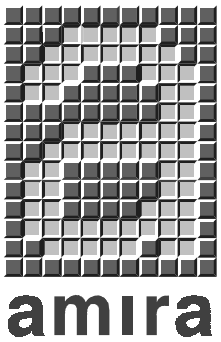


DTS200

Laboratory Setup Three - Tank - System

printed 23. May 2002

Bismarckstraße 67 · im Technologiezentrum · D-47057 Duisburg · Telefon 0203/37809-50 · Telefax 0203/37809-56

Assembly and Start-Up

Date: 05. November 2001

1 Assembly and Start-Up 1-1

1 Assembly and Start-Up

1.1 Unpacking

After the DTS200 has been unpacked, all components are to be checked visually for damages as well as for completeness. Should anything be damaged notify us and save the item and the packaging material until final clarification.

The standard shipment of the DTS200 consists of:

- A premounted tank system complete with pumps, sensors and system cable.
- An actuator consisting of two servo amplifiers for the pumps (SERVO), a power supply unit with an integrated signal adaption unit (POWER) and a module for measuring outputs (SENSOR), contained within a 19" module box and a mains supply lead.
- An additional filling tube.
- A bottle of water colorant.

Depending on the desired system option, the shipment also includes the following items:

- Option 200-02
  A PC plug-in card DAC98 for a PC-AT-computer with 50-pol. connection lead and the software suitable for the control.
- Option 200-05
  An electrical disturbance module (SIGNAL ERROR) in 19" plug-in design, which is contained in the actuator box at the time of delivery.

1.2 Setting up the System

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).

Before selecting a place to set up the system, you should consider the following points:

Tank system:

- Choose a place, where the tank system is not exposed to extreme temperatures. In particular direct sun light and direct heat ratiation, e. g. by a radiator, are to be avoided.
- The tank system must be placed on a solid surface.
- Make sure that the area you selected is able to support the weight of all system components without difficulties.
- To guarantee a perfect control, the tank system must be completely level.
- Choose a hard surface area. Soft surfaces like carpets can hold a static charge. In case of contact, this can lead to discharge and to damages of sensitive circuits inside the actuator.

Actuator:

- The air must be able to circulate freely above, below as well as behind the actuator and the personal computer respectively.
- Do not place any heavy objects on top of the actuator.
- It is important, not to expose the actuator to extreme temperatures. Avoid intensive or direct sun radiation as well as any other heat sources.

- Humidity and dust must be avoided. High humidity can lead to malfunctions and damages of the actuator.

1.3 The Rear Panel

Before putting the connections of the DTS200 into their positions, have a good look at the rear panel and locate the position of the connection sockets.

1.3.1 The Power-Switch

The power switch turns the power supply (mains supply) to the actuator ON and OFF.

1.3.2 Mains input

The mains supply lead is to be connected to the mains input.

1.3.3 Fuse

The fuse is a glass-tube fuse with M1.6 A (medium slow-blowing) in order to secure the 230 V mains supply.

1.3.4 System

The lead for the connection between the actuator and the tank system is to be connected here.

1.3.5 PC-Connector

In case you have selected Option 200-02, plug in the 50-pol. lead for the connection between the actuator and the A/D-D/A card of the personal computer..

Note

The mains supply lead is to be connected only after the peripheral devices are connected. When peripheral devices are to be connected or disconnected, you must make sure that all components are switched off.

1.4 The Front Panel

Turn the actuator around so you can see the front panel. The following figure displays the components located on the front panel.

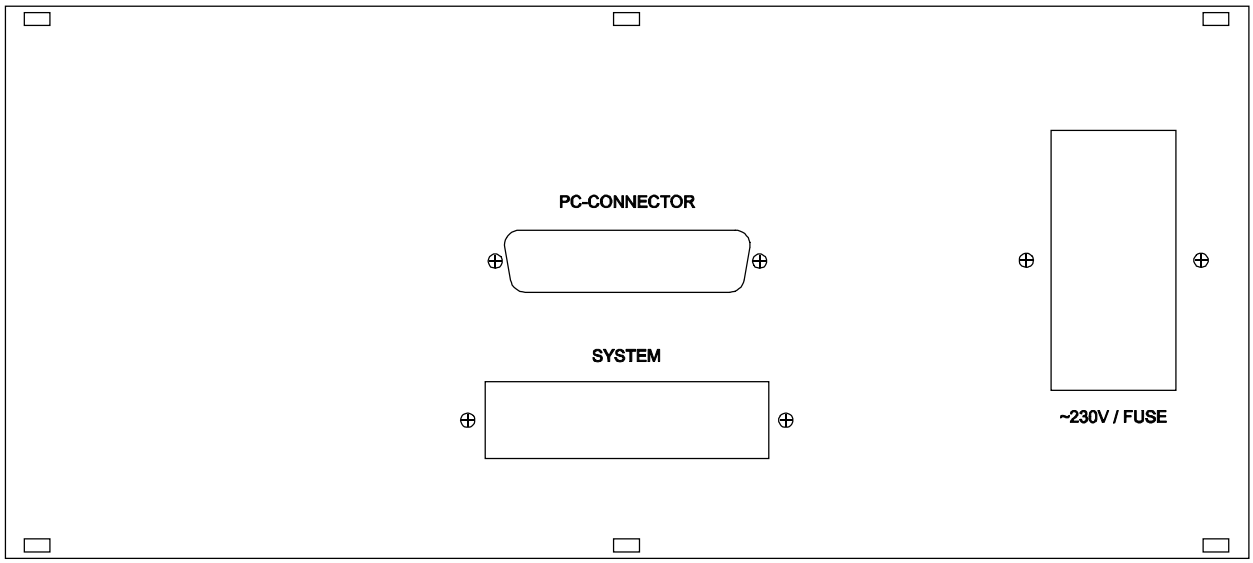


Figure 1.1 : Rear panel with denotations

1.4.1 Actuators (SERVO module)

Both servo controllers for the pumps are located on the left-hand side of the front panel. The left module (servo controller 1) controls the flow of pump 1 for the left tank, the right module (servo controller 2) controls pump 2 for the right tank. Two light emitting diodes indicate the state of operation of each module.

- Ready (green) : The voltage supply is active.
- Limit (red) : The maximum liquid level of the corresponding tank is reached and the pump is shut down (overflow protection).
- Switch Automatic/Manual : This switch allows to change from computer based control (Automatic) to manual control (Manual) of the pumps.
- Potentiometer : For manual control of the servo controller, the flow of the corresponding pump is adjusted by the potentiometer.

1.4.2 Power Servo

This module provides the AC supply voltages for the two servo amplifiers. It does not contain any control or display element.

1.4.3 Mains Supply and Signal Adaption Unit (POWER module)

The power module contains the power supply for the electronics and the amplifiers for the adaption of the sensor signals.

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

1.4.4 Measurement outputs (SENSOR module)

The sensor module provides for external processing of the three sensor signals, the two control signals as well as ground at its 6 jacks.

- Tank1 : Processed signal of the liquid level sensor of tank 1.
- Tank2 : Processed signal of the liquid level sensor of tank 2.
- Tank3 : Processed signal of the liquid level sensor of tank 3.

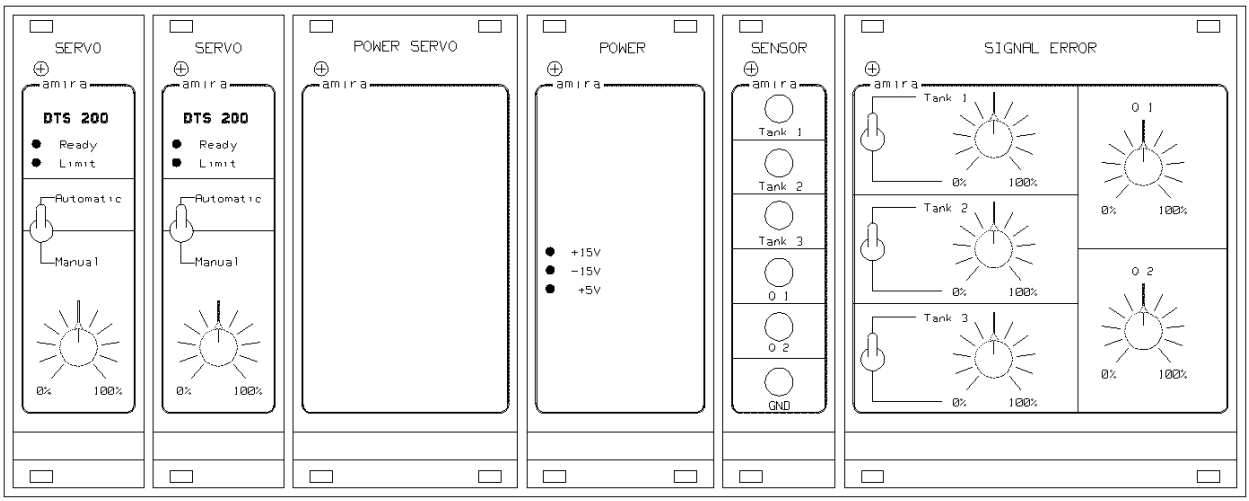


Figure 1.2 : Front panel with denotations

- Q1 : Processed control signal for the servo controller 1.
- Q2 : Processed control signal for the servo controller 2.
- GND : Electrical ground potential corresponding to the described measurement outputs.

Note:

The signals of the liquid level sensors are processed as follows:
The sensor signal is amplified such that the resulting range is +9V to -9V. We use this reduced range (instead of +/- 10V) due to the fact that the offset of the sensor signals may be variable. Mainly air bubbles inside the pressure sensor line which cannot be avoided during the filling operation may produce different offset values. The variation of the resulting pressure signal offset will always be less than +/-1V so that the signal will remain in the limited range of +/-10V. At least after the first tank filling operation the pressure sensors have to be calibrated. This is performed with the assistance of the PC controller program (see "Practical Instructions").

1.4.5 Electrical Disturbance Module (SIGNAL ERROR module)

The signal error module is present only in case your system is equipped with Option 200-05. Otherwise , a blank panel will be installed in its place.

The module allows for a scaling of the processed sensor signals and the control signals in a range of 0% to 100%. By means of three switches, a simulation of total sensor failures is possible.

- Potentiometer Tank 1 : Scales the amplified signal of the liquid level sensor of tank 1 in a range of 0% to 100%.
- Switch Tank 1 : The switch setting "0%" simulates a complete failure of the liquid level sensor of tank 1. This is accomplished by grounding the voltage coming from the sensor amplifier before it is scaled by the corresponding potentiometer.
- Potentiometer Tank 2 : Scales the amplified signal of the liquid level sensor of tank 2 in a range of 0% to 100%.
- Switch Tank 2 : The switch setting "0%" simulates a complete failure of the liquid level sensor of tank 2. This is accomplished by grounding the voltage coming from the sensor amplifier before it is scaled by the corresponding potentiometer.
- Potentiometer Tank 3 : Scales the amplified signal of the liquid level sensor of tank 3 in a range of 0% to 100%.
- Switch Tank 3 : The switch setting "0%" simulates a complete failure of the liquid level sensor of tank 3. This is accomplished by grounding the voltage coming from the sensor amplifier before it is scaled by the corresponding potentiometer.
- Potentiometer Q 1 : Scales the control signal for the servo controller of pump 1 in a range of 0% to 100% (this is ineffective for the switch setting "Manual" of servo controller 1).
- Potentiometer Q 2 : Scales the control signal for the servo controller of pump 2 in a range of 0% to 100% (this is ineffective for the switch setting "Manual" of servo controller 2).

1.5 Connecting the System Components

1.5.1 Option 200-02

If your DTS200 is equipped with option 200-02 a personal computer is used as a controller. The PC plug-in card included in option 200-02 must be installed in your PC-AT computer. Please consult the manual of your computer for the installation of an additional card. All neccessary components are included with the shipment of option 200-02. Please consider that the preadjusted basis address of 300 Hex must not be occupied by another card.

You can change this address on the card (see corresponding chapter of the PC plug-in card), but then you will have to enter the changed address by means of the dialog to configure the card driver (see chapter program operation).

After the personal computer has been set up and put into operation (for information consult the corresponding manual) it can simply be connected to the jack "PC-Connector" at the rear panel of the actuator. Plug in the 50-pol. lead for the connection between the actuator and the PC plug-in card here.

1.5.2 Option 200-05

Option 200-05 is an electrical disturbance module which can operate together with option 200-02. At the time of delivery this module is already mounted in the box of the actuator.

Adjust the potentiometers for the signals Tank 1, Tank 2 and Tank 3 to 100%. To start operation, turn the potentiometers for the control signals Q 1 and Q 2 to 100%. None of the toggle switches is switched to the position "0%" (switch knob down).

When all potentiometers are turned to 100%, the actuator functions without the disturbance module.

1.6 Output Stage Release

If you use your own 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 the pumps stop immediately.

You need two digital signals for the output stage release. DO1 (pin 35 of the 50-pol. connector) gets first a high-level with pulse to low, duration 40 - 100 µs. After going high DO2 (pin 36 of the 50-pol. connector) needs within the next 100 ms a rect-signal in the range of 10 Hz and 1 kHz. (see fig. 1.3)
If you want to switch off this safety function move jumper JP4 about one position in direction to the integrated circuit IC2 (see Figure 1.4).

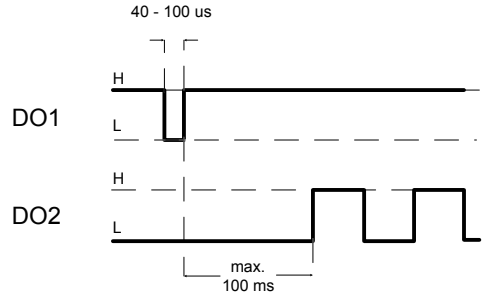


Figure 1.3 : Signals for the output stage release

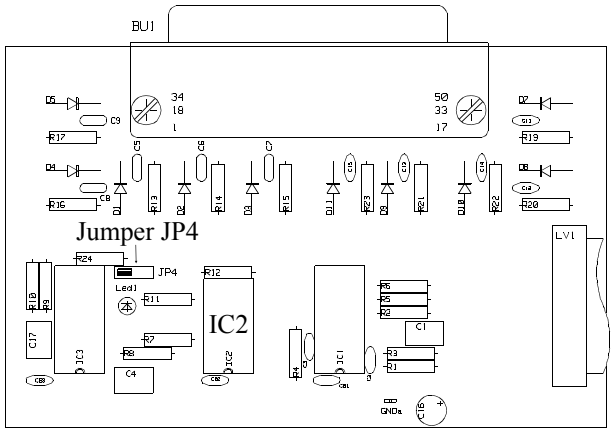


Figure1.4: Signal distribution, output stage release

1.7 Filling the Tank System

In order to avoid deposition of calcium, the tank system should only be filled with deionized or distilled water (total volume of about 40 litres). The system is filled by using the upper pump. Unscrew the left spigot nut of the upper pump and connect the delivered filling tube here. Close all four drain valves and open both connecting valves.

Connect the system cable of the tank system to the actuator. Connect the actuator to the required supply voltage. There is no need to connect a PC. Set the switch of the left SERVO-amplifier to the "Manual"-position and switch on the actuator. By means of the potentiometer, located below the switch, the pump is controlled. Fill each tank to a height of approx. 60 cm.

Using the delivered colorant you are able to fix the depth of colour of the water. This fluid contains beside the food colour also antifungal and germicidal substances. It can be added drop by drop to the water.

Open the four drain valves. Now there are approx. 28 liters of water inside the main reservoir. Close the drain valves again and fill the three tanks to height of approx. 25 cm. Now the system contains the required 40 liters of water.

Remove the filling tube and connect again the original tube to the pump by screwing the spigot nut. Make sure that the screwed connection is watertight.

1.8 Venting the Pressure Sensor Lines

After filling the system with water, venting of the pressure sensor lines inside the base of each tank is strictly recommended. To start with this operation, each tank has to be filled up to approx. 20 cm by controlling the pumps manually. At the rear side of each tank the pressure sensor is screwed normally to the tanks base. A drilled hole leading to the threaded connection of the pressure sensor is closed by a 4mm screw. This sealing is to be unscrewed until water is flowing from the bottom of the tank into the drilled hole inside its base. The screw is tightened again when water is flowing out of the tanks base at the screw location. Carrying-out this operation for each tank completes venting the pressure sensor lines.

1.9 Maintenance

The cover of the reservoir on which the three cylindrical tanks and the pumps are mounted must not be removed!

To avoid calcium deposition in the tank system or growth of algae, the system should only be used with distilled or deionized water. The plexiglas can be cleaned with a standard cleaner for synthetic material and a soft, lint-free cloth.

The pumps and all other components do not require maintenance.

1.10 Transport

The filled tank system must not be transported! It can be emptied by means of one of the pumps. Open the four drain valves so that all the water can flow into the main reservoir. Unscrew the right spigot nut of the upper pump. It should be taken into account that a little bit of water flows out of the dismantled tube. Connect the delivered filling tube to the pump. The corresponding pump can be manually controlled on the actuator. There is no need to connect a PC. The remaining water below the syphon tube of the container can remain in the system during transportation. If the system is not used for a longer time, it is recommended to drain off the remaining water using the drain valve at the back, right-hand side of the system. Naturally the total volume of water can be drained off using this valve.

1.11 Start-Up and Test of Function

After all components of the DTS200 have been connected and adjusted as described in the previous sections, you can start-up the system according to the following procedure:

1. Fill the tank system (1.7).

2. Turn off all components and connect the mains supply.

3. When Option 200-02 is used, turn on the personal computer and start the installation program SETUP.EXE from the enclosed floppy disk with Windows 3.1 or Windows 95/98/ME (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 **DTS20W16.EXE** may be started immediately.

4. When the electrical disturbance module (Option 200-05) is used, the potentiometers for the control signals and the sensors must be adjusted to 100%.

5. Turn on the actuator.

At this time the controller is still not started. Toggle the switches at the servo amplifiers to "Manual". Using the potentiometers located below the switches you can now test the function of the pumps. At the same time the sensors are testet by measuring the output voltages of the sensor amplifiers at the measurement outputs. The voltages have to be in a range between -10V and +10V.

With this, start-up of your DTS200 is complete if you don't use the PC controller program of opt. 200-02.

To test the DTS200 with the controller program of opt. 200-02 please ensure that the switches of the servo amplifiers are turned to the position "Automatic". Select the item "Open Loop Control" from the menu "Run" so that the current readings from the system are displayed in the monitor window. The readings for the levels should be around zero in case of empty tanks. To check the pumps and sensors please select the item "Adjust Setpoint" from the menu "Run" and acknowledge just with "OK" by pressing "Return". The pumps should fill Tank 1 and Tank 2 with a flow rate of 30 resp. 15 ml/s and the sensor readings should increase accordingly.

Differences between the water levels that are shown on the screen and those of the real plant are due to the fact that the sensors are not yet calibrated.

1.12 Locating Errors

First try to eliminate faults with the help of the following table. In case you cannot solve problems with your DTS200 yourself, please contact us.

Problem	Possible cause
The LEDs do not light up	Mains supply switched on? Check the connection to the mains supply. Check the fuse for the mains supply (rearpanel).
Green LEDs voltage supply do not light up	Check the fuses on the mains supply card.
Green LED of Servo-Module does not light up	Check the secondary fuse of the corresponding servo module.
Limit LEDs of the servos light up, without exceeding the limit of the liquid level (60 cm)	Check the lead connection between actuator and tank system.
Controller does not work	Check the connection between actuator and PC.

Practical Instructions

Date: 16. December 1998

1 Introduction

The control of nonlinear systems, particularly multi-variable systems, plays a more and more important part in the scope of the advancing automation of technical processes. Due to the ever increasing requirements of process control (e.g. response time, precision, transfer behavior) nonlinear controller designs are neccessary. One design method with practical applications was developed among others by Prof. Dr.–Ing. E. Freund and successfully used for trajectory control of robots. In this context it is about the so-called principle of nonlinear control and decoupling. The decoupling refers to the input/output behavior, i.e.: after successful decoupling every input affects only the corresponding output. Thus it is possible, to subdivide the multi-variable system into subsystems which are mutually decoupled. These subsystems are linear, i.e. simple to analyze. By means of determining freely choosable parameters, the transfer behavior of the subsystems can be designed subject to some restrictions.

During this laboratory experiment, the application of this principle to a three–tank–system with two inputs and three measurable state variables is to be examined. To that end, the step response and the disturbance behavior of the control system following the controller design is analyzed and compared with those of a standard PI-controlled system.

Because the theroretical foundations are formulated in a general manner, you should be able to apply the "Nonlinear System Decoupling and Control" even to more complicated systems after carrying out the experiment.

2 Theoretical Foundations

The nonlinear control and decoupling is applicable to nonlinear, time-variant multi-variable systems. The system description must have the following form:

$\dot{\underline{x}}(t)=\underline{A}(\underline{x},t)+\underline{B}(\underline{x},t)\,\underline{u}(t)$ Eq.2.1

$\underline{y}(t)=\underline{C}(\underline{x},t)+\underline{D}(\underline{x},t)\,\underline{u}(t)$ Eq.2.2

with the initial conditions: $\underline{x}(t_0)=\underline{x}_0$ and the definitions:

$\underline{x}(t)$: state vector with dimension n $(x_1(t),x_2(t),....,x_n(t))$

$\underline{u}(t)$: input vector of dimension m

$\underline{y}(t)$: output vector of dimension m

$\underline{A}(\underline{x},t)$: (n x 1)–matrix

$\underline{B}(\underline{x},t)$: (n x m)–matrix

$\underline{C}(\underline{x},t)$: (m x 1)–matrix

$\underline{D}(\underline{x},t)$: (m x m)–matrix

A fundamental requirement is that the numbers of system inputs and outputs must be identical.

Now a state feedback in the following form is introduced:

$\underline{u}(t)=\underline{F}(\underline{x},t)+\underline{G}(\underline{x},t)\,\underline{w}(t),$ Eq.2.3

where:

$\underline{F}(\underline{x},t)$: column vector of dimension m

$\underline{G}(\underline{x},t)$: in $(\underline{x},t)$ non-singular (m x m)–matrix

$\underline{w}(t)$: new m–dimensional input vector (setpoint vector)

Figure 2.1 shows graphically the closed loop system with $\underline{D}=0$:

Now one has to determine $\underline{F}$ and $\underline{G}$ such that the i–th input w_i (i=1,...,m) only affects the i–th output y_i. Moreover, it is possible to adjust the dynamics of the decoupeled subsystems by placing the poles.

The difference order d_i referring to the i–th output signal y_i is of great importance for the decoupling. The difference order indicates which total time derivative of the output y_i is directly affected by the input u_i. It is a measure of the number of poles that can be placed

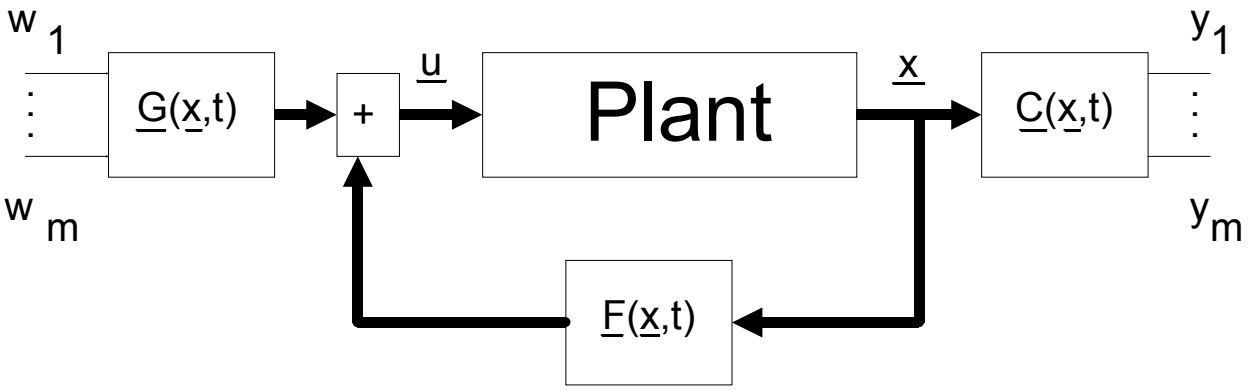


Figure 2.1 : Block diagram with system decoupling

arbitrarily.

In order to define the difference order, in the following the i–th component of the output vector is used instead of all the m components:

$$y_i(t)=C_i(\underline{x},t)+\underline{D_i}(\underline{x},t)\,\underline{u}(t) \qquad \text{Eq.2.4}$$

where:

$C_i(\underline{x},t)$: i–th component of the vector $\underline{C}(\underline{x},t)$

$\underline{D_i}(\underline{x},t)$: i–th row vector of the matrix $\underline{D}(\underline{x},t)$

In case $\underline{D_i}(\underline{x},t) \neq \underline{0}$, the difference order d_i is now defined as follows:

$d_i=0$

In this case, as will be shown later, decoupling is possible using direct compensation.

In case $\underline{D_i}(\underline{x},t)=\underline{0}$, the result is:

$d_i>0$

In order to determine the exact value in this case, the i–th output equation:

$$y_i=C_i(\underline{x},t) \qquad \text{Eq.2.5}$$

is repeatedly differentiated with respect to time, which is shown exemplaryly in the following. The first diffentiation yields:

$$\begin{aligned} y_i'(t) &= d/dt\,[C_i(\underline{x},t)] \\ &= \partial/\partial t\, C_i(\underline{x},t) + \partial/\partial \underline{x}\,[C_i(\underline{x},t)]\,\underline{x}'(t) \end{aligned} \qquad \text{Eq.2.6}$$

where:

$$\partial/\partial \underline{x}\,[C_i(\underline{x},t)] = (\partial/\partial x_1\,[C_i(\underline{x},t)],....,\partial/\partial x_m\,[C_i(\underline{x},t)]) \qquad \text{Eq.2.7}$$

Substituting Eq.2.1 into Eq.2.7 results in:

$$\begin{aligned} y_i'(t) &= \partial/\partial t\, C_i(\underline{x},t) \\ &\quad + \{\partial/\partial x\,[C_i(\underline{x},t)]\}\,\{\underline{A}(\underline{x},t)+\underline{B}(\underline{x},t)\,\underline{u}(t)\} \\ &= \partial/\partial t\, C_i(\underline{x},t) \\ &\quad + \{\partial/\partial \underline{x}\,[C_i(\underline{x},t)]\}\,\underline{A}(\underline{x},t) \\ &\quad + \{\partial/\partial \underline{x}\, C_i(\underline{x},t)\}\,\underline{B}(\underline{x},t)\,\underline{u}(t) \end{aligned} \qquad \text{Eq.2.8}$$

For simplification, now the following nonlinear, time-variant operator $N_A^{\,k}$ is defined:

$$\begin{aligned} N_A^{\,k}\, C_i(\underline{x},t) &= \partial/\partial t\,(N_A^{\,k-1}\, C_i(\underline{x},t)) \\ &\quad + \{\partial/\partial \underline{x}\,(N_A^{\,k-1}\, C_i(\underline{x},t))\}\,\underline{A}(\underline{x},t) \end{aligned} \qquad \text{Eq.2.9}$$

where:

k=1,2,...

and the initial value:

$$N_A^{\,0}\, C_i(\underline{x},t) = C_i(\underline{x},t)$$

Using this operator in Eq.2.8, the result is:

$$\begin{aligned} y_i'(t) &= N_A^{\,1}\, C_i(\underline{x},t) \\ &\quad + [\partial/\partial \underline{x}\,(N_A^{\,0}\, C_i(\underline{x},t))]\,\underline{B}(\underline{x},t)\,\underline{u}(t) \end{aligned} \qquad \text{Eq.2.10}$$

In case:

$$[\partial/\partial \underline{x}\,(N_A^{\,0}\, C_i(\underline{x},t))]\,\underline{B}(\underline{x},t) \neq \underline{0},$$

the difference order $d_i=1$ is defined. If in other case this term is equal to the null vector, the second diffentiation must be carried out:

$$\begin{aligned} y_i''(t) &= N_A^{\,2}\, C_i(\underline{x},t) \\ &\quad + [\partial/\partial \underline{x}\,(N_A^{\,1}\, C_i(\underline{x},t))]\,\underline{B}(\underline{x},t)\,\underline{u}(t) \end{aligned} \qquad \text{Eq.2.11}$$

Proceeding in this manner, the final result of the j–th differentiation is:

$$y_i^{(j)} = N_A^{\,j}\, C_i(\underline{x},t) + [\partial/\partial \underline{x}\,(N_A^{\,j-1}\, C_i(\underline{x},t))]\,\underline{B}(\underline{x},t)\,\underline{u}(t)$$

where eventually:

$$[\partial/\partial \underline{x}\,(N_A^{\,j-1}\, C_i(\underline{x},t))]\,\underline{B}(\underline{x},t) \neq \underline{0}$$

for the first time. This means, the input vector $\underline{u}(t)$ directly affects the j–th differentiation of the output signal $y_i(t)$. With this, the difference order of the i–th subsystem results in:

$d_i = j$

In summary the difference order can be defined as follows:

a) if $\underline{D}_i(\underline{x},t) \neq \underline{0} : d_i=\underline{0}$ Eq.2.12

b) if $\underline{D}_i(\underline{x},t) = \underline{0} : d_i=\min\{j:[\partial/\partial\underline{x}\,(N_A^{j-1}\,C_i(\underline{x},t))]$
$\underline{B}(\underline{x},t)\neq\underline{0}\}$ Eq.2.13

For simplification it is assumed in the following , that d_i (i=1,..,m) is constant with respect to all $\underline{x}(t)$ and t.

Using the above mentioned method for all m outputs of the nonlinear, time-variant system, the result is:

$\underline{y}^*(t)=\underline{C}^*(\underline{x},t)+\underline{D}^*(\underline{x},t)\,\underline{u}(t)$ Eq.2.14

where:

$\underline{y}^*(t)=(y_1^{(d1)}(t),....,y_m^{(dm)}(t))^T$

$\underline{C}^*(\underline{x},t)$: m–dimensional column vector

$\underline{D}^*(\underline{x},t)$: (m x m)–matrix;

In this case the i–th component of $\underline{C}^*(\underline{x},t)$ can be stated as follows:

$C^*_i(\underline{x},t)=N_A^{di}\,C_i(\underline{x},t)$ Eq.2.15

Moreover the i–th row vector of the matrix $\underline{D}^*(\underline{x},t)$ is defined by:

$$\underline{D}^*_i(\underline{x},t)=\begin{cases}\underline{D}_i(\underline{x},t) & \text{für } d_i=0\\ [\delta/\delta\underline{x}\,N_A^{di-1}\,C_i(\underline{x},t)]\,\underline{B}(\underline{x},t) & \text{für } d_i\neq 0\end{cases}$$

Eq.2.16

On the assumption that the rank of the matrix $\underline{D}^*(\underline{x},t)$ is constant, all its row vectors are unequal to the null vector.

Eq.2.14 is the initial equation to derive the decoupling matrices and the introduction of assignable dynamics of the decoupled subsystems.

