForm.SavePlot1Click 1-14

TMainForm.Print1Click 1-15

TMainForm.PrintSetup1Click 1-15

TMainForm.ExitItemClick 1-15

TMainForm.IOInterface1Click 1-15

TMainForm.DAC98Click 1-16

TMainForm.DIC24Click 1-16

TMainForm.DACSetupClick 1-16

TMainForm.System1Click 1-17

TMainForm.StateControllerSetupClick 1-17

TMainForm.FuzzyControllerSetupClick 1-17

TMainForm.Run1Click 1-18

TMainForm.StateController1Click 1-18
TMainForm.FuzzyController2Click 1-18
TMainForm.StopController1Click 1-19
TMainForm.IdentifyCart1Click 1-19
TMainForm.IdentifyPendulum1Click 1-19
TMainForm.CalibrateSensors1Click 1-20
TMainForm.StartMeasuring1Click 1-20
TMainForm.SetpointGenerator1Click 1-20
TMainForm.View1Click 1-20
TMainForm.PlotMeasuredData1Click 1-21
TMainForm.PlotFileData1Click 1-21
TMainForm.ParametersfromPLDFile1Click 1-21
TMainForm.Fuzzy3D1Click 1-21
TMainForm.Timing1Click 1-22
TMainForm.Contents1Click 1-22
TMainForm.SearchforHelpon1Click 1-22
TMainForm.HowtoUseHelp1Click 1-22
TMainForm.About1Click 1-23
TMainForm.Timer1Timer 1-23
TMainForm.PaintBox1Click 1-23
TMainForm.MeasLabelClick 1-23
TSingleInstance.WndProc 1-24
TAboutBox.FormShow 1-24
TCalibrateDlg.TopBBtnClick 1-24
TCalibrateDlg.MiddleBBtnClick 1-25
TCalibrateDlg.BottomBBtnClick 1-25
TCalibrateDlg.FormShow 1-25
TCalibrateDlg.HelpBtnClick 1-26
TShow3DFuzDlg.rescale 1-26

TShow3DFuzDlg.recalc 1-26

TShow3DFuzDlg.calcrot 1-26

TShow3DFuzDlg.calctrans 1-27

TShow3DFuzDlg.FormShow 1-27

TShow3DFuzDlg.FormHide 1-28

TShow3DFuzDlg.FuzzyCBoxChange 1-28

TShow3DFuzDlg.DrawSquare 1-29

TShow3DFuzDlg.DrawCoors 1-29

TShow3DFuzDlg.DrawCoors2 1-29

TShow3DFuzDlg.DrawMark 1-30

TShow3DFuzDlg.PaintBoxPaint 1-30

TShow3DFuzDlg.ScrollBar1Change 1-30

TShow3DFuzDlg.ScrollBar2Change 1-31

TShow3DFuzDlg.PaintBoxMouseDown 1-31

TShow3DFuzDlg.PaintBoxMouseMove 1-31

TShow3DFuzDlg.PaintBoxMouseUp 1-31

TShow3DFuzDlg.KoorsCBoxClick 1-32

TShow3DFuzDlg.Koors2CBoxClick 1-32

TShow3DFuzDlg.ColorCBoxClick 1-32

TShow3DFuzDlg.LowResCBoxClick 1-32

TShow3DFuzDlg.PrintBBtnClick 1-33

TShow3DFuzDlg.MarkCBoxClick 1-33

TShow3DFuzDlg.Timer1Timer 1-33

TShow3DFuzDlg.HelpBtnClick 1-33

TFuzzyParameterDlg.Big 1-34

TFuzzyParameterDlg.Small 1-34

TFuzzyParameterDlg.FormCreate 1-34

TFuzzyParameterDlg.FormDestroy 1-34

TFuzzyParameterDlg.FormShow 1-35

TFuzzyParameterDlg.Sel1BBtnClick 1-35
TFuzzyParameterDlg.Ed1BBtnClick 1-35
TFuzzyParameterDlg.CancelEdBBtnClick 1-36
TFuzzyParameterDlg.SaveEdBBtnClick 1-36
TFuzzyParameterDlg.OKBtnClick 1-36
TFuzzyParameterDlg.HelpBtnClick 1-37
TIdCartDlg.ScrollBar1Change 1-37
TIdCartDlg.OKBtnClick 1-37
TIdCartDlg.FormShow 1-38
TIdCartDlg.HelpBtnClick 1-38
TMeasureDlg.OKBtnClick 1-38
TMeasureDlg.HelpBtnClick 1-39
TPLDInfoDlg.FormShow 1-39
TPLDInfoDlg.HelpBtnClick 1-39
TPlotDlg.OKBtnClick 1-40
TPlotDlg.HelpBtnClick 1-40
TPrintPlotDlg.PrinterBitBtnClick 1-40
TPrintPlotDlg.OKBtnClick 1-41
TPrintPlotDlg.FormShow 1-41
TPrintPlotDlg.HelpBtnClick 1-41
TGeneratorDlg.OKBtnClick 1-42
TGeneratorDlg.FormShow 1-42
TGeneratorDlg.HelpBtnClick 1-42
TStartPendulumDlg.Timer1Timer 1-43
TStateParameterDlg.FormCreate 1-43
TStateParameterDlg.TabSetClick 1-43
TStateParameterDlg.FormShow 1-44
TStateParameterDlg.OKBtnClick 1-44
TStateParameterDlg.HelpBtnClick 1-44

TStartFuzzDlg.OKBtnClick 1-45
TStartFuzzDlg.HelpBtnClick 1-45
TStartStateDlg.OKBtnClick 1-45
TStartStateDlg.HelpBtnClick 1-45
TTimingForm.UpdateData 1-46
TTimingForm.ResetButtonClick 1-46
TTimingForm.FormShow 1-46
TTimingForm.HideButtonClick 1-46
TTimingForm.HelpButtonClick 1-47
FloatToStr2 1-47
FloatToStr3 1-47
FloatToStr4 1-47
FloatToStr5 1-48
StrToFloatMinMax 1-48
StrToFloatStrMinMax 1-48
MinMaxi 1-49
DetectNT 1-49
1.4 Overview of Classes and DLL Interfaces 1-50
1.5 References of the DLL Interfaces 1-53
1.5.1 The DLL Interface PENSRV16 1-55
DoService 1-55
SetParameter 1-55
GetParameter 1-56
GetData 1-56
LockMemory 1-57
SetDriverHandle 1-57
ReadFuzzy 1-57
ReadStatePar 1-58
WriteStatePar 1-58

IsDemo 1-58
CalibrateSen 1-59
RawSensor 1-59
MeasureStart 1-60
MeasureLevel 1-60
MeasureStatus 1-60
OpenFuzzy3D 1-61
CloseFuzzy3D 1-61
InfoFuzzy3D 1-61
CalcFuzzy3D 1-62
1.5.2 The Class PS600STA in the PENSRV16.DLL 1-63
PS600STA::PS600STA() 1-64
PS600STA::Calc 1-64
PS600STA::SetTa 1-65
PS600STA::Reset 1-65
PS600STA::Load 1-65
PS600STA::Save 1-66
PS600STA::SetOrder 1-66
PS600STA::GetOrder 1-66
PS600STA::SetStateObserver 1-66
PS600STA::SetDistObserver 1-67
PS600STA::geterrors 1-67
PS600STA::GetFt 1-67
PS600STA::GetLBD 1-67
PS600STA::GetABD 1-68
PS600STA::GetFBD 1-68
PS600STA::GetBBD 1-68
PS600STA::GetDCON 1-68
PS600STA::GetDLBD 1-68

