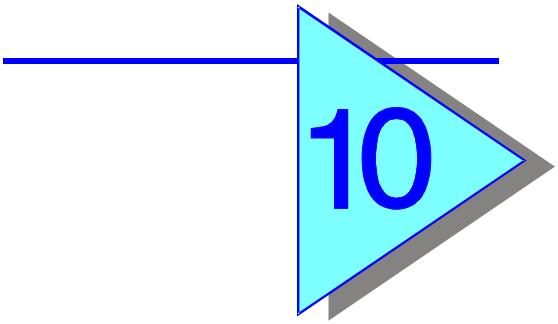


Block-oriented simulation with BORIS

Overview

Range of Application of BORIS

The block oriented simulation system BORIS allows the simulation of nearly any structured dynamic systems and is therefore in - connection with the hardware interface and the optional C-Code-generation - suitable for the following applications:

- Measurement and signal analysis,
- Analysis and synthesis of feedback control systems,
- System optimization,
- Digital technology,
- Configuration and parameterization of hardware modules.

Besides the known conventional systems even systems with fuzzy or neuro components can be handled.

For the easy and comfortable realization of ambigous and complex animations and process visualizations the separately available *Flexible Animation Builder* is recommended.

Some features of BORIS

Some of the essential features of BORIS (depending on the purchased version):

- Extensive system library:
  - Signal generators (triangle, rectangle, sinus, pulse, noise, programmable test functions)
  - Linear standard elements (PT_1, PT_2, ...)
  - Linear transfer elements of a higher level

  - Nonlinear characteristic curve elements; free definable characteristic curves; algebraic functions
  - Linear and nonlinear standard controller components
  - Fuzzy controller and neural networks
  - User-defined function blocks based on DLLs
  - Different output blocks (time responses, trajectory plot, analog and digital meter, oscilloscope, status displays)
  - Action blocks for an interactive intervention to the simulation (e. g. switch, potentiometer)
  - In- and output of signals from files resp. in files
  - Spectrum analysis by Fast-Fourier-Transformation
  - Statistic functions
  - Digital modules (e. g. logical gate and flip-flops)
  - Communication modules (e. g. DDE and TCP/IP-blocks)
- Definition of hierarchical blocks (superblocks)
- Number of system blocks only limited by the memory of the computer
- Optimal placement of system blocks; scrollable working area of nearly any size
- Different integration methods
- Automatic or half-automatic design of PID-controllers by design rules
- Numerical parameter optimization based on evolutionary strategies
- Process interface by A/D- und D/A-cards, serial measuring modules, process control systems etc.
- Transfer of any system structures into ANSI-C-Code by the efficient C-Code-Generator

In the following chapter you will get an introduction into the work with BORIS by a detailed example.

A first simulation with BORIS

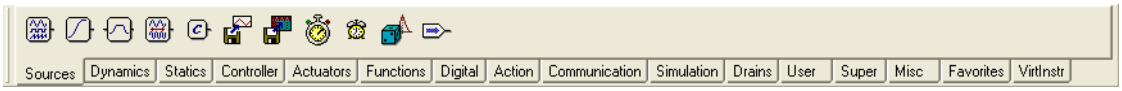
A simple example

The user interface of BORIS is basically following the Windows standard and the typical WinFACT-conventions. In the following we will learn the basic handling of the program by a simple example.

The example described in the following you can find in the examples directory under the name DEMO1.BSY.