If the relation:

$\underline{u}(t)=\underline{F}_1(\underline{x},t)=-\underline{D}^{*-1}(\underline{x},t)\,\underline{C}^*(\underline{x},t)$ Eq.2.17

is substituted in Eq.2.14, the result is:

$\underline{y}^*(t)=\underline{0}$ Eq.2.18

This means, each of the m outputs of the multi–variable system is decoupled. Extending the above relation in the following manner:

$\underline{u}(t)=\underline{F}_1(\underline{x},t)+\underline{G}(\underline{x},t)\,\underline{w}(t)$ Eq.2.19

where:

$\underline{G}(\underline{x},t)=\underline{D}^{*-1}(\underline{x},t)\,\underline{L}$ Eq.2.20

$\underline{L}=\text{diag}\{l_i\}$, (i=1,2,...,m)

yields:

$\underline{y}^*(t)=\underline{L}\,\underline{w}(t)$ Eq.2.21

This form of feedback additionally allows a free adjustment of the input amplification of the i–th subsystem. In order to make it possible to influence the dynamic of the decoupled subsystem the above relation is changed as follows:

$\underline{u}(t)=\underline{F}(\underline{x},t)+\underline{G}(\underline{x},t)\,\underline{w}(t)$ Eq.2.22

where:

$\underline{F}(\underline{x},t)=\underline{F}_1(\underline{x},t)+\underline{F}_2(\underline{x},t)$ Eq.2.23

$\underline{F}_2(\underline{x},t)=-\underline{D}^{*-1}(\underline{x},t)\,\underline{M}^*(\underline{x},t)$ Eq.2.24

and

$\underline{M}^*(\underline{x},t)$: m–dimensional vector

Substituting this relation in Eq.2.14 yields:

$$\underline{\dot{y}}^{*}(t)=-\underline{M}^{*}(\underline{x},t)+\underline{L}\,\underline{w}(t) \qquad \text{Eq.2.25}$$

The vector $\underline{M}^{*}(\underline{x},t)$ has to be determined such that the dynamic of the m subsystems can be changed using pole shifting and new couplings are avoided as well.

A suitable relation of the i–th component of the vector $\underline{M}^{*}(\underline{x},t)$ can be stated as

$$M^{*}_{i}(\underline{x},t)=\begin{cases}0 & \text{für } d_i=0\\ \Sigma a_{ki}\, N_A^k\, C_i\,(\underline{x},t) & \text{für } d_i\neq 0\end{cases} \qquad \text{Eq.2.26}$$

The constant factors a_{ki} are freely assignable with:

$i=1,2,..,m$ und $k=0,1,..,(d_i-1)$

In can be shown (substituting Eq.2.26 in Eq.2.25) that each component of Eq.2.25 with di≠0, using the above choice for $M^{*}_{i}(\underline{x},t)$, can be described in the following form:

$$y_i^{(di)}(t)+a_{(di-1)i}y_i^{(di-1)}(t)+...+a_{0i}y_i(t)=l_i\,w_i(t) \qquad \text{Eq.2.27}$$

Accordingly each subsystem results in a linear differential equation of d_i–th order.

Thus, the nonlinear, time–variant, multi–variable system is decoupled. The transfer behavior of the subsystems with $d_i\neq 0$ is described by Eq.2.27.

In summary $\underline{F}$ and $\underline{G}$ are determined as follows:

$$\underline{F}(\underline{x},t)=-\underline{D}^{*-1}(\underline{x},t)\,\{\underline{C}^{*}(\underline{x},t)+\underline{M}^{*}(\underline{x},t)\}$$
$$\underline{G}(\underline{x},t)=\underline{D}^{*-1}(\underline{x},t)\,\underline{L}$$

Moreover it has to be considered that the difference order d_i is invariant with respect to a transformation of the system into different coordinate systems. The requirements of the determination of d_i however depend strongly on the choice of the basis system. This is also true with respect to the further steps of determining the algorithms of decoupling and control.

3 The System "Three-Tank-System"

3.1 The Plant

The following figure 3.1 shows the principal structure of the plant.

The used abbreviations are described in the following.

The plant consits of three plexiglas cylinders T1, T3 and T2 with the cross section A. These are connected serially with each other by cylindrical pipes with the cross section S_n. Located at T2 is the single so called nominal outflow valve. It also has a circular cross section S_n. The outflowing liquid (usually distilled water) is collected in a reservoir, which supplies the pumps 1 and 2. Here the circle is closed.

H_{max} denotes the highest possible liquid level. In case the liquid level of T1 or T2 exceeds this value the corresponding pump will be switched off automatically. Q_1 and Q_2 are the flow rates of the pumps 1 and 2.

Technical data:

— A=0.0154 m^2
— S_n=5*10^{-5} m^2
— H_{max}=62cm (+/– 1cm)
— Q_{1max}=Q_{2max}=100mltr/sec=6.0ltr/min

The remarkable feature of the used "Three-section-diaphragm-pumps" with fixed piston stroke (Pump 1 and 2) is the well defined flow per rotation. They are driven by DC motors.

For the purpose of simulating clogs or operating errors, the connecting pipes and the nominal outflow are equipped with manually adjustable ball valves, which allow to close the corresponding pipe.

For the purpose of simulating leaks each tank additionally has a circular opening with the cross section S_l and a manually adjustable ball valve in series. The following pipe ends in the reservoir.

The pump flow rates Q_1 and Q_2 denote the input signals, the liquid levels of T1 and T2 denote the output signals, which have to be decoupled, of the controlled plant. The

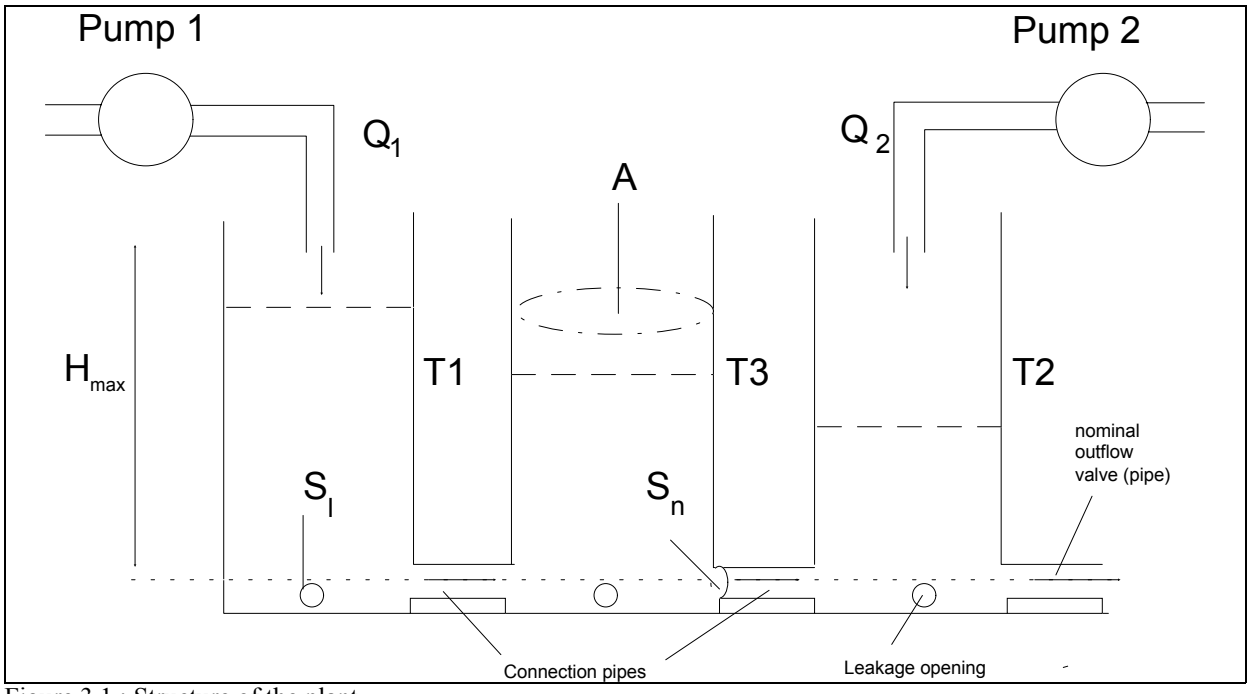


Figure 3.1 : Structure of the plant

necessary level measurements are carried out by piezo-resistive difference pressure sensors. At each of it a pressure line measures the pressure of the atmosphere.

3.1.1 The Controller

Here a digital controller realized on a microcumputer system is used. The connection to the controlled plant is carried out by 12-bit converters.

The A/D–converter is required to read the liquid levels of the tanks T1, T3 and T2. It is initialized for the voltage range: –10V...10V. This results in a resolution of 4.88mV. Two D/A–converters are used to control the pumps. The following figure 3.2 shows the principal structure of the complete control loop.

The control software is designed such that the decoupled subsystems can be controlled additionally by PI–controllers. Furthermore a PI–control of the liquid levels of tank 1 and 2 is possible even without decoupling. The respective controller configuration can be seen at the screen.

Each controller adjustment as well as the setpoints can be changed on–line.

The control signals and the state vector components (i.e. the pump flow rates and the liquid levels) can be stored in the RAM of the microcomputer and can be plotted afterwards on a plotter. The controller stays active during the graphical output, but inputs cannot be carried out during this time.

Operating the program will be described later in detail.

3.1.2 The Signal Adaption Unit

The purpose of the signal adaption unit is to adapt the voltage levels of the plant and the converter to each other. Here the output voltages of the sensors are adapted to the maximum resolution of the A/D– converters and on the other hand the output voltage range of the D/A–converters is adapted the servo amplifier of the corresponding pump.

3.1.3 The Disturbance Modul

Using switches, sensor failures of the level measurement can be simulated. The corresponding sensor output

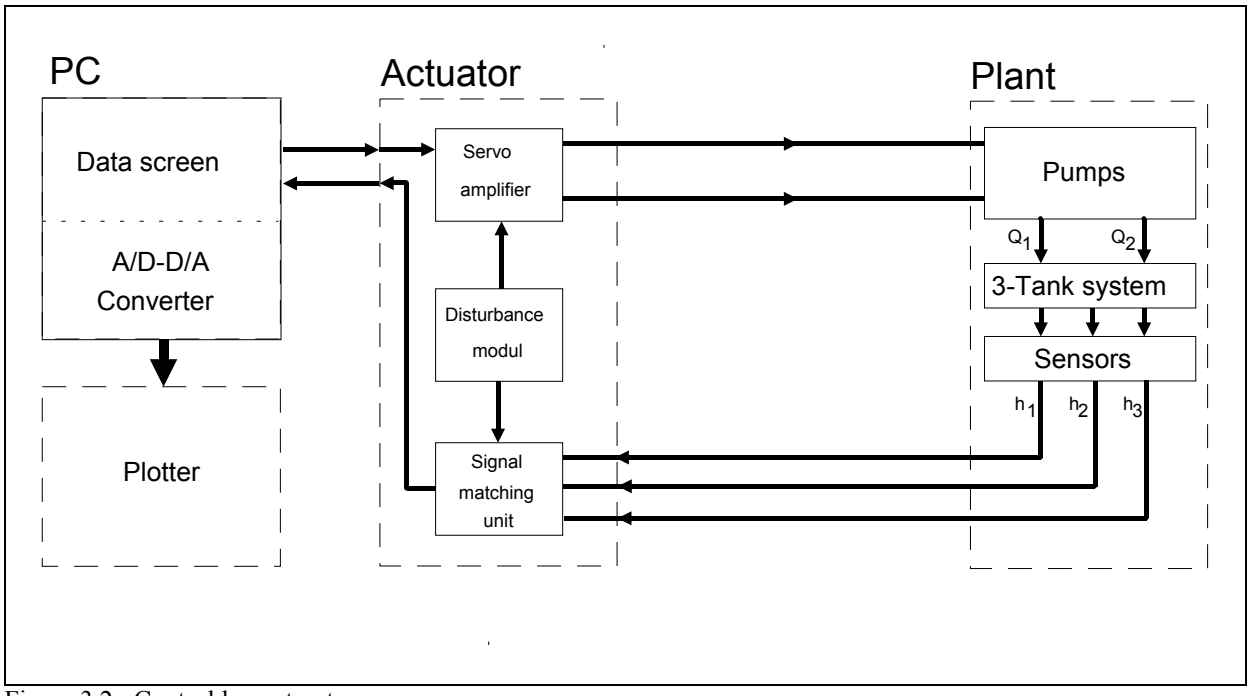


Figure 3.2 : Control loop structure

supplies a voltage which corresponds to the liquid level 0.

Alternatively it is possible to scale the measured liquid level between the real height (100%) and the height 0 (0%) using a potentiometer.

Two other potentiometers allow to simulate defects of the pumps. To do this the control signal can be reduced up to 0% of its original value, which is equivalent to decreasing the pump flow rate.

3.2 Mathematical Model

The following figure 3.3 once again shows the structure of the plant to define the variables and the parameters.

az_i : outflow coefficients [ ]

h_i : liquid levels [m]

Q_{ij} : flow rates [m^3/sec]

Q_1,Q_2 : supplying flow rates [m^3/sec]

A : section of cylinder [m^2]

S_l : section of leak opening [m^2]

S_n : section of connection pipe [m^2]

where i=1,2,3 and (i,j)=[(1,3);(3,2);(2,0)]

If the balance equation:

A dh/dt=Sum of all occuring flow rates Eq.3.1

is used for all of the three tanks, the result is:

$A\ dh_1/dt=Q_1-Q_{13}$ Eq.3.2

$A\ dh_3/dt=Q_{13}-Q_{32}$ Eq.3.3

$A\ dh_2/dt=Q_2+Q_{32}-Q_{20}$ Eq.3.4

The still unknown quantities Q_{13}, Q_{32} and Q_{20} can be determined using the generalized Torricelli–rule. It can be stated as

$q=az\ S_n\ sgn(\Delta h)\ (2g\ |\Delta h|)^{1/2}$ Eq.3.5

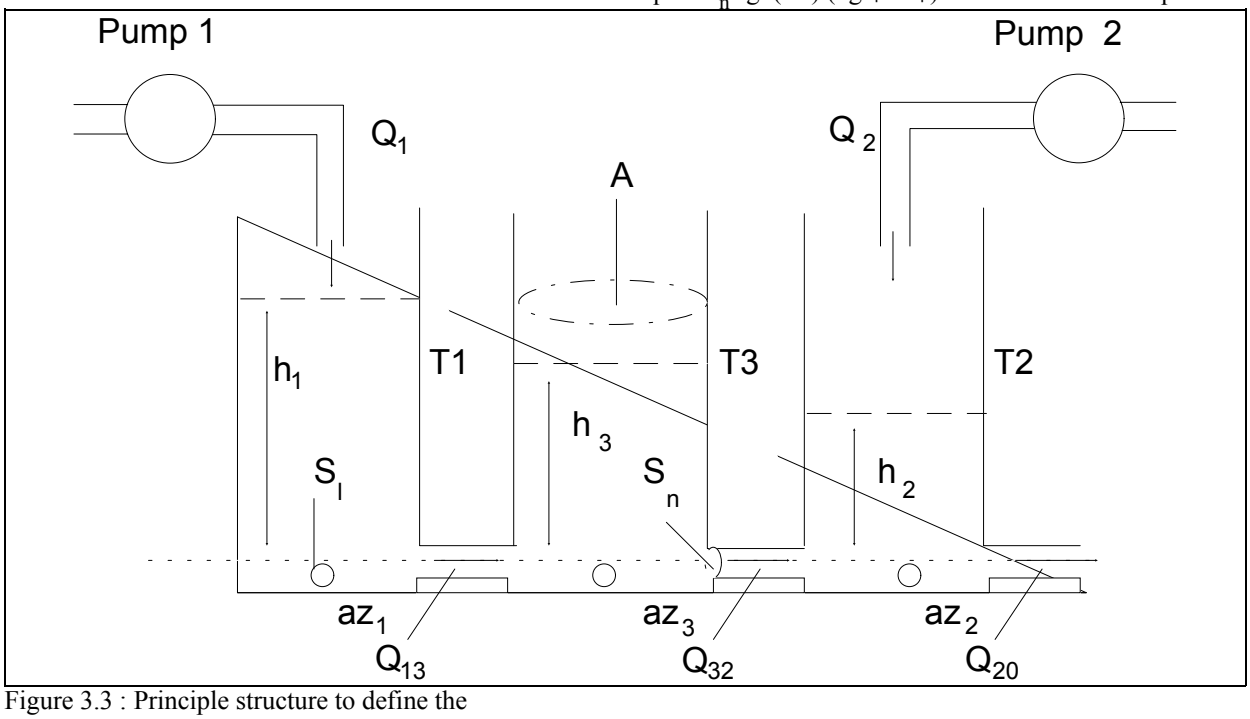


Figure 3.3 : Principle structure to define the variables and parameters

where:

g : earth acceleration

sgn(z) : sign of the argument z

Δh : liquid level difference between two tanks connected to each other

az : outflow coefficient (correcting factor, dimensionless, real values ranging from 0 to 1)

q : resulting flow rate in the connecting pipe

So the result for the unknown quantities is:

$$Q_{13}=az_1\, S_n\, \mathrm{sgn}(h_1-h_3)\,(2g\,|h_1-h3|)^{1/2} \qquad \text{Eq.3.6}$$

$$Q_{32}=az_3\, S_n\, \mathrm{sgn}(h_3-h_2)\,(2g\,|h_3-h2|)^{1/2} \qquad \text{Eq.3.7}$$

$$Q_{20}=az_2\, S_n\,(2g\,h_2)^{1/2} \qquad \text{Eq.3.8}$$

With the vector definitions:

$$\underline{h}=(h_1,h_2,h_3)^T \qquad \text{Eq.3.9}$$

$$\underline{Q}=(Q_1,Q_2)^T \qquad \text{Eq.3.10}$$

$$\underline{A}(\underline{h})=(-Q_{13},Q_{32}-Q_{20},Q_{13}-Q_{32})^T\ 1/A \qquad \text{Eq.3.11}$$

$$\underline{y}=(h_1,h_2)^T \qquad \text{Eq.3.12}$$

and the matrix definition:

$$\underline{B}=\frac{1}{A}\begin{bmatrix}1 & 0\\ 0 & 1\\ 0 & 0\end{bmatrix} \qquad \text{Eq.3.13}$$

the above system of equations can be transformed to:

$$d\underline{h}/dt=\underline{A}(\underline{h})+\underline{B}\,\underline{Q} \quad \text{: state equation} \qquad \text{Eq.3.14}$$

$$\underline{y}=(h_1,h_2)^T \quad \text{: output equation} \qquad \text{Eq.3.15}$$

With this the plant is described completely. Because the physical values h and Q are not accessible directly, converters and sensors are used. These are parts of the plant and are described using their characteristics (characteristics of pumps and sensors). Furthermore the values of the outflow coefficients are required to get a quantitative description of Eq.3.14 and to carry out the nonlinear decoupling in practice. Therefore, the neccessary data are determined interactively using the control program before the actual control starts.

The model described by Eq.3.14/Eq.3.15 is now used as the initial basis for the nonlinear decoupling of the Three–Tanks–System. The corresponding controller design is carried out within the framework of the preparations. The solutions of these preparations are therefore the fundamental basis to carry out the experiment.

4 Preparations for the Laboratory Experiment

4.1 Controller Design

a) Determine the difference orders d_1,d_2 of the model as described by Eq.3.14/Eq.3.15.

Hint: Either use the general definition or transform Eq.3.14/Eq.3.15 directly to a form similar to Eq.2.14 with $\underline{D}^*_i(\underline{x},t)$ unequal $\underline{0}$.

What is the behaviour of each subsystem after the decoupling?

b) Determine the vector $\underline{C}^*(\underline{x},t)$ and the matrix $\underline{D}^*(\underline{x},t)$. What is the rank of $\underline{D}^*(\underline{x},t)$? Can the system be decoupled?

c) Determine the vector $\underline{F}_1(\underline{x},t)$ and the matrix $\underline{G}(\underline{x},t)$.

d) Determine the vector $\underline{M}^*(\underline{x},t)$ with freely assignable parameters a_{ki} using a suitable formulation. Determine the vector $\underline{F}_2(\underline{x},t)$.

e) Determine the overall transfer behaviour of the decoupled subsystems. Roughly draw the step responses. What is the condition each of the real controller parameter l_i and a_{ki} has to meet so that the transfer behaviour of the subsystems is stable with an overall amplification of one?

4.2 Disturbance behaviour in case of a leak in tank 2

a) How do the model equations Eq.3.14/ Eq.3.15 change in case of a leak with the section S_l and the outflow coefficient az_l in tank 2 ? The leak is assumed to be located at the level of the connection pipes.

b) What is the behaviour of the subsystems in case the controller design determined in 4.1 is used? To do this, formulate the corresponding differential equations of the systems. Are the subsystems still decoupled?

c) Compute the stationary output value of the 2nd subsystem (output signal h_2) in case of this disturbance and a constant input signal. Is there a steady state difference compared with the disturbance-free case? If so, what is its value?

4.3 PI–Control of the decoupled subsystems

The decoupled subsystems can be PI–controlled additionally in accordance with the following block diagram:

a) Compute the overall transfer function: $G_{gi}(s)=H_i(s)/WE_i(s)$

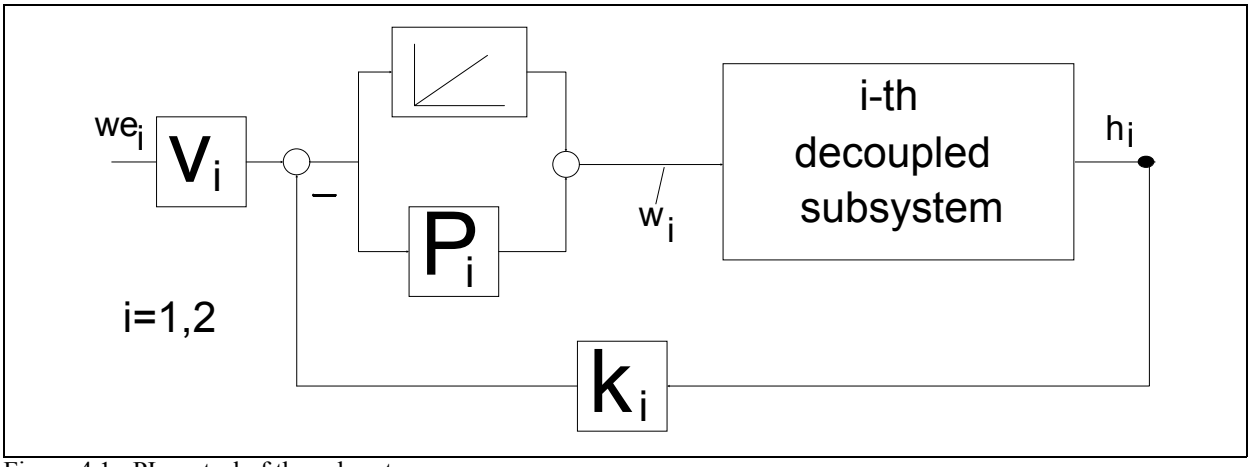


Figure 4.1 : PI-control of the subsystems

b) What is the condition the real parameters v_i,k_i and P_i have to fulfill so that each PI–controlled loop is stable with an overall amplification of one?

c) What is the overall transfer behaviour in case P_i=0?

5 Carrying-Out the Experiment

Start the controller program (further details will be found in the chapter instructions for operating the computer).

5.1 Determination of Characteristics and Outflow Coefficients

Before you start with the experiments the system has to be calibrated. To do this you select the menu item **Parameter** from the upper menu bar. A pulldown menu will appear offering further items for the system calibration.

a) Use the menu item **Characteristic Liquid Level Sensor** to determine the characteristic: liquid level <=> ADC input voltage by experiment. Control the pumps manually in this process (switches to "Manual").

b) Change to the determination of the outflow coefficients by means of the menu item **Outflow Coefficient** and use experimental measuring. Approximately 60 sec after the interactive start, the determination is performed by the program which computes the following equations:

$$az_1=A\ h_1'/(-S_n\ (2g\ (h_1-h_3))^{1/2})$$

$$az_2=A\ (h_1'+h_2'+h_3')/(-S_n\ (2g\ h_2)^{1/2})$$

$$az_3=A\ (h_1'+h_3')/(-S_n\ (2g\ (h_3-h_2))^{1/2})$$

These equations can be derived from the modell equations Eq.3.14 with $Q_1=Q_2=0$ and $h_1>=h_3>=h_2$.
Record the outflow coefficients.

c) Use the menu item **Characteristic Pump Flow Rate** to determine the characteristic: DAC output voltage <=> pump supply using experimental measuring (Do not use the default values!).

Here, according to an amount of 8 DAC output voltages, the difference of levels which is measured in a time intervall of 20 sec is used to determine the pump flow rates (supplies).

Please follow exactly the instructions of the controller program.

After the determination of the plant parameters the menu item **File / Save System Parameter** may be used to store these values to the hard disk.

The window called "DTS200 Monitor" is located in the lower part of the screen. Its first line displays the current system state respectively the selected control structure. The following lines contain current data of the plant and of the controller like liquid levels and flowrates. The status of the plot buffer used to store the measurements is displayed in addition.

A graphical evaluation of any experiment result is possible only if previously the storage of measurements was switched on.

Now open the connection valves and the nominal outflow valve! Start the controller by means of the menu item **RUN**.

5.2 Behavior of Reference and Disturbance Variable without PI-control

The following tasks require the control structure with a decoupling network (select the menu item **Decoupling-Controller** from the menu **RUN**). The accompanying parameter is adjustable by selection of the menu item **Decoupling-Controller** from the pulldown menu **Parameter**. As mentioned with the preparation tasks the parameters are set according to 'Decoup'=$l_i=a_{0i}$ meaning that the amplification of the decoupled subsystems is equal to 1.

a) Step Responses

At first adjust the decoupling parameter to 'Decoup'=0.03 and choose the setpoints w_1=32cm (tank 1) and w_2=20cm (tank 2) (Use the menu item **Adjust Setpoint** from the menu **RUN**, enter the corresponding setpoint values with a constant signal shape and prompt with 'OK'). Wait until the steady state is reached. Now switch on the storing of measured data with a measuring time=240 sec (Use the menu item **Measuring** from the menu **RUN**, enter appropriate values and prompt with 'OK').

Now generate a step of the setpoint w_1 (tank 1) to 37cm (Use the menu item **Adjust Setpoint** from the menu **RUN**, enter the corresponding setpoint value with a constant signal shape and prompt with 'OK'). After the steady state is reached (approximately after 50% of the measuring time) change the decoupling parameter to 'Decoup'=0.04 and generate a step of the setpoint w_2 (tank 2) to 25cm (Use the menu item **Adjust Setpoint** from the menu **RUN**, enter the corresponding setpoint value with a constant signal shape and prompt with 'OK') The measuring is finished when the status line of the plot buffer displays the message "Buffer filled". Using the menu **View** and its menu item **Plot Measured Data** the measured data are displayable in a graphic representation on the screen. The graphic may be sent to a plotter or printer in addition. Furthermore the data may be saved in a file using the menu item **File / Save Recorded Data**.

Notes for the user:

The series of events, storing data and plot output will occur during all of the following experiments. Therefore it will not be mentioned explicitly in the following.

b) Disturbance behaviour in case of leaks

Adjust the decoupling parameter to 'Decoup'=0.2 and the setpoints to 40cm (tank 1) and 15cm (tank 2).

Store measurements (150 sec):
Create a leak in tank 3 by opening the corresponding valve. Close the valve after 30sec. Now use 'Decoup'=0.1sec and create a leak in tank 2. Wait until the steady states of the water levels are reached. Close the valve after this.

c) Disturbance behaviour in case of closed connection valve

Adjust the setpoints to 25cm (tank 1) und 20cm (tank 2) with 'Decoup'=0.1. Wait until the steady states of the water levels are reached.

Store measurements (210sec):
Close the connection valve between tank 3 and tank 2. Wait until the water levels of tank 3 and tank 1 are the same. Now create a step of the setpoint w_1 to 50cm.

At the end of the measuring do not forget to open the connection valve between tank 3 and tank 2 again.

d) Disturbance behaviour in case of sensor failure

If the disturbance modul is not present this experiment cannot be performed.

Choose the parameter 'Decoup'=0.2 and adjust the setpoints to 25cm (tank 1) and 20cm (tank 2).

Store measurements (100sec):
Create a sensor failure in tank 1 until 50% of the measuring time is reached (Knob down, switch "Tank1" of the disturbance module). Then switch on the sensor again.

5.3 Behaviour of the Reference Variable with PI–Control

To obtain an overall amplification of 1 for the closed control loop the controller parameters k_i are set equal to v_i.

a) Reference behaviour of the decoupled, PI–controlled subsystems

Choose the parameter 'Decoup'=0.05 and adjust the parameters of the PI-controller to $P_i = 0$, $K_i = 0.1\text{sec}^{-1}$ (Use the menu item **PI-Controller** from the menu **Parameter**, enter the corresponding values and prompt with 'OK'). Now adjust the setpoints to 30cm (tank 1) and 20cm (tank 2). Then activate the PI–controller with decoupling mode (Select the menu item **PI-Controller** from the pulldown menu **RUN**). Wait until the steady states of the water levels are reached.

Store mesurements (120sec):
Create a step of the setpoint w_1 to 34cm.

b) Reference behaviour of the coupled, PI–controlled subsystems

Activate the decoupling controller by using the menu item **Decoupling Controller**. Adjust the setpoints to 30cm (tank 1) and 15cm (tank 2). Close the connection valve between tank 3 and tank 2. Wait until the water levels h_2 and h_3 are settled to steady state. Adjust the proportional portion of the PI-controller to 0 and select $k_i=0.5\text{sec}^{-1}$.

In the following, only the reference behaviour of the water level h_2 will be considered.

Store measurements (450sec):
Now activate the PI–controller without decoupling. The water levels of tank 1 and tank 2 are now not any longer decoupled; they are controlled be the PI–controller.

Adjust the parameter P_i=15sec after approximately 50% of the measuring time.

6 Evaluation of the Experiments

ref. 5.2a):
Explain the characteristic behaviour and examine the time constants.

ref. 5.2b):
A leak in tank 3 does not change the steady states of the ouptput variables (water levels of tank 1 and tank 2). A leak in tank 2 results in difference of the steady states. Why? Use this difference to calculate the opening of the leak valve. Use a value of 0.7 for the outflow coefficient of the leak valve.

Will a PI–controller settle the leak disturbance in tank 2? Use the differential equations of the system to show this on the assumption that the control loop is stable.

ref. 5.2c):
Explain the characteristic behaviour. Does the nonlinear controller design decouple the subsystems furthermore in case of such a failure?

ref. 5.2d):
Explain the characteristic behaviour. Does the nonlinear controller design furthermore decouple the output variable h_2? What would be the result of a sensor failure in tank 3?

ref. 5.3a):
Will such parameters lead to stable or unstable control loops? Explain the system behaviour.

ref. 5.3b):
Is the control of the output variable h_2 stable if you use P_2=0 or P_2=10sec? Show this by linearization of the plant model of tank 2 around the reference variable and by calculating the transfer function of the control loop.

Which is the order of the linearized plant in the nominal state?

7 Reference

/1/ : E. Freund u. H. Hoyer:
Das Prinzip der nichtlinearen Systementkopplung mit der Anwendung auf Industrieroboter
Zeitschrift :"Regelungstechnik", Nr. 28 im Jahr 1980, S.80–87 und S. 116–126

8 Solutions

8.1 Solutions of the Preparation Problems

ref. 4.1a) and 4.1b):

The equations Eq. 3.14 and Eq. 3.15 can be transformed to:

$$y_1' = h_1' = (-Q_{13} + u_1)/A \qquad \text{Eq.8.1}$$

$$y_2' = h_2' = (Q_{32} - Q_{20} + u_2)/A \qquad \text{Eq.8.2}$$

The result is that the input variable u_i (i=1,2) of each subsystem already affects the first differentiation of the corresponding output variable y_i. It follows:

$$d_1 = d_2 = 1 \qquad \text{Eq.8.3}$$

So the subsystems behave like a first order lag.