PS600STA::GetDABD 1-69
PS600STA::GetDFBD 1-69
PS600STA::GetDBBD 1-69
PS600STA::GetLA 1-69
PS600STA::GetLF 1-69
PS600STA::GetLB 1-70
1.5.3 The Class PS600FUZ in the PENSRV16.DLL 1-71
PS600FUZ::PS600FUZ() 1-71
PS600FUZ::~PS600FUZ() 1-72
PS600FUZ::Calc 1-72
PS600FUZ::Load 1-73
PS600FUZ::Save 1-73
PS600FUZ::SetAngObserver 1-73
PS600FUZ::SetPosObserver 1-74
PS600FUZ::getname 1-74
PS600FUZ::getfname 1-74
PS600FUZ::geterrors 1-74
1.5.4 The Class STOREBUF in the PENSRV16.DLL 1-75
STOREBUF::ResetBufIndex 1-76
STOREBUF::STOREBUF 1-76
STOREBUF::~STOREBUF() 1-76
STOREBUF::StartMeasure 1-77
STOREBUF::WriteValue 1-77
STOREBUF::SetOutChan 1-77
STOREBUF::ReadValue 1-77
STOREBUF::GetBufLen 1-78
STOREBUF::GetBufTa 1-78
STOREBUF::GetStatus 1-78
STOREBUF::GetBufferLevel 1-78

1.5.5 The Class AFBUF in the PENSRV16.DLL 1-79

AFBUF::AFBUF() 1-79

AFBUF::~AFBUF() 1-79

AFBUF::NewFBuf 1-80

AFBUF::ReadFBuf 1-80

AFBUF::WriteFBuf 1-80

1.5.6 The Class TWOBUFFER in the PENSRV16.DLL 1-81

TWOBUFFER::TWOBUFFER() 1-82

TWOBUFFER::New2Buffer 1-82

TWOBUFFER::Write2Buffer 1-82

TWOBUFFER::Read2Buffer 1-82

1.5.7 The Class Signal in the PENSRV16.DLL 1-83

SIGNAL::SIGNAL( ) 1-83

SIGNAL::InitTime 1-83

SIGNAL::MakeSignal 1-84

SIGNAL::ReadNextValue 1-84

SIGNAL::SetRange 1-85

SIGNAL::WriteBuffer 1-85

SIGNAL::Stuetzstellen 1-85

1.5.8 The DLL Interface PLOT 1-86

ReadPlot 1-87

WritePlot 1-87

Plot 1-88

GetPlot 1-89

PrintPlot 1-89

GetPldInfo 1-90

2 Driver Functions for PS600 2-1

2.1 The Class DICDRV 2-1

DICDRV::DICDRV 2-2
DICDRV::~DICDRV 2-2
DICDRV::ReadPosition 2-2
DICDRV::ReadWinkel 2-3
DICDRV::SetKraft 2-3
DICDRV::GetXCenter 2-3
DICDRV::SetXCenter 2-3
DICDRV::GetWCenter 2-4
DICDRV::EichOK 2-4
DICDRV::CheckSystem 2-4
DICDRV::CheckFree 2-4
DICDRV::LeftSwitch 2-5
DICDRV::RightSwitch 2-5
DICDRV::StartInterrupt 2-5
DICDRV::TriggerEndstufe 2-5
DICDRV::CalcModel 2-6
DICDRV::SetOrder 2-6
DICDRV::WinDelay 2-6
2.2 The Class WDAC98 2-7
WDAC98::ReadAnalogVolt 2-7
WDAC98::WriteAnalogVolt 2-8
WDAC98::ReadDigital 2-8
WDAC98::ReadAllDigital 2-8
WDAC98::WriteDgital 2-8
WDAC98::GetCounter 2-9
WDAC98::GetTimer 2-9
WDAC98::ReadDDM 2-9
WDAC98::ResetDDM 2-9
WDAC98::ResetAllDDM 2-9

3 The Fuzzy Library 3-1

3.1 Introduction to the Structure of the Fuzzy Library 3-1

3.2 Description of the Classes 3-2

3.2.1 General 3-2

3.2.2 Overview of the Classes 3-2

3.2.3 References of the Classes, their Data and Element Functions 3-4

3.2.3.1 The Class FuzzySet 3-4

FuzzySet::FuzzySet 3-4

FuzzySet::FuzzySet 3-4

FuzzySet::FuzzySet 3-5

FuzzySet::FuzzySet 3-5

FuzzySet::~FuzzySet 3-5

FuzzySet::getname 3-5

FuzzySet::cleary 3-6

FuzzySet::getxvector 3-6

FuzzySet::getyvector 3-6

FuzzySet::getstuetzen 3-6

FuzzySet::insert 3-6

FuzzySet::normalize 3-7

FuzzySet::coa 3-7

FuzzySet::crisp 3-7

FuzzySet::conclude 3-7

FuzzySet::tout 3-8

FuzzySet::= 3-8

FuzzySet::*= 3-9

FuzzySet::+= 3-9

FuzzySet::<< 3-9

3.2.3.2 The class FuzzyVar 3-10

FuzzyVar::FuzzyVar 3-10
FuzzyVar::~FuzzyVar 3-10
FuzzyVar::getname 3-11
FuzzyVar::getsetname 3-11
FuzzyVar::getsetcount 3-11
FuzzyVar::add 3-11
FuzzyVar::check 3-12
FuzzyVar::norm 3-12
FuzzyVar::getmode 3-12
FuzzyVar::getmaxx 3-12
FuzzyVar::getminx 3-12
FuzzyVar::get 3-13
FuzzyVar::set 3-13
FuzzyVar::clear 3-13
FuzzyVar::out 3-13
FuzzyVar::getval 3-14
FuzzyVar::setval 3-14
FuzzyVar::vsort 3-15
FuzzyVar::<< 3-15
3.2.3.3 The class FuzzyRule 3-16
FuzzyRule::FuzzyRule 3-16
FuzzyRule::~FuzzyRule 3-16
FuzzyRule::getname 3-16
FuzzyRule::addIn 3-17
FuzzyRule::addOut 3-17
FuzzyRule::Do 3-17
FuzzyRule::tout 3-17
FuzzyRule::<< 3-18
3.2.3.4 The Class Fuzzy 3-19

Fuzzy::Fuzzy 3-20
Fuzzy::Fuzzy 3-20
Fuzzy::~Fuzzy 3-20
Fuzzy::read 3-21
Fuzzy::write 3-21
Fuzzy::generate 3-21
Fuzzy::calc 3-22
Fuzzy::getinputcount 3-22
Fuzzy::getoutputcount 3-22
Fuzzy::geterrors 3-22
Fuzzy::getrulecatch 3-23
Fuzzy::speed 3-23
Fuzzy::Get3DInfo 3-24
Fuzzy::getname 3-24
Fuzzy::tout 3-24
Fuzzy::parser 3-27
Fuzzy::calcsetup 3-27
Fuzzy::gettoken 3-28
Fuzzy::defvar 3-28
Fuzzy::defset 3-29
Fuzzy::defrule 3-29
Fuzzy::deflabel 3-29
Fuzzy::getlabel 3-30
Fuzzy::killstructures 3-30
Fuzzy::killfuzzybase 3-30
Fuzzy::killlabel 3-30
Fuzzy::out 3-31
Fuzzy::<< 3-31
Fuzzy::>> 3-31