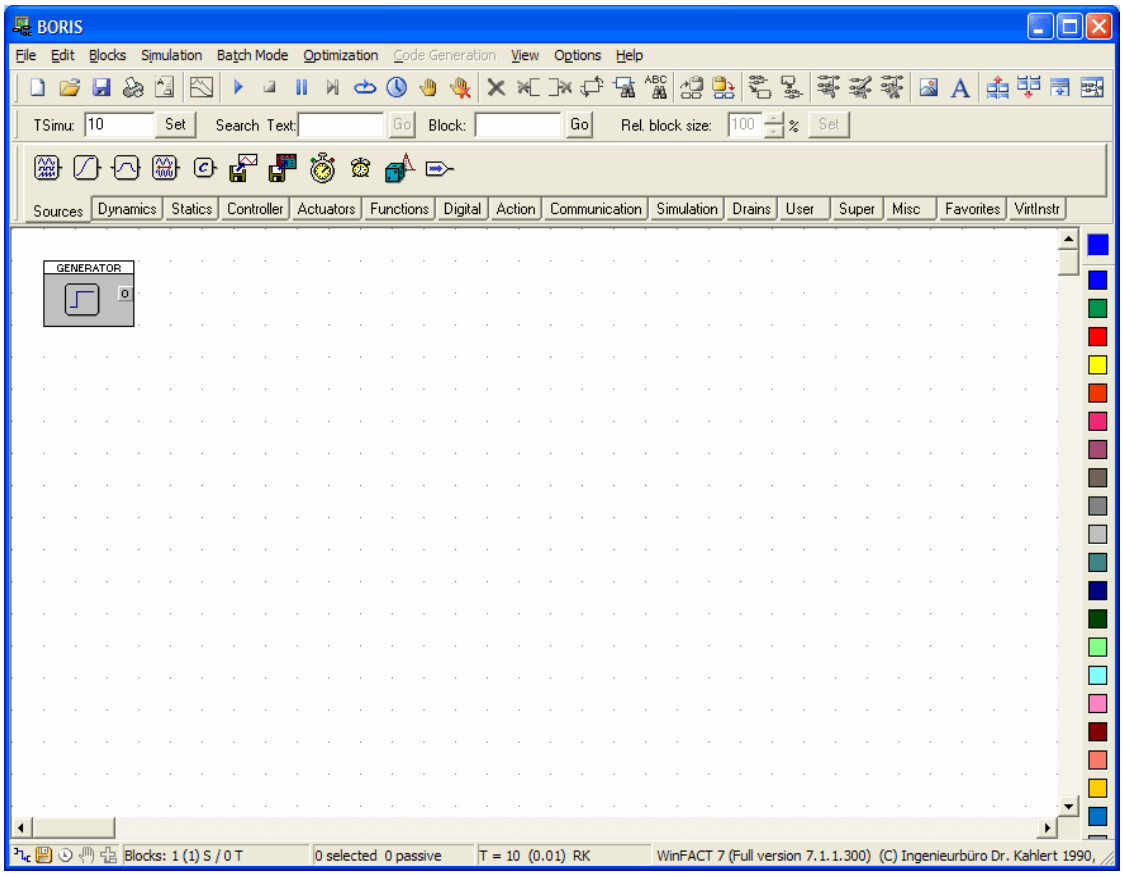
We assume a second order linear plant with the gain $K = 2$ and the time constants $T_1 = 1$ und $T_2 = 2$. At first we are interested in the response $y(t)$ of the system to a sinus input signal $u(t) = u_0 \sin \omega t$ with the amplitude $u_0 = 1$ and the frequency $\omega = 3$. How can we solve this problem with the help of BORIS?

First of all we take a look at the main window of BORIS directly after starting the program. At the upper border you find two horizontal toolbars. The lower one - the so called *system block toolbar* - enables the direct access to all system blocks and is divided into different block groups (*palettes*) which can be activated by the registers.



System block toolbar of BORIS

For the generation of our input signal we need a *signal generator* which we get by a click on the first button of the palette *Sources*. Thereupon the generator appears in the upper left corner of the drawing window. The insertion of the system block is receipted in the status bar of the main window further on.



Main window of BORIS after inserting the signal generator

Afterwards we will insert the plant itself. For that purpose we press the button *PT1T2* of the palette *Dynamics*. The block appears on the right side of the generator.

Moving blocks

We want to move the block to the right. For this purpose we select the block by a click with the left mouse button. The selection of the block is indicated by an inverted block title. With pressed mouse button we can now place the block at the desired position.