A comparison with Eq.3.14 yields:

$$\underline{C}^* = \frac{1}{A}\begin{bmatrix} -Q_{13} \\ Q_{32} - Q_{20} \end{bmatrix} \qquad \text{Eq.8.4}$$

$$\underline{D}^* = \frac{1}{A}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \text{Eq.8.5}$$

D^{*-1} exists, because the rank $[\underline{D}^*]=2$. So the system can be decoupled.

ref. 4.1c):

with:

$$\underline{D}^{*-1} = A\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \text{Eq.8.6}$$

follows:

$$\underline{F}_1(\underline{x},t) = -A\,\underline{C}^* \qquad \text{Eq.8.7}$$

Using $\underline{L}=\mathrm{diag}\{l_i\}$ (i=1,2) the matrix $\underline{G}(\underline{x},t)$ can be determined:

$$\underline{G}(\underline{x},t) = A\begin{bmatrix} I_1 & 0 \\ 0 & I_2 \end{bmatrix} \qquad \text{Eq.8.8}$$

ref. 4.1d):

Because of $d_i=1$, an additional parameter as a freely placeable pole can be introduced in each subsystem. Using the formulation:

$$\underline{M}^*(\underline{x},t) = (a_{01}\,x_1\,,\, a_{02}\,x_2)^T \quad ;\ (a_{01},a_{02}) \text{ arb.} \qquad \text{Eq.8.9}$$

yields:

$$\underline{F}_2(\underline{x},t) = -A\,\underline{M}^*(\underline{x},t) \qquad \text{Eq.8.10}$$

ref. 4.1e):

If Eq.8.3 with $\underline{F}=\underline{F}_1+\underline{F}_2$ is substituted in Eq.8.41, the result is:

$$h_i' = -a_{0i}\,h_i + l_i\,w_i \quad ;\ i=1,2 \qquad \text{Eq.8.11}$$

This results in the transfer function $G_i(s)$:

$$G_i(s) = l_i/(s+a_{0i}) \quad \text{(first order lag)} \qquad \text{Eq.8.12}$$

with the step response $\ddot{U}G_i(t)$:

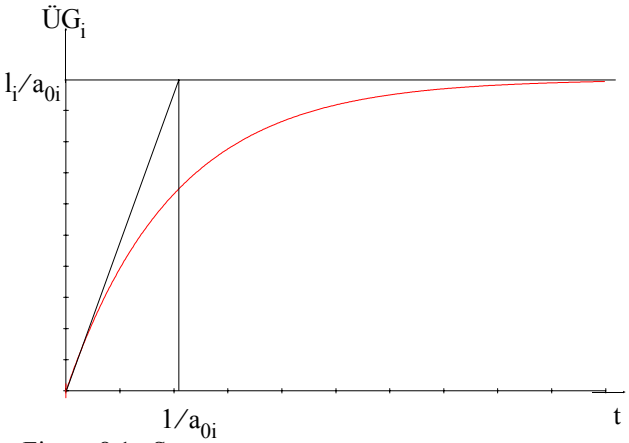


Figure 8.1 : Step response

Every stable subsystem has an overall amplification of one if the following is true:

$l_i=a_{0i}>=0$ Eq.8.13

ref. 4.2a):

A leak in tank 2 results in the following system equations:

$h_1'=(u_1-Q_{13})/A$ (unchanged) Eq.8.14

$h_2'=(u_2+Q_{32}-Q_{20}-Q_{2l}(h_2))/A$ Eq.8.15

where:

$Q_{2l}(h_2)=az_l S_l (2g h_2)^{1/2}$ Eq.8.16

ref. 4.2b):

Using the controller design of 4.1):

$u_1=Q_{13}+A (l_1 w_1-a_{01} h_1)$ Gl8.17

$u_2=Q_{20}-Q_{32}+A (l_2 w_2-a_{02} h_2)$ Gl8.18

the result is:

$h_1'=-a_{01} h_1+l_1 w_1$ (unchanged) Eq.8.19

$h_2'=-a_{02} h_2+l_2 w_2-Q_{2l}(h_2)/A$ Eq.8.20

So the subsystems are decoupled furthermore. The behaviour of h_1 is unchanged. The behaviour of h_2 is described by Eq.8.20.

ref. 4.2c):

The final value of h_2 is given by the condition $h_2'=0$. With $a_{02}=l_2$ the result of Eq.8.20 is:

$h_2'=0 => h_2=h_{2R}=w_2+c/l_2^2-(c (2 w_2+c/l_2^2))^{1/2}/l_2$ Eq.8.21

where:

$c=g az_l^2 S_l^2/A^2$ Eq.8.22

The difference d of the steady state results in:

$d=w_2-h_{2R}=-c/l_2^2+(c (2 w_2+c/l_2^2))^{1/2}/l_2$ Eq.8.23

ref. 4.3a):

With $G_i(s)=l_i/(s+a_{0i})$ and $l_i=a_{0i}$ the result is:

$G_{gi}(s)=v_i l_i (P_i s+1)/NN$ Eq.8.24

where:

$NN=s^2+l_i (k_i P_i+1) s+l_i k_i$ Eq.8.25

ref. 4.3b):

Using the fundamental stability criterion, the control loops are stable if all of the parameters v_i ,k_i ,P_i and l_i have a positive value.

Using an overall amplification of one, the final value theorem of the laplace transformation results in:

$v_i=k_i$ Eq.8.26

ref. 4.3c):

Using $P_i=0$ and $v_i=k_i$ the result is a behaviour of a second order lag with the following characteristic coefficients:

Amplification: $Ver_i=1$ Eq.8.27

Time constant: $Zei_i=1/(l_i k_i)^{1/2}$ Eq.8.28

Damping: $Dae_i=(l_i/k_i)^{1/2}/2$ Eq.8.29

8.2 Solutions to Carrying-Out the Experiment and the Questions

ref. 5.2a):

Figure 8.2 shows the graphic representation of the water levels and figure 8.3 the pump flow rates.
The adjusted time constants with 33.3sec and 25sec can be seen from the characteristics.
The output variables h_1 and h_2 are decoupled. A step of the setpoint w_1 does not change the behaviour of $h_2(t)$ and vice versa.

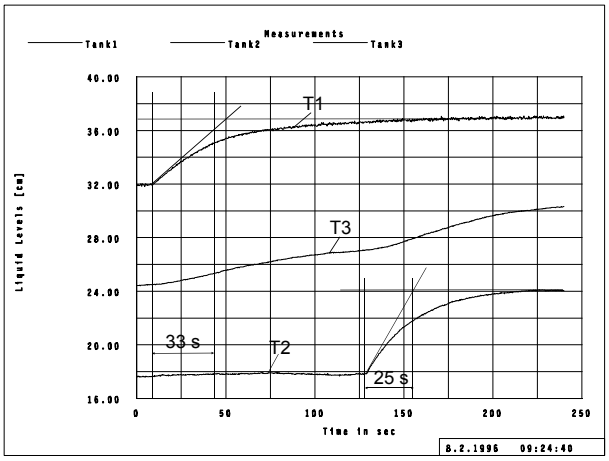


Figure 8.2 : State variables ref. experiment 5.2a)

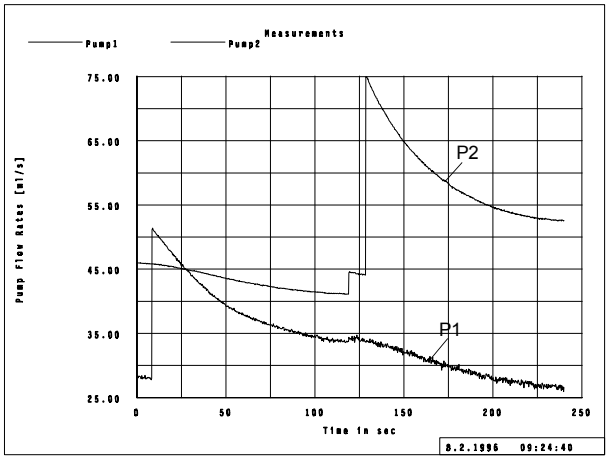


Figure 8.3 : Flow rates ref. experiment 5.2a)

ref. 5.2b):

Figures 8.4 and 8.5 show the resulting graphical representations. A leak in tank 2 was realized with decoupling mode additionally.

A leak in tank 3 does not change the output equations of the controller design (Eq.8.1 und Eq.8.2). h_1 and h_2 behave furthermore according to Eq.8.11.

According to chapter 8.1 a leak in tank 2 results in a difference d of the steady state of the reference variable w_2. If Eq.8.23 is solved for the opening of the leak valve S_l the result is:

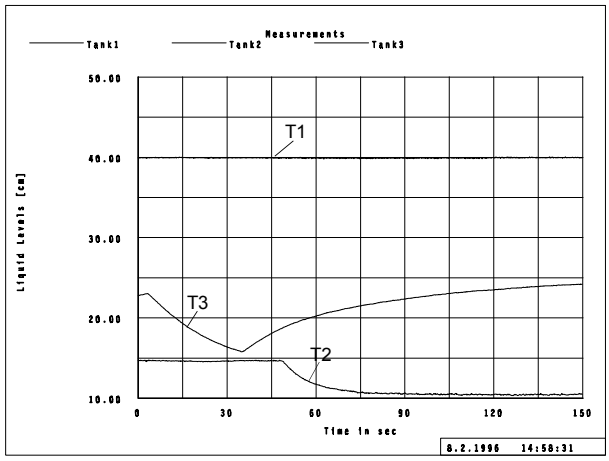


Figure 8.4 : State variables ref. experiment 5.2b)

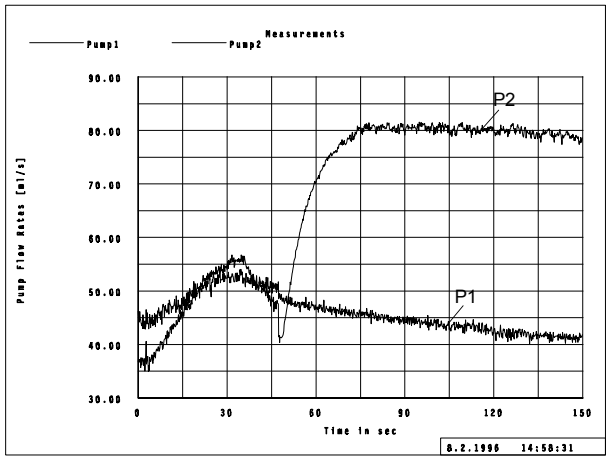


Figure 8.5 : Flow rates ref. experiment 5.2b)

$S_l=A\ (z/g)^{1/2}/az_l$ Eq.8.30

where:

$z=d^2\ l_2^2/(2(w_2-d))$ Eq.8.31

The interactively determined (chapter 5.1) outflow coefficient of tank 2 should have a value in a range of 0.6 ... 0.8. Using a measured value of 0.73 the result is a section of the leak opening as documented in figure 3.3. For comparison: the real opening of the leak valve has a value of 0.5cm^2 according to the technical data of the manufacturer.

Using the equation Eq.8.15, the parameter denotation from Figure 8.5 and a PI–controller for this subsystem the results are the following system differential equations:

$w_2'=v_2\ we_2 - k_2\ h_2 + P_2\ (v_2\ we_2' - k_2\ h_2')$ Eq.8.32

$h_2'=-az_l\ S_l\ (2g\ h_2)^{1/2}/A + l_2\ w_2 - a_{02}\ h_2$ Eq.8.33

With $v_2=k_2$ and under stationary condition (differentiations with respect to time are set to zero) the result is directly:

$we_2=h_2$ Eq.8.34

So a leak disturbance in tank 2 is settled by a PI–controller.

ref. 5.2c):

Figures 8.6 and 8.7 show the resuling graphic representations.

If the connection valve between tank 3 and tank 2 is closed this stands for $az_3=0$. With that Eq.8.2 is:

$h_2'=(-Q_{20}+u_2)/A$ Eq.8.35

With Eq.8.18 it follows:

$h_2' + a_{02}\ h_2 = l_2\ w_2 - Q_{32}(h_3,h_2)/A$ Eq.8.36

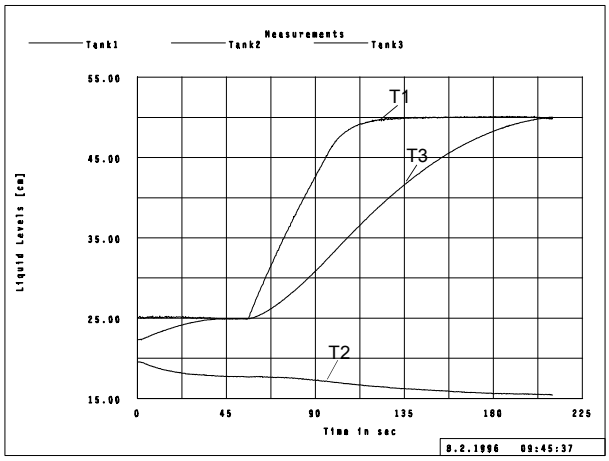


Figure 8.6 : State variables ref. experiment 5.2c)

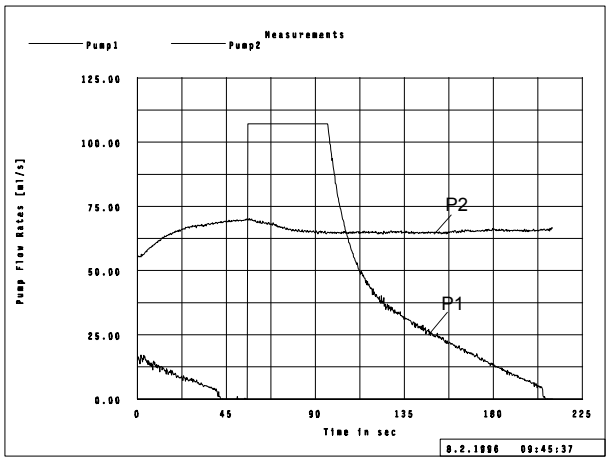


Figure 8.7 : Flow rates ref. experiment 5.2c)

The output variable h_2 is now coupled with h_3. The system behaviour of the output variable h_1 is furthermore described by Eq.8.11.

This behaviour can be seen from the characteristic: h_1 behaves similar to a first order lag and does not change its steady state in case of such a disturbance. The steady state of the output variable h_2 depends on the water level of tank 3.

ref. 5.2d):

Figure 8.8 und 8.9 show the resulting graphic representations.

A total pressure sensor failure in tank 1 means the assignment $h_1=0$ in Eq.8.17. So the result is:

$$u_1=Q_3\hat{} + A\ l_1\ w_1 \qquad \text{Eq.8.37}$$

where:

$$Q_3\hat{}=Q_{13}(h_1=0,h_3)=-az_1\ Sn\ (2g\ h_3)^{1/2} \qquad \text{Eq.8.38}$$

If the above equation is substituted in Eq.8.1, the result is:

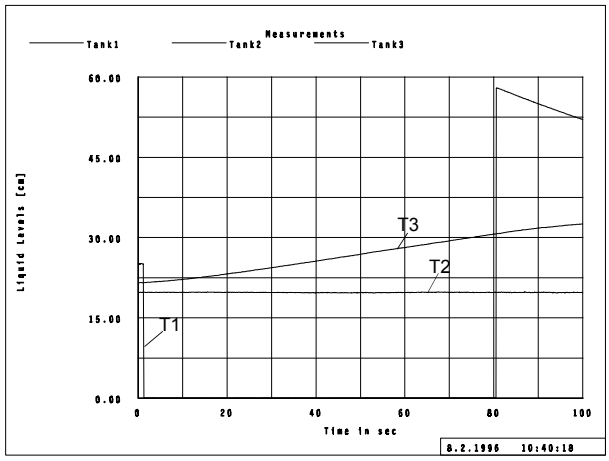


Figure 8.8 : State variables ref. experiment 5.2d)

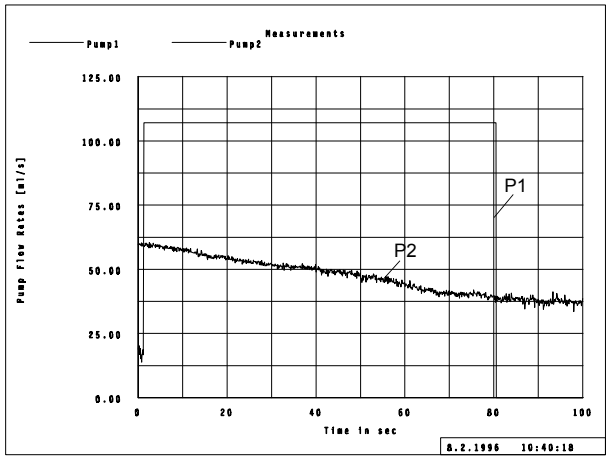


Figure 8.9 : Flow rates ref. experiment 5.2d)

$h_1'=(Q_3\text{^}-Q_{13}(h_1,h_3))/A + l_1\, w_1$ Eq.8.39

With this h_1 is coupled with the real level h_3. Using Eq.8.39, the steady state value of h_1 can be determined in dependency on h_3. According to the chosen parameters this value is over 60cm. The behaviour of the output variable h_2 is furthermore described by Eq.8.11.

A total pressure sensor failure results in an assignment h_3=0 in Eq.8.17 and Eq.8.18. So it can be shown accordingly that both of the output variables are coupled with the actual level h_3.

ref. 5.3a):

Figures 8.10 and 8.11 shows the graphic representations.

All of the parameters are chosen with a positive value; because of that the control loops are stable. The decoupled subsystems behave like a first order lag with the damping:

$Dae=(0.05/0.1)^{1/2}/2=0.35<1$ Eq.8.40

The corresponding settling behaviour can be seen from the characteristic.

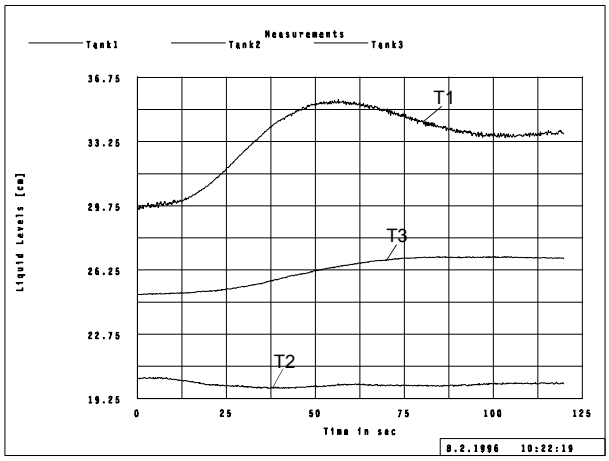


Figure 8.10 : State variables ref. 5.3a)

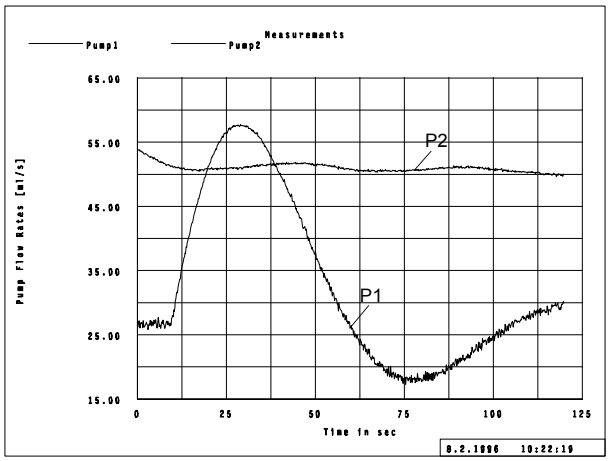


Figure 8.11 : Flow rates ref. experiment 5.3a)

ref. 5.3b):

Figures 8.12 and 8.13 show the resulting graphic representations.

Linearization of the system model of tank 2 around the operating point H_2 yields:

$$h_2' = (-\text{const}\, h_2 + q_2)/A \qquad \text{Eq.8.41}$$

where:

$$\text{const} = 0.5\, az_2\, S_n\, ((2g)/H_2)^{1/2} \qquad \text{Eq.8.42}$$

If Eq.8.41 is transformed into laplace, with:

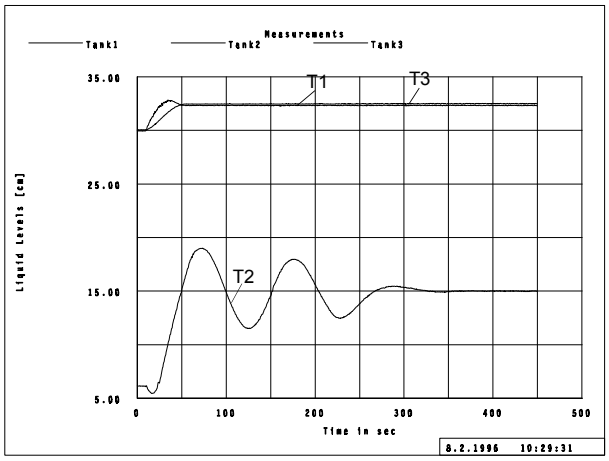


Figure 8.12 : State variables ref. experiment 5.3b)

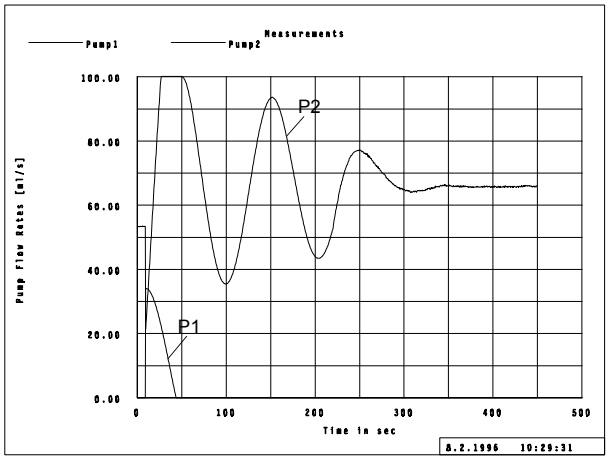


Figure 8.13 : Flow rates ref. experiment 5.3b)

$$q_2 = v_2\, (1/s + P_2)\, (we_2 - h_2) \qquad \text{Eq.8.43}$$

the result is the following transfer function G(s):

$$G(s) = v_2\, (1 + P_2 s)/(A\, s^2 + (v_2\, P_2 + \text{const})\, s + v_2) \qquad \text{Eq.8.44}$$

All of the variables are defined as differences from the operating point.

The poles of G(s) are:

$$s_{1,2} = -(\text{const} + v_2\, P_2)/(2\, A) +/- (((\text{const} + v_2\, P_2)/(2\, A))^2 - v_2/A)^{1/2} \qquad \text{Eq.8.45}$$

Using P_2=0 and P_2=10sec, each pole has a negative real part. Because of that the control loop is stable in both cases.

With az_2=0.7 the result at the operating point is: const=2.0 cm^2/sec.

Because of that the real parts of the poles with P_2=0 are close to the boundary of the stability region. Using P_2=10sec, the real parts are strong negative accordingly.

This can be seen from the characteristic: With P_2=0 the behaviour is oscillating, with P_2=10 the behaviour is damped.

The order of the linearized plant is 3 in the nominal state.

So it can be noted that the nonlinear decoupling results in an order reduction of the control loops.

Program Operation

(WINDOWS-Version)

Date: 02. November 2001

1 Program Operation 1-1

1 Program Operation

1.1 Program Start

The correct execution of the Windows program **DTS20W16.EXE** requires that the following files are available in the actual directory:

DTS20W16.EXE
DEFAULT.CAL
DTS200.HLP
DTSW16.INI
DTSSRV16.DLL
TIMER16.DLL
PLOT16.DLL
DAC98.DRV
DAC6214.DRV
DUMMY.DRV

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). The driver DUMMY.DRV is required only for the Demo version of the executable program. The file DEFAULT.CAL is used to store the calibration data of the system. A detailed description of this file is given in chapter 1.10. The help file DTS200.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 DTSW16.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.

After starting the program **DTS20W16.EXE** the main menu appears on the screen as shown in figure 1.1.

The window titled "DTS - Monitor" is located below the menu bar of the main window. Its first field indicates the active controller. The following field displays in the middle an animation picture of the three tanks including their liquid levels. To the left and to the right side of this picture two panels indicate the controller inputs (setpoint liquid level and difference signal) as well as the controller outputs (pump control signal) for tank 1 and tank 2. When the open loop control is active, the setpoint signal is the same as the controller output signal and the liquid level difference disappears. The last field indicates the status of the measurement.

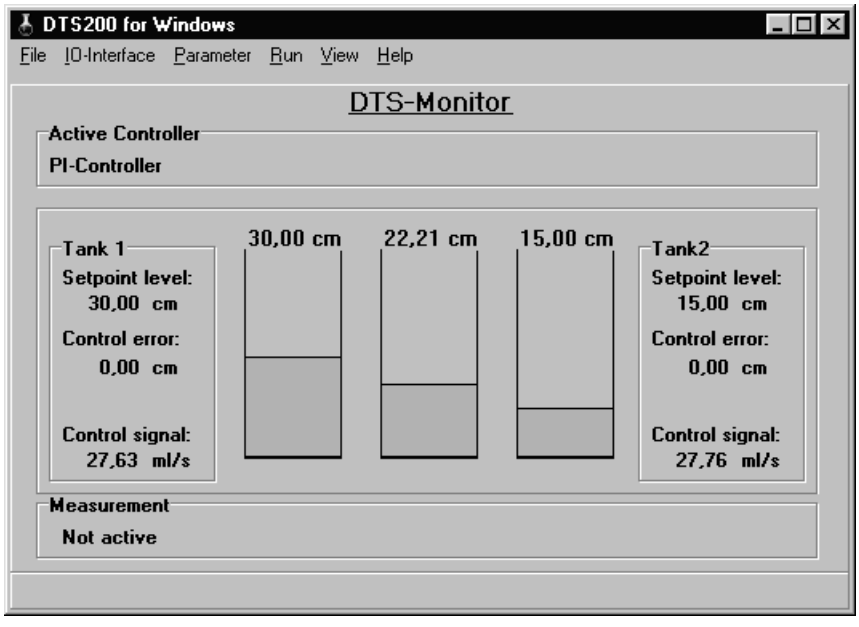


Figure 1.1: The main menu of the DTS200 controller

A menu bar is located in the upper part of the screen. Its sub-menus will be described in the following sections.

1.2 Menu 'File'

The pulldown menu **File** provides functions to save and load recorded data or system parameters, to print plot windows as well as to terminate the program (see figure 1.2).

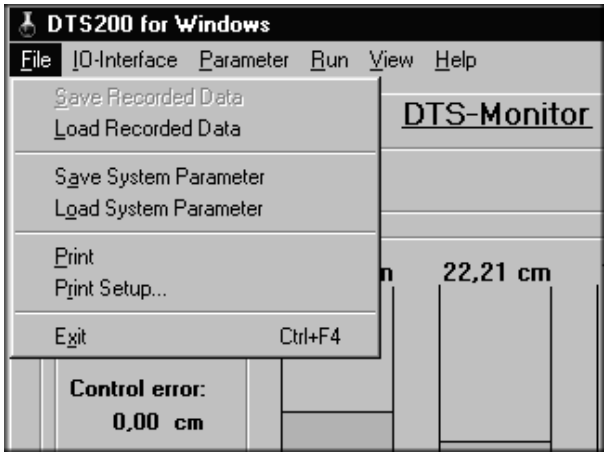


Figure 1.2: The sub menu 'File'

The function **Save Recorded Data** is enabled after the first measurement accquisition. It writes the measured data together with the selected system configuration like controller type and parameters to a file (documentation file). The file name is selected by the user by means of a file save dialog window. The extension of the file name has to be *.PLD.

The function **Load Recorded Data** opens a file open dialog window allowing the user to select a documentation file (*.PLD). The file data (from previous measurements) may be displayed in a graphic using the function **Plot File Data** from the menu **View**. The system configuration read from this file may be displayed in an information box using the function **Parameter From *.PLD File** from the menu **View**.

The function **Save System Parameter** stores the calibration data of the tank system (sensor characteristics, outflow coefficients and pump characteristics) to a file with an assignable name on the hard disk. (The file format is described with Calibration Data File). This option should be used to create the file DEFAULT.CAL when this file is missing. Otherwise a warning message will appear any time the program is started. This menu item is disabled when any controller is active.

The function **Load System Parameter** reads the calibration data of the tank system (sensor characteristics, outflow coefficients and pump characteristics) from a selectable file. This menu item is disabled when any controller is active.

The function **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.

The function **Print Setup ...** opens the Windows dialog to select a printer and to adjust its options.

Selecting the menu item **Exit** (equivalent to pressing Ctrl+F4)will terminate the program when the message box "Exit program ?" is confirmed. Another message box has to be confirmed in case a new sensor calibration has been carried-out without saving the system parameters.

1.3 Menu 'IO-Interface'

The pulldown menu **IO-Interface** provides functions to manipulate the driver for the PC plug-in card (see figure 1.3a).

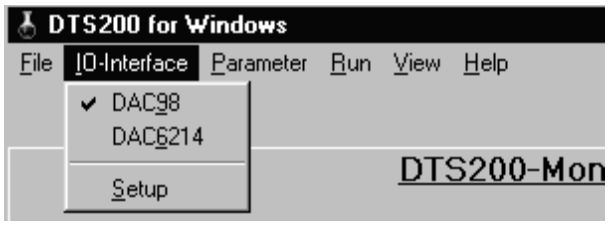


Figure 1.3a: The sub menu 'IO-Interface'

The first two items

DAC98
DAC6214

represent the selectable drivers (DAC98.DRV, DAC6214.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 DTS20W16.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 DTSW16.INI which is controlled by the program automatically. When this file is missing the default driver is always the DAC98.DRV.

The function **Setup** opens a dialog (see figure 1.3b) to adjust the drivers hardware address of the installed IO-adapter card. This address has to match the hardware settings !

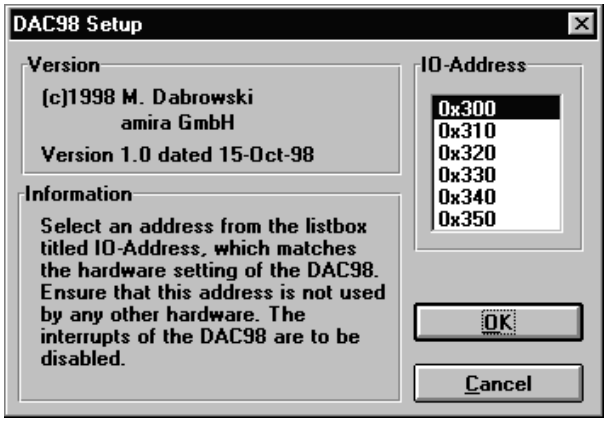


Figure 1.3b: DAC98 setup dialog

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.

1.4 Menu 'Parameter'

The pulldown menu **Parameter** as shown in figure 1.4a contains functions to adjust or calibrate system parameters.

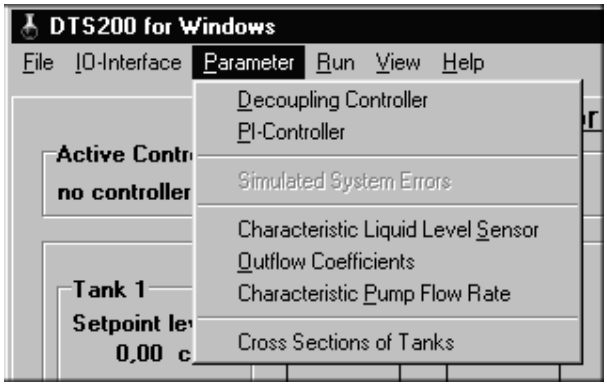


Figure 1.4a: The sub menu 'Parameter'

The function **Decoupling-Controller** allows the user to adjust the common amplification factor of the decoupling network interactively. The resulting control loop behaves like a control loop with a proportional controller with an integrating plant. The coefficients l*i* = a0*i* = Decoup (see theoretical backgrounds) are equal to the proportional amplification. This value is active in addition, when decoupling mode is activated for the PI controller.

The function **PI-Controller** opens a dialog (see figure 1.4b) allowing the user to adjust the proportional and the integral portion of the PI controller interactively. When 'decoupled' is enabled the same factor as mentioned above is adjustable meaning an active decoupling network in addition.