3.3 Description of the File Formats . . . 3-32

3.3.1 The Format of the Fuzzy Description File (*.FUZ) . . . 3-32

3.3.2 The Format of the Error Output File ERROR.OUT . . . 3-34

3.4 A Very Simple Example . . . 3-35

3.4.1 The Fuzzy Description File of a Temperature Control . . . 3-35

3.4.2 The C++ Sources of a Temperature Control . . . 3-36

4 Functions of the PLOT16.DLL 4-1

Version . . . 4-2

CreateSimplePlotWindow . . . 4-2

ShowPlotWindow . . . 4-3

ClosePlotWindow . . . 4-4

UpdatePlotWindow . . . 4-4

GetValidPlotHandle . . . 4-4

AddPlotTitle . . . 4-4

AddAxisPlotWindow . . . 4-5

AddXData: . . . 4-6

AddTimeData: . . . 4-6

AddYData: . . . 4-6

SetCurveMode . . . 4-7

SetPlotMode . . . 4-7

PrintPlotWindow . . . 4-9

CreateEmptyPlotWindow . . . 4-10

5 Interface Functions of the TIMER16.DLL 5-1

LibMain . . . 5-1

SetService . . . 5-2

SelectDriver . . . 5-2

StartTimer . . . 5-3

IsTimerActive 5-3
StopTimer 5-4
GetMinMaxTime 5-4
GetSimTime 5-4
SetupDriver 5-5

6 Windows Drivers for DAC98, DAC6214 and DIC24 6-1

OpenDriver 6-1
SendDriverMessage 6-2
CloseDriver 6-4

1 Source Files of the PENDW16 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 Windows95/98. 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 PENDW16.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 PS6PEND).

You will then find the following subdirectories:
PENDDSK
PENDSERV
EXE

Where PENDDSK contains the *Delphi Project File* PENDW16.DPR together with all the accompanying Pascal source files to generate the desktop, PENDSERV contains the *Borland Project File* PEND.IDE with all the accompanying C++ source files to create the controller-DLL (PENSRV16.DLL). Finally the subdirectory EXE contains all the additional files required by the executable program as can be seen from the following table:

Executable, Libraries and Drivers	**Tandem Pendulum Data Files**	**Additional Data Files**
BC450RTL.DLL	PENDULUM.STA	P_CART.STA
DAC98.DRV	PENDULUM.FBW,	
DIC24.DRV	XCONTROL.FUZ,	
PLOT16.DLL	WCONTROL.FUZ,	
PS6PEND.HLP	XERROR.FUZ,	
PENDW16.INI	NOP.FUZ,	
PENDW16.EXE	,	
PENSRV16.DLL	,	
TIMER16.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 Tandem Pendulum system instead of accessing the PC adapter card) may be obtained simply by setting the macro __SIMULATION__ in the include file PSDEFINE.H and generating a new PENSRV16.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 **PSDEFINE.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 inverted pendulum 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 __FUZZY__
#define __SIMULATION__          //to create Demo version
#define ScopeBufSize            11
#define SIMTIME                 0.03
#define PCREADY_DISABLED        0x01
#define RIGHT_LIMIT_SWITCH      0x02
#define LEFT_LIMIT_SWITCH       0x04
#define MAX_ANGLE_1             0x08
#define MAX_ANGLE_2             0x10
```

The file **PS600DAT.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];         // P346_2 for inverted pendulum
        char    name[10];           // PS600
        char    Titel[10];          // PENDULUM
        char    Version[10];        // last version
        char    Date[10];           // last modification
        char    Dummy[10];          // Reserve
};
```

```
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. (60 bytes)
The structure CTRLSTATUS. (96 bytes)
The structure DATASTRUCT. (8 bytes)
The data array with float values (4 bytes per value).

The size of the data array is defined in the structure DATASTRUCT. With the PS600 inverted pendulum the number of the stored channels is always 11 (the length of the measurement vector is 11, i.e. equal to 44 bytes). When the state controller was active during the measuring the vector contains the following signals (estimated values from Luenberger observer):

the position setpoint	in [m],
the measured position	in [m],
the measured angle	in [rad],
the control force	in [N],
the measured cart speed	in [m/s],
the angular velocity	in [rad/s],
the friction compensation	in [N],
the estimated position	in [m],
the estimated speed	in [m/s],
the estimated angle	in [rad],
the estimated angular velocity	in [rad/s].

When the fuzzy controller was active during the measuring the vector contains the following signals:

the position setpoint	in [m],
the measured position	in [m],
the measured angle	in [rad],
the control force	in [N],
the measured cart speed	in [m/s],
the angular velocity	in [rad/s],
the cart friction compensation	in [N],
the pendulum friction comp.	in [rad],
a dummy zero signal,	
a dummy zero signal,	
a dummy zero signal.	

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)

System1Click(*Sender*: TObject)

StateControllerSetupClick(*Sender*: TObject)

FuzzyControllerSetupClick(*Sender*: TObject)

Run1Click(*Sender*: TObject)

StateController1Click(*Sender*: TObject)

FuzzyController2Click(*Sender*: TObject)

StopController1Click(*Sender*: TObject)

IdentifyCart1Click(*Sender*: TObject)

IdentifyPendulum1Click(*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:
TopBBtnClick(*Sender*: TObject)

MiddleBBtnClick(*Sender*: TObject)

BottomBBtnClick(*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)

OKBtnClick(Sender: TObject)

HelpBtnClick(Sender: TObject)

The file IDCART.PAS contains the procedures:

ScrollBar1Change(*Sender*: TObject)

OKBtnClick(*Sender*: TObject)

FormShow(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 STARTPEN.PAS contains the procedure:

Timer1Timer(*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

ResetButtonClick(*Sender*: TObject)

FormShow(*Sender*: TObject)

HideButtonClick(*Sender*: TObject)

HelpButtonClick(*Sender*: TObject)

The file TOOLS.PAS contains the functions:

FloatToStr2(*f* : Single) : string

FloatToStr3(*f* : Single) : string

FloatToStr4(*f* : Single) : string

FloatToStr5(*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 PENSRV16 and TIMER16.

Global Data:

type ServiceParameter = record

controller :	WORD;
stateobserver :	WORD;
fuzzyobserver :	WORD;
spshape :	WORD;
ft :	array[0..3] of 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;
        la  :               array[0..15] of double;
        lf  :               array[0..3] of double;
        lb  :               array[0..3] of double;
        name :              array[0..79] of char;
        spoffset, spamplitude, spperiode : double;
        sysorder :          WORD;
        stateError, fuzzyError :    WORD;
        dummy1 :            WORD;
 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;
        cardNo :            WORD;
```

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 on 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 PENDW16 program) may be started only once. The boolean variable *calibrated* is reset and a bitmap for the animation picture is prepared.

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 PENDW16 is started with the selection of the controller DLL, here PENSRV16.DLL, and the driver for the adapter card (the drivers file name is contained in the file PENDW16.INI). The check marks below the menu item "IO-Interface" are set accordingly. The parameter structure *param* is read from the PENSRV16.DLL. 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 PENSRV16.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. When PENDW16.INI does not contain a driver file DUMMY.DRV, the sensor inputs are calibrated automatically (**CalibrateSensors1Click**) in case of a real system. And with this the timer for the sampling period of the controller (**StartTimer**, TIMER16.DLL) is started. At the end the timer for the periodic update of the monitor window is started in addition.