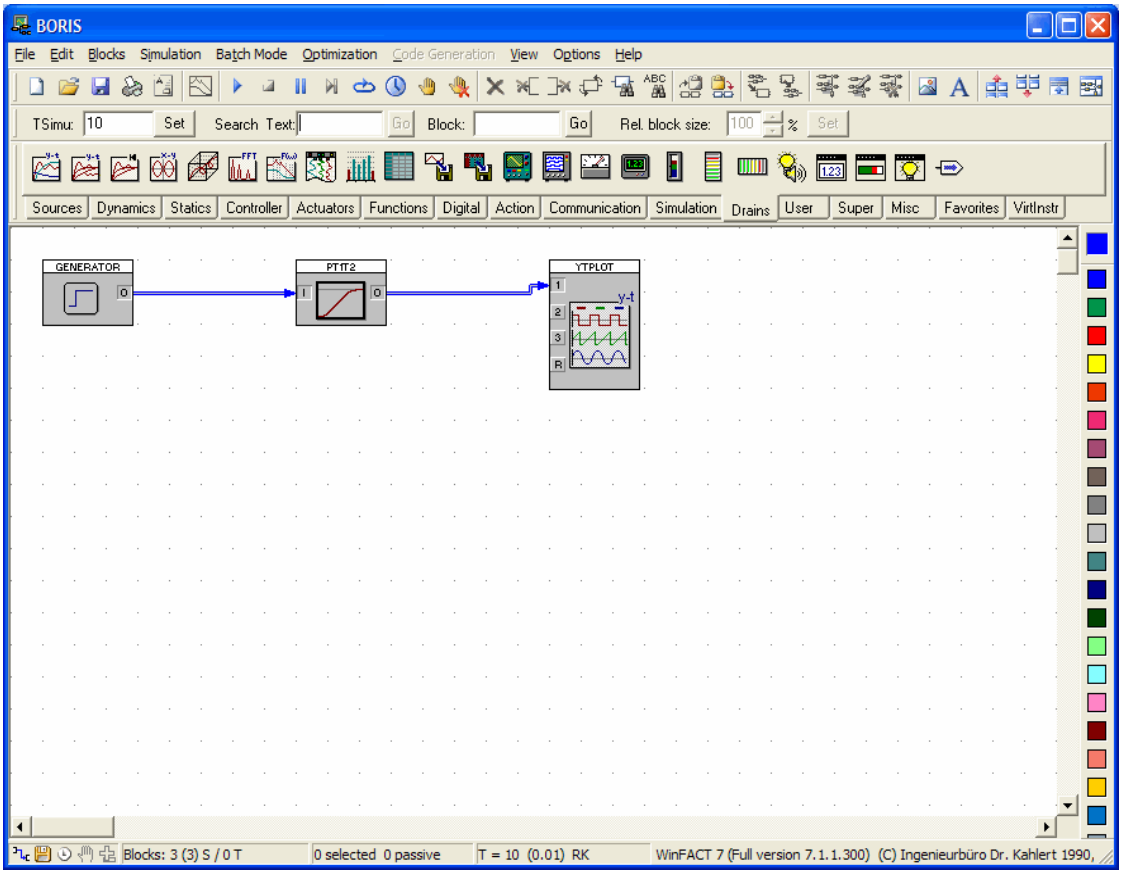
Alternatively a new block can be inserted by drag and drop. For that purpose keep the block button pressed. Then you can move the block to the desired place on the working area and drop it down.

Now we need an output block for the later display of the simulation results. Therefore we will choose a block of the type *Time response* which we insert by using the toolbar (palette *Drains*) and move to a suitable place afterwards. Simultaneously to the representation of the block symbol a window will be opened which is reduced to a symbol at first. Later on this window will contain the current simulation graphic.

In the next step we will draw the connections. Please observe the following:

*Connections have to be drawn **from the output to the input**!*

At first we click with the left mouse button on the output of our signal generator which is represented by the raised field marked with an ”O”. The cursor changes its shape into a stylized soldering iron. We move the cursor to the input ”I" of our plant - its color turns into black and shows a cross wire – and press the left mouse button again. The connection will be drawn automatically. In the same way we connect the output of the plant to the first input of the *Time response* block.