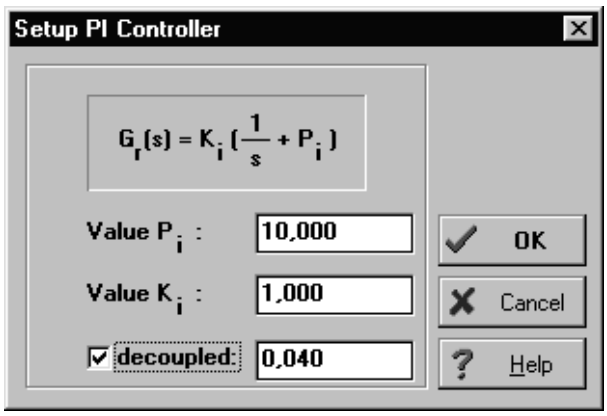


Figure 1.4b: Dialog for PI controller parameters

The menu item **Characteristic Liquid Level Sensor** starts a measuring to determine the sensor parameters such that a straight line defined by a gradient and an offset maps the sensor voltage in [Volt] to a real liquid level in [cm]. A dialog is opened automatically where the user is informed which actions are to be carried-out. At first the default calibration levels at 20 cm and 50 cm may be changed. Then the user is asked to control both pumps manually such that the calibration levels one after the other are reached in the 3 tanks. Reaching a specific calibration level is to be prompted by pressing 'measure 1' for the first and pressing 'measure 2' for the second calibration level. The function will then determine the accompanying sensor characteristics (gradient and offset as mentioned above) for each tank. After pressing 'measure 2' the dialog is terminated immediately. To abort the operations the 'Cancel' button can be used before 'measure 2' is activated. In that case the previous calibration data will be left unchanged. The same happens when the user activates the 'measure 1/2' buttons one after the other without really changing the tank levels by controlling the pumps manually.

The menu item **Simulated System Errors** is available only for the Demo version of the program (see chapter 1.8).

The menu item **Outflow Coefficient** opens a dialog (see figure 1.4c) to start an automatic determination of the outflow coefficients. By means of a dialog the user is asked to adjust the maximum initial liquid levels of 60 cm in all tanks by controlling the pumps manually. After

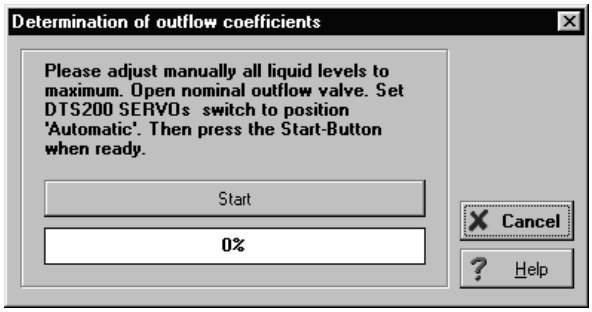


Figure 1.4c: Dialog for outflow coefficients

pressing the 'Start' button the determination of the outflow coefficients is carried out automatically. The liquid levels are measured the first time after a delay time of 60 sec. The second measurement is taken after another 15 sec. The progress in percentage of the measuring time is displayed in form of a gauge bar. At the end the outflow coefficients are calculated by substituting the level differences in the mathematical model of the tank system. The resulting outflow coefficients are then displayed in an information box which may be obtained in addition by activating **Outflow Coefficients** from the main menu item **View**. The measuring may be aborted at any time by pressing the 'Cancel' button.

The menu item **Characteristic Pump Flow Rate** starts an automatic determination of the pump characteristics by controlling the pumps with a constant signal for a specific time and measuring the resulting liquid level changes. By means of a dialog (see figure 1.4d) the user is asked to close all valves and to adjust the initial liquid levels of 10 cm in all tanks by controlling the pumps manually. Then the pump data are determined automatically by switching on the pumps for a specific period of time with a step-wise (12.5%) increasing control signal. The current liquid levels in the tanks 1 and 2 are measured after each step. The flow rate is calculated from the liquid difference and the on-switching time of the pumps for each control signal. The results are displayed in a table with nine rows. To terminate the dialog an 'Ok' button is presented at the end of the measurements. The 'Cancel' button may be used before the end of the measurements to abort the operations. But any result obtained in the mean time (any filled row of the table) is taken as a valid pump calibration base point.

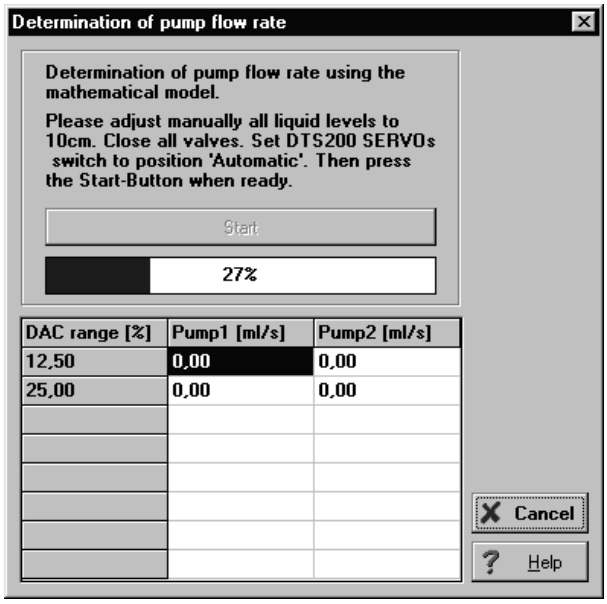


Figure 1.4d: Dialog for flow rate calibration

The menu item **Cross Section of Tanks** allows for displaying or editing the effective cross sections of the three tanks by means of a dialog (see figure 1.4e) . Standard values are 154 (cm^2) for tanks without and 149 (cm^2) for tanks with an inner flow pipe.

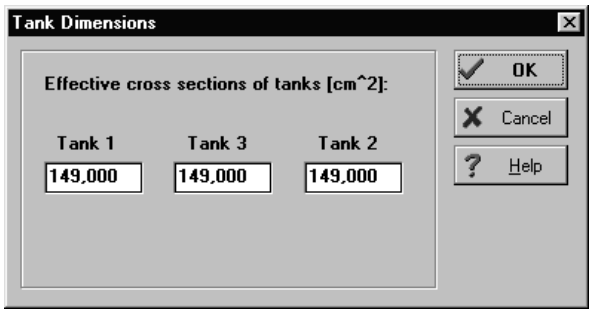


Figure 1.4e: Dialog for tank dimensions

1.5 Menu 'Run'

The pulldown menu **Run** from figure 1.5a provides adjustments of the system configuration.

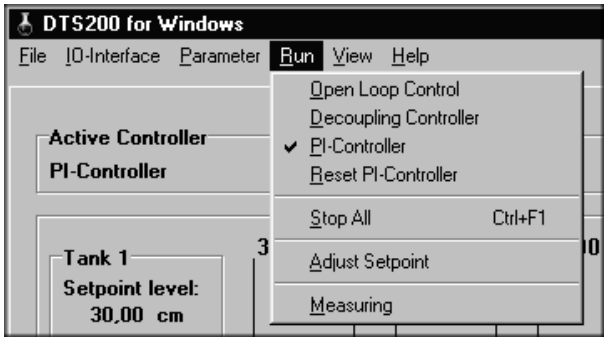


Figure 1.5a: The sub menu 'Run'

The function **Open Loop Control** installs an open loop control. When another controller was active previously the setpoint generators (for the pump flow rate) are reset.

The function **Decoupling Controller** installs a control loop with decoupling network (P controller). When another controller was active previously the setpoint generators (for the liquid levels in the tanks 1 and 2) are reset.

The function **PI-Controller** installs a control loop with PI controller. When another controller was active previously the setpoint generators (for the liquid levels in the tanks 1 and 2) are reset.

The function **Reset PI-Controller** resets the controller variables from the previous sampling period.

The function **Stop All** (short cut Ctrl F1) switches off any of the selected controllers, disables the menu item **Adjust Setpoint** and enables the menu items **Load/Save System Parameter** as well as the three items to control the system calibration.

The function **Adjust Setpoint** is available only when one of the above system configurations is activated. This menu item opens a dialog window (see figure 1.5b) to adjust 2 setpoints or control signals, that means level setpoints (tank 1 and tank 2) for the controlled tank system or pump flow rates for the system in an open loop control. The signal shape (constant, rectangle, triangle, ramp, sine) together with an offset, an amplitude and a timing period are adjustable for each setpoint signal. The period is meaningless in case of a constant signal shape. The real signal is always formed by the sum of offset and amplitude. The selected signal becomes valid after terminating the dialog with 'Ok'.

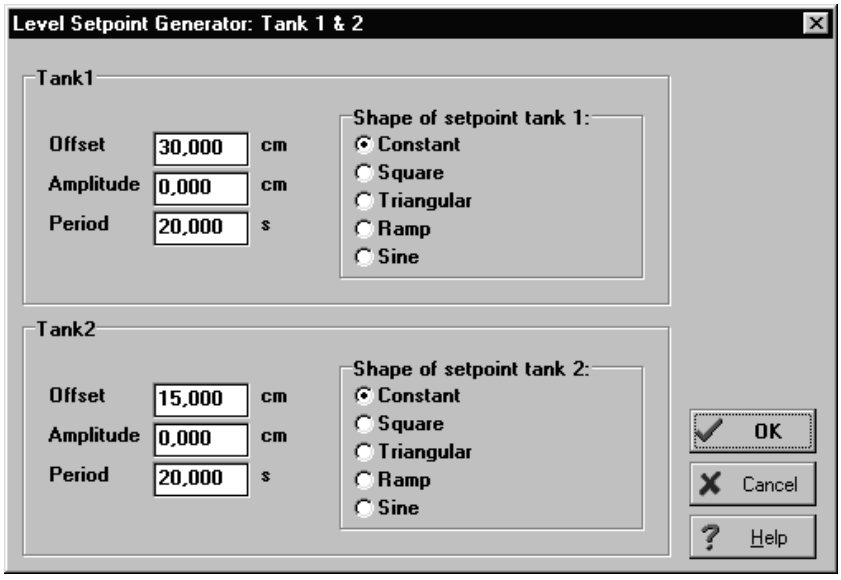


Figure 1.5b: The dialog window 'Adjust Setpoint'

The function **Measuring** is always enabled. This menu item opens a window to adjust the measuring time and to assign trigger conditions to start recording the measurements. Figure 1.5c 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 currently adjusted system configuration like controller type and parameters are stored in a data structure which may be saved in a documentation file (*.PLD).

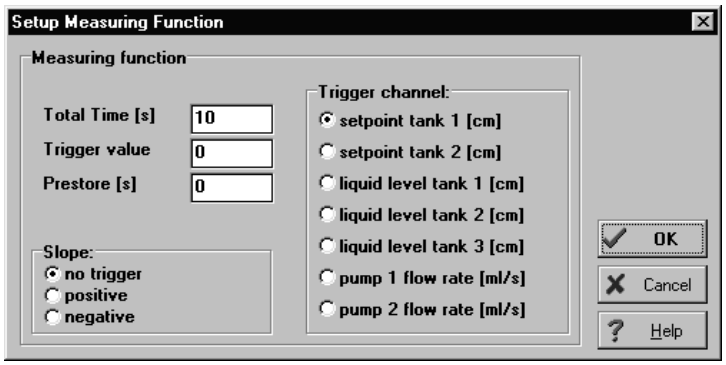


Figure 1.5c: The dialog window 'Measuring'

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.

1.6 Menu 'View'

The pulldown menu **View** from figure 1.6a provides functions for graphic representations of currently recorded measurements or of data from the documentation file (*.PLD).

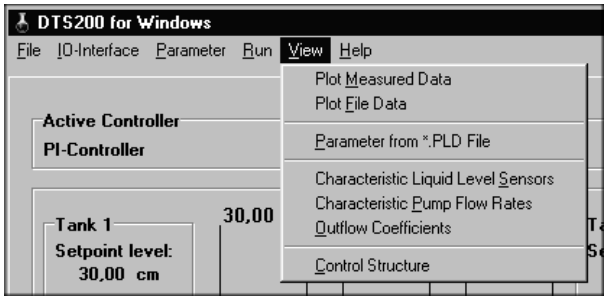


Figure 1.6a: The sub menu 'View'

The menu item **Plot Measured Data** opens a dialog window to select the data which are to be displayed in a graphic representation. The selectable data are:

Liquid levels (3 curves)
Liquid levels and setpoints (5 curves)
Pump flow rates (2 curves)
Pump control signals (2 curves)
Levels and flow rates (5 curves)

Pressing the 'Ok' button will display the selected graphic immediately (see figure 1.6b as an example) while 'Cancel' will abort this dialog and return to the main window. An open plot window may be stored to the clipboard or to an assignable Windows Meta File by activating corresponding items from its system menu. See figure 1.6b for an example.

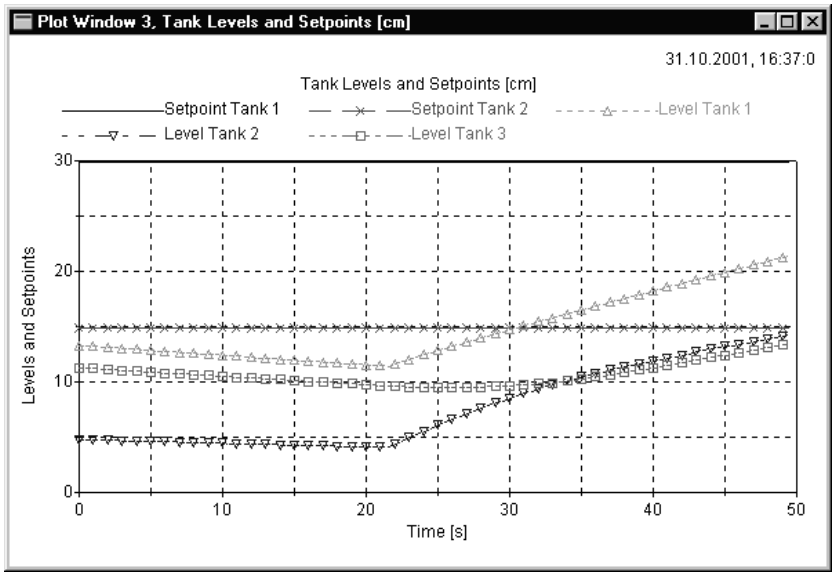


Figure 1.6b: Example of plot window

The menu item **Plot File Data** operates similar to the item **Plot Measured Data**. But the data are loaded from the file selected by the menu item **Load Plot Data** from the menu **File**.

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 **Characteristic Liquid Level Sensor** displays a graphic representation of the 3 sensor characteristics.

The menu item **Characteristic Pump Flow Rate** displays a graphic representation of the 2 pump characteristics.

The menu item **Outflow Coefficients** displays the outflow coefficients in an information box.

The menu item **Control Structure** displays a separate window (see figure 1.6c) containing the block diagram of the two control loops with respect to the selected controller structure. This window will appear also by clicking on the label "Active Controller" located in the upper panel of the "DTS-Monitor" window.

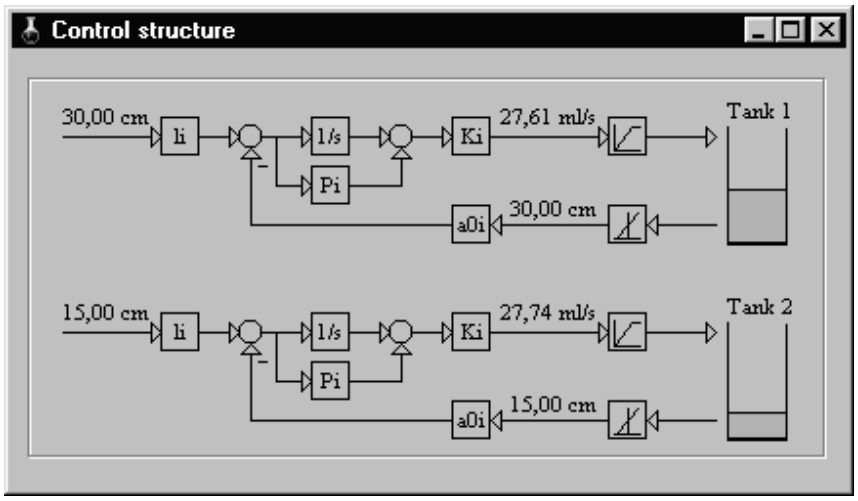


Figure 1.6c: Example of a control structure

1.7 Menu 'Help'

The pulldown menu **Help** as shown in figure 1.7a provides functions to control the Windows help function and to obtain general information about the program.

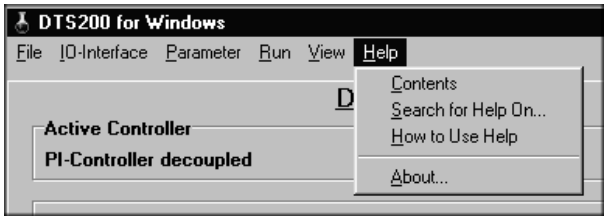


Figure 1.7a: The sub menu 'Help'

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

Activating the menu item **About** opens an information box displaying the program version, the copyright and the IO-adapter card requirements (see figure 1.7b).

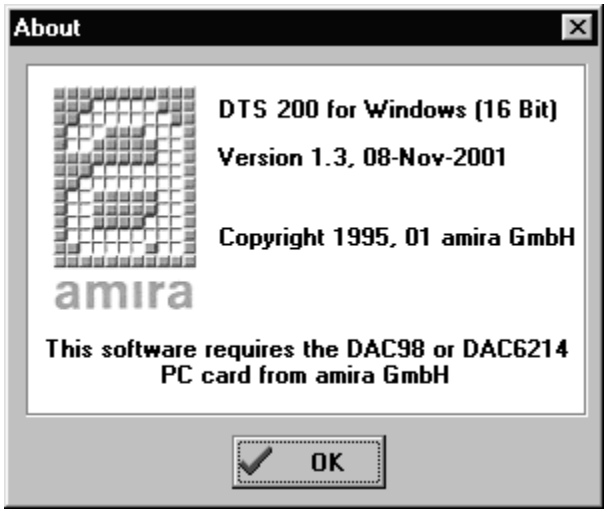


Figure 1.7b: 'About' information box

1.8 The Demo Version

The demo version of the program DTS20W16.EXE is indicated by the title "DTS - Monitor (Demo-Version)" in the monitor window (see figure 1.8a). It operates with a mathematical model instead of reading sensor signals from the IO-adapter card and writing control signals to this card. Besides the functions to control the calibration and to select the IO-Interface all of the menu items are available.

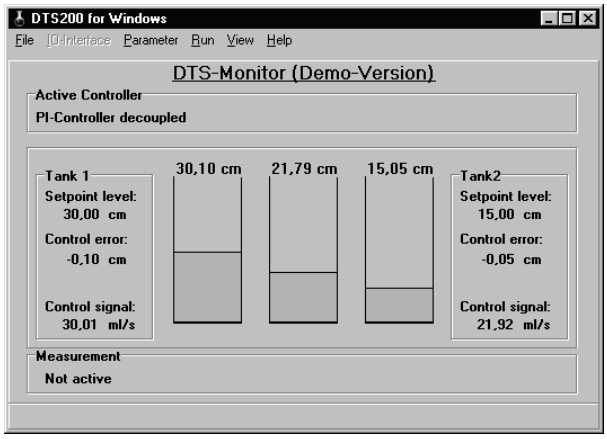


Figure 1.8a: Main window of Demo-Version

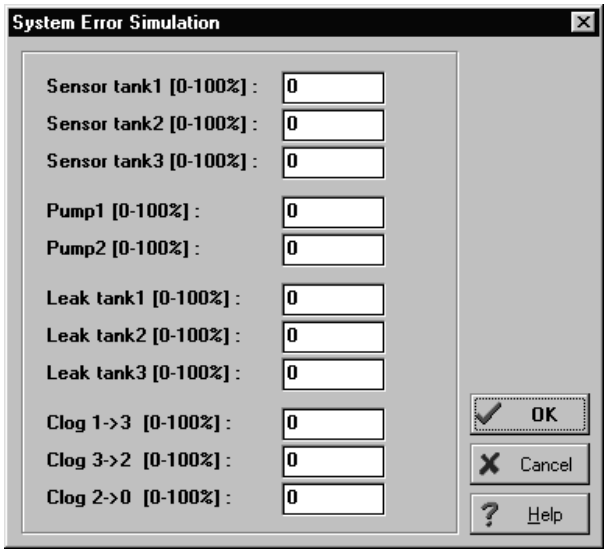


Figure 1.8b: Dialog to set system errors

But two special features are introduced. At first the simulation runs 10 times faster than the real system. At second the menu item **Simulated System Errors** (see figure 1.8b) from the main menu **Parameter** is provided to simulate signal errors for the three level sensors and the 2 pumps. Single or multiple leaks and clogs may be defined in addition. The errors are defined in the range from 0 to 100% meaning an undisturbed signal for 0% and a complete missing signal for 100%.

During calculation of the mathematical model the errors are considered in the range from 0 to 1. This results for tank 1 in example:

$$A_{Tank}\,\frac{\delta h_1}{\delta t} = Q_{in} - Q_{out} - Q_{Leak}$$

$$Q_{in} = Q_{Pump1}\,(\,1 - Pump1_{\,Error}\,)$$

$$Q_{out} = c_1\,A_{Valve}\,(\,1 - Clog1{-}{>}3_{Error}\,)\,\sqrt{2\,g\,|\,h_1 - h_3\,|}$$

$$Q_{Leak} = c_1\,A_{Valve}\,Leak1_{Error}\,\sqrt{2\,g\,h_1}$$

$$h_{1\,Sensor} = h_1\,(\,1 - Sensor1_{Error}\,)$$

where

h_1 , h_3 = real liquid levels of the tanks 1 and 3

A_{Tank} , A_{Valve} = cross sections of tank resp. valve

c_1 = outflow coefficient connection valve 1 -> 3

1.9 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 (70 bytes),
The structure CTRLSTATUS (36 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 DTS200 the number of the stored channels is always 7 (the length of the measurement vector is 7, i.e. equal to 28 bytes). The vector contains the following signals:

the setpoint for tank 1 (in cm),
the setpoint for tank 2 (in cm),
the measured liquid level from tank 1 (in cm),
the measured liquid level from tank 2 (in cm),
the measured liquid level from tank 3 (in cm),
the control signal for pump 1 (in ml/s),
the control signal for pump 2 (in ml/s)

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

1.10 Format of the Calibration Data File DEFAULT.CAL

The calibration data file DEFAULT.CAL which may be loaded or saved by **Load System Parameter** resp. **Save System Parameter** from the main menu item **File** contains the sensor characteristics (gradient and offset), the outflow coefficients as well as nine base points for each pump characteristic in an ASCII format. When this file is missing during starting the program **DTS20W16**, it may be created with the default calibration data just by activating **Save System Parameter** without carrying-out any calibration. In this case the result would be as follows:

```
-3.42857      [gradient]
30.8571       [offset]
 -3.42857     [gradient]
30.8571       [offset]
-3.42857      [gradient]
30.8571       [offset]
0.5 0.6 0.5   [Outflow Coef.]
0       [ml/s at   0.0 % Controller output
12.5    [ml/s at  12.5 % Controller output]
25      [ml/s at  25.0 % Controller output]
37.5    [ml/s at  37.5 % Controller output]
50      [ml/s at  50.0 % Controller output]
62.5    [ml/s at  62.5 % Controller output]
75      [ml/s at  75.0 % Controller output]
87.5    [ml/s at  87.5 % Controller output]
100     [ml/s at  100.0 % Controller output]
0       [ml/s at   0.0 % Controller output]
12.5    [ml/s at  12.5 % Controlle output]
25      [ml/s at  25.0 % Controller output]
37.5    [ml/s at  37.5 % Controller output]
50      [ml/s at  50.0 % Controller output]
62.5    [ml/s at  62.5 % Controller output]
75      [ml/s at  75.0 % Controller output]
87.5    [ml/s at  87.5 % Controller output]
100     [ml/s at 100%% Controller output]
149 149 149 [Tank cross sections]
```

Technical Data

Date: 05. November 2001

(Technical data are subject to change)

1 Technical Data 1-1

1 Technical Data

1.1 Tank system, pumps, sensors

Dimension Sizes and Weight	Value	Unit
Length	1300	mm
Depth	360	mm
Height	880	mm
Weight	40	kg

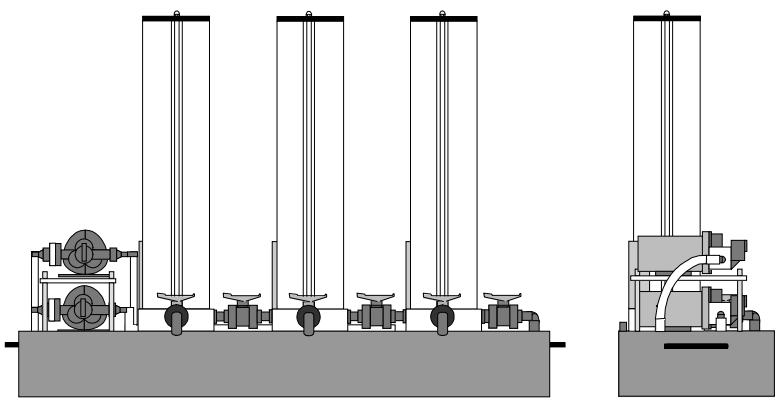
Reservoir	Value	Unit
Length	1210	mm
Depth	360	mm
Height	150	mm
Capacity ca.	55	l

Cylinder tank	Value	Unit
Diameter outside	150	mm
Diameter inside Diameter inner flow pipe	140 25	mm mm
Height incl. socket and cover	720	mm
max. liquid level	630	mm
Capacity ca.	9	l

Ball Valves Effective area of section	Value	Unit
Connection pipes	0.5	cm^2
Nominal outflow	0.5	cm^2
Leakage openings	0.5	cm^2

Pumps DC-motor with "three chamber diaphragm pump"	Value	Unit
Rated voltage	12	V
Rated current (open flow) Rated current (max.)	1.4 4.5	A A
Flow rate (open flow)	7	l/min
Pressure	1.4	bar
Weight	1.5	kg
Length	210	mm
Depth	128	mm
Height	115	mm

Sensors Principle of measurement: plate capacitor	Value	Unit
Pressure range	0...0.1	bar
Supply voltage	12...30	V
Output signal nominal	4...20	mA

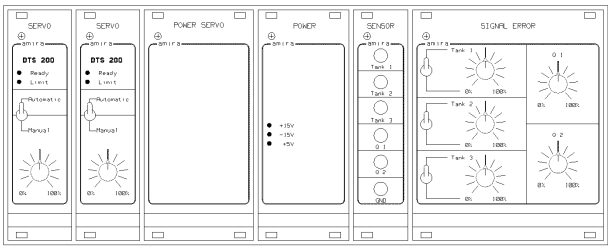


Sensors Principle of measurement: plate capacitor	Value	Unit
max. allowable burden	37.5* (Ub-12)	Ohm
max. current consumption	23	A
Characteristic	linear	
Deviation from linearity	< 0.5	%
Repeatability and hysteresis	<0.02	%
Response time (63%)	5	ms

Inputs	Value	Unit
Pump supply voltages	0 ... +12	V
Sensor supply voltages	18	V

Outputs	Value	Unit
Liquid level from sensors - Current range - Height range - Resolution	 4 ... 14 0 ... 63 0.16	 mA cm mA/cm

1.2 Actuator



Dimension Sizes and Weight	Value	Unit
Length	360	mm
Depth	310	mm
Height	147	mm
Weight	7.5	kg

Input	Value	Unit
Input voltage	230	V~
Frequency	50/60	Hz
Power	150	W
Fuse primary	1.6	A M

1.2.1 Servo modules

Inputs	Value	Unit
Voltage supply Fuse (15 V~)	15 +15 2.0	V~ V A M
Sensor signal overflow protection	1 ... +5.6	V
Control signals	0 ... +10	V

Outputs	Value	Unit
Pump voltage supply (PWM)	+15	V
Rated current	1.6	A
Dyn. peak current	4.5	A
Overflow protection 0 = Overflow 1 = No overflow	TTL signal	
Switch position 0 = Automatic 1 = Manual	TTL signal	

1.2.2 Mains supply with signal adaption unit

Mains supply	Value	Unit
Input voltage 2 x Fuse primary	230 100	V~ mA T
Frequency	50/60	Hz
Mains supply	Value	Unit
Output voltages (short-circuit protected)	± 15 +5 +8	V V V

Inputs signal adaption **unit**	Value	Unit
3 sensor signals (at 274 Ohm)	1.1 ... 3.8	V
2 control signals	-10...+10	V
TTL-signal release control signal	0 ... +5	V

Outputs signal adaption unit	Value	Unit
3 liquid levels - voltage range - level range - resolution	 +10 ... -10 0 ... 70 0.286	 V cm V/cm
2 control sig. for servos	0 ... +10	V

1.2.3 Module "POWER SERVO"

Mains supply	Value	Unit
Input voltage Fuse primary	230 630	V~ mA T
Frequency	50/60	Hz
Output voltage	2*15	V~
Rated current	2*2.6	A

1.2.4 Measurement Outputs of the Module "Sensor"

Measurement jacks	Value	Unit
Tank 1 ... Tank 3 (= Outputs of sig. adapt. unit or disturbance mod.) - Range of liquid levels - Nominal value for 0cm - Nominal value for 60cm	 +10 ... -10 +9 -9	 V V V
Q1 and Q2 (= Outputs of PC-controller) - Flow rate 0 ml/s - Flow rate 100 ml/s	 -10 +10	 V V

1.2.5 Signal distribution (Rear Panel)

Inputs signal distribution unit (50-pol. Connector)	Value	Unit
D/A0 (Pin 47) Control signal servo 1	 -10 ... +10	 V
D/A1 (Pin 48) Control signal servo 2	 -10 ... +10	 V
DO1 (Pin 35) Pulse (release output stage) Hihg-level with pulse to low, duration 40-100 µs	TTL-Signal	
DO2 (Pin 36) Rect (release output stage) min. 10 Hz, max. 1 kHz	TTL-Signal	

Outputs signal distribution (50-pol. connector)	Value	Unit
A/D0 (Pin 16) Level tank 1	-10 ... +10V	V
A/D1 (Pin 17) Level tank 2	-10 ... +10V	V
A/D2 (Pin 32) Level tank 3	-10 ... +10V	V

1.2.6 Controller

PC with Opt. 200-02	
A/D-D/A-card for PC	DAC98

Inputs	Value	Unit
3 sensor signals at 12 Bit A/D-Converter	-10 ... 10	V

Outputs	Value	Unit
Control signals for servo amplifier from 12 Bit D/A-Converter	-10 ... +10	V

1.2.7 Elektrical disturbance module (Option 200-05)

Option 200-05	Value	Unit
Sensor signals for liquid levels - scaleable - can be switched off	0 - 100	%
Control signals - scaleable	0 - 100	%

1.3 Pin reservations of the plug connections

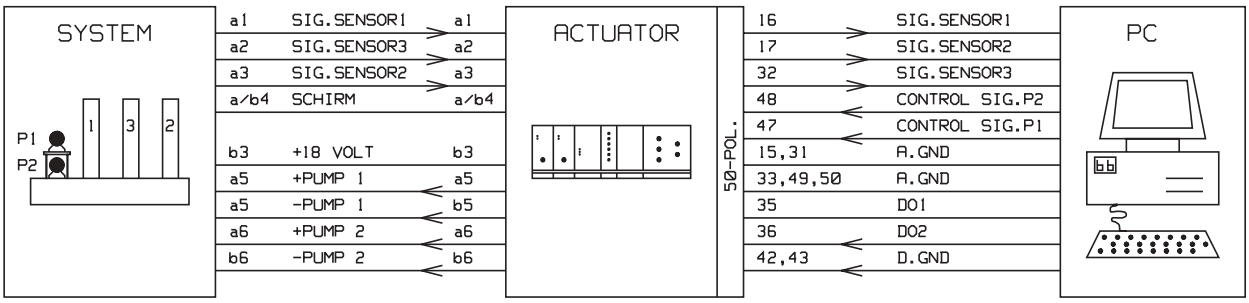
1.3.1 Connection Actuator to the PC Plug-in Card