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. Otherwise the boolean variable *defaultfiles* is reset.

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 PENDULUM.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 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. Otherwise the boolean variable *defaultfiles* is reset.

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 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 PENDW16 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 PENDW16.INI. After stopping the timer (**StopTimer**) controlling the sampling period of the controller the driver DAC98.DRV is selected for the PENSRV16.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/DIC24". The name of the driver file DIC24.DRV is written to the file PENDW16.INI. After stopping the timer (**StopTimer**) controlling the sampling period of the controller the driver DIC24.DRV is selected for the PENSRV16.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.System1Click

System1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **System1Click** is an event handler activated by clicking once on the menu item "Edit/Switch to system CART" or "Switch to system PENDULUM". When the user prompts the message "Do you really want to switch from system "pendulum" to system "cart" or "Do you really want to switch from system "cart" to system "pendulum" positively, the order of the system (2 for the cart and 4 for the pendulum) under control as well as the title of the menu item are set accordingly. The state feedback vector is reset in addition only, when the system "pendulum" is changed to the system "cart". A corresponding message will inform the user. The updated parameter structure *param* is transferred again to the PENSRV16.DLL (**SetParameter**).

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 Setup". 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 Setup". 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 is set. Where the parameters of the setpoint generator are reset only when no controller was active previously. If in that case a pendulum of a real system is to be controlled the modal dialog **StartPendulumDlg** will appear asking the user to move the pendulum into an upright position. Only a successful controller start will set the corresponding controller type parameter of the structure *param*. This structure is then transferred (**SetParameter**) to the PENSRV16.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 is set. Where the parameters of the setpoint generator are reset only when no controller was active previously. If in that case a pendulum of a real system is to be controlled the modal dialog **StartPendulumDlg** will appear asking the user to move the pendulum into an upright position. Only a successful controller start will set the corresponding controller type parameter of the structure *param*. This structure is then transferred (**SetParameter**) to the PENSRV16.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 PENSRV16.DLL.

TMainForm.IdentifyCart1Click

IdentifyCart1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **IdentifyCart1Click** is an event handler activated by clicking once on menu item "Run/Identify Cart". Any active controller is stopped (**StopController1Click**) and the modal dialog **IdCartDlg** is called with its mode set to identify the cart. At the end the controller type is reset again by **StopController1Click**.

TMainForm.IdentifyPendulum1Click

IdentifyPendulum1Click(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **IdentifyPendulum1Click** is an event handler activated by clicking once on menu item "Run/Identify Pendulum". Any active controller is stopped (**StopController1Click**) and the modal dialog **IdCartDlg** is called with its mode set to identify the pendulum. At the end the controller type is reset again by **StopController1Click**.

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 **CalibrateDlg** 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 modal 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/Setpoint Generator" or by pressing "F6". The conditions for the setpoint of the cart 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 PS6PEND.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 PENSRV16.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 cart position and the pendulum angle, the control signal, the state of the measurement acquisition as well as an updated animation picture. The 'Active controller' panel will display additional error messages, when any controller was stopped by the program itself (disabled Servo, limit switch or maximum pendulum angle).

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.

TCalibrateDlg.TopBBtnClick

TopBBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **TopBBtnClick** is an event handler activated by clicking once on the upper "Start" button inside the "PS600 Inverted Pendulum Calibration Dialog". The timer for the sampling period of the controller is started if it is not running already. The upper panel in the dialog appears in a green background colour indicating an active test of the system connections (**CalibrateSen(0)**). If the timer is not running or the test of the system connections failed the colour of the upper panel 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 panel is changed to white and the next calibration step (**MiddleBBtnClick**) is started automatically.

TCalibrateDlg.MiddleBBtnClick

MiddleBBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **MiddleBBtnClick** is an event handler activated by clicking once on the "Start" button in the middle of the "PS600 Inverted Pendulum Calibration Dialog". The panel on the middle of the dialog appears in a green background colour indicating an active sensor calibration for the zero-angle of the pendulum (**CalibrateSen(2)**). Any pendulum oscillation should be damped manually until its amplitude is less than about 1. If the sensor calibration failed the colour of the panel 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 panel in the middle is changed to white and the next calibration step (**BottomBBtnClick**) is started automatically.

TCalibrateDlg.BottomBBtnClick

BottomBBtnClick(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **BottomBBtnClick** is an event handler activated by clicking once on the lower "Start" button inside the "PS600 Inverted Pendulum Calibration Dialog". The lower panel of the dialog appears in a green background colour indicating an active sensor calibration for the zero-position of the cart (**CalibrateSen(1)**). The procedure to move the cart to the right limit switch, then to the left limit switch and at last near to the zero-position is carried-out automatically. When the check box labelled "View plots" is marked the control signal as well as the measured position are presented in two graphics at the screen. If the sensor calibration failed the colour of the lower panel 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 and the "OK" button to terminate the dialog is enabled.

TCalibrateDlg.FormShow

FormShow(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TCalibrateDlg is displayed on the screen. The three panels assigned to the calibration steps of the "PS600 Inverted Pendulum Calibration Dialog" are displayed with a blue background colour. Only the "Start" button of the upper panel is enabled. The complete calibration is carried-out automatically after pressing the "Start" button.

TCalibrateDlg.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 "PS600 Inverted Pendulum Calibration Dialog". The corresponding section of the help file PS6PEND.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, scrollbars 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 last settings for the rotation angles around the X-axis and around the Y-axis are read from PENDW16.INI. The listbox to select a fuzzy description file is filled with the 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. The last settings for the rotation angles around the X-axis and around the Y-axis are written to PENDW16.INI.

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 "Coordinate 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 "Coordinate 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 PS6PEND.HLP will be displayed in a window on the screen.

TFuzzyParameterDlg.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.

TFuzzyParameterDlg.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.

TFuzzyParameterDlg.FormCreate

FormCreate(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormCreate** creates an instance of type TFuzzyParameterDlg. A string list is generated to store the names of the fuzzy description files.

TFuzzyParameterDlg.FormDestroy

FormDestroy(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormDestroy** is called before the object of type TFuzzyParameterDlg is removed from the memory. The string list containing the names of the fuzzy is erased.

TFuzzyParameterDlg.FormShow

FormShow(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TFuzzyParameterDlg is displayed on the screen. At first only the upper part of the "Fuzzy Controller Parameters" dialog is presented containing four fields ("Position Controller", "Angle Controller", "Position Observer" and "Angle Observer") displaying the names of the accompanying fuzzy description files. These names are loaded 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.

TFuzzyParameterDlg.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.

TFuzzyParameterDlg.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.

TFuzzyParameterDlg.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.

TFuzzyParameterDlg.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.

TFuzzyParameterDlg.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.

TFuzzyParameterDlg.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 PS6PEND.HLP will be displayed in a window on the screen.

TIdCartDlg.ScrollBar1Change

ScrollBar1Change(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **ScrollBar1Change** converts the current positions of the scroll marks to the strings for the accelerating power and the measuring time.