Main window after insertion of all blocks and drawing the connections

Parameterizing blocks

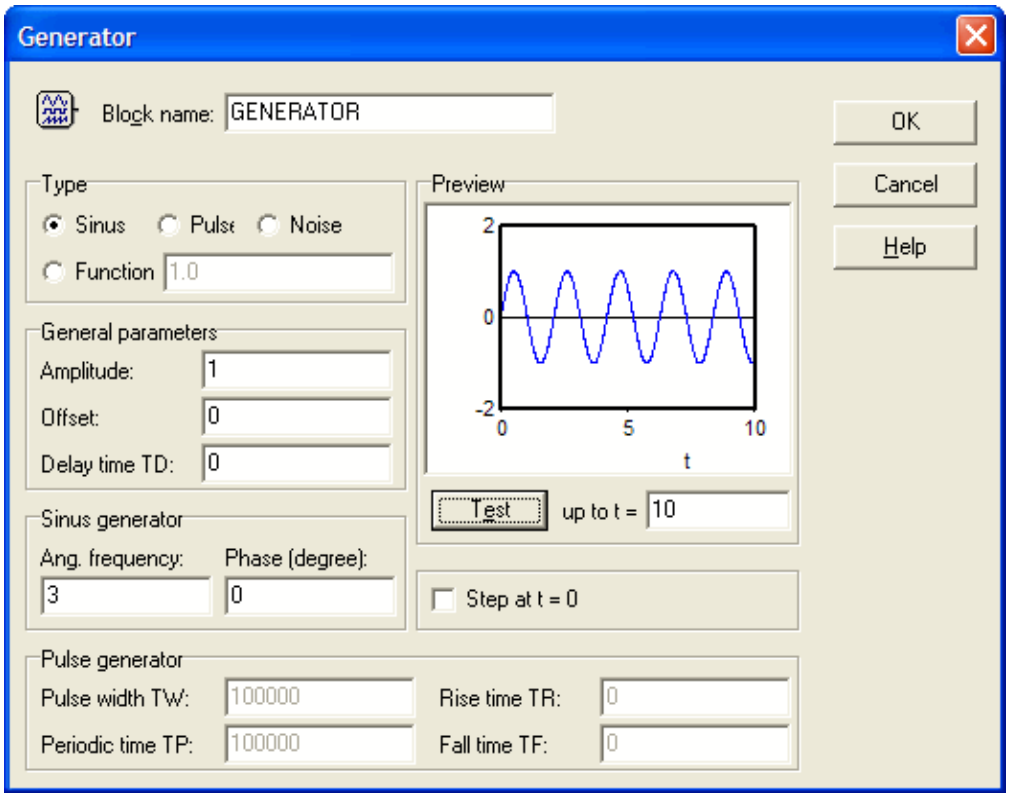
Now the structure of the system is specified. Before the start of the simulation of course the blocks have to be parameterized. At first we will do this with the generator. The easiest way of setting the parameters of the generator is a double click on the system block with the left mouse button. Afterwards the corresponding input dialog where we can specify the modifications will appear.

For our example we have to execute the following modifications which have already been documented in the dialog below.

➢ In the radio group *Type* we have to choose *Sinus*.

➢ In the field *Ang. frequency* we have to enter the value 3.