PC-Connector (DAC98)		
Pin-No.	Pin-Den.	Reservation
1	n.c.	n.c.
2	n.c.	n.c.
3	n.c.	n.c.
4	n.c.	n.c.
5	n.c.	n.c.
6	n.c.	n.c.
7	n.c.	n.c.
8	n.c.	n.c.
9	DIN0	n.c.
10	DIN1	n.c.
11	DIN2	n.c.
12	DIN3	n.c.
13	n.c.	n.c.
14	n.c.	n.c.
15	AGND	AGND
16	AIN0	A/D 0 Level tank 1
17	AIN1	A/D 1 Level tank 2
18	n.c.	n.c.
19	n.c.	n.c.
20	n.c.	n.c.
21	n.c.	n.c.
22	n.c.	n.c.
23	n.c.	n.c.
24	n.c.	n.c.
PC-Connector (DAC98)		

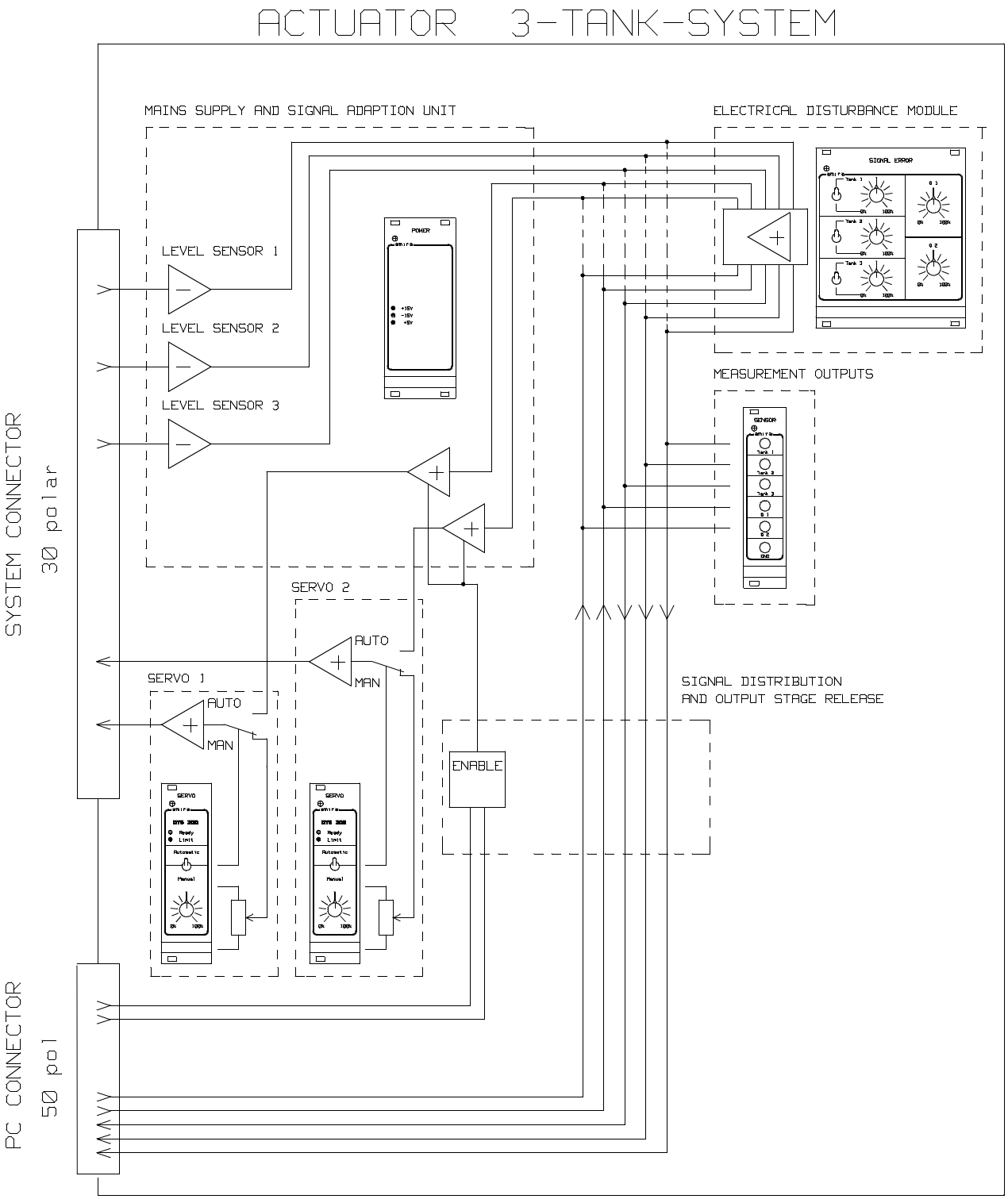
Pin-No.	Pin-Den.	Reservation
25	n.c.	n.c.
26	n.c.	n.c.
27	n.c.	n.c.
28	n.c.	n.c.
29	n.c.	n.c.
30	n.c.	n.c.
31	AGND	AGND
32	AIN2	A/D 1 Level tank 3
33	AIN3	n.c.
34	Dout0	n.c.
35	Dout1	D/O 1 Pulse (release output stage) Hihg-level with pulse to low, duration 40-100 µs
36	Dout4	D/O 2 Rect (release output stage) min. 10 Hz, max. 1 kHz
37	Dout3	n.c.
38	n.c.	n.c.
39	n.c.	n.c.
40	n.c.	n.c.
41	n.c.	n.c.
42	DGND	DGND
43	DGND	DGND
44	n.c.	n.c.
45	n.c.	n.c.
46	n.c.	n.c.
47	Aout0	D/A 0 Control signal servo 1
48	Aout1	D/A 1 Control signal servo 2
49	AIN4	AGND
50	AIN5	AGND

1.3.2 Connection Actuator to the Three-Tanks-System

System Connector					
Pin-No.	Reservation	Pin-No.	Reservation	Pin-No.	Reservation
a1	Signal sensor 1	b1	n.c.	c1	n.c.
a2	Signal sensor 3	b2	n.c.	c2	n.c.
a3	Signal sensor 2	b3	18 V	c3	n.c.
a4	Screen	b4	Screen	c4	n.c.
a5	Pump 1 +	b5	Pump 1 -	c5	n.c.
a6	Pump 2 +	b6	Pump 2 -	c6	n.c.
a7	n.c.	b7	n.c.	c7	n.c.
a8	n.c.	b8	n.c.	c8	n.c.
a9	n.c.	b9	n.c.	c9	n.c.
a0	n.c.	b0	n.c.	c0	n.c.



Connections between mechanic, actuator and PC


ACTUATOR 3-TANK-SYSTEM
MAINS SUPPLY AND SIGNAL ADAPTION UNIT
ELECTRICAL DISTURBANCE MODULE
SIGNAL ERROR
POWER
LEVEL SENSOR 1
LEVEL SENSOR 2
LEVEL SENSOR 3
MEASUREMENT OUTPUTS
SENSOR
SERVO 2
AUTO
MAN
SERVO 1
AUTO
MAN
SIGNAL DISTRIBUTION
AND OUTPUT STAGE RELEASE
ENABLE
SERVO
SYSTEM CONNECTOR
30 polar
PC CONNECTOR
50 pol

The PC Plug-in Card DAC98

Date: 04. January 1999

1 The PC Plug-in Card DAC98 (PCA902)

1.1 Introduction

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

1.2 Features

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

1.3 Specifications

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

1.4 Installation of the DAC98

1.4.1 Adjustment of the Base Address

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

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

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

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

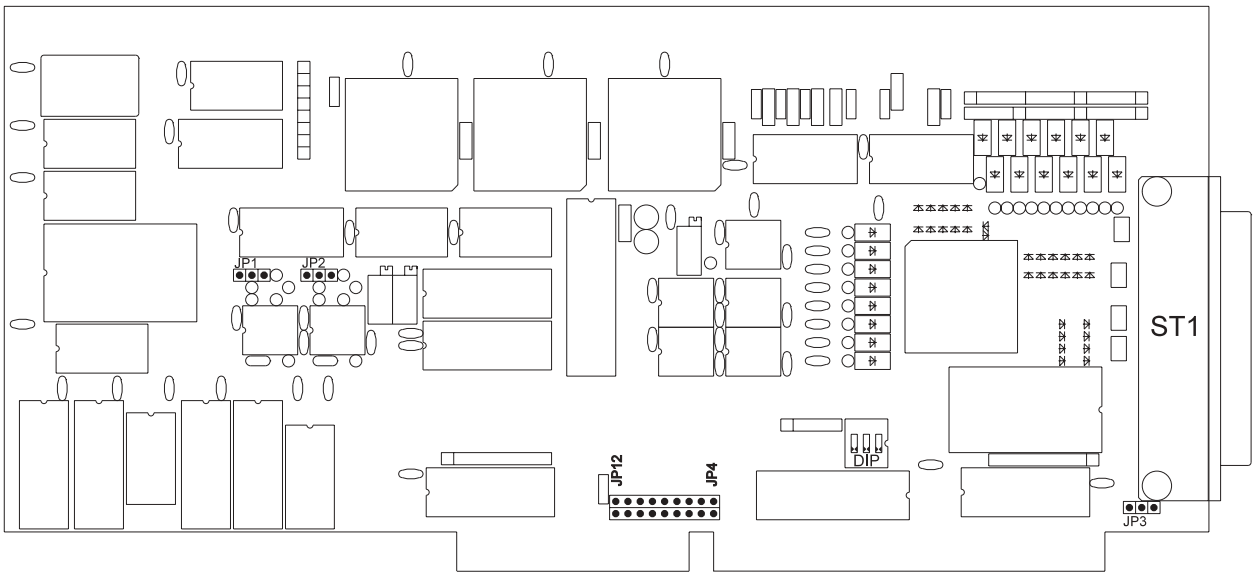
I/O Address (Hex)	3	2	1
300(*)	1	1	1
310	0	1	1
320	1	0	1
330	0	0	1
340	1	1	0
350	0	1	0
360	1	0	0
370	0	0	0

1.4.2 Adjustment of the Interrupt Channel

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

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

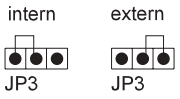
The PC hardware may be damaged when more than one jumper is installed or in case the selected interrupt channel is in use by another card.



Jumpers of the DAC98

1.4.3 The Operation Modes of the 16 Bit Timer/Counter

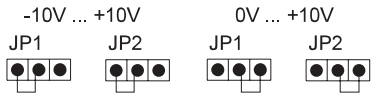
The 16 bit counter either counts external events or it counts the timer clock. The jumper JP3 adjusts the corresponding operation mode.



The default setting is timer clock counting mode.

1.4.4 Adjustment of the Analog Output Signal Range

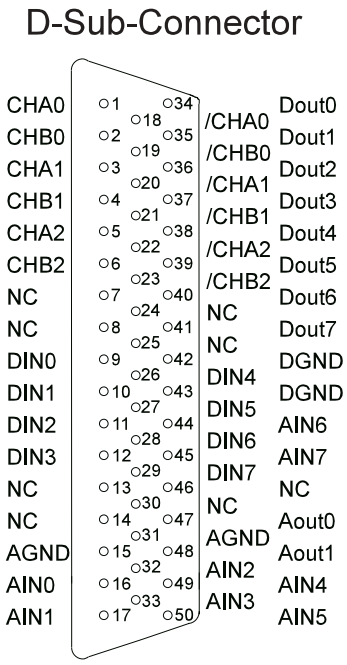
Two different output voltage ranges (-10 V to +10 V or 0 V to +10 V) are selectable for each D/A converter. The jumpers JP1 (analog out 0) and JP2 (analog out 1) provide this selection for each channel.



1.4.5 Pin-Reservations of the DAC98

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

ST1: 50-polar D-SUB connector



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

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

Note:

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

1.4.6 Installation of the Card in the PC

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

b) Disconnect all cables of your PC.

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

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

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

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

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

1.5 Programming of the DAC98

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

1.5.1 The Registers of the DAC98

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

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

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

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

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

1.5.2 Configuration of the DAC98

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

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

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

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

1.5.3 Reading the Identification String

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

1.5.4 A/D Conversion

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

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

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

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

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

1.5.5 D/A Conversion

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

1.5.6 Programming the Timer Chip 8253

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

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

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

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

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

1.5.7 Reading the Digital Inputs

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

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

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

1.5.8 Setting the Digital Outputs

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

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

1.5.9 Internal Digital Functions

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

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

1.5.10 Reading an Incremental Encoder Input Channel

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

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

1.5.11 Interrupt / Clock Signal

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

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

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

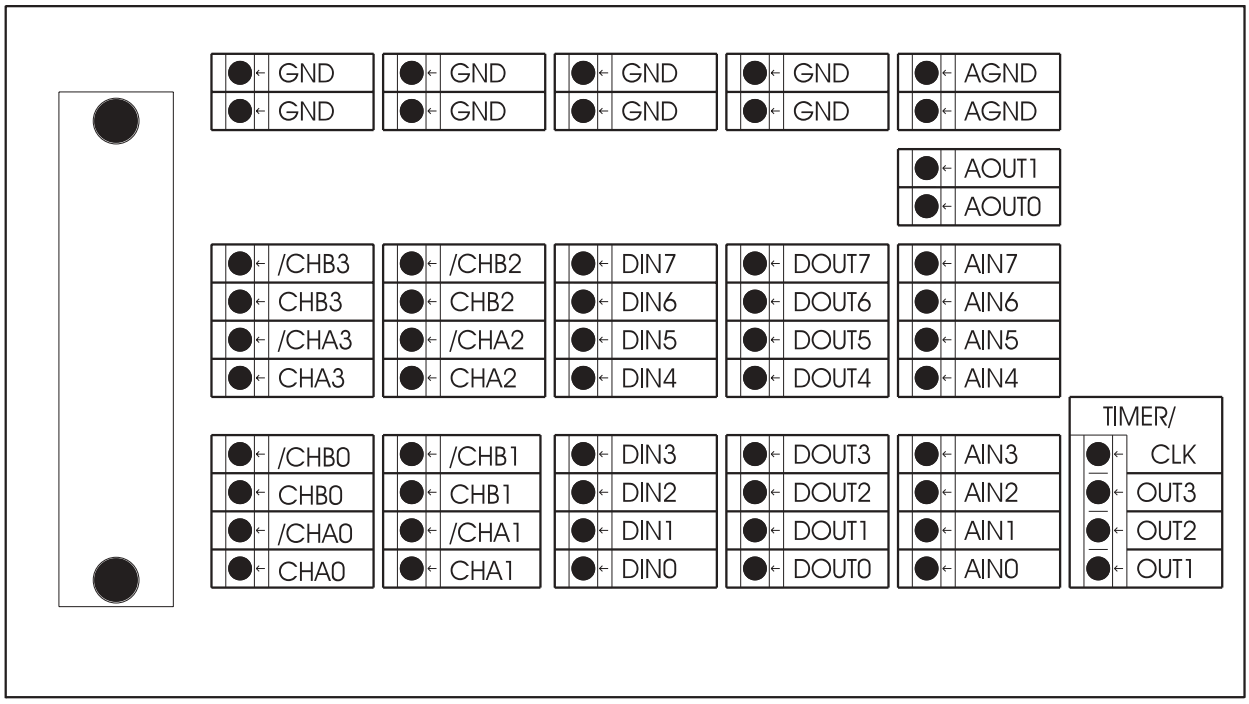
Note: To maintain any selected clock the corresponding bits have to be the same in following write operations to the interrupt / clock signal register.

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

2 DAC98 Adapter Card

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

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



3 Operating Instructions for the Test Program

3.1 Installation

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

Now switch on your computer and start MS Windows.

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

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

according to the drive assignment.

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

Attention !

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

3.2 Program Start

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

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

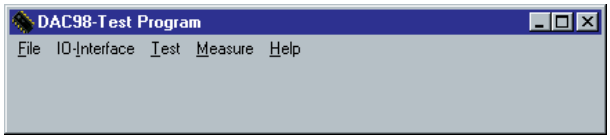


Figure 3.1: The main window of the DAC98 test software

3.3 Menu 'File'

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

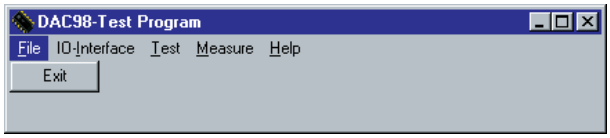
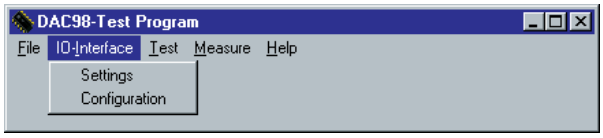


Figure 3.2: The menu 'File'

3.4 Menu 'IO-Interface'

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



Figue 3.3: The menu 'IO-Interface'

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

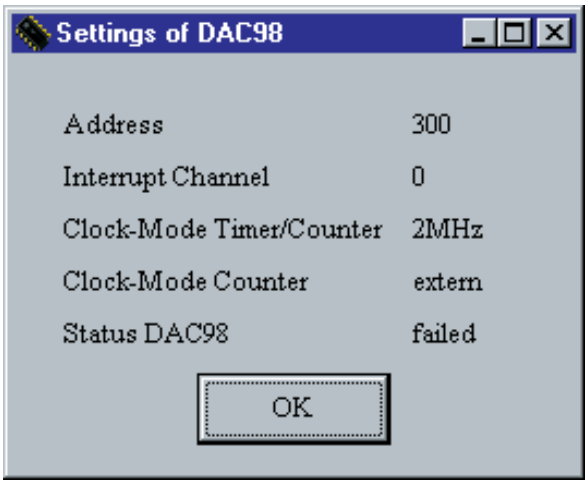


Figure 3.4: The window 'Settings'

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!

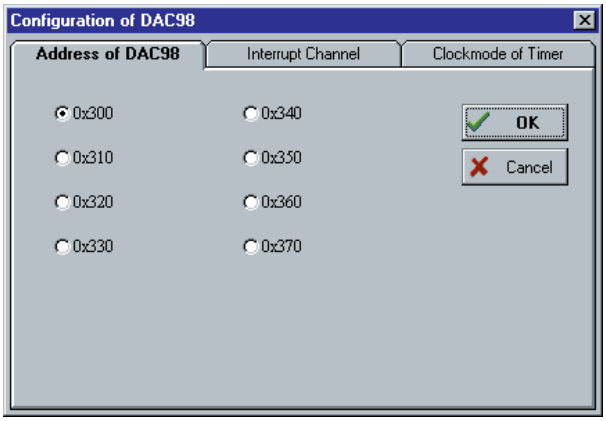


Figure 3.5: The window 'Configuration'

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.

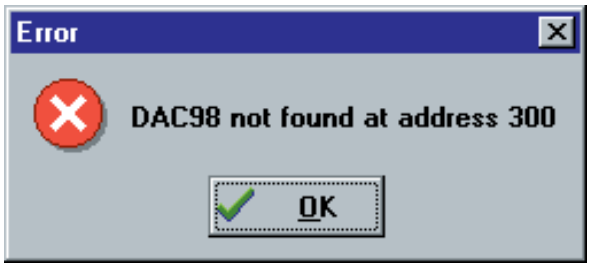


Figure 3.6: The window 'Error'

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.

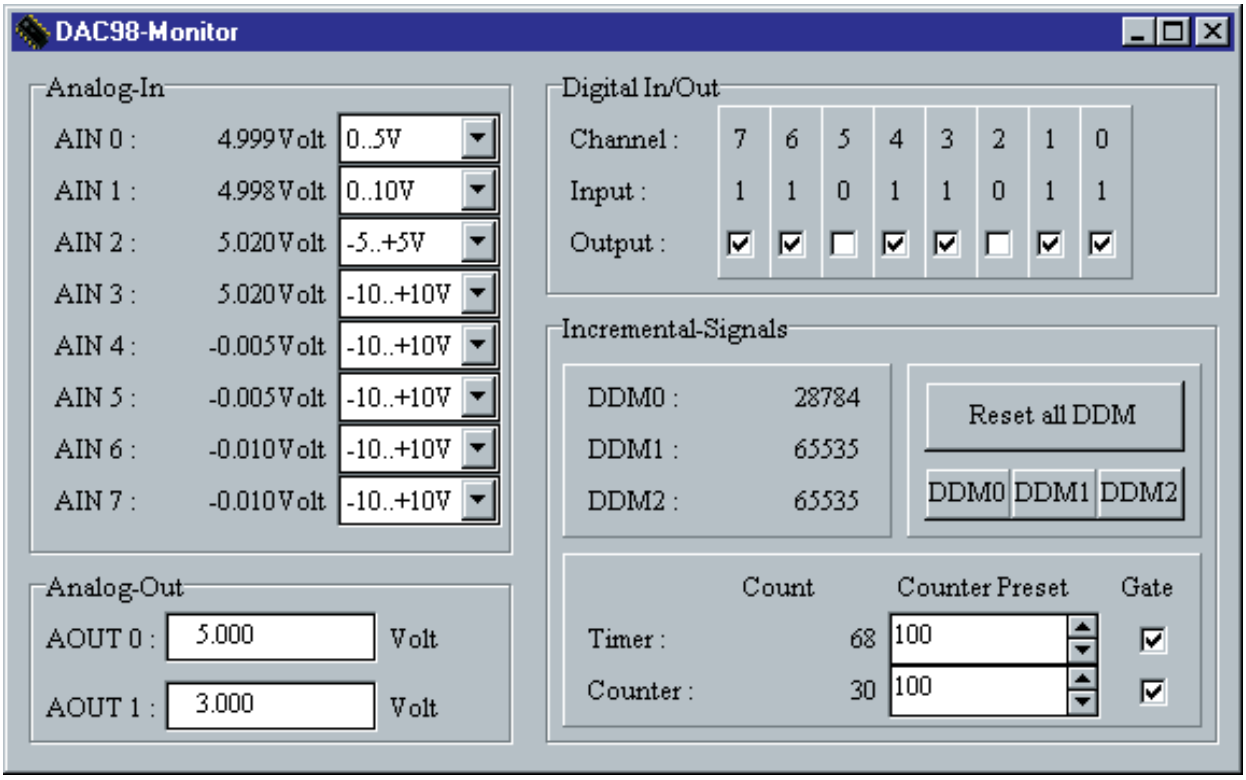


Figure 3.7: The window 'Showall'

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.

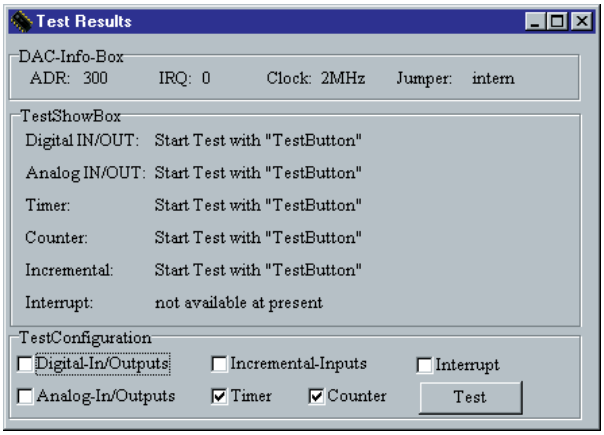


Figure 3.8: The window 'Test Components'

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.

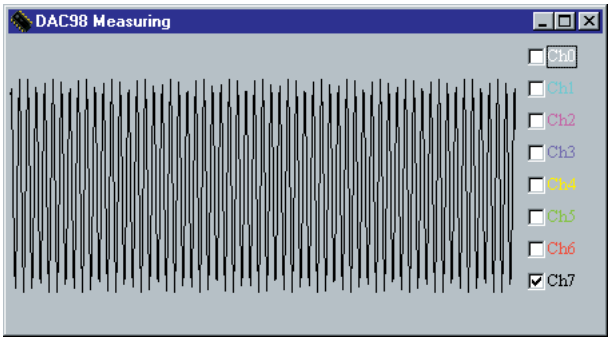


Figure 3.9: The window 'Measure'

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.

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**.

DTS200 Windows Software V1.3

Printed 23. May 2002

1 Source Files of the DTS200 Controller Program 1-1

1 Source Files of the DTS200 Controller Program

1.1 General

The program is a 16-bit application, which may be started only once by the operating systems Windows 3.1 or Windows95. The desktop is created by means of the program language 'Pascal', while the actual controller is realized by a DLL developed with the program language 'C++'. Both program parts are available in source completely. The program package is completed by
the TIMER16.DLL to handle the cyclic controller calls,
the card drivers DAC6214.DRV or DAC98.DRV to access the PC adapter card,
the PLOT16.DLL for the graphic output,
the help file DTS200.HLP.

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 DTS200).

You will then find the following subdirectories:

DTSDESK
CONTRDLL
EXE

Where DTSDESK contains the *Delphi Project File* DTS200W.DPR together with all the accompanying Pascal source files to generate the desktop, CONTRDLL contains the *Borland Project File* DTSDLL.IDE with all the accompanying C++ source files to create the controller-DLL (DTSSRV16.DLL). Finally the subdirectory EXE contains all the additional files required by the executable program (DTS200.HLP, DTSW16.INI, DEFAULT.CAL, DAC*.DRV, TIMER16.DLL, PLOT16.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 three-tank-system instead of accessing the PC adapter card) may be obtained simply by setting the macro __DTS_DEMO__ in the include file DTSDEFIN.H and generating a new DTSSRV16.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 to a different subdirectory (i.e. DEMO) afterwards.

1.2 Global Data and Functions

The file **DTSDEFIN.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 __DTS_DEMO__ is defined all program functions besides system calibration are available for a simulated three tank system. The simulation of the system behaviour runs ten times faster than the real system. The PC adapter card is not required in this program mode. To control the real system the macro __DTS_DEMO__ must not be defined!

Used definitions:

```
//#define __DTS_DEMO__
#define TRUE                1
#define FALSE               0
#define SAMPLETIME      0.05
#ifdef __DTS_DEMO__
        #define SIMTIME     10.0 *SAMPLETIME
#define ERROR               -1
#define OK                  0
#define AVALVE              0.5     // Sn     [cm^2]
#define ATANK               154.0  // Aq     [cm^2]
#define TWOG                1962.0 // 2.0 * g [cm/s^2]
```

The file **DTS20DAT.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];   // P335
        char    name[10];     // DTS200
        char    Titel[20];    // DTS200 - Monitor
        char    Version[10];  // 1.30
        char    Date[10];     // 08.11.01
        char    Dummy[10];    // Reserve
};
static struct PROJECT PRJ = {
        "P335",
        "DTS200",
        "DTS200-Monitor",
        "V1.30WIN",
        "08-NOV-01",
        "not used"
};
CTRLSTATUS {
        int OpenLoop;         // Flag open loop control is active
```

```
        int PX;                 // Flag P controller is active
        int PI;                 // Flag PI controller is active
        int DeCoupled;          // Flag decoupling is active
        double P;               // Coefficient P of the P or PI controller
        double K;               // Coefficient Ki of the PI controller
        long timeofmeasure; // Date and time of the measurement acquisition
};

struct DATASTRUCT {     // Structure to reconstruct the measured data
        int      nchannel;      // Length of the stored measurement vectors (number of channels)
        int     nvalues;        // Number of the measurement vectors (measurement acquisitions)
        float   deltatime;      // Time between two measurements
};
```

The Format of the Documentation Data File *.PLD

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

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

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

The size of the data array is defined in the structure DATASTRUKT. With the DTS200 the number of the stored channels is always 7 (the length of the measurement vector is 7, i.e. equal to 28 bytes). The vector contains the following signals:

the setpoint for tank 1 (in cm),
the setpoint for tank 2 (in cm),
the measured liquid level from tank 1 (in cm),
the measured liquid level from tank 2 (in cm),
the measured liquid level from tank 3 (in cm),
the control signal for pump 1 (in ml/s),
the control signal for pump 2 (in ml/s),

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 program's 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:

FormCreate(*Sender*: TObject)

FormShow(*Sender*: TObject)

CanCloseForm(*Sender*: TObject; *var CanClose*: Boolean)

FormClose(*Sender*: TObject; *var Action*: TCloseAction)

FormDestroy(*Sender*: TObject)

FileMenuClick(*Sender*: TObject)

SavePlotData1Click(*Sender*: TObject)

LoadPlotData1Click(*Sender*: TObject)

SaveSystemParameter1Click(*Sender*: TObject)

LoadSystemParameter1Click(*Sender*: TObject)

FilePrint(*Sender*: TObject)

FilePrintSetup(*Sender*: TObject)

FileExit(*Sender*: TObject)

IOInterface1Click(*Sender*: TObject)

DAC98Click(*Sender*: TObject)

DAC6214Click(*Sender*: TObject)

DACSetupClick(*Sender*: TObject)

Parameter1Click(*Sender*: TObject)

DecouplingController1Click(*Sender*: TObject)

PIController1Click(*Sender*: TObject)

SimulatedSystemErrors1Click(*Sender*: TObject)

CharacteristicLiquidLevelSensor1Click(*Sender*: TObject)

OutflowCoefficient1Click(*Sender*: TObject)

CharacteristicPumpFlowRate1Click(*Sender*: TObject)

CrossSectionsofTanks1Click(*Sender*: TObject)

Run1Click(*Sender*: TObject)

OpenLoopControl1Click(*Sender*: TObject)

DecouplingController2Click(*Sender*: TObject)

PIController2Click(*Sender*: TObject)

ResetPIController1Click(*Sender*: TObject)

StopAll1Click(*Sender*: TObject)

AdjustSetpoint1Click(*Sender*: TObject)

Measuring1Click(*Sender*: TObject)

View1Click(*Sender*: TObject)

PlotMeasuredData1Click(*Sender*: TObject)

PlotFileData1Click(*Sender*: TObject)

ParametersfromPLDFile1Click(*Sender*: TObject)

CharacteristicPumpFlowRates1Click(*Sender*: TObject)

OutflowCoefficients1Click(*Sender*: TObject)

ControlStructure1Click(*Sender*: TObject)

HelpContents(*Sender*: TObject)

HelpSearch(*Sender*: TObject)

HelpHowToUse(*Sender*: TObject)

HelpAbout(*Sender*: TObject)

Timer1Timer(*Sender*: TObject)

GBControlClick(*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 CALSENS.PAS contains the procedures:

BitBtn1Click(*Sender*: TObject)

FormShow(*Sender*: TObject)

CancelBtnClick(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file CTRLPICT.PAS contains the procedures:

FormCreate(*Sender*: TObject)

UpdateData

Paint(*Sender*: TObject)

FormDestroy(*Sender*: TObject) und die Funktionen:

DrawController(*xStart, yStart, TankNr* : Integer)

DrawArrow(*xStart, yStart, Length* : Integer)

DrawArrowVert(*xStart, yStart, Length* : Integer)

The file MEASURE.PAS contains the procedures:

OKBtnClick(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file OUTFLOW.PAS contains the procedures:

BitBtn1Click(*Sender*: TObject)

BitBtn2Click(*Sender*: TObject)

FormShow(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file PIPARM.PAS contains the procedures:

FormShow(*Sender*: TObject)

OKBtnClick(*Sender*: TObject)

CheckBox1Click(*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 PPARM.PAS contains the procedures:

FormShow(*Sender*: TObject)

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 PUMPCHAR.PAS contains the procedures:

FormShow(*Sender*: TObject)

BitBtn1Click(*Sender*: TObject)

CancelBtnClick(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file SETPOINT.PAS contains the procedures:

OKBtnClick(*Sender*: TObject)

FormShow(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file SIMULAT.PAS contains the procedures:

OKBtnClick(*Sender*: TObject)

HelpBtnClick(*Sender*: TObject)

The file TANKDIM.PAS contains the procedures:

FormShow(*Sender*: TObject)

OKBtnClick(*Sender*: TObject)

The file TOOLS.PAS contains the functions:

FloatToStr2(*f* : Single) : string

FloatToStr3(*f* : Single) : string

FloatToStr4(*f* : Single) : string

StrToFloatMinMax(*s* : string; *min,max* : double) : double

StrToFloatStrMinMax(*s* : string; *var val* : double; *min,max* : double) : string

MinMaxi(*val, min, max* : Integer) : Integer

FloatToStr4(*f* : Single) : string

The file DLLS.PAS contains besides the global data definitions the interface definitions for the DLL's DTSSRV16 and TIMER16.

Global Data:

```
type ServiceParameters = record
        controller, decoupled : WORD;
        p, i, decoup : double;
        sp1shape : WORD;
        sp1offset, sp1amplitude, sp1periode : double;
        sp2shape : WORD;
        sp2offset, sp2amplitude, sp2periode : double;
        At1, At2, At3 : double;
        end;
type ServiceData = record
        t1setpoint, t2setpoint : double;
        t1level, t2level, t3level : double;
        p1flow, p2flow : double;
        end;


param : ServiceParameters;
data : ServiceData;
```

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 DTS200W program) may be started only once.

TMainForm.FormShow

FormShow(*Sender*: TObject)

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

Description The procedure **FormShow** is called when the object of type TMainForm is displayed on the screen. This is equivalent to starting the application (the program DTS200W). The driver for the PC adapter card is loaded at first (its name is contained in the file DTSW16.INI). The markings below the menu item "IO-Interface" are set accordingly. At next it is determined by calling up the DTSSRV16.DLL (**IsDemo**) if it is a DEMO version. Only in this case the menu item "IO-Interface" is disabled, the menu item "Simulated System Errors" is enabled and the string "(Demo-Version)" is appended to the title of the monitor. The parameter structure *param* is set to default values (no controller, proportional factor=10, integral factor=1, decoupling factor=0.04, setpoint tank 1 constant 30 cm, setpoint tank 2 constant 15 cm) and transferred to the DTSSRV16.DLL (**SetParameter**). The state for reading the documentation file (extension *.PLD) is reset. The timer handling the sampling period of the controller is started (**StartTimer**, TIMER.DLL).

TMainForm.CanCloseForm

CanCloseForm(*Sender*: TObject; *var CanClose*: Boolean)

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

var CanClose is set on return according to prompts.

Description The procedure **CanCloseForm** controls the termination of the program by **FormClose** by setting the variable *CanClose*. At first the message box "Exit program?" is shown. Prompting this message by 'Ok' will set *CanClose* to TRUE, otherwise to FALSE. When a sensor calibration was carried-out previously without saving the system parameters to a file, another message box "Exit without saving calibration data?" is shown. If this message is not prompted by 'Ok', saving the system parameters (**SaveSystemParameter1Click**) is carried-out automatically.

TMainForm.FormClose

FormClose(*Sender*: TObject; *var Action*: TCloseAction)

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

var Action unused.

Description The procedure **FormClose** stops the timer controlling the monitor outputs.

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 object *single* is released.

TMainForm.FileMenuClick

FileMenuClick(*Sender*: TObject)

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

Description The procedure **FileMenuClick** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "File". When measurements are available in the memory the menu item "Save Recorded Data" is enabled, else it is disabled. When no controller is active, the menu items "Save System Parameter" and "Load System Parameter" are enabled, else they are disabled.

TMainForm.SavePlotData1Click

SavePlotData1Click(*Sender*: TObject)

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

Description The procedure **SavePlotData1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "File/Save Recorded Data". A Windows system dialog appears offering the storage (**WritePlot**) of the current measurements contained in the memory to a documentation file with an adjustable name (extension *.PLD).

TMainForm.LoadPlotData1Click

LoadPlotData1Click(*Sender*: TObject)

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

Description The procedure **LoadPlotData1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "File/Load Measured Data". A Windows system dialog appears, which allows for selecting a documentation file with an adjustable name (extension *.PLD) to load measurements (**ReadPlot**) into the memory.

TMainForm.SaveSystemParameters1Click

SaveSystemParameters1Click(*Sender*: TObject)

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

Description The procedure **SaveSystemParameters1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "File/Save System Parameters". A Windows system dialog appears offering the storage (**SaveCalibration**) of the current calibration data (sensor and pump characteristics, outflow coefficients, cross sections of tanks) to a file with an adjustable name (extension *.CAL).

TMainForm.LoadSystemParameters1Click

LoadSystemParameters1Click(*Sender*: TObject)

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

Description The procedure **LoadSystemParameters1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "File/Load System Parameters". A Windows system dialog appears, which allows for selecting a file with an adjustable name (extension *.CAL) to load calibration data (sensor and pump characteristics, outflow coefficients, cross sections of tanks) into the memory (**LoadCalibration**).

TMainForm.FilePrint

FilePrint(*Sender*: TObject)

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

Description The procedure **FilePrint** generates a modal dialog (**PrintPlotDlg**) to select previously created plot windows, which are to be printed to an output device (i.e. printer).

TMainForm.FilePrintSetup

FilePrintSetup(*Sender*: TObject)

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

Description The procedure **FilePrintSetup** calls the standard printer setup dialog of Windows to select and adjust the output device.

TMainForm.FileExit

FileExit(*Sender*: TObject)

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

Description The procedure **FileExit** terminates the running application, that means exits the program DTS200W.

TMainForm.IOInterface1Click

IOInterface1Click(*Sender*: TObject)

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

Description The procedure **IOInterface1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "IO-Interface". The menu items "DAC98" and "DAC6214" are enabled only when the accompanying drivers DAC98.DRV and DAC6214.DRV are existent. The currently selected driver is emphasized with a check mark.

TMainForm.DAC98Click

DAC98Click(*Sender*: TObject)

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

Description The procedure **DAC98Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "IO-Interface/DAC98". The name DAC98.DRV of the driver file is written to the file DTSW16.INI. After stopping the timer handling the sampling period of the controller the driver DAC98.DRV is loaded. Then the timer is started again and the check marks for the selected driver are set accordingly.

TMainForm.DAC6214Click

DAC6214Click(*Sender*: TObject)

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

Description The procedure **DAC6214Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "IO-Interface/DAC6214". The name DAC6214.DRV of the driver file is written to the file DTSW16.INI. After stopping the timer handling the sampling period of the controller the driver DAC6214.DRV is loaded. Then the timer is started again and the check marks for the selected driver are set accordingly.

TMainForm.DACSetupClick

DACSetupClick(*Sender*: TObject)

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

Description The procedure **DACSetupClick** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "IO-Interface/DAC Setup". After stopping the timer handling the sampling period of the controller a dialog appears to adjust the base address of the adapter card. The timer is started again after terminating this dialog.

TMainForm.Parameters1Click

Parameters1Click(*Sender*: TObject)

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

Description The procedure **Parameters1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Parameter". The menu items to record the calibration data (sensor and pump characteristics, outflow coefficients) are enabled when no controller is active. These menu items are disabled in case of an active controller or a DEMO version.

TMainForm.DecouplingController1Click

DecouplingController1Click(*Sender*: TObject)

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

Description The procedure **DecouplingController1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Parameters/Decoupling Controller". A modal dialog (**DecoupParamDlg**) appears to adjust the decoupling parameter.

TMainForm.PIController1Click

PIController1Click(*Sender*: TObject)

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

Description The procedure **PIController1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Parameters/PI-Controller". A modal dialog (**PIParamDlg**) appears to adjust the proportional and integral factor of the PI controller as well as the decoupling parameter only if the decoupling structure was selected in addition.

TMainForm.SimulatedSystemErrors1Click

SimulatedSystemErrors1Click(*Sender*: TObject)

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

Description The procedure **SimulatedSystemErrors1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Parameters/Simulated System Errors". A modal dialog (**SimulationDlg**) appears to adjust simulated system errors (sensor and control signal errors as well as leaks and clogs) only for the DEMO version.

TMainForm.CharacteristicLiquidLevelSensor1Click

CharacteristicLiquidLevelSensor1Click(*Sender*: TObject)

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

Description The procedure **CharacteristicLiquidLevelSensor1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Parameters/Characteristic Liquid Level Sensor". After switching off any currently active controller a modal dialog (**CalcSensDlg**) appears supporting the user with recording the sensor characteristics.

TMainForm.OutflowCoefficient1Click

OutflowCoefficient1Click(*Sender*: TObject)

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

Description The procedure **OutflowCoefficient1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Parameters/OutflowCoefficient". After switching off any currently active controller a modal dialog (**OutFlowCoefDlg** appears supporting the user with recording the outflow coefficients.

TMainForm.CharacteristicPumpFlowRate1Click

CharacteristicPumpFlowRate1Click(*Sender*: TObject)

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

Description The procedure **CharacteristicPumpFlowRate1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Parameters/Characteristic Pump Flow Rate". After switching off any currently active controller a modal dialog (**PumpCharDlg**) appears supporting the user with recording the pump characteristics.

TMainForm.CrossSectionsofTanks1Click

CrossSectionsofTanks1Click(*Sender*: TObject)

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

Description The procedure **CrossSectionsofTanks1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Parameters/Cross Sections of Tanks". The modal dialog (**TankDimsDlg**) appears allowing the user view or to enter the effective cross sections of the tanks.

TMainForm.Run1Click

Run1Click(*Sender*: TObject)

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

Description The procedure **Run1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Run". The menu items to select a controller are enabled and the menu items to adjust the setpoints and to start the measurement acquisition are disabled when no controller is active. In the opposite case the menu items to select a controller are disabled and the menu items to adjust the setpoints and to start the measurement acquisition are enabled.

TMainForm.OpenLoopControl1Click

OpenLoopControl1Click(*Sender*: TObject)

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

Description The procedure **OpenLoopControl1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Run/Open Loop Control". The flags indicating a selected controller (for the menu items as well as in the parameter structure *param*) are set to OPENLOOP and the parameters of the two setpoint generators are reset. The parameter structure changed in such a way is transferred to the DTSSRV16.DLL (**SetParameter**). All adjustments are carried-out only in case open loop control was inactive previously.

TMainForm.DecouplingController2Click

DecouplingController2Click(*Sender*: TObject)

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

Description The procedure **DecouplingController2Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Run/Decoupling Controller". The flags indicating a selected controller (for the menu items as well as in the parameter structure *param*) are set to DECOUPLED (proportional decoupling controller) and the parameters of the two setpoint generators are set to standard values (setpoint tank 1 constant 30 cm, setpoint tank 2 constant 15 cm). The parameter structure changed in such a way is transferred to the DTSSRV16.DLL (**SetParameter**). All adjustments are carried-out only in case the proportional decoupling controller was inactive previously.

TMainForm.PIController2Click

PIController2Click(*Sender*: TObject)

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

Description The procedure **PIController2Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Run/PI-Controller". The flags indicating a selected controller (for the menu items as well as in the parameter structure *param*) are set to PICONTROLLER (PI controller) and the parameters of the two setpoint generators are set to standard values (setpoint tank 1 constant 30 cm, setpoint tank 2 constant 15 cm). The parameter structure changed in such a way is transferred to the DTSSRV16.DLL (**SetParameter**). All adjustments are carried-out only in case the PI controller was inactive previously.

TMainForm.ResetPIController1Click

ResetPIController1Click(*Sender*: TObject)

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

Description The procedure **ResetPIController1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Run/Reset PI-Controller". The function **ResetPIController** is called to reset the PI controller states.

TMainForm.StopAll1Click

StopAll1Click(*Sender*: TObject)

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

Description The procedure **StopAll1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Run/Stop All". The flags indicating a selected controller (for the menu items as well as in the parameter structure *param*) are reset. The parameter structure changed in such a way is transferred to the DTSSRV16.DLL (**SetParameter**).

TMainForm.AdjustSetpoint1Click

AdjustSetpoint1Click(*Sender*: TObject)

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

Description The procedure **AdjustSetpoint1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Run/Adjust Setpoint". A modal dialog (**GeneratorDlg**) appears for adjusting the tank level setpoints or the pump control signals respectively.

TMainForm.Measuring1Click

Measuring1Click(*Sender*: TObject)

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

Description The procedure **Measuring1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "Run/Measuring". A modal dialog (**MeasureDlg**) appears for adjusting the conditions of a measurement acquisition.

TMainForm.View1Click

View1Click(*Sender*: TObject)

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

Description The procedure **View1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "View". The menu item "Plot Measured Data" is enabled when measurements are available in the memory, else it is disabled. When data from a documentation file (extension *.PLD) have been loaded, the menu items "Plot File Data" and "Parameter from *.PLD File" are enabled, else they are disabled.

TMainForm.PlotMeasuredData1Click

PlotMeasuredData1Click(*Sender*: TObject)

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

Description The procedure **PlotMeasuredData1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "View/Plot Measured Data". A modal dialog (**PlotDlg**) appears to select those components of the last measurement acquisition which are to be represented 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 for clicking once on the calling object (*Sender*), which was activated by the menu item "View/Plot File Data". A modal dialog (**PlotDlg**) appears to select those components of a loaded documentation file which are to be represented 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 for clicking once on the calling object (*Sender*), which was activated by the menu item "View/Parameters from *.PLD File". A modal dialog (**PLDInfoDlg**) displaying the essential controller settings from a loaded documentation file appears directly.

TMainForm.CharacteristicLiquidLevelSensors

CharacteristicLiquidLevelSensors(*Sender*: TObject)

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

Description The procedure **CharacteristicLiquidLevelSensors** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "View/Characteristic Liquid Level Sensors". A plot window representing the three sensor characteristics will appear directly.

TMainForm.CharacteristicPumpFlowRates1Click

CharacteristicPumpFlowRates1Click(*Sender*: TObject)

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

Description The procedure **CharacteristicPumpFlowRates1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "View/Characteristic Pump Flow Rates". A plot window representing the two pump characteristics will appear directly.

TMainForm.OutflowCoefficients1Click

OutflowCoefficients1Click(*Sender*: TObject)

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

Description The procedure **OutflowCoefficients1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "View/Outflow Coefficients". A message box displaying the current outflow coefficients will appear directly.

TMainForm.ControlStructure1Click

ControlStructure1Click(*Sender*: TObject)

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

Description The procedure **ControlStructure1Click** is an event handler for clicking once on the calling object (*Sender*), which was activated by the menu item "View/Control Structure". The window showing to two control loop structures is set to be visible.

TMainForm.HelpContents

HelpContents(*Sender*: TObject)

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

Description The procedure **HelpContents** is activated by the menu item "Help/Contents" to display the content of the help file.

TMainForm.HelpSearch

HelpSearch(*Sender*: TObject)

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

Description The procedure **HelpSearch** is activated by the menu item "Help/Search for Help On" to start the Windows dialog to search for selectable keywords.

TMainForm.HelpHowToUse

HelpHowToUse(*Sender*: TObject)

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

Description The procedure **HelpHowToUse** is activated by the menu item "Help/How to Use help" to start the Windows dialog displaying hints how to use the help function.

TMainForm.HelpAbout

HelpAbout(*Sender*: TObject)

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

Description The procedure **HelpAbout** is activated by the menu item "Help/About" displaying information about the actual program in a message box.

TMainForm.Timer1Timer

Timer1Timer(*Sender*: TObject)

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

Description The procedure **Timer1Timer** is activated every 100 ms by a timer. Each time the data structure *data* belonging to the controller in the DTSSRV16.DLL is read and the contents of the monitor in the main window are updated. The monitor displays the current controller type, setpoints, measured values, control errors, control signals, pump flow rates as well as the state of the measurement acquisition. If the window containing the two control loop structures is visible, it is updated in addition.

TMainForm.GBControlClick

GBControlClick(*Sender*: TObject)

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

Description The procedure **GBControlClick** is an event handler for clicking once on the label "Active Controller" in the upper filed of the "DTS-Monitor" window. The window showing the two control loop structures is set to be visible.

TSingleInstance.WndProc

WndProc(*var Msg*: TMessage)

Parameters: *var Msg* is the actual 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.

TCalSensDlg.BitBtn1Click

BitBtn1Click(*Sender*: TObject)

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

Description The procedure **BitBtn1Click** is an event handler for clicking once on the "measure_1" button from the dialog to calibrate the liquid level sensors. This button as well as the input field for the lower calibration point in [cm] are disabled. The content of this input field is converted to a number and taken as the lower calibration point (DACDRV::**SenSetLevel**) only when it is inside the range from 0 to 60 [cm]. The current sensor value in [Volt] for the liquid level of each tank (the user has to adjust this level by controlling the pumps manually) is recorded and assigned to the calibration point (DACDRV::**SenEichLevel**). Finally the "measure_2" button is enabled.

TCalSensDlg.BitBtn2Click

BitBtn2Click(*Sender*: TObject)

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

Description The procedure **BitBtn2Click** is an event handler for clicking once on the "measure_2" button from the dialog to calibrate the liquid level sensors. This button as well as the input field for the upper calibration point in [cm] are disabled. The content of this input field is converted to a number and taken as the upper calibration point (DACDRV::**SenSetLevel**) only when it is inside the range from 0 to 60 [cm]. The current sensor value in [Volt] for the liquid level of each tank (the user has to adjust this level by controlling the pumps manually) is recorded and assigned to the calibration point (DACDRV::**SenEichLevel**). Finally the modal dialog is terminated.

TCalSensDlg.FormShow

FormShow(*Sender*: TObject)

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

Description The procedure **FormShow** is called, when the object of type TCalSensDlg is displayed on the screen. The input fields for the lower and upper calibration point in [cm] are enabled and filled with the current values for these points (DACDRV::**SenGetLevel**). The "measure_1" button is enabled while the "measure_2" button is disabled to guarantee the sequence of the sensor calibration.

TCalSensDlg.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 calibrate the liquid level sensors. The accompanying section of the help file will be displayed in a window on the screen.

TFCtrlPict.FormCreate

FormCreate(*Sender*: TObject)

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

Description The procedure **FormCreate** creates an instance of an object type TFCtrlPict. A bitmap to show the two control loop structures is initialized.

TFCtrlPict.UpdateData

UpdateData

Description The procedure **UpdateData** is called from the main window (timer in **TMainForm) to show an updated bitmap of the two control loop structures.**

TFCtrlPict.Paint

Paint(*Sender*: TObject)

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

Description The procedure **Paint** draws a background picture of the two control loop structures and copies it to the visible window.

TFCtrlPict.DrawController

TFCtrlPict.DrawController(*xStart, yStart, TankNr* : Integer) : Integer

Parameters: *xStart* Picture start (pixel) in horizontal direction.

yStart Picture start (pixel) in vertical direction.

TankNr Number of the tank (1 or 2).

Description The function **TFCtrlPict.DrawController** draws the structure of one tank control loop to a bitmap either for tank 1 or tank 2. The structure is an open loop control, a decoubling network or a PI-control structure with or without decoupling. The tank level is shown in an animated picture. The level setpoint, the mesasured level as well as the control signal are displayed as decimal numbers in addition.

Return without meaning.

TFCtrlPict.DrawArrow

TFCtrlPict.DrawArrow(*xStart, yStart, Length* : Integer) : Integer

Parameters: *xStart* Starting position (pixel) in horizontal direction.

yStart Starting position (pixel) in vertical direction.

Length Length of the arrow.

Description The function **TFCtrlPict.DrawArrowVert** draws an arrow of lenght *Length* + 5 pixel for the arrow head to the right (if *Length* < 0 to the left) beginning with the starting position.

Return The ending position (pixel) of the arrow in horizontal direction.

TFCtrlPict.DrawArrowVert

TFCtrlPict.DrawArrowVert(*xStart, yStart, Length* : Integer) : Integer

Parameters: *xStart* Starting position (pixel) in horizontal direction.

yStart Starting position (pixel) in vertical direction.

Length Length of the arrow.

Description The function **TFCtrlPict.DrawArrow** draws an arrow of lenght *Length* + 5 pixel for the arrow head upward (if *Length* < 0 downward) beginning with the starting position.

Return The ending position (pixel) of the arrow in vertical direction.

TFCtrlPict.FormDestroy

FormDestroy(*Sender*: TObject)

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

Description The procedure **FormDestroy** is called before the object of type TFCtrlPict is removed from the memory. In this connection the bitmap memory is released.

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 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 Meßzeit) and for the trigger level (*trigger* = 0 to 100) 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 setpoints, liquid levels as well as the pump control signals.

TMeasureDlg.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 adjust the conditions for the measurement acquisition. The accompanying section of the help file will be displayed in a window on the screen.

TOutFlowCoefDlg.BitBtn1Click

BitBtn1Click(*Sender*: TObject)

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

Description The procedure **BitBtn1Click** is an event handler for clicking once on the "Start" button from the dialog to determine (calibrate) the outflow coefficients. After a delay time of 60 sec the actual measurements are taken in between 15 sec (**OfcEichStart, OfcEichStop**). If the "Cancel" button is not pressed during the overall time of 75 sec, the progress of which is displayed in a bar in percentage below the "Start" button, the determined outflow coefficients are displayed in a message box. Prompting this message will also terminate the dialog.

TOutFlowCoefDlg.FormShow

FormShow(*Sender*: TObject)

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

Description The procedure **FormShow** is called, when the object of type TOutFlowCoefDlg is displayed on the screen. The "Start" button is enabled and the display for the progress of the measurement time is reset.

TOutFlowCoefDlg.CancelBtnClick

CancelBtnClick(*Sender*: TObject)

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

Description The procedure **CancelBtnClick** is an event handler for clicking once on the "Cancel" button. The dialog is either aborted directly or indirectly by resetting the flag of a current measurement.

TOutFlowCoefDlg.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 determine the outflow coefficients. The accompanying section of the help file will be displayed in a window on the screen.

TPIParamDlg.FormShow

FormShow(*Sender*: TObject)

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

Description The procedure **FormShow** is called, when the object of type TPIParamDlg is displayed on the screen. The proportional and integral factor from the PI controller as well as the decoupling parameter from the global parameter structure *param* are converted to ASCII-strings which are inserted in the three input fields of the dialog. When the control structure is selected with decoupling the corresponding check mark is emphasized.

TPIParamDlg.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 adjust the proportional and integral factor from the PI controller as well as the decoupling parameter. The contents of the input fields of the dialog are converted to numbers when each of these numbers does not exceed the range from 0 to 100. The control structure is set with or without decoupling according to the check mark. The global parameter structure *param* is transferred to the controller of the DTSSRV16.DLL.

TPIParamDlg.CheckBox1Click

CheckBox1Click(*Sender*: TObject)

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

Description The procedure **CheckBox1Click** is an event handler for clicking once on the check mark for the decoupling structure. The input field to enter the decoupling parameter is enabled or disabled accordingly.

TPIParamDlg.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 adjust the parameters of the PI controller. The accompanying section of the help file 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 essential controller settings from a loaded documentation file are displayed in a dialog. The settings consist of the controller structure (PI controller with/without decoupling, decoupling controller, open loop control), the accompanying parameters, the measuring time as well as the sampling period of the measuring.

TPLDInfoDlg.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 display the essential controller settings from a loaded documentation file. The accompanying section of the help file 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 to select the channels of a measuring which are to be represented in a plot window. The selectable channels are the liquid levels, the liquid levels together with the setpoints, the pump flow rates, the pump control signals as well as the liquid levels together with the pump flow rates.

TPlotDlg.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 the channels of a measuring which are to be represented in a plot window. The accompanying section of the help file will be displayed in a window on the screen.

TDecoupParamDlg.FormShow

FormShow(*Sender*: TObject)

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

Description The procedure **FormShow** is called, when the object of type TDecoupParamDlg is displayed on the screen. The decoupling parameter from the global parameter structure *param* is converted to an ASCII-string which is inserted in the input field of the dialog.

TDecoupParamDlg.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 adjust the decoupling parameter. The content of the input field of the dialog is converted to a number when this number does not exceed the range from 0 to 100. The global parameter structure *param* is transferred to the controller of the DTSSRV16.DLL.

TDecoupParamDlg.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 adjust the decoupling parameter. The accompanying section of the help file 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 will be displayed in a window on the screen.

TPumpCharDlg.FormShow

FormShow(*Sender*: TObject)

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

Description The procedure **FormShow** is called, when the object of type TPumpCharDlg is displayed on the screen. The "Start" as well as the "Cancel" button are enabled, the bar indicating the progress of the measurement is reset and a table containing 3 columns and 9 rows is initialized.

TPumpCharDlg.BitBtn1Click

BitBtn1Click(*Sender*: TObject)

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

Description The procedure **BitBtn1Click** is an event handler for clicking once on the "Start" from the dialog to determine (calibrate) the pump characteristics (each containing 9 base points for the pump control signal and the accompanying pump flow rate). The pump flow rates are determined in a loop with 8 passes. In each pass the pump control signals are incremented by 12,5% and supplied to the pumps during a time period of 10 sec (**PfrEichStart, PfrEichStop**). After another settling time 0f 2,5 sec the tank levels are measured. At the end of each pass the next row of the table is filled with the values for the control signal and the determined pump flow rate in [ml/s] (**PfrEichGet**). The progress of the loop is indicated in a bar below the "Start" button in percentage. If the "Cancel" button is not pressed during the loop an "Ok" button will appear to terminate the dialog.

TPumpCharDlg.CancelBtnClick

CancelBtnClick(*Sender*: TObject)

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

Description The procedure **CancelBtnClick** is an event handler for clicking once on the "Cancel" button. The dialog is either aborted directly or indirectly by resetting the flag of a current measurement.

TPumpCharDlg.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 determine the pump characteristics. The accompanying section of the help file 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 to adjust the two setpoints, respectively the two pump control signal generators. For each generator 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 - 60 cm for the setpoints, respectively 0 - 100 ml/s for the pump control signals 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 DTSSRV16.DLL. Finally the dialog is terminated and both generators operate synchronously.

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 two generators are preset according to the parameters of the global parameter structure *param*. The two generators provide the liquid level setpoints for tank1 and tank 2 in the closed control loop or the pump control signals in the open control loop.

TGeneratorDlg.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 adjust the two setpoints, respectively the two pump control signal generators. The accompanying section of the help file will be displayed in a window on the screen.

TSimulationDlg.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 adjust the simulated system errors (sensor and pump errors, leaks and clogs) in the DEMO version. The contents of the input fields of the dialog are converted to numbers when each of these numbers does not exceed the range from 0 to 100 (%). The numbers are stored in the corresponding global parameters.

TSimulationDlg.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 adjust the simulated system errors in the DEMO version. The accompanying section of the help file will be displayed in a window on the screen.

TTankDimsDlg.FormShow

FormShow(*Sender*: TObject)

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

Description The procedure **FormShow** is called, when the object of type TTankDimsDlg is displayed on the screen. The parameter structure *param* is read and its corresponding values of the tanks cross setions are entered as an string to the edit fields of the dialog.

TTankDimsDlg.OKBtnClick

OKBtnClick(*Sender*: TObject)

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

Description The procedure **OKBtnClick** is an event handler auf ein for clicking once on the ’Ok’ button from the dialog to adjust the tank cross sections. Only when the three numbers are limited to the range from 10 to 1000 (cm^2), its values are copied to the parameter structure *param* which is then send to the DTSSRV16.DLL.

FloatToStr2

FloatToStr2(*f* : Single) : string

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

Description The function **FloatToStr2** converts a floating point value (4 bytes for single) to a its string representation with a maximum of 7 significant digits and 2 digits behind the decimal point.

Return Is the string representation of the floating point value.

FloatToStr3

FloatToStr3(*f* : Single) : string

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

Description The function **FloatToStr3** converts a floating point value (4 bytes for single) to a its string representation with a maximum of 7 significant digits and 3 digits behind the decimal point.

Return Is the string representation of the floating point value.

FloatToStr4

FloatToStr4(*f* : Single) : string

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

Description The function **FloatToStr4** converts a floating point value (4 bytes for single) to a its string representation with a maximum of 7 significant digits and 4 digits behind the decimal point.

Return Is the string representation of the floating point value.

StrToFloatMinMax

StrToFloatMinMax(*s* : string; *min,max* : double) : double

Parameters: *s* is the string representation of a floating point value.

min is the lower limit for a floating point value.

max is the upper limit for a floating point value.

Description The function **StrToFloatMinMax** converts a string to the corresponding floating point value. When this value exceeds the lower or upper limit it is set equal to the exceeded limit and a corresponding message appears on the screen.

Return Is the possibly limited floating point value.

StrToFloatStrMinMax

StrToFloatStrMinMax(*s* : string; *var val* : double; *min,max* : double) : string

Parameters: *s* is the string representation of the floating point value.

var val is on return the possibly limited floating point value.

min is the lower limit for a floating point value.

max is the upper limit for a floating point value.

Description The function **StrToFloatStrMinMax** converts a string to the corresponding floating point value. When this value exceeds the lower or upper limit it is set equal to the exceeded limit and a corresponding message appears on the screen. The possibly limited floating point value is again converted to its string representation with a maximum of 7 significant digits and 3 digits behind the decimal point.

Return Is the string representation of the possibly limited floating point value.

MinMaxi

MinMaxi(*val, min, max* : Integer) : Integer

Parameters: *val* is the integer value, which is to be checked.

min is the lower limit for an integer value.

max is the upper limit for an integer value.

Description The function **MinMaxi** checks if an integer value is inside a limited range. When the integer value exceeds the lower or upper limit it is set equal to the nearest limit.

Return Is the possibly limited integer value.

1.4 Overview of Classes and DLL Interfaces

The files DTSSRV16.H, DTSSRV16.CPP contain:

int CALLBACK **LibMain**(HINSTANCE *hInstance*, WORD *wDatenSeg*, WORD *cbHeapSize*, LPSTR *lpCmdLine*)

int CALLBACK **WEP**(int *nParameter*)

BOOL CALLBACK **LockMemory**(BOOL *bStart*, HDRVR *hDrv*)

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

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*)

void CALLBACK **ResetPIController**(void)

double CALLBACK **SenGetLevel**(int *mode*)

void CALLBACK **SenSetLevel**(int *mode*, double *val*)

void CALLBACK **SenEichLevel**(int *mode*)

double CALLBACK **SenGetVolt**(int *s*, double *h*)

void CALLBACK **OfcEichStart**(void)

void CALLBACK **OfcEichStop**(void)

double CALLBACK **OfcEichGet**(int *i*)

void CALLBACK **PfrEichStart**(int *i*)

void CALLBACK **PfrEichStop**(int *i*, double *time*)

double CALLBACK **PfrEichGet**(int *p*, int *i*)

int CALLBACK **LoadCalibration**(char **fname*)

int CALLBACK **SaveCalibration**(char **fname*)

int CALLBACK **MeasureStart**(double *time*, double *trigger*, double *prestore*, int *tchannel*, int *slope*)

double CALLBACK **MeasureLevel**(void)

int CALLBACK **MeasureStatus**(void)

void CALLBACK **SetDemo**(double *se1*, double *se2*, double *se3*, double *sp1*, double *sp2*, double *sl1*, double *sl2*, double *sl3*, double *sv13*, double *sv32*, double *sv20*)

int CALLBACK **IsDemo**(void)

The files DTS20SYS.H, DTS20SYS.CPP contain:

class **LLS**

LLS()

double **calccm** (double *volt*)
void **SetL1**(double *in*)
void **SetL2**(double *in*)
void **SetV1**(double *in*)
void **SetV2**(double *in*)
void **newsensor**(void)
double **GetLevel**(int *which*)
int **Load**(void)
int **Save**(void)
int **GetStatus**(void)

class **OFC**
OFC()
void **SetMaxLevel**(double*h1*, double*h2*, double*h3*)
void **SetMinLevel**(double*h1*, double*h2*, double*h3*)
void **SetTime**(double *t*)
void **CalcOfc**(void)
int **CalcOfcA**(double *AT1*, double *AT2*, double *AT3*)
int **Load**(void)
int **Save**(void)
int **GetStatus**(void)

class **PFR**
PFR(void)
double **pumpi2Q**(int *i*)
double **pumpQ2V**(double *q*)
double **GetVoltage**(void)
void **SetOut**(int *in*)
void **SetMeasureLevel1**(double *l1*)
void **SetMeasureLevel2**(double *l2*)
void **SetMeasureTime**(double *t*)
double **CalcQzu**(int *index*)
double **CalcQzuA**(int *index*, double *At*)
int **Load**(void)
int **Save**(void)
int **GetStatus**(void)

class **TUSTINPI**
TUSTINPI(void)
void **SetKP**(double *k*)
double **GetKP**(void)
void **SetKI**(double *k*)
double **GetKI**(void)
void **SetTA**(double *ta*)
double **GetTA**(void)
void **ResetControl**(void)
double **Control**(double *x*d)
void **CalcZCoef**(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*, int &*decoupled*, double &*p*, double &*i*, double &*decoup*, char ***s*, int &*n*, int &*c*, double &*d*)

1.5 References of the DLL Interfaces

Global Data:

```
typedef struct{
        WORD    controller;     // controller structure (open loop, P or PI controller)
        WORD    decoupled;      // flag for decoupling controller
        double  p;              // controller parameter proportional part
        double  i;              // controller parameter integral part
        double  decoup;         // decoupling parameter
        WORD    sp1shape;       // shape of setpoint signal 1 (constant, rectangle, sine etc)
        double  sp1offset;      // offset of setpoint signal 1
        double  sp1amplitude;   // amplitude of setpoint signal 1
        double  sp1periode;     // period of setpoint signal 1
        WORD    sp2shape;       // shape of setpoint signal 2
        double  sp2offset;      // offset of setpoint signal 2
        double  sp2amplitude;   // amplitude of setpoint signal 2
        double  sp2periode;     // period of setpoint signal 2
        double  At1;            // effective cross section of tank 1
        double  At2;            // effective cross section of tank 2
        double  At3;            // effective cross section of tank 3
}ServiceParameters;

typedef struct{
        double t1setpoint;  // liquid level setpoint tank 1 or pump flow rate 1
        double t2setpoint;  // liquid level setpoint tank 2 or pump flow rate 2
        double t1level;     // measured liquid level tank 1
        double t2level;     // measured liquid level tank 2
        double t3level;     // measured liquid level tank 3
        double p1flow;      // control signal pump 1
        double p2flow;      // control signal pump 2
}ServiceData;
```

ServiceParameters *par*	is a global structure of type ServiceParameters (see DTSSRV16.H).
ServiceData *dat*	is a global structure of type ServiceData (see DTSSRV16.H).
HGLOBAL *mHnd*	is a handle for the code memory of the DTSSRV16.DLL.
UINT *mData* = 0	is a handle for the data memory of the DTSSRV16.DLL.
HDRVR *hDriver* = NULL	is a handle for the IO-adapter card driver (*.DRV).
DWORD *dwCounter* = 0L	is a counter for calling the function **DoService**.
DACDRV **drv*	is a pointer to an instance of the class **DACDRV** (driver interface).
double *Out1*=0.0	is the control signal for pump 1.
double *Out2*=0.0	is the control signal for pump 2.
TUSTINPI **DTS200PI1*	is a pointer to a controller object for tank 1.

TUSTINPI **DTS200PI2* is a pointer to a controller object for tank 2.

float *scopebuf*[ScopeBufSize] is the measurement-vector.

STOREBUF **scope*(ScopeBufSize, scopebuf, 1024)
is a pointer to an object to handle the storage of a maximum of 1024 elements of type *scopebuf*.

SIGNAL **SpGenTank1* is a pointer to a setpoint generator object for tank 1 (pump 1).

SIGNAL **SpGenTank2* is a pointer to a setpoint generator object for tank 2 (pump 2).

1.5.1 The DLL Interface DTSSRV16

LibMain

int CALLBACK **LibMain**(HINSTANCE *hInstance*, WORD *wDatenSeg*, WORD *cbHeapSize*, LPSTR *lpCmdLine*)

Description The function **LibMain** is the entry function of the 16 bit DTSSRV16.DLL. None of the parameters is used directly. New objects for the driver, the scope buffer, the signal generators and the controllers are created and assigned to the corresponding global pointers.

Return Is TRUE in case of successful initialization, else FALSE.

WEP

int CALLBACK **WEP**(int *nParameter*)

Description The function **WEP** is the exit function of the 16 bit DTSSRV16.DLL. None of the parameters is used directly. When the code and data memory of the complete DTSSRV16.DLL is locked, it will be unlocked. The objects for the driver, the scope buffer, the signal generators and the controllers are deleted.

Return Is TRUE in case of successful operations, else FALSE.

LockMemory

BOOL CALLBACK **LockMemory**(BOOL *bStart*, HDRVR *hDrv*)

Parameters bStart is a flag with the meaning:
=TRUE, code and data memory of the DTSSRV16.DLL will be locked,
=FALSE, code and data memory of the DTSSRV16.DLL will be unlocked.

hDrv is a handle for the IO-adapter card driver.

Description The function **LockMemory** controls the lock status of the code and data memory of the complete DTSSRV16.DLL. With *bStart*=TRUE this memory is locked and the IO-adapter card driver *hDriver* (for the DTSSRV16.DLL) is set equal to *hDrv*. With *bStart*=FALSE this memory will be unlocked again.

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

Return Is equal to TRUE, when the handle of the code memory of the DTSSRV16.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.

DoService

BOOL CALLBACK **DoService**(DWORD *counter*)

Parameters *counter* is counter for the number of calls.

Description The function **DoService** is the service routine for the controller interrupt. It is called at every sampling period when the interrupt is active. The following operations are carried-out in sequence:

trigger the output stage (rectangle signal),
read the liquid levels of 3 tanks,
store the measured values to the measurement vector,
determine the setpoints by means of the generators,
store the setpoints to the measurement vector,
calculate the pump control signals (closed/open loop control, none),
calculate cross flow rates and nominal outflow,
correct control signals in case of decoupling,
store the pump control signals to the measurement vector,
write the pump control signals,
store the measurement-vector.

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

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 ServiceParameters) only when the size of the source structure is less or equal to the size of the destination structure. In this case the controller parameters of the two PI controllers, the setpoint generators as well as the cross sections of the tanks are set accordingly.

Return Is equal to TRUE when the size of the source structure is less 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 the current parameters of the tank cross sections 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 or equal to the size of the destination structure.

Return Is equal to TRUE when the size of the source structure is less or equal to the size of the destination structure, else return is equal to FALSE.

ResetPIController

void CALLBACK **ResetPIController**(void)

Description The function **ResetPIController** resets the integral parts of the two PI controllers.

The functions

SenGetLevel

double CALLBACK **SenGetLevel**(int *mode*)

SenSetLevel

void CALLBACK **SenSetLevel**(int *mode*, double *val*)

SenEichLevel

void CALLBACK **SenEichLevel**(int *mode*)

SenGetVolt

double CALLBACK **SenGetVolt**(int *s*, double *h*)

OfcEichStart

void CALLBACK **OfcEichStart**(void)

OfcEichStop

void CALLBACK **OfcEichStop**(void)

OfcEichGet

double CALLBACK **OfcEichGet**(int *i*)

PfrEichStart

void CALLBACK **PfrEichStart**(int *i*)

PfrEichStop

void CALLBACK **PfrEichStop**(int *i*, double *time*)

PfrEichGet

double CALLBACK **PfrEichGet**(int *p*, int *i*)

LoadCalibration

int CALLBACK **LoadCalibration**(char **fname*)

SaveCalibration

int CALLBACK **SaveCalibration**(char **fname*)

Description directly call the functions of the class **DACDRV** with the same names (LoadCalibration -> Load, SaveCalibration -> Save). The description of the functions used for calibrating the tank system is to be found in the corresponding sections.

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 interrupt 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)

SetDemo

void CALLBACK **SetDemo**(double *se1*, double *se2*, double *se3*, double *sp1*, double *sp2*, double *sl1*,
double *sl2*, double *sl3*, double *sv13*, double *sv32*, double *sv20*)

Description call directly the corresponding functions *scope*.**GetBufferLevel**, *scope*.**GetStatus** of the class **STOREBUF** and *drv*.**SetDemo** of the class **DACDRV**.

See also STOREBUF::**GetBufferLevel**, STOREBUF::**GetStatus**, DACDRV::**SetDemo**

IsDemo

int CALLBACK IsDemo(void)

Description The function **IsDemo** returns a 1 only when the DTSSRV16.DLL is a DEMO version (generated with the macro __DTS_DEMO__ , 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.

1.5.2 The Class LLS in the DTSSRV16.DLL

The class **LLS** provides the determination and handling of a sensor characteristic for one tank. The sensor characteristic is a straight line of the form *h[cm] = u[Volt] * m + b* to calculate the liquid level *h* in cm with respect to the sensor voltage *u* in Volt.

Public data:

double *m*	is the gradient of the sensor characteristic.
double *b*	is the offset of the sensor characteristic.

Private data:

double *l1*	is the liquid level for the lower calibration point (in cm).
double *l2*	is the liquid level for the upper calibration point (in cm).
double *v1*	is the liquid level for the lower calibration point (in Volt).
double *v2*	is the liquid level for the upper calibration point (in Volt).
int *status*	is the status of the calibration data: =0, default calibration data =1, calibration data from file =2, calibration data from measurement

LLS::LLS()

void **LLS**(void)

Description The constructor of the class **LLS** initializes values for the calibration data and calculates the gradient and the offset of the accompanying sensor characteristic (from 9 Volt correspond to 0 cm and -8,5 Volt correspond to 60 cm). The status of the calibration is reset to 0. The calibration points for the measurements are set to *l1* = 20 and *l2* = 50 cm.

LLS::calccm

double **calccm**(double *volt*)

Parameters double *volt* is the sensor value in Volt.

Description The function **calccm** returns a liquid level in cm with respect to a sensor value (parameter *volt*) by determining the accompanying straight line equation.

Return Liquid level (double) in cm for sensor value in Volt.

LLS::calcv

double **calcv**(double *cm*)

Parameters double *cm* is the liquid level in cm.

Description The function **calcv** returns a sensor value in Volt with respect to a liquid level (parameter *cm*) by determining the accompanying straight line equation.

Return Sensor value (double) in Volt for a liquid level in cm.

LLS::SetL1

void **SetL1**(double *in*)

Parameters double *in* is the liquid level of the lower calibration point.

Description The function **SetL1** adjusts the liquid level in cm for the lower calibration point to the value of the parameter *in*.

LLS::SetL2

void **SetL2**(double *in*)

Parameters double *in* is the liquid level of the upper calibration point.

Description The function **SetL2** adjusts the liquid level in cm for the upper calibration point to the value of the parameter *in*.

LLS::SetV1

void **SetV1**(double *in*)

Parameters double *in* is the sensor value of the lower calibration point.

Description The function **SetV1** adjusts the sensor value in Volt for the lower calibration point to the value of the parameter *in*.

LLS::SetV2

void **SetV2**(double *in*)

Parameters double *in* is the sensor value of the upper calibration point.

Description The function **SetV2** adjusts the sensor value in Volt for the upper calibration point to the value of the parameter *in*.

LLS::newsensor

void **newsensor**(void)

Description The function **newsensor** calculates the gradient and the offset of the straight line equation of a sensor using the pair of values (level in cm / sensor in Volt) from two calibration points. The status of the calibration is set to 2.

LLS::GetLevel

double **GetLevel**(int *which*)

Parameters int *which* selects the return value:
=1, liquid level of the lower calibration point,
else, liquid level of the upper calibration point.

Description The function **GetLevel** returns the liquid level in cm for the lower or upper calibration point depending on the parameter *which*.

Return Liquid level (double) of the lower or upper calibration point.

LLS::Load

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

Parameters char **name* is a pointer to the file name.

Description The function **Load** reads the gradient and offset for a straight line equation from a file with the name *name* (ASCII format). The status of the calibration is set to 1.

Return State of the file access:
=1 (TRUE), when reading from file was successful,
=0 (FALSE), in any other case.

LLS::Save

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

Parameters char **name* is a pointer to the file name.

Description The function **Save** stores the gradient and offset of a straight line equation to a file with the name *name* (ASCII format).

Return State of the file access:
=1 (TRUE), when writing to file was successful,
=0 (FALSE), in any other case.

LLS::GetStatus

int **GetStatus**(void)

Description The function **GetStatus** returns the status of the calibration.

Return Status (int) of the calibration data:
=0, default calibration data
=1, calibration data from file
=2, calibration data from measurement

1.5.3 The Class OFC in the DTSSRV16.DLL

The class **OFC** determines the outflow coefficients of a three-tank-system. Upper liquid levels and lower liquid levels reached after a known period of time are substituted in the model equations to calculate the outflow coefficients.

Private data:

double *ofc[3]*	contains the outflow coefficients: [0] = tank 1-3, [1] = nominal outflow (tank 2), [2] = tank 3-2.
double *maxlevel[3]*	are the upper liquid levels to determine the outflow coefficients.
double *minlevel[3]*	are the lower liquid levels to determine the outflow coefficients.
double *dlevel[3]*	are the differences between the upper and lower liquid levels.
double *level[3]*	are the mean values of the upper and lower liquid levels.
double *time*	is the time interval between the upper and lower liquid levels (switch-on time of the pumps).
int *status*	is the status of the outflow coefficients: =0, default outflow coefficients =1, outflow coefficients from file =2, outflow coefficients from measurement

---

OFC::OFC()

void **OFC**(void)

Description The constructor of the class **OFC** sets initial values for the outflow coefficients (tanks 1-3 and 3-2 = 0,5 nominal outflow = 0,6). The status of the outflow coefficients is reset to 0.

OFC::SetMaxLevel

void **SetMaxLevel**(double *h1*, double *h2*, double *h3*)

Parameters double *h1* is the upper liquid level of tank 1.

double *h2* is the upper liquid level of tank 2.

double *h3* is the upper liquid level of tank 3.

Description The function **SetMaxLevel** adjusts the upper liquid levels in cm to determine the outflow coefficients to the values of the given parameters.

OFC::SetMinLevel

void **SetMinLevel**(double *h1*, double *h2*, double *h3*)

Parameters double *h1* is the lower liquid level of tank 1.

double *h2* is the lower liquid level of tank 2.

double *h3* is the lower liquid level of tank 3.

Description The function **SetMinLevel** adjusts the lower liquid levels in cm to determine the outflow coefficients to the values of the given parameters.

OFC::SetTime

void **SetTime**(double *t*)

Parameters double *t* is the time interval between the liquid levels.

Description The function **SetTime** adjusts the time interval the system needed to reach the lower liquid levels starting from the upper liquid levels to the value of the parameter *t*.

OFC::CalcOfc

void **CalcOfc**(void)

Description The function **CalcOfc** calculates the outflow coefficients with respect to the global variables for the upper and lower liquid levels and for the time interval using the mathematical model of the three-tank-system.

OFC::CalcOfcA

void **CalcOfcA**(double *AT1*, double *AT2*, double *AT3*)

Parameters double **AT1* is the effective cross section of tank 1.

double **AT2* is the effective cross section of tank 2.

double **AT3* is the effective cross section of tank 3.

Description The function **CalcOfcA** calculates the outflow coefficients with respect to the global variables for the upper and lower liquid levels and for the time interval using the mathematical model of the three-tank-system. Here the calculation uses the possibly different cross sections of the tanks. Any cross section outside the range <=0 and >1000 is reset to 154 (cm^2). In case the time interval is 0 and the level differences are <0.01, the calculation is skipped and the function returns with FALSE, otherwise with TRUE and a state *status*=2.

Return State of the calculation:
=1 (TRUE), when calculation successful,
=0 (FALSE), in any other case.

OFC::Load

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

Parameters char **name* is a pointer to the file name.

Description The function **Load** reads the outflow coefficients from a file with the name *name* (ASCII format). The status of the outflow coefficients is set to 1.

Return State of the file access:
=1 (TRUE), when reading from file was successful,
=0 (FALSE), in any other case.

OFC::Save

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

Parameters char **name* is a pointer to the file name.

Description The function **Save** stores the outflow coefficients to a file with the name *name* (ASCII format).

Return State of the file access:
=1 (TRUE), when writing to file was successful,
=0 (FALSE), in any other case.

OFC::GetStatus

int **GetStatus**(void)

Description The function **GetStatus** returns the status of the outflow coefficients.

Return Status (int) of the outflow coefficients:
=0, default outflow coefficients
=1, outflow coefficients from file
=2, outflow coefficients from measurement

1.5.4 The Class PFR in the DTSSRV16.DLL

The class **PFR** provides determination and handling of a pump characteristic for a tank system. The pump characteristic is defined by 9 base points. Each base point contains the correlation between a known control signal in Volt and the resulting pump flowrate in ml/s. Values between the base points are obtained by linear interpolation. The control signal for the base points is preassigned in the range from -10 Volt to +10Volt with a step of 2,5 Volt. The pump flowrate is calculated from the changes in the liquid levels during a known time period.

Public data :

int *i*	is the index of a base point belonging to a pump characteristic.
int *status*	is the status of the pump characteristic: =0, default pump characteristic =1, pump characteristic from file =2, pump characteristic from measurement
int *calibration*	is a flag for the pump characteristic.
int *delay*	is a flag for the pump characteristic.
double *deltatime*	is the time period between the initial and final value of a liquid level.
double *level1*	is the initial value of the liquid level.
double *level2*	is the final value of the liquid level.
double *pump[9]*	contains the base points of the pump characteristic (flowrate [ml/s]).

PFR::PFR()

void **PFR**(void)

Description The constructor of the class **PFR** initializes default values for the pump characteristic in the range of 0 to 100 ml/s with a step of 12,5 ml/s according to the base points of the control signal in the range of -10 V to +10 V with a step of 2,5 V. The status of the pump characteristic as well as the flags are reset to 0.

PFR::pumpi2Q

double **pumpi2Q**(int *k*)

Parameters int *k* is the index for the base point of the pump characteristic.

Description The function **pumpi2Q** returns the value of the base point of the pump characteristic referenced by an index (parameter *k*)

Return Pump flowrate (double) in ml/s belonging to the index.

PFR::pumpQ2V

double **pumpQ2V**(double *q*)

Parameters double *q* is the desired pump flowrate.

Description The function **pumpQ2V** returns the control signal in Volt determined by interpolation of the pump characteristic which is required for a desired pump flowrate in ml/s (Parameter *q*).

Return Control signal (double) in Volt required for a desired pump flowrate in ml/s.

PFR::GetVoltage

double **GetVoltage**(void)

Description The function **GetVoltage** returns the control signal in Volt referenced by the current index (public data *i*).

Return Control signal (double) in Volt referenced by an index.

PFR::SetOut

void **SetOut**(int *in*)

Parameters int *in* is the index for the pump characteristic.

Description The function **SetOut** adjusts the current index to the value of the parameter *in* and sets the flag *calibration* to 1.

PFR::SetMeasureLevel1

void **SetMeasureLevel1**(double *l1*)

Parameters double *l1* is the initial liquid level for calibration.

Description The function **SetMeasureLevel1** adjusts the initial value of the liquid level used for the pimp calibration to the value of the parameter *l1*.

PFR::SetMeasureLevel2

void **SetMeasureLevel2**(double *l2*)

Parameters double *l1* is the final liquid level for calibration.

Description The function **SetMeasureLevel2** adjusts the final value of the liquid level used for the pump calibration to the value of the parameter *l2*.

PFR::SetMeasureTime

void **SetMeasureTime**(double *t*)

Parameters double *l1* is the time period between the initial and final value of the liquid levels.

Description The function **SetMeasureTime** adjusts the time period, which was needed to reach the final value of the liquid level starting with the initial value, to the (measured) value of the parameter *t*.

PFR::CalcQzu

double **CalcQzu**(int *i*)

Parameters int *i* is the base point index for the pump characteristic.

Description The function **CalcQzu** returns the flowrate belonging to a base point of the pump characteristic referenced by an index (parameter *i*). The flowrate is calculated using the current values of the initial and final liquid levels as well as the time period. The status of the pump characteristic is set to 2 when the flowrate of the last base point is calculated.

Return Pump flowrate (double) in ml/s referenced by an index.

PFR::CalcQzuA

double **CalcQzuA**(int *i*, double *At)*

Parameters int *i* is the base point index for the pump characteristic.

double *At* is the effective cross setion of the tank.

Description The function **CalcQzuA** returns the flowrate belonging to a base point of the pump characteristic referenced by an index (parameter *i*). The flowrate is calculated using the current values of the initial and final liquid levels as well as the time period. The calculation is carried-out with respect to the cross section of the tank which the pump is supplying. A cross section outside the range <=0 and > 1000 is reset to 154(cm^2).The status of the pump characteristic is set to 2 when the flowrate of the last base point is calculated.

Return Pump flowrate (double) in ml/s referenced by an index.

PFR::Load

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

Parameters char **name* is a pointer to the file name.

Description The function **Load** reads the base points of the pump characteristic from a file with the name *name* (ASCII format). The status of the pump characteristic is set to 1.

Return State of the file access:
=1 (TRUE), when reading from file was successful,
=0 (FALSE), in any other case.

PFR::Save

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

Parameters char **name* is a pointer to the file name.

Description The function **Save** stores the base points of the pump characteristic together with the accompanying control signal to a file with the name *name* (ASCII format).

Return State of the file access:
=1 (TRUE), when writing to file was successful,
=0 (FALSE), in any other case.

PFR::GetStatus

int **GetStatus**(void)

Description The function **GetStatus** returns the status of the pump characteristic.

Return The status (int) of the pump characteristic:
=0, default pump characteristic
=1, pump characteristic from file
=2, pump characteristic from measurement.

1.5.5 The Class TUSTINPI in the DTSSRV16.DLL

The class **TUSTINPI** provides the calculation of a digital PI controller obtained by an approximation of an analog PI controller. The coefficients of the digital controller result from applying the Tustin relation on the transfer function of the analog controller:

analog controller: G(s) = ki * 1/s + kp

with the Tustin relation: s = 2/Ta * (z-1) / (z+1)

digital controller: y(k+1) = y(k) + z0 * xd(k) + z1 * xd(k-1)

where

z0 = ki * Ta/2 + kp,
z1 = ki * Ta/2 - kp

Private data:

double *ta*	is the sampling period of the controller.
double *ki*	is the integral portion of the analog controller.
double *kp*	is the proportional portion of the analog controller.
double *online_zk0*	z0 coefficient of the active digital controller.
double *zk0*	z0 coefficient of the digital controller.
double *online_zk1*	z1 coefficient of the active digital controller.
double *zk1*	z1 coefficient of the digital controller.
double *y*	is the control signal from the previous sampling period.
double *xd_1*	is the control error signal from the previous sampling period.
int *newz*	is a flag indicating the coefficients are used by the active controller.

TUSTINPI::TUSTINPI()

void **TUSTINPI** (void)

Description The constructor of the class **TUSTINPI** adjusts initial values for the coefficients of the analog controller (to 1 resp. 0), for the values of the digital controller belonging to the previous sampling period (*y=xd_1=0*) and calculates the coefficients of the digital controller.

TUSTINPI::SetKP

void **SetKP**(double *k*)

Parameters double *k* is the proportional amplification of the analog controller.

Description The function **SetKP** adjusts the proportional portion of the analog PI controller to the value of the given parameter and calculates the coefficients of the accompanying digital controller.

TUSTINPI::GetKP

double **GetKP**(void)

Description The function **GetKP** returns the adjusted proportional portion of the analog PI controller.

Return The proportional portion (double) of the analog controller.

TUSTINPI::SetKI

void **SetKI**(double *k*)

Parameters double *k* is the integral portion of the analog controller.

Description The function **SetKI** adjusts the integral portion of the analog PI controller to the value of the given parameter and calculates the coefficients of the accompanying digital controller.

TUSTINPI::GetKI

double **GetKI**(void)

Description The function **GetKI** returns the adjusted integral portion of the analog PI controller.

Return The integral portion (double) of the analog controller.

TUSTINPI::SetTA

void **SetTA**(double *t*)

Parameters double *t* is the sampling period of the digital controller.

Description The function **SetTA** adjusts the sampling period in sec of the digital controller to the value of the given parameter and calculates the coefficients of the accompanying digital controller.

TUSTINPI::GetTA

double **GetTA**(void)

Description The function **GetTA** returns the adjusted sampling period of the digital controller.

Return The sampling period (double) of the digital controller.

TUSTINPI::ResetControl

void **ResetControl**(void)

Description The function **ResetControl** resets the past values of the digital controller (control signal *y* and control error signal *xd* from the previous sampling period) to 0.

TUSTINPI::Control

double **Control**(double *xd*)

Parameters double *xd* is the current control error signal of the digital controller.

Description The function **Control** calculates the digital controller with respect to the given control error signal *xd* and returns the control signal limited to the range +/-100. In case the flag *newz* was set, at first the coefficients of the digital controller are copied to those of the active controller. The flag is reset after this operation.

Return The control signal (double) of the digital controller.

TUSTINPI::CalcZCoef

void **CalcZCoef**(void)

Description The function **CalcZCoef** calculates the coefficients of the digital controller by applying the Tustin relation on the transfer function of the analog controller. The Flag *newz* is set to 1.

1.5.6 The Class STOREBUF in the DTSSRV16.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 initialzed
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.

1.5.7 The Class AFBUF in the DTSSRV16.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).

1.5.8 The Class TWOBUFFER in the DTSSRV16.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.

1.5.9 The Class Signal in the DTSSRV16.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.

1.5.10 The DLL Interface PLOT

Included in the DTSSRV16.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".
char **SensorYAxisName*	is the description of the Y-axis for the drawing with the sensor characteristics.
char **SensorXAxisName*	is the description of the X-axis for the drawing with the sensor characteristics.
char **SensorScopeNames*	contains the curve descriptions (strings) for the linestyle table of the drawing with the sensor characteristics.
char **SensorTitleName*	is the drawing title for the plot the sensor characteristics.
char **PumpYAxisName*	is the description of the Y-axis for the drawing with the pump characteristics.
char **PumpXAxisName*	is the description of the X-axis for the drawing with the pump characteristics.
char **PumpScopeNames*	contains the curve descriptions (strings) for the linestyle table of the drawing with the pump characteristics.
char **PumpTitleName*	is the drawing title for the plot the pump characteristics.

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*,
=3, at that time meaningless, return -4,
=4, data from the sensor characteristics,
=5, data from the plot characteristics.

channel defines the curves related to "plot channels":
=0, Tank Levels [cm] (3 curves),
=1, Tank Levels and Setpoints [cm] (5 curves),
=2, Pump Flowrates [ml/s] (2 curves),
=3, Pump Control Signals [ml/s] (2 curves),
=4, Tank Levels [cm] and Pump Flowrates [ml/s] (5 curves),

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*, the global array ***ppData* or the characteristics of the sensors or the pumps depending on the parameter *command*.

Return

The state of the graphic output:
=0, successful graphic output of measured curves,
=-1, invalid values for *command*,
=-2, invalid values for *channel*,
=-3, length of the global array ***ppData* is 0,
=-4, length of the global measurement buffer *scope* is 0.

See also

CreateSimplePlotWindow, SetCurveMode, AddAxisPlotWindow, AddXData, AddPlotTitle, ShowPlotWindow.

GetPlot

int CALLBACK **GetPlot**(int *start*, char **lpzName*)

Parameters *start* is a flag indicating the first plot window.

**lpzName* is a pointer to the title of the plot window, the Windows handle of which was found.

Description The function **GetPlot** determines the Windows handle of an existing plot window referenced by a local index *index*. The Windows handle is copied to the global list *handlelist* and the index is incremented. If the Windows handle is unequal to 0 up to 60 characters of the title of the corresponding plot window are copied to *lpzName*. With *start*=TRUE the Windows handle of the plot window with index=0 is determined.

Return The state of the handle determination:
=0, handle = 0, plot window with current index could not be found,
=1, handle determined for current index, title copied.

See also **GetValidPlotHandle**.

PrintPlot

int CALLBACK **PrintPlot**(int *idx*, HDC *dcPrint*, int *iyOffset*)

Parameters *idx* is the index for the global list of handles referencing existing plot windows.

dcPrint is the device context of the output device.

iyOffset is the beginning of the printout in vertical direction as a distance in [mm] from the upper margin of a page.

Description The function **PrintPlot** prints the content of the plot window with the Windows handle from the global list *handlelist*[*idx*] to the device with the device context *dcPrint*. The printout has a width of 180 mm and a height of 140 mm. It is located at the left margin with a distance of *iyOffset* mm from the upper margin of a (i.e. DIN A4) page.

Return Is always equal to 0.

See also **PrintPlotWindow**.

GetPldInfo

int CALLBACK **GetPldInfo**(int &*controller*, int &*decoupled*, double &*p*, double &*i*, double &*decoup*, char **s*, int &*n*, int &*c*, double &*d*)

Parameters

&*controller* is a reference to the controller structure (open loop control, P controller decoupled, PI controller).

&*decoupled* is a reference to the PI controller structure (with/without decoupling).

&*p* is a reference to the proportional factor of the PI controller.

&*i* is a reference to the integral factor of the PI controller.

&*decoup* is a reference to the decoupling parameter.

***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 DTS200

2.1 The Class DACDRV

The class **DACDRV** provides the interface between the DTS200 controller program and the driver functions of the PC plug-in card. The class **WDAC98** containing the driver functions is the basic class of **DACDRV**. In addition this class contains the mathematical model of the Three-Tank-System when it is compiled with '#define __DTS_DEMO__' (see file DTSDEFIN.H). With this all program functions except for the calibration can be carried out for a simulated Three-Tank-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 DACDRV.H, DACDRV.CPP contain the class **DACDRV** with the functions:

DACDRV(void)

~**DACDRV**(){}

void **SetTankDims**(double *at1*, double *at2*, double *at3*)

double **ReadTank1**(void)

double **ReadTank2**(void)

double **ReadTank3**(void)

void **SetPumpFlow1**(double *q*)