TIdCartDlg.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 "Start" button of the "Cart/Pendulum Identification Dialog". After reading the structures *param* and *data* from the PENSRV16.DLL the accelerating power is read from the current position of the upper scrollbar. Its sign is converted to minus, when the current cart position is positive. The resulting power is assigned to the constant amplitude of the setpoint generator values of the structure *param* which is then transferred to the PENSRV16.DLL. After starting a measurement acquisition with a measuring time according to the position of the lower scrollbar an endless loop begins. This loop is terminated only when the measuring time is over or the controller type is unequal to IDENTIFICATION. The last will occur if the cart reaches one of the limit switches. The accelerating power is reset to zero when the cart reaches a distance of 0.3m from the middle position. At the end the measured cart speed or the measured angle of the pendulum (depending on the public variable *mode*) are displayed in a graphic on the screen.

TIdCartDlg.FormShow

FormShow(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TIdCartDlg is displayed on the screen. The procedure **ScrollBar1Change** is called at first. When the public variable *mode* is TRUE, the dialog title is set to "Pendulum Identification Dialog". Else the title will be "Cart Identification Dialog".

TIdCartDlg.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 "Cart/Pendulum Identification Dialog". The corresponding section of the help file PS6PEND.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* = -0.6 to 0.6) 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 cart position, the measured values for the cart position and the pendulum 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 PS6PEND.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 PS6PEND.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 cart position, the measured cart position, the measured pendulum angle, the control signal, the calculated cart speed and angular velocity, the friction compensation for the cart and the pendulum (the last only with the fuzzy controller). The observed signals are meaningful only for an active state controller. The selected channel is presented in a graphic on the screen by calling **PlotMeasObs**.

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 PS6PEND.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 PS6PEND.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 "Position Setpoint Generator" to adjust the setpoint for the cart 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 -0.6 to +0.6m 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 PENSRV16.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 cart 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 "Position Setpoint Generator". The corresponding section of the help file PS6PEND.HLP will be displayed in a window on the screen.

TStartPendulumDlg.Timer1Timer

Timer1Timer(*Sender*: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **Timer1Timer** is called every 1000 ms by means of a timer belonging to the modal dialog titled "Start Pendulum Controller", which asks the user to move the pendulum manually into an upright position. The dialog is terminated either by means of the "Cancel" button or when the absolute values of the pendulum angles is less than about 4.5 degrees.

TStateParameterDlg.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 TStateParameterDlg. With this a multiple-page dialog with four pages is generated.

TStateParameterDlg.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.

TStateParameterDlg.FormShow

FormShow(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **FormShow** is called, when the object of type TStateParameterDlg 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 components of the matrices *A, B, L, F* of the reduced-order state observer are changeable with the second page "State Observer". The third page with the tab stop label "Friction Compensation" allows for editing the matrices *A, B, L, F* of the (friction) disturbance observer and the parameter *Const.* of the constant friction compensation. The components of the matrices *A, B, F* of the complete state observer are changeable with the fourth page "Luenberger Observer".

TStateParameterDlg.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 PENSRV16.DLL. An error message is presented in the other case.

TStateParameterDlg.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 PS6PEND.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 one or two checkboxes to select the friction compensation for the cart and/or the pendulum 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 PS6PEND.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 cart (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 PS6PEND.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**). Depending on the listbox selection 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 "PS600 Timing" dialog. The flag *ResFlag* to reset the minimum and maximum values is reset.

TTimingForm.ResetButtonClick

ResetButtonClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **ResetButtonClick** is an event handler activated by clicking once on the "Reset" button in the "PS600 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 "PS600 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 (**ResetButtonClick**) is carried-out. The default display values are from the real sampling period.

TTimingForm.HideButtonClick

HideButtonClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HideButtonClick** is an event handler activated by clicking once on the "Hide" button in the "PS600 Timing" dialog. The procedure **TMainForm.Timing1Click** will reset the visibility of the "PS600 Timing" dialog.

TTimingForm.HelpButtonClick

HelpButtonClick(Sender: TObject)

Parameters: *Sender* is a reference to the calling object.

Description The procedure **HelpButtonClick** is an event handler activated by clicking once on the "Help" button in the dialog "PS600 Timing". The corresponding section of the help file PS6PEND.HLP will be displayed in a window on the screen.

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 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 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 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.

FloatToStr5

FloatToStr5(*f* : Single) : string

Parameters: *f* is the floating point value, which is to be converted.

Description The function **FloatToStr5** converts a floating point value (4 bytes for single) to its string representation with a maximum of 7 significant digits and 5 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 PENSRV16.H, PENSRV16.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 PS600STA.H, PS600STA.CPP contain the class **PS600STA** with:

void **PS600STA** (void)

double **Calc**(double *w*, double *Position*, double *Winkel*, double *Winkel_2*)

void **SetTa**(float *ta*)

void **Reset**(void)

int **Load**(char* *name*)

int **Save**(char* *name*)

void **SetOrder**(int *order*)

int **GetOrder**(void)

void **SetStateObserver**(Observer *o*) void **SetDistObserver**(Observer *o*)

int **geterrors**(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)

double* **GetLA**(void)

double* **GetLF**(void)

double* **GetLB**(void)

The files PS600FUZ.H, PS600FUZ.CPP contain the class **PS600FUZ** :

void **PS600FUZ** (void)

void ~**PS600FUZ** (void)

double **Calc**(double *w*, double *Position*, double *Winkel*, double *Winkel_2*)

int **Load**(char* *name*)

int **Save**(char* *name*)

void **SetAngObserver**(Observer *o*)

void **SetPosObserver**(Observer *o*)

char* **getname**(void)

char* **getfname**(int *i*)

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        fuzzy_observer;      // flag for type of friction compensation in fuzzy controller
        WORD        sp1shape;            // shape of setpoint signal (constant, rectangle, sine etc.)
        double      ft[4];               // state feedback vector
        double      lbd[4];              // reduced order observer matrix L
        double      abd[4];              // reduced order observer matrix A
        double      fbd[4];              // reduced order observer matrix F
        double      bbd[2];              // reduced order 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
        double      la[16];              // Luenberger observer matrix A
        double      lf[4];               // Luenberger observer matrix F
        double      lb[4];               // Luenberger observer vector b
        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
        WORD        sysorder;            // order of controlled system
        WORD        stateError;          // flag for error in state controller
        WORD        fuzzyError;          // flag for error in fuzzy controller
        WORD        dummy;
}ServiceParameter ;

typedef struct{
        double      setpoint;            // setpoint for cart position
        double      pos;                 // measured value of cart position
        double      dpos;                // calculated value of cart speed
        double      angle;               // measured value of long pendulum angle 1
        double      dangle;              // calculated value of angular velocity 1
        double      out;                 // control signal for cart drive
        double      fuzhelp;             // output 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 PENSRV16.H)
ServiceData *dat*	is a global structure of type ServiceData (see also PENSRV16.H)
HGLOBAL *mHnd*	is a handle for the code memory of the PENSRV16.DLL
UINT *mData* = 0	is a handle for the data memory of the PENSRV16.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
PS600STA *StateCon*	is a state controller object
PS600FUZ *FuzzyCon*	is a fuzzy controller object
int *changed*	is a flag for detecting multiple output stage release errors, max. angle or limit switches 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 PENSRV16

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:

Return directly during sensor calibration mode,
Check for active controller (state controller, fuzzy controller, identification),
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 cart position, pendulum angle,
Check for maximum angle or limit switches, set *changed* accordingly,
Reset controller to none, when *changed* unequal to zero,
Calculate the control signal (state controller/fuzzy controller/identification/none),
Output of control signal,
If PC control is disabled increment release counter, else reset
If release counter is equal to 5 set flag *changed*,
Store the measurement-vector.