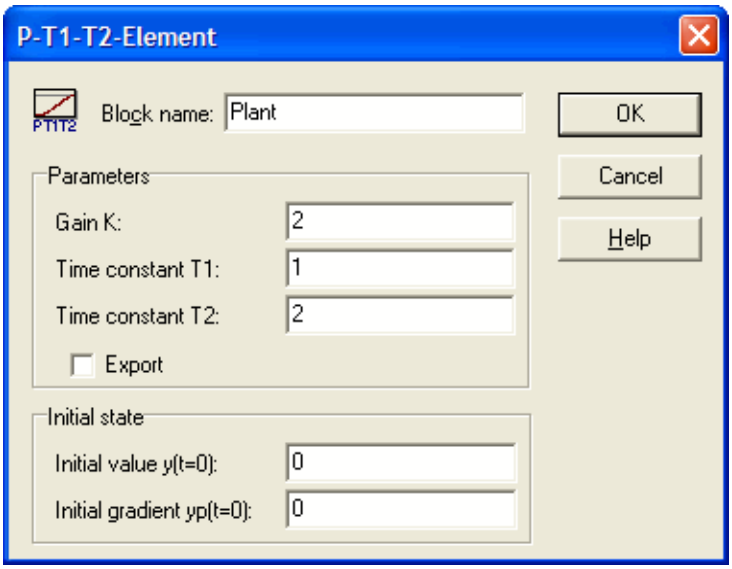
After these modifications we can test the settings by using the button *Test*.



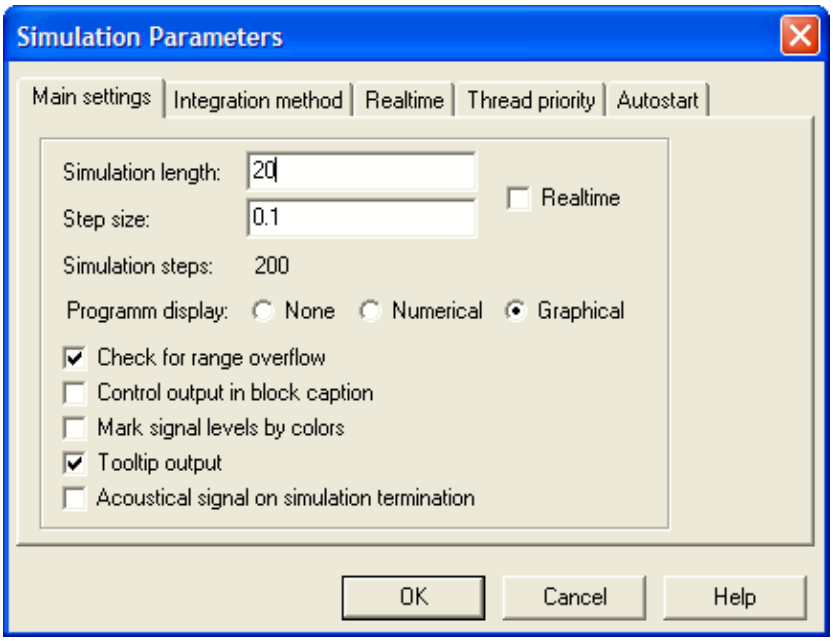
Parameter dialog for generator

By the same way we will parameterize the plant. Additionally we will change the block name into *Plant* for clarification.

Finally we have to set the simulation parameters. Therefore we choose the clock symbol in the upper horizontal toolbar, the so called *System toolbar* and get to the appropriate input dialog. We choose a simulation time of 20 and a step size of 0.1 for the simulation. With these adjustments we get a result of 200 simulation steps.