void **SetPumpFlow2**(double *q*)

#ifdef __DTS_DEMO__

void **SetDemo**(double *se1*, double *se2*, double *se3*, double *sp1*, double *sp2*, double *sl1*, double *sl2*, double sl3, double *sv13*, double *sv32*, double *sv20*)

void **GetSensorErrors**(double& *SensErr1*, double& *SensErr2*, double& *SensErr3*)

void **GetPumpErrors**(double& *PumpErr1*, double& *PumpErr2*)

void **ResetSimSystem**(void)

void **SetNoise**(double *SignalNoise*)

double **SensNoise**(void)

#endif

double **SenGetLevel**(int *mode*)

void **SenSetLevel**(int *mode*, double *val*)

void **SenEichLevel**(int *mode*)

double **SenGetVolt**(int *mode*, double *val*)

void **OfcEichStart**(void)

void **OfcEichStop**(void)

double **OfcEichGet**(int *i*)

void **PfrEichStart**(int *i*)

void **PfrEichStop**(int *i*, double *time*)

double **PfrEichGet**(int *p*, int *i*)

int **Save**(char* *FileName*)

int **Load**(char* *FileName*)

int **eichok**(void)

int **CheckSystem**(void)

int **CheckFree**(void)

virtual void **StartInterrupt**(void)

virtual void **TriggerEndstufe**(void)

Private Data:

LLS *DTS200sen1* is an object to determine and handle the characteristic of level sensor 1
LLS *DTS200sen2* is an object to determine and handle the characteristic of level sensor 2
LLS *DTS200sen3* is an object to determine and handle the characteristic of level sensor 3
OFC *DTS200ofc* is an object to determine and handle the outflow coefficients
PFR *DTS200pump1* is an object to determine and handle the characteristic of the pump 1
PFR *DTS200pump2* is an object to determine and handle the characteristic of the pump 2
double *Level1* is the liquid level of tank 1 in [cm]
double *Level2* is the liquid level of tank 2 in [cm]
double *Level3* is the liquid level of tank 3 in [cm]
double *Volt1* is the liquid level of tank 1 in [Volt]
double *Volt2* is the liquid level of tank 2 in [Volt]
double *Volt3* is the liquid level of tank 3 in [Volt]
int *Tick* is a flag for the sensor calibration
unsigned long *EichTick* is a time counter for the pump calibration
#ifdef __DTS_DEMO__
double *q1* is the flowrate of the pump 1
double *q2* is the flowrate of the pump 2
double *q13* is the cross flow from tank 1 to tank 3

double *q32*	is the cross flow from tank 3 to tank 2
double *q20*	is the outflow of tank 2
double *ql1*	is the leakage flow from tank 1
double *ql2*	is the leakage flow from tank 2
double *ql3*	is the leakage flow from tank 3
double *SensError1*	is the error of sensor 1
double *SensError2*	is the error of sensor 2
double *SensError3*	is the error of sensor 3
double *PumpError1*	is the error of pump 1
double *PumpError2*	is the error of pump 2
double *l1*	is a measure for the leakage at tank 1
double *l2*	is a measure for the leakage at tank 2
double *l3*	is a measure for the leakage at tank 3
double *v13*	is a measure for the clog between tank 1 and tank 3
double *v32*	is a measure for the clog between tank 3 and tank 2
double *v20*	is a measure for the clog in the outflow from tank 2
int *AddNoise*	is a flag for the noise signal
double *noise*	is the amplitude of the sensor noise signal

#endif

Private Data:

double *At1*	is the effective cross section of tank 1
double *At2*	is the effective cross section of tank 2
double *At3*	is the effective cross section of tank 3

DACDRV::DACDRV

DACDRV(void)

Description The constructor of the class **DACDRV** initializes an object of the class **WDAC98** as well as all objects required to determine and handle the characteristics and outflow coefficients (classes **LLS**, **PFR**, **OFC**). The liquid levels in [cm] and [Volt] and for the Demo-Version the error signals as well as the real pump control signals are reset to 0. The effective cross sections of the tanks are set to 154. Finally the file DEFAULT.CAL containing the calibration data (characteristics and outflow coefficients) of the tank system is loaded. A message box with the error message "Can't open file: DEFAULT.CAL" will appear, when this file does not exist.

DACDRV::~DACDRV

~DACDRV(void)

Description The destructor of the class **DACDRV** resets the real pump control signals to 0.

DACDRV::SetTankDims

void **SetTankDims**(double *at1*, double *at2*, double *at3*)

Parameters *at1* is the effective cross section of tank 1.

at2 is the effective cross section of tank 2.

at3 is the effective cross section of tank 3.

Description The function **SetTankDims** copies the given values to the variables for the effective cross sections of the tanks only when they are limited to the range >0 and < 1000.

DACDRV::ReadTank1

double **ReadTank1**(void)

Description The function **ReadTank1** reads the sensor for the liquid level in tank 1 and calculates the accompanying liquid level in [cm] with respect to the corresponding sensor characteristic. This value is returned by the function.

For the DEMO-Version the return value is equal to the liquid level *h* calculated by means of the mathematical model. This value may be superimposed by an error and noise signal according to the following relation:

level[cm] = *h* * (1 - sensor error) + noise signal

The noise signal is added only when the corresponding flag is set. Negative liquid levels are limited to 0,0.

Return: The liquid level of tank 1 (in DEMO-Version with errors).

DACDRV::ReadTank2

double **ReadTank2**(void)

Description The function **ReadTank2** reads the sensor for the liquid level in tank 2 and calculates the accompanying liquid level in [cm] with respect to the corresponding sensor characteristic. This value is returned by the function.

For the DEMO-Version the return value is equal to the liquid level *h* calculated by means of the mathematical model. This value may be superimposed by an error and noise signal according to the following relation:

level[cm] = *h* * (1 - sensor error) + noise signal

The noise signal is added only when the corresponding flag is set. Negative liquid levels are limited to 0,0.

Return: The liquid level of tank 2 (in DEMO-Version with errors).

DACDRV::ReadTank3

double **ReadTank3**(void)

Description The function **ReadTank3** reads the sensor for the liquid level in tank 3 and calculates the accompanying liquid level in [cm] with respect to the corresponding sensor characteristic. This value is returned by the function.

For the DEMO-Version the return value is equal to the liquid level *h* calculated by means of the mathematical model. This value may be superimposed by an error and noise signal according to the following relation:

level[cm] = *h* * (1 - sensor error) + noise signal

The noise signal is added only when the corresponding flag is set. Negative liquid levels are limited to 0,0.

Return: The liquid level of tank 3 (in DEMO-Version with errors).

DACDRV::SetPumpFlow1

void **SetPumpFlow1**(double *q*)

Parameters *q* is the desired flowrate of pump 1

Description The function **SetPumpFlow1** determines the control signal in [Volt] required for a desired flowrate *q* of the pump 1 by evaluation of the pump characteristic. This control signal is provided at the D/A converter output. During the determination of the pump characteristic (*calibration* = 1) the accompanying base point values are provided at the output.

The D/A converter output is omitted in the DEMO-Version. The flowrate for the mathematical model is calculated as follows

flowrate [ml/s] = *q* * (1 - pump error)

DACDRV::SetPumpFlow2

void **SetPumpFlow2**(double *q*)

Parameters *q* is the desired flowrate of pump 2

Description The function **SetPumpFlow2** determines the control signal in [Volt] required for a desired flowrate *q* of the pump 2 by evaluation of the pump characteristic. This control signal is provided at the D/A converter output. During the determination of the pump characteristic (*calibration* = 1) the accompanying base point values are provided at the output.

The D/A converter output is omitted in the DEMO-Version. The flowrate for the mathematical model is calculated as follows

flowrate [ml/s] = *q* * (1 - pump error)

The flowrates of both pumps are limited to the allowed range. The mathematical model of the Three-Tank-System is calculated with respect to the possibly adjusted leakage flows and clogs.

coef1 = outflow coefficient1 * valve section
coef2 = outflow coefficient2 * valve section
coef3 = outflow coefficient3 * valve section

leakage flows:
ql1 = coef2 * *l1* * sqrt(2*g * old_Level1)
ql2 = coef2 * *l2* * sqrt(2*g * old_Level2)
ql3 = coef2 * *l3* * sqrt(2*g * old_Level3)

cross flows:
delta_h13 = old_Level1 - old_Level3
delta_h32 = old_Level3 - old_Level2
q13 = coef1 * (1 - *v13*) * sqrt(2*g * delta_h13)
q32 = coef2 * (1 - *v32*) * sqrt(2*g * delta_h32)

$$q20 = \text{coef3} * (1 - v20) * \text{sqrt}(2*g * \text{old\_Level2})$$

T = sampling period
A = tank cross section
$Level1 = (flowrate1 - q13 - ql1) * T/A + old\_level1$
$Level2 = (flowrate2 + q32 - q20 - ql2) * T/A + old\_level2$
$Level3 = (q13 - q32 - ql3) * T/A + old\_level3$

The calculated liquid levels are limited to the range 0 to 60 cm.

DACDRV::SetDemo

void **SetDemo**(double *se1*, double *se2*, double *se3*, double *sp1*, double *sp2*, double *sl1*, double *sl2*, double *sl3*, double *sv13*, double *sv32*, double *sv20*)

Parameters

se1 is the error of sensor 1

se2 is the error of sensor 2

se3 is the error of sensor 3

sp1 is the error of pump 1

sp2 is the error of pump 2

sl1 is a measure for the leakage at tank 1

sl2 is a measure for the leakage at tank 2

sl3 is a measure for the leakage at tank 3

sv13 is a measure for the clog between tank 1 and tank 3

sv32 is a measure for the clog between tank 3 and tank 2

v20 is a measure for the clog in the outflow of tank 2

Description

The function **SetDemo** adjusts the sensor and pump errors as well as the measures for leakages and clogs for the simulation of the Three-Tank-System. The errors are considered as follows:

liquid level [cm] = h * (1 - sensor error)
flow rate [ml/s] = q * (1 - pump error)
leakage flow [ml/s] = coef * leakage error * sqrt(2*g * level)
cross flow [ml/s] = coef * (1 - clog) * sqrt(2*g * delta_level)

DACDRV::GetSensorErrors

void **GetSensorErrors** (double& *SensErr1*, double& *SensErr2*, double& *SensErr3*)

Parameters *SensErr*1 is a pointer reference to the error of sensor 1
SensErr2 is a pointer reference to the error of sensor 2
SensErr3 is a pointer reference to the error of sensor 3

Description The function **GetSensorErrors** returns the adjusted sensor errors referenced by pointers.

DACDRV::GetPumpErrors

void **GetPumpErrors** (double**&** *PumpErr1*, double& *PumpErr2*)

Parameters *PumpErr1* is a pointer reference to the error of pump 1
PumpErr2 is a pointer reference to the error of pump 2

Description The function **GetPumpErrors** returns the adjusted pump errors referenced by pointers.

DACDRV::ResetSimSystem

void **ResetSimSystem**(void)

Description The function **ResetSimSystem** resets the liquid levels of the simulated Three-Tank-System to 0.

DACDRV::SetNoise

void **SetNoise**(double *SignalNoise*)

Parameters *SignalNoise* is the amplitude of the noise signal in [cm].

Description The function **SetNoise** adjusts the amplitude of the noise signal which is superimposed to the liquid level sensor signal of the simulated Three-Tank-System and activates the addition of the noise signal.

DACDRV::SensNoise

double **SensNoise**(void)

Description The function **SensNoise** returns the noise signal for the sensors of the liquid levels of the simulated Three-Tank-System. The pseudo binary noise signal is calculated as follows:

noise signal [cm] = *noise* * (1 - random number[0...2])

Return: The noise signal (double) in [cm] for the liquid levels.

DACDRV::SenGetLevel

double **SenGetLevel**(int *mode*)

Parameters *mode* determines the return value:
=1, liquid level for the lower calibration point,
else, liquid level for the upper calibration point.

Description The function **SenGetLevel** returns the liquid level in [cm] for the lower or upper calibration point of tank 1 depending on the parameter *mode*.

Return: Liquid level (double) of the lower or upper calibration point.

DACDRV::SenSetLevel

void **SenSetLevel**(int *mode*, double *val*)

Parameters *mode* determines the calibration point:
=1, liquid level for the lower calibration point,
else, liquid level for the upper calibration point.

val is the liquid level for the lower or upper calibration point.

Description The function **SenSetLevel** adjusts the liquid level in [cm] for the lower or upper calibration point for all tanks depending on the parameter *mode* to the value of *val*.

DACDRV::SenEichLevel

void **SenEichLevel**(int *mode*)

Parameters *mode* determines the calibration point:
=1, liquid level for the lower calibration point,
else, liquid level for the upper calibration point.

Description The function **SenEichLevel** reads successively in 8 sampling periods (synchronized by *Tick*) the sensor values in [Volt] for the liquid levels (in the calibration points), calculates the corresponding mean values and takes these values for the lower or upper calibration point depending on the parameter *mode*.

The controller interrupt has to be active. Its service routine has to read the sensor values and has to reset *Tick* to 0 (here by the function **TriggerEndstufe**).

DACDRV::SenGetVolt

double **SenGetVolt**(int *mode*, double *h*)

Parameters *mode* determines the return value:
=1, sensor value in [Volt] for tank 1
=2, sensor value in [Volt] for tank 2
=3, sensor value in [Volt] for tank 3
else return value of 0 double *h* is the liquid
level in [cm] of a tank

Description The function **SenGetVolt** returns the sensor value in [Volt] for the sensors of tank 1, 2 or 3 depending on the parameter *mode* for a given liquid level *h* in [cm] by evaluation of the accompanying straight line equation (sensor characteristic).

Return: The sensor value (double) in [Volt] for a liquid level in [cm].

DACDRV::OfcEichStart

void **OfcEichStart**(void)

Description The function **OfcEichStart** adjusts the initial values of the liquid levels for the determination of the outflow coefficients to the current values of the liquid levels in all tanks and stores the counter content of the interrupt service routine to *EichTick*.

DACDRV::OfcEichStop

void **OfcEichStop**(void)

Description The function **OfcEichStop** adjusts the final values of the liquid levels for the determination of the outflow coefficients to the current values of the liquid levels in all tanks and reads the counter content of the interrupt service routine. The time since calling the function **OfcEichStart** is calculated by comparison with *EichTick*. The three outflow coefficients are calculated using this time and the differences of the initial and final liquid levels.

DACDRV::OfcEichGet

double **OfcEichGet**(int *i*)

Parameters *i* determines the outflow coefficient:
i=0, tank 1-3
i=1, nominal outflow tank2
i=2, tank 3-2

Description The function **OfcEichGet** returns the outflow coefficient referenced by the parameter *i*.

Return: The outflow coefficient (double) referenced by *i*.

DACDRV::PfrEichStart

void **PfrEichStart**(int *i*)

Parameters *i* determines the base point (control signal in [Volt]) for the pump characteristic.

Description The function **PfrEichStart** adjusts the initial values for the liquid levels used to determine the pump characteristics to the current values of the liquid levels in the tanks 1 and 2. The control signal for both pumps is set to the value of the base point referenced by the parameter *i*. The counter content of the interrupt service routine is stored to *EichTick*.

DACDRV::PfrEichStop

void **PfrEichStop**(int *i*, double *time*)

Parameters *i* determines the base point (pump flowrate in [ml/s]) for the pump characteristic.

time is the delay time for settling of the liquid levels.

Description The function **PfrEichStop** resets the control signal for both pumps to 0 and calculates the time since calling the function **PfrEichStart** by comparing the counter content of the interrupt service routine with *EichTick*. After passing the delay time *time* for the settling of the liquid levels in the tanks 1 and 2 these values are taken as the final liquid levels to determine the pump flow rates. The pump flowrates are calculated for the base point referenced by the parameter *i*. After the determination of the last base point (*i*=8) the flag indicating pump calibration is reset again.

DACDRV::PfrEichGet

double **PfrEichGet**(int *p*, int *i*)

Parameters *p* determines the pump (=1- pump 1, =2- pump 2).

i determines the base point for the pump flowrate (=0,..,8).

Description The function **PfrEichGet** returns the pump flowrate for the pump referenced by *p* and for the base point referenced by the parameter *i*.

Return: Pump flowrate (double) referenced by *i* for pump *p*.

DACDRV::Save

int **Save**(char* *FileName*)

Parameters **FileName* pointer to the name of the file to which the calibration data are be written.

Description The function **Save** stores the sensor characteristics (offset and gradient of the straight line equation), the outflow coefficients, the pump characteristics (9 base points for each characteristic) as well as the effective cross sections of the tanks to the file with the given name. The function returns 1 when the file exists, otherwise 0.

Return: File status:
=1, file exists,
=0, file doesn't exist.

DACDRV::Load

int **Load**(char* *FileName*)

Parameters **FileName* pointer to the name of the file from which the calibration data are be read.

Description The function **Load** reads the sensor characteristics (offset and gradient of the straight line equation), the outflow coefficients, the pump characteristics (9 base points for each characteristic) as well as the effective cross sections of the tanks from the file with the given name. The function returns 1 when the file exists, otherwise 0.

Return: File status:
=1, file exists,
=0, file doesn't exist.

DACDRV::eichok

int **eichok**(void)

Description The function **eichok** is a dummy routine reserved for future applications.

Return: Is always equal to 1

DACDRV::CheckSystem

int **CheckSystem**(void)

Description The function **CheckSystem** is a dummy routine reserved for future applications.

Return: Is always equal to 1

DACDRV::CheckFree

int **CheckFree**(void)

Description The function **CheckFree** is a dummy routine reserved for future applications.

Return: Is always equal to 1

DACDRV::StartInterrupt

virtual void **StartInterrupt**(void)

Description The function **StartInterrupt** activates the output stage release by sending the necessary trigger pulse and starting the rect signal. The interrupt behaviour is retained.

DACDRV::TriggerEndstufe

virtual void **TriggerEndstufe**(void)

Description The function **TriggerEndstufe** toggles the level of the rect signal for the output stage release and resets *Tick* to 0.

2.2 The Class WDAC98

The class **WDAC98** realizes the interface between the class DACDRV and the driver functions (DAC98.DRV, DAC6214.DRV) of the PC adapter card. Calling the DRV-functions is carried out by "SendMessage"-functions using commands and parameters as described with the driver software (see also IODRVCMD.H).

The files WDAC98.CPP and WDAC98.H contain the class **WDAC98** with the functions:

double **ReadAnalogVolt**(int *channel*)

void **WriteAnalogVolt**(int *channel*, double *val*)

int **ReadDigital**(int *channel*)

void **WriteDigital**(int *channel*, int *value*)

unsigned int **GetCounter**(void)

unsigned long **GetTimer**(void)

unsigned int **ReadDDM**(int *channel*)

void **ResetDDM**(int *channel*)

void **ReadAllDDM**(unsigned int &*cnt0*, unsigned int &*cnt1*, unsigned int &*cnt2*)

void **ResetAllDDM**(void)

WDAC98::ReadAnalogVolt

double **ReadAnalogVolt**(int *channel*)

Parameters *channel* is the number of the analog input channel, which is to be read.

Description The function **ReadAnalogVolt** reads the analog input channel specified by *channel* and returns the corresponding voltage value. The value is in the range from -10.0 to +10.0 with the assumed unit [Volt].

Return: The input voltage of the analog channel in the range from -10.0 to +10.0.

WDAC98::WriteAnalogVolt

void **WriteAnalogVolt**(int *channel*, double *val*)

Parameters *channel* is the number of the analog output channel, to which a value is to be written.

val is the value for the analog output.

Description The function **WriteAnalogVolt** writes the value *val* in the range from -10.0 to +10.0 (with the assumed unit [Volt]) as an analog voltage to the specified analog output channel. Values outside of the mentioned range are limited internally.

WDAC98::ReadDigital

int **ReadDigital**(int *channel*)

Parameters *channel* is the number of the digital input channel, which is to be read.

Description The function **ReadDigital** reads the state (0 or 1) of the specified digital input channel and returns this value.

Return: The state (0 or 1) of the specified digital input.

WDAC98::WriteDgital

void **WriteDgital**(int *channel*, int *val*)

Parameters *channel* is the number of the digital output channel, to which a value is to be written.

value is the new state of the digital output.

Description The function **WriteDgital** writes the value *val* (0 or 1) to the specified digital output channel and with this sets its state.

WDAC98::GetCounter

unsigned int **GetCounter**(void)

Description The function **GetCounter** returns the content of 16-bit-counter register.

Return: The content of the 16-bit-counter register.

WDAC98::GetTimer

unsigned long **GetTimer**(void)

Description The function **GetTimer** returns the content of the 32-bit-timer register.

Return: The content of the 32-bit-timer register.

WDAC98::ReadDDM

unsigned int **ReadDDM**(int *channel*)

Parameters *channel* is the number of the DDM device, which is to be read.

Description The function **ReadDDM** returns the content of the counter register of the specified DDM device (incremental encoder).

Return: The content of the specified DDM counter register.

WDAC98::ResetDDM

unsigned int **ResetDDM**(int *channel*)

Parameters *channel* is the number of the DDM device, which is to be reset.

Description The function **ResetDDM** resets the content of the counter register of the specified DDM device (incremental encoder).

WDAC98::ReadAllDDM

void **ReadAllDDM**(unsigned int &*cnt0*, unsigned int &*cnt1*, unsigned int &*cnt2*)

Parameters &*cnt0* is a reference to the content of the counter register of the DDM device No. 0, which is to be read.

&*cnt1* is a reference to the content of the counter register of the DDM device No. 1, which is to be read.

&*cnt2* is a reference to the content of the counter register of the DDM device No. 2, which is to be read.

Description The function **ReadAllDDM** should read the contents of the counter registers of the DDM devices 0, 1 and 2 at the same time and return the values by references.

This function is still not realized but reserved for future applications.

WDAC98::ResetAllDDM

void **ResetAllDDM**(void)

Description The function **ResetAllDDM** resets the contents of the counter register of all DDM devices (incremental encoders) at the same time.

3 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.

4 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.

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	

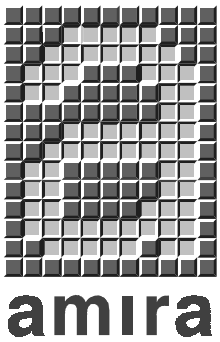
Table of the special commands with the flag DRV_USER:				
PARAMETER1				Return
Command	Channel Number			
	DAC98	DAC6214	DIC24	
DRVCMD_INIT initializes the card and has to be the first command				0
DRVINFO_AREAD returns the number of analog inputs				8 for DAC98, 6 for DAC6214, 0 for DIC24
DRVINFO_AWRITE 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**.



Extension Kit Electrical Control Valve and Adapter Box

for

Laboratory Setup DTS200 Three - Tank - System

 printed 10.08.2000

Bismarckstraße 67 · im Technologiezentrum · D-47057 Duisburg · Telefon 0203/37809-50 · Telefax 0203/37809-56

1 Mounting Instructions for Electrical Control Valves

1.1 Control Valve as Connection Valve between the Tanks

1. Drain off all three tanks.

2. Put an absorbent cloth under the connection valves between the tanks because some remaining water may flow out by dismantling the valves.

3. Loosen the mounting screws of all tanks.

4. Loosen both spigot nuts of the connection valves and move them towards the tanks. The valve can be radially pulled out.

Steps 5 and 6 are only necessary if the delivery of the control valve contains additional fittings. If not, continue with step 7.

5. Now unscrew the remaining fittings from the tanks. Remove the sealing material which may be left in the threaded holes of the tanks.

6. Mount the fittings which were delivered together with the electrical control valve (use a spanner). Don't forget to insert the fittings into the spigot nuts first. Screw the fittings into threaded holes of the tanks carefully.

7. Mount the control valve now by screwing the spigot nuts. The electrical connectors should be directed to the back of the tank system.

Steps 8-11 are only necessary if there is no fastening for the valves located on the base plate.

8. Now mark the mounting holes through the base of the valves.

9. Dismantle the control valves again.

10. Now drill the threaded holes at the marked locations on the base plate (6 mm metric screw- thread). The holes may not be drilled through the base plate.

11. Mount the control valve now by screwing the spigot nuts. Afterwards fix the valves to the base plate by using the delivered hexagon head scews (6 mm metric screw-thread, 15 mm length) with the corresponding plain washers. Use a spanner with 10 mm size.

12. Finally the tanks are screwed down to the base plate.

Now the system is ready for work. Check the closeness of all screwed connections. In case of possible leakage the corresponding connections have to be tightened.

The electrical connections are displayed on the control valves.

1.2 Control Valve as Drain Valve

1. Drain off all three tanks.

2. Put an absorbent cloth under the drain valve because some remaining water may flow out by dismantling the valve.

3. Loosen the spigot nut of the drain valve which has to be replaced and move the spigot nut towards the tank. The valve can be radially pulled out.

Steps 4 and 5 are only necessary if the delivery of the control valve contains additional fittings. If not, continue with step 6.

4. Now unscrew the remaining fitting from the tank. Remove the sealing material which may be left in the threaded holes of the tank.

5. Mount the fitting which was delivered together with the electrical control valve (use a spanner). Don't forget to insert the fitting into the spigot nut first. Screw the fitting into threaded hole of the tank carefully.

6. Mount the control valve now by screwing the spigot nut. The electrical connectors should be directed to the right side of the tank system.

Steps 7-10 are only necessary if there is no fastening for the valves located on the base plate.

7. Now mark the mounting holes through the base of the valve.

8. Dismantle the control valve again.

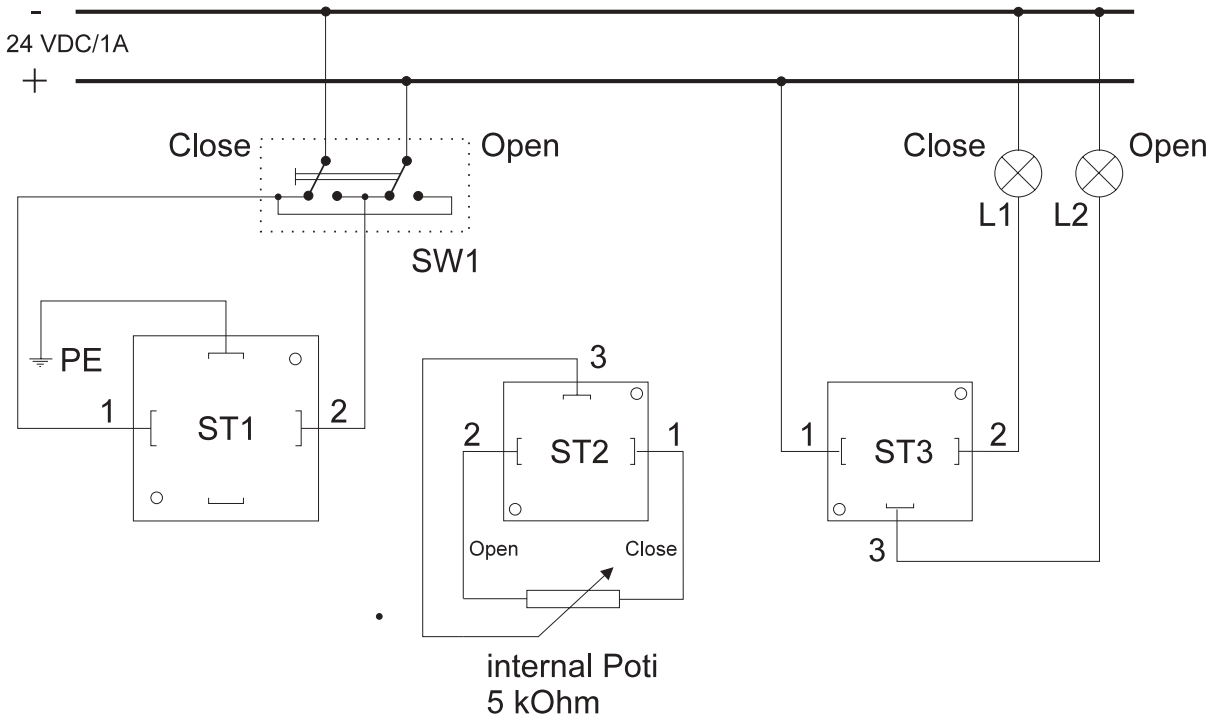
9. Now drill the threaded holes at the marked locations on the base plate (6 mm metric screw- thread). The holes may not be drilled through the base plate.

10. Mount the control valve now by screwing the spigot nuts. Afterwards fix the valve to the base plate by using the delivered hexagon head scews (6 mm metric screw-thread, 15 mm length) with the corresponding plain washers. Use a spanner with 10 mm size.

Now the system is ready for work. Check the closeness of all screwed connections. In case of possible leakage the corresponding connections have to be tightened.

The electrical connections are displayed on the control valves.

1.3 Electrical Connections



Depending on the type of the control valve there are two or three electrical connectors mounted on one side of the valve's actuator.

ST1 is the electrical connector of the valve motor. The pins 1 and 2 have to be supplied with a voltage of 24VDC/1A. The upper pin is PE (protection earth). The external switch SW1 (pole reversal circuit) should demonstrate how to open and to close the valve.

ST2 (optional) is the electrical connector of the internal potentiometer (5 kOhm). In the closed position the pins 2 and 3 are of high impedance. In the open position the pin 2 and 3 are of low impedance. Depending on the adjustment of the potentiometer inside the actuator the limit values are not exactly 5 kOhm (close) and 0 Ohm (open). This doesn't matter because in normal operation the potentiometer range will be calibrated either by external hardware or software.

ST3 is the electrical output of the internal limit switches.

Pin 1 : Common
Pin 2 : indicates that the valve is closed
Pin 3 : indicates that the valve is open

The diagram shows an example circuit where the limits are indicated by means of two lamps L1 and L2.

2 The Adapter Box

The Adapter Box serves as an interface between electrical control valves (with or without potentiometer output) and the PC adapter card i.e. DAC98 from amira.

2.1 The Rear Panel

The mains input unit is located on the right above the type plate. It contains a fuse holder, the mains inlet and the power switch. Two 50-pol. sockets labeled "PC-Connector 1" and "PC-Connector 2" are located on the rear panel in addition.

2.1.1 PC-Connector 1

This socket is connected to the module slots with the numbers 1 to 4 of the adapter box.

2.1.2 PC-Connector 2

This socket is connected to the module slots with the numbers 5 to 6 of the adapter box.

2.2 The Front Panel

The modules of the adapter box are described from left to right as follows.

2.2.1 Power Supply 1

This module provides the 24 V AC power supply for the six module slots.

2.2.2 Power Supply 2 (POWER Modul)

This module provides the DC power supplies for the six module slots. Three LEDs indicate the availability of the DC voltages.

- +15V(green): +15V power supply is available
- -15V(green): -15V power supply is available
- +5V(green): +5V power supply is available

2.2.3 Converter Module

Up to six converter modules may be mounted into the adapter box. Any unused module slot will be covered by a blind plate.

A converter module allows for operating an electrical control valve. It is connected to the control valve by means of three connectors. 5 LEDs indicate different states of the valve. A key allows for selecting either the external operating mode of the valve or the direct operating mode (open/close).

- LED "Power"(green): 24V DC power supply is available.
- LED "Valve Open"(red): Indicates a completely opened valve and that its drive is switched off.
- LED "Open"(green): Indicates that the valve is moving to the open position. This operation mode is achieved either by an external signal connected to 50-pol. socket or by pressing the key to its upper position.
- Key "Extern Control": If this key is in its middle position the valve may be controlled by an external signal connected to 50-pol. socket. The required signals are described further below. When the key is pressed upwards the valve is moving to the open position. Pressing the key downwards will move the valve to the close position accordingly. Operating the key as described above overwrites the function of external signals.
- LED "Close"(green): Indicates that the valve is moving to the close position. This operation mode is achieved either by an external signal connected to 50-pol. socket or by pressing the key to its lower position.
- LED "Valve Closed"(red): Indicates a completely closed valve and that its drive is switched off.

2.3 Technical Data

2.3.1 Adapter Box

Sizes and weight:

Length	471 mm
Depth	340 mm
Height	152 mm
Weight	6 kg

Mains supply:

Input voltage	230 V AC
Frequency	50/60 Hz
Fuse	1A M

2.3.2 Converter Module

Inputs:

Supply voltage	24V AC, 1A +15V, -15V, +5V
Open valve	TTL signal
Close valve	TTL signal

All inputs are active high (5V).

Outputs:

Limit switch valve opened	TTL signal
Limit switch valve closed	TTL signal
Analog signal valve position (from potentiometer)	Range +/- 10 V

The leads coming from the front panel of the converter module are directly to be plugged to the control valve. The connectors are to be fixed by screws.

2.4 Pin Reservation of the 50-pol. Sockets

The logic states of the signals to open or close a valve are described in table 2.1. The pin reservation of the 50-pol. connector is decribed in table 2.2.

Open control valve	Close control valve	
0	0	Drive switched off
0	1	Drive closes
1	0	Drive opens
1	1	invalid

Table 2.1: Logic of the input signals

	PC-Connector 1				PC-Connector 2	
	module 1	module 2	module 3	module 4	module 5	module 6
Input open control valve	34	36	38	40	34	36
Input close control valve	35	37	39	41	35	37
Output limit switch "Opened"	9	11	26	28	9	11
Output limit switch "Closed"	10	12	27	29	10	12
Digital Ground	42,43	42,43	42,43	42,43	42,43	42,43
Measured valve position (potentiometer)	16	17	32	33	16	17
Analog ground for measured position	15,31	15,31	15,31	15,31	15,31	15,31

Table 2.2: Pin reservation of the 50-pol. sockets