Attention: This function is to be called only by the TIMER16.DLL!

Return Is always 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 reset to zero or set to the value of *changed*. In the last case *changed* is reset to zero.

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 PENSRV16.DLL will be locked,

=FALSE, code and data memory of the PENSRV16.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 PENSRV16.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 PENSRV16.DLL is valid, else return is equal to FALSE.

SetDriverHandle

BOOL CALLBACK **SetDriverHandle**(HDRVR *hDrv*)

Parameters *hDrv* is an 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 PENSRV16.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, check for correct system connections,
=1, zero-position of cart,
=2, zero-angles of pendulums.

Description

The function **CalibrateSen** checks for correct system connections and determines the calibration data for the incremental encoder signals to measure the cart position and the pendulum angle depending on the parameter *mode*. With *mode*=0 two signals from the connection between the actuator and the mechanics are tested. The signal levels should be equal to a mask for the inverted pendulum system. The zero-position of the cart is determined with *mode*=1. Hereby the servo amplifier is controlled (proportional controller with a ramp setpoint) such that the cart reaches at first the right limit switch, then the left limit switch where the position sensor can be calibrated. At the end the cart is driven near to the zero-position. This procedure is carried-out in a loop with a periodic check of the system connections and the output stage release circuit until either the calibration is successful or a time-out value of 60s is reached. With *mode*=2 the procedure tries to detect the zero-angle of the pendulum. The detection is taken as valid, when the amplitudes of the pendulum angle remains within about 1 degree for a period of 4 seconds. Otherwise the measurement will end after a time-out of 120s. For possible errors see below.

Return

Error state of the calibration procedure:
=0, no error,
=-1, invalid value for *mode*,
=-2, system detection failed,
=-3, PC control disabled,
=-4, time-out during waiting for small pendulum angle,
=-6, time-out during driving the cart.

RawSensor

int CALLBACK **RawSensor**(int *mode*)

Parameters

mode defines the return value:
=0, increments of position sensor,
=1, increments of pendulum angle sensor.

Description

The function **RawSensor** returns the sensor data (raw data) of the incremental encoders depending on the value of *mode*.

Return

Increments for position or angle.

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 PS600STA in the PENSRV16.DLL

The class **PS600STA** provides functions to calculate the state controller and to determine missing state variables as well as disturbance signals. It inherits from the basic class **CONTROLLER**.

Public Data:

enum *Observer*	{ NONE=0, CONST, ACTIVE };

Private Data:

double *t*	sampling period (class **CONTROLLER**)
int *errors*	error counter for file loading (class **CONTROLLER**)
char *filename[MAXPATH]*	current name of the parameter file (class **CONTROLLER**)
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[2]*	B-vector of the state observer
double *dcon*	parameter of the constant friction compensation
double *dhaft*	parameter of the static 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 *la[16]*	A-matrix of the Luenberger observer
double *lf[4]*	F-matrix of the Luenberger observer
double *lb[4]*	B-matrix of the Luenberger observer
double *lx[4]*	current Luenberger observer state vector
double *lxold[4]*	previous Luenberger observer state vector
double *Position_alt*	measured cart position from the previous sampling period
double *Winkel_alt*	measured pendulum angle from the previous sampling period
int *start*	flag: reset of the observer
int *order*	order of the controlled system
Observer *state*	mode of the state observer
Observer *dist*	mode of the disturbance observer

PS600STA::PS600STA()

void **PS600STA** (void) : **CONTROLLER**( )

Description The constructor of the class **PS600STA** 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 PENULUM.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.

PS600STA::Calc

double **Calc**(double *w*, double *Position*, double *Winkel*, double *Winkel_2*)

Parameters *w* is the setpoint of the cart position.

Position is the measured value of the cart position.

Winkel is the measured value of the pendulum angle.

Description The function **Calc** is the main function of this class. It carries-out the calculation of the state controller. The calculations are carried-out depending on the order of the controlled system (=2 for cart only, =4 for inverted pendulum). When the flag *start* is set, the initial states of the observer and the controller are reset. The missing state variables cart speed and angular velocity of the pendulum 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 cart 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 cart speed or reset to 0.0 depending on the mode *dist* of the disturbance observer. Following the calculation of the two observers for the missing state variables and the friction compensation the complete state vector of the Luenberger observer is determined in addition. After limiting this estimated state vector (which is not used for the control!) 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.

PS600STA::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.

PS600STA::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.

PS600STA::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.

PS600STA::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.

PS600STA::SetOrder

void **SetOrder**(int *order*)

Parameters *order* is the order of the system, which is to be controlled.

Description The function **SetOrder** sets the order of the system, which is to be controlled in the following (valid values are: =2 for cart only, =4 for inverted pendulum).

PS600STA::GetOrder

int **GetOrder**(void)

Description The function **GetOrder** returns the order of the currently controlled system (=2 for cart only, =4 for inverted pendulum).

Return The order of the currently controlled system.

PS600STA::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 cart speed as well as the angular velocity of the pendulum. 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).

PS600STA::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 cart 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 cart speed (CONST) or reset to 0.0 (NONE).

PS600STA::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*.

PS600STA::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*.

PS600STA::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.

PS600STA::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.

PS600STA::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.

PS600STA::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.

PS600STA::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.

PS600STA::GetDLBD

double* **GetDLBD**(void)

Description The function **GetDLBD** returns a pointer to the *L*-matrix of the disturbance observer (compensation of cart friction).

Return A pointer to the *L*-matrix of the disturbance observer.

PS600STA::GetDABD

double* **GetDABD**(void)

Description The function **GetDABD** returns a pointer to the *A*-matrix of the disturbance observer (compensation of cart friction).

Return A pointer to the *A*-matrix of the disturbance observer.

PS600STA::GetDFBD

double* **GetDFBD**(void)

Description The function **GetDFBD** returns a pointer to the *F*-matrix of the disturbance observer (compensation of cart friction).

Return A pointer to the *F*-matrix of the disturbance observer.

PS600STA::GetDBBD

double* **GetDBBD**(void)

Description The function **GetDBBD** returns a pointer to the *B*-matrix of the disturbance observer (compensation of cart friction).

Return A pointer to the *B*-matrix of the disturbance observer.

PS600STA::GetLA

double* **GetLA**(void)

Description The function **GetLA** returns a pointer to the *A*-matrix of the complete Luenberger state observer.

Return A pointer to the *A*-matrix of the complete Luenberger state observer.

PS600STA::GetLF

double* **GetLF**(void)

Description The function **GetLF** returns a pointer to the *F*-matrix of the complete Luenberger state observer.

Return A pointer to the *F*-matrix of the complete Luenberger state observer.

PS600STA::GetLB

double* **GetLB**(void)

Description The function **GetLB** returns a pointer to the *B*-matrix of the complete Luenberger state observer.

Return A pointer to the *B*-matrix of the complete Luenberger state observer.

1.5.3 The Class PS600FUZ in the PENSRV16.DLL

The class **PS600FUZ** provides functions to apply a fuzzy controller. It serves as an interface to the fuzzy algorithms contained in **Fuzzy.lib**. It inherits from the basic class **CONTROLLER**.

Public Data:

enum Observer	{NONE=0, CONST, ACTIVE};
double hlpval	output of the outer fuzzy controller in the cascade.

Private Data:

int *errors*	error counter for file access (class **CONTROLLER**)
Fuzzy **PosController*	fuzzy cart position controller
Fuzzy **AngController*	fuzzy angle controller
Fuzzy **PosObserver*	fuzzy position observer
Fuzzy **AngObserver*	fuzzy angle observer
Observer *posobserver*	mode of the fuzzy position observer
Observer *angobserver*	mode of the fuzzy 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
double *hlpval*	internal control signal of cascaded fuzzy controller

PS600FUZ::PS600FUZ()

void **PS600FUZ** (void) : CONTROLLER

Description The constructor of the class **PS600FUZ** assigns NULL to all the pointers to fuzzy objects, reads the "fuzzy-controller" file PENDULUM.FBW (see also **Load**) and generates the accompanying fuzzy objects. The modes of the fuzzy position observer as well as of the fuzzy 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.

PS600FUZ::~PS600FUZ()

void **~PS600FUZ** (void)

Description The destructor of the class **PS600FUZ** removes all the fuzzy objects from the memory.

PS600FUZ::Calc

double **Calc**(double *w*, double *Position*, double *Winkel*)

Parameters *w* is the setpoint of the cart position.

Position is the measured value of the cart position.

Winkel is the measured value of the pendulum angle.

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 cart speed and angular velocities 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 angle observer is activated an additional signal (angle offset) to compensate the pendulum friction is calculated by the fuzzy object with the two input signals angular velocity and cart 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 cart speed produce the setpoint signal for the pendulum angle controller in the lower cascade. The fuzzy object of the pendulum angle controller is driven by the two input signals, angle control error and the angular velocity of the pendulum. Its output signal is the control signal for the plant. When the fuzzy position observer is activated a second additional signal (power offset) to compensate the cart friction is calculated by the fuzzy object with the two input signals cart speed and control signal from the previous sampling period. Otherwise the second additional signal is reset to 0.0.

The additional signal for the cart friction compensation 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 additional signal.

Return The control signal with cart friction compensation of the fuzzy controller.

PS600FUZ::Load

int **Load**(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 **Load** 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 *errors* returned.

Return The internal *errors*.

PS600FUZ::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.

PS600FUZ::SetAngObserver

void **SetAngObserver**(Observer *o*)

Parameters *o* is the new mode of the fuzzy angle observer.

Description The function **SetAngObserver** determines the mode of the fuzzy angle observer to calculate the additional signal for the compensation of the pendulum friction. The additional signal is determined either by means of the fuzzy angle observer (ACTIVE) or is reset to 0.0 (NONE).

PS600FUZ::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 calculate the additional signal for the compensation of the cart friction. The additional signal is determined either by means of the fuzzy position observer (ACTIVE) or is reset to 0.0 (NONE).

PS600FUZ::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*.

PS600FUZ::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]*.

PS600FUZ::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 PENSRV16.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 PENSRV16.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 PENSRV16.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 PENSRV16.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 PENSRV16.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 cart position with setpoint [m] (2 curves),
=1, Measured cart position [m] (1 curve),
=2, Measured pendulum angle 1 [rad] (1 curve),
=3, Controller output [N] (1 curve),
=4, Measured cart speed [m/s] (1 curve),
=5, Measured angular velocity [rad/s] (1 curve),
=6, Cart friction compensation [N] (2 curve2),
=7, Pendulum friction compensation (Fuzzy)[N] or Observed position [m] (2 curves)
=8, Observed angle [rad] (2 curves),
=9, Observed speed [m/s] (2 curves),
=10, Observed angular velocity.

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*. When especially the value of *channel* is greater or equal to 100, this value is subtracted and any graphic for a state variable (position, angles and their derivations) contain the corresponding observed values in addition. Plot titles and axes descriptions for some plot channels depend on the type of the controller which was active during the measurement acquisition (see **fixTitles** in the source code).

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 PS600

2.1 The Class DICDRV

The class **DICDRV** provides the interface between the PS600 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 inverted pendulum system when it is compiled with '#define __SIMULATION__' (see file PSDEFINE.H). With this all program functions except for the calibration can be carried-out for a simulated inverted pendulum 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 **ReadPosition**(void)

double **ReadWinkel**(void)

void **SetKraft**(double *n*)

void **GetXcenter**(void)

void **SetXcenter**(double *mval*)

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 **CalcModell**(int *reset*)

void **SetOrder**(int *o*)

The files TOOLS.H, TOOLS.CPP contain function:

void **WinDelay**(int *ms*)

Public Data:

short *IncOffset_0* is the incremental encoder signal for the zero-position of the cart.

short *IncOffset_1* is the incremental encoder signal for the zero-angle of the pendulum.

short *order* is the order of the model.

double *position* is the cart position of the mathematical model.

double *angle* is the long pendulum angle of the mathematical model.

double *power* is the driving force for the cart of the model.

double *x[4]* is the state vector of the mathematical model.

double *PosIncs* is the incremental encoder signal for the cart position.

double *AngIncs* is the incremental encoder signal for the pendulum angle.

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_0, IncOffset_1*. The two analog outputs are reset to 0. With the DEMO-version initial values are set for the cart position and the pendulum angle and the model variables are reset to zero.

DICDRV::~DICDRV

~DICDRV(void)

Description The destructor of the class **DICDRV** resets the real control signal to 0.

DICDRV::ReadPosition

double **ReadPosition**(void)

Description The function **ReadPosition** reads the incremental encoder signal to measure the cart position, stores this value to *PosIncs* and returns its conversion to a position in [m].

With the DEMO-version the cart position *position* calculated by the mathematical model is returned.

Return The cart position in [m].

DICDRV::ReadWinkel

double **ReadWinkel**(void)

Description The function **ReadWinkel** reads the incremental encoder signal to measure the angle of the pendulum, stores this value to *AngIncs* and returns its conversion to an angle in [rad].

With the DEMO-version the pendulum angle *angle* calculated by the mathematical model is returned.

Return The angle of the pendulum in [rad].

DICDRV::SetKraft

void **SetKraft**(double *n*)

Parameters *n* is the driving force for the cart in [N].

Description The function **SetKraft** calculates the control signal in [Volt] required for the given driving force *n* in [N] for the cart drive. This control signal is limited to +/-10V 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 *power* and the mathematical model of the inverted pendulum system is calculated (**CalcModell**).

DICDRV::GetXCenter

void **GetXCenter**(void)

Description The function **GetXCenter** resets the control signal for the cart drive to 0 and takes the current incremental encoder signal (**ReadDDM**) as a valid value for the zero-position *IncOffset_0* of the cart.

DICDRV::SetXCenter

void **SetXCenter**(double *mval*)

Parameters *mval* is the new zero-position of the cart in [m].

Description The function **SetXCenter** sets the increments of the zero-position *IncOffset_0* according to the given value *mval* (used to calibrate the position sensor).

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_1* of the pendulum angle.

DICDRV::EichOK

int **EichOK**(void)

Description The function **EichOK** is a dummy function reserved for future use.

Return A value of 1.

DICDRV::CheckSystem

int **CheckSystem**(void)

Description The function **CheckSystem** checks if the two system identification signals are equal to the mask for a inverted pendulum system. When this condition is reached at least five times in ten successive readings the return value is 1, else it is 0 (system lead not connected or defect?, wrong lead/system).

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 at least five times in ten successive readings, 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 cart position is activated not more than two times in ten successive readings, else it returns 1.

Return A value of 1 for an activated left limit switch, else 0.

DICDRV::RightSwitch

int **RightSwitch**(void)

Description The function **RightSwitch** returns a value of zero, when the right limit switch for a maximum cart position is activated not more than two times in ten successive readings, else it returns 1.

Return A value of 1 for an activated right limit switch, else 0.

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::CalcModel

virtual void **CalcModel**(int *reset*)

Parameters *reset* is the reset flag for the model calculation.

Description The function **CalcModel** calculates the nonlinear state space model of the inverted pendulum system with the DEMO-version when the flag *reset* is FALSE, otherwise the state variables are reset to zero before the calculation is carried-out. A constant cart friction of 3[N] is simulated in addition.

DICDRV::SetOrder

virtual void **SetOrder**(int *o*)

Parameters *o* is the order (2 or 4) of the state space model.

Description The function **SetOrder** sets the order of the state space model, when it is different from the current order. In this case the state vector is reset to zero in addition. Only values of 2 for the cart or 4 for the pendulum are valid.

DICDRV::WinDelay

virtual void **WinDelay**(int *ms*)

Parameters *ms* is the delay time in milliseconds.

Description The function **WinDelay** uses the multi media timer to produce a delay time of *ms* milliseconds. The delay time is limited to the range from 1 to 100 milliseconds and incremented by one to reach at least a delay time of 1 ms.

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*)

int **ReadAllDigital**(void)

void **WriteDigital**(int *channel*, int *value*)

unsigned int **GetCounter**(void)

unsigned long **GetTimer**(void)

unsigned int **ReadDDM**(int *channel*)

void **ResetDDM**(int *channel*)

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::ReadAllDigital

int **ReadAllDigital**(void)

Description The function **ReadAllDigital** reads the state of all input channels and returns this value.

Return: The state of all digital input channels.

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::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)
int **Get3DInfo**(Fuzzy3DInfo **info*)
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:

```
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
};
```

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 last pass.

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::Get3DInfo

int **Get3DInfo**(Fuzzy3DInfo **info*)

Parameters Fuzzy3DInfo **info* is a pointer to a structure of type Fuzzy3DInfo.

Description The function **Get3DInfo** copies the names and range values of those fuzzy variables referenced by their indexes. The indexes *idx, idy* and *idz* are taken from the Fuzzy3DInfo structure pointed to by *info*. For valid indexes the names, minimum and maximum values of the referenced variables are copied to this structure. When the index *idy* is invalid (less than 0 or greater than the number of input variables of this fuzzy object) only, its value is set equal to the valid value of *idx*. This allows for a three-dimensional representation of the characteristic area of a fuzzy object containing only one input variable.

Return Status (int) of operations:
0, no errors
-1, invalid pointer *info* (=NULL)
-2, size conflict in Fuzzy3DInfo structure
-3, invalid index *idx* (< 0 or > number of input variables)
-4, invalid index *idz* (< 0 or > number of output variables)

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

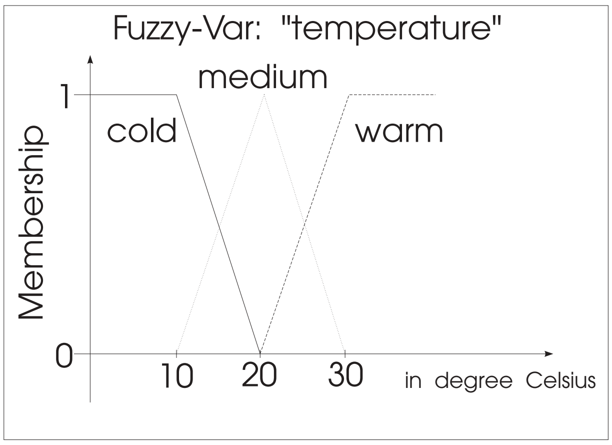


Figure 3.1: The fuzzy variable 'temperature'

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>isalreadydefined.
<n>Errorsdetected.
```

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 (The recent version of this DLL is 1.1).

The interface of the TIMER16.DLL contains the following functions:

UINT **SetService**(LPSTR *lpServiceName*)
UINT **SelectDriver**(LPSTR *lpDriverName*)
UINT **StartTimer**(double *Time*)
UINT **StopTimer**(void)
UINT **IsTimerActive**(void)
void **GetMinMaxTime**(DWORD *&min* , DWORD *&max*, BOOL *res*)
float **GetSimTime**(void)
UINT **SetupDriver**(void)

LibMain

int **LibMain**(HINSTANCE , WORD, WORD, LPSTR)

Parameters All parameters will be left out of consideration.

Description The function **LibMain** only resets the addresses of the functions of the "Service" to NULL and returns 1.

Return Is always equal to 1.

SetService

UINT **SetService**(LPSTR *lpServiceName*)

Parameters

lpServiceName is a pointer to the name of the "Service"-DLL which contains the controller and is to be called periodically.

Description

The function **SetService** stores the given name (including the extension "DLL") to the variable *szServiceName* and tries to load the DLL with this name. In case a DLL with the given name cannot be loaded, an error message ("SetService ’ServiceName’ LoadLibrary failed!") is presented and the function returns 0 immediately. Otherwise the addresses of the functions **DoService, SetParameter, GetData, LockMemory, IsDemo** and **SetDriverHandle** which should be contained in the DLL are determined. If one of these addresses cannot be determined an error message ("SetService - GetProcAddress ’function name’ failed!") appears on the screen and the function returns 0 immediately.

Attention: It is strongly recommended to call this function as the first function of the TIMER16.DLL. Furthermore it has to be called before any function of the "Service"-DLL is called!

Return

Is equal to 1 in case of successful loading the "Service"-DLL and correct address determination, else equal to 0.

SelectDriver

int **SelectDriver**(LPSTR *lpDriverName*)

Parameters

lpDriverName is a pointer to the name of the new driver for the PC adapter card.

Description

The function **SelectDriver** stores the given name (including the extension "DRV") to the variable *szDRiverName*, which determines the driver for the PC adapter card, only when no timer is running and no other card driver is open.

Attention: It is strongly recommended to call this function for the first time directly after calling **SetService**!

Return

Error state :
TERR_OK (0) on successful operations,
TERR_RUNNING (1), when a timer is still running,
TERR_FAIL (99), when another card driver is open.

StartTimer

UINT **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.

IsTimerActive

UINT **IsTimerActive**(void)

Description The function **IsTimerActive** returns the state of the timer controlling the sampling period.

Return Timer state :

0: timer is not running,
1: timer is running.

StopTimer

UINT **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/calculation time in ms. If this value is equal to 0 at entry the calculation time of **DoService** function is returned, else the sampling period.

&max is a reference to the maximum sampling period/calculation time in ms.

res is a flag to reset the minimum and maximum value of the sampling period/calculation time.

Description The function **GetMinMaxTime** returns the minimum and maximum value of the real sampling period or the calculation time during the sampling period (when *min*=0 at entry) determined up to this time. With *res*=1 the minimum and maximum value are set to the nominal sampling period or a calculation time of 0.

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

UINT **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.

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. Each driver may be opened only once meaning that only one PC adapter card may be handled by this driver. 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.

The address of the PC adapter card handled by this driver is read from a specific entry of the file SYSTEM.INI from the public Windows directory. When this entry is missing the default address 0x300 (=768 decimal) will be taken.

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 and stores it to SYSTEM.INI, i. e. [DAC98] Adress=768
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**.