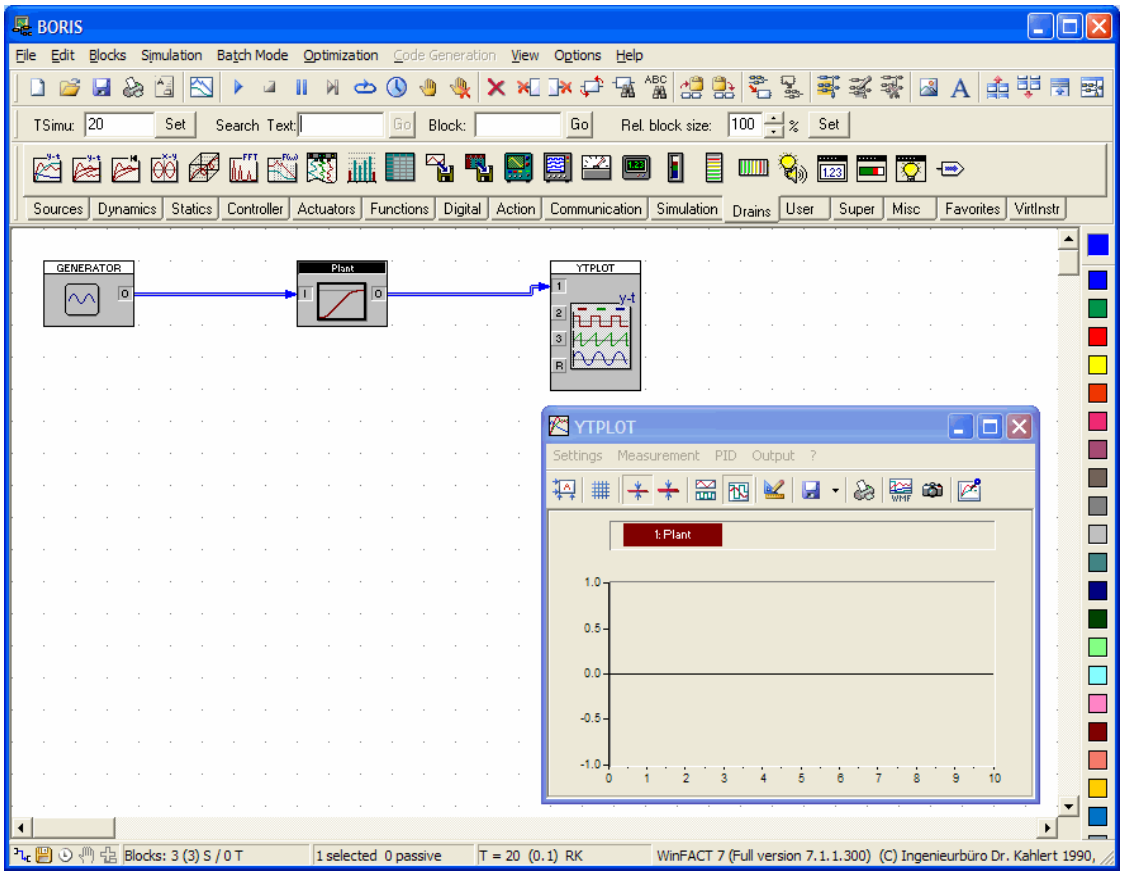


Parameterization of the plant



Adjustment of simulation parameters

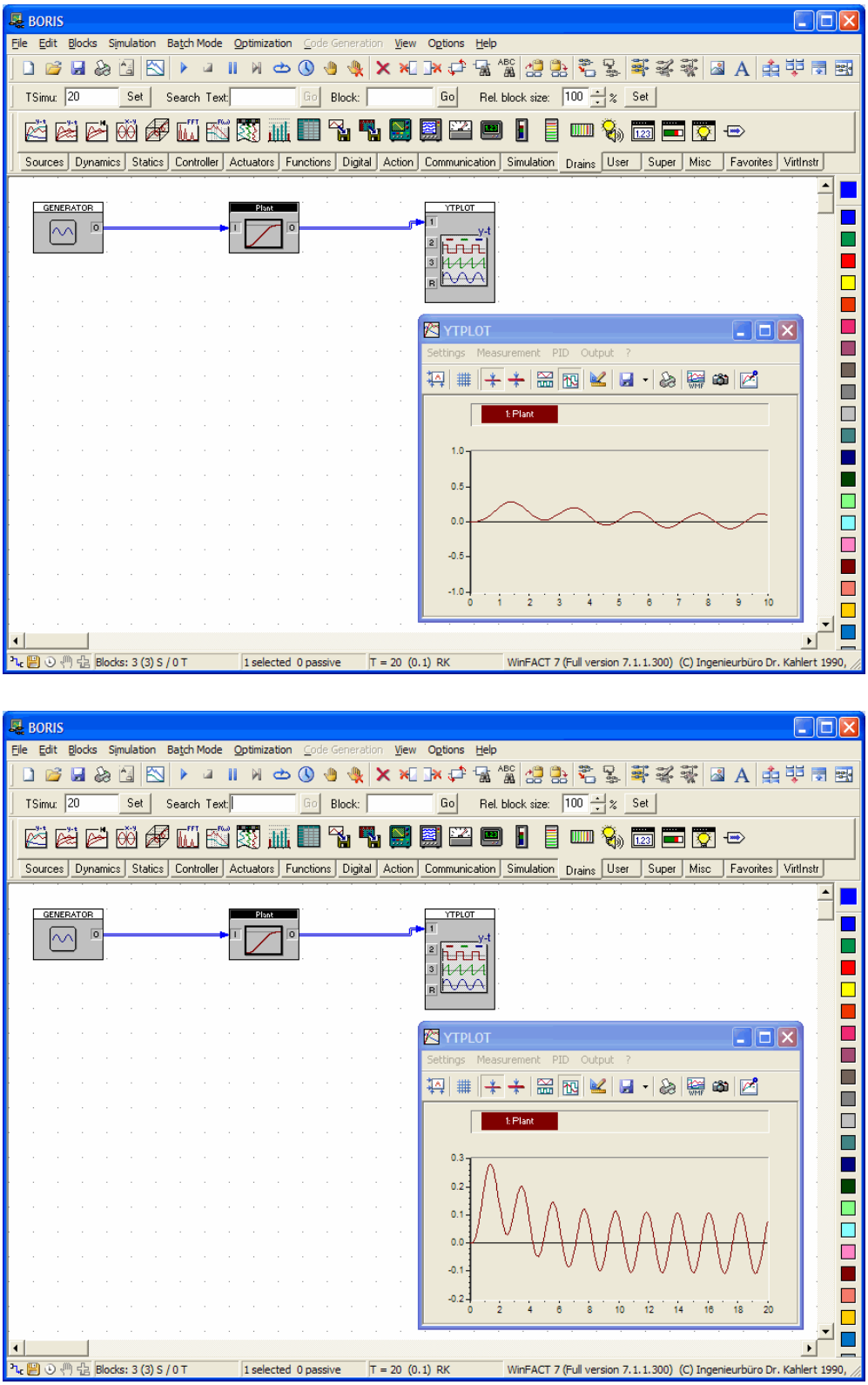
Now our system is ready for the simulation. To track the simulation progress during the simulation we normalize the time response display from the symbol state by a doubleclick. Thereupon it appears in normal size but - of course – does not show any results. Now we can change the size of the window optionally. Because we have increased the simulation time up to 20 we have to adjust the time axis using the menu option SETTINGS of the display window.



Time response display window in normal size

To start the simulation we now only have to click at the arrow in the system toolbar. The progress is notified in the status window of the main window and of course in the time response display window.

After the end of the simulation we notice that the scaling of the amplitude values in the time response display is rather unfavourable. This can be changed by a click on the left button of the toolbar of the window. Thereupon the curve will be drawn automatically with the optimal scaling.



Presentation of the simulation progress (top) and time response display window after the automatical scaling (bottom)

Proceeding from the existing system we want to pass over to a more complex example. Our plant should be completed to a closed control loop which contains the following components in addition to the plant:

- a P-controller with a gain of $K_R = 2$
- a measuring element in the feedback (PT_1-element with $K = 1$, $T = 0.5$)
- a summing junction for the comparison of reference and current value

The example described in the following you can find within the examples directory under the name DEMO2.BSY.

Because we need the present blocks further more it is sufficient to delete the existing connections. Therefore we have to select the plant by a click with the left mouse button and then delete the output connection by the corresponding button of the toolbar. In the following we are able to delete the input connection to the generator by the button left beside.

We only want to give a rough description of the following steps. At first we will insert the following blocks:

the P-controller (palette *Dynamics*)

the PT_1-measuring element (palette *Dynamics*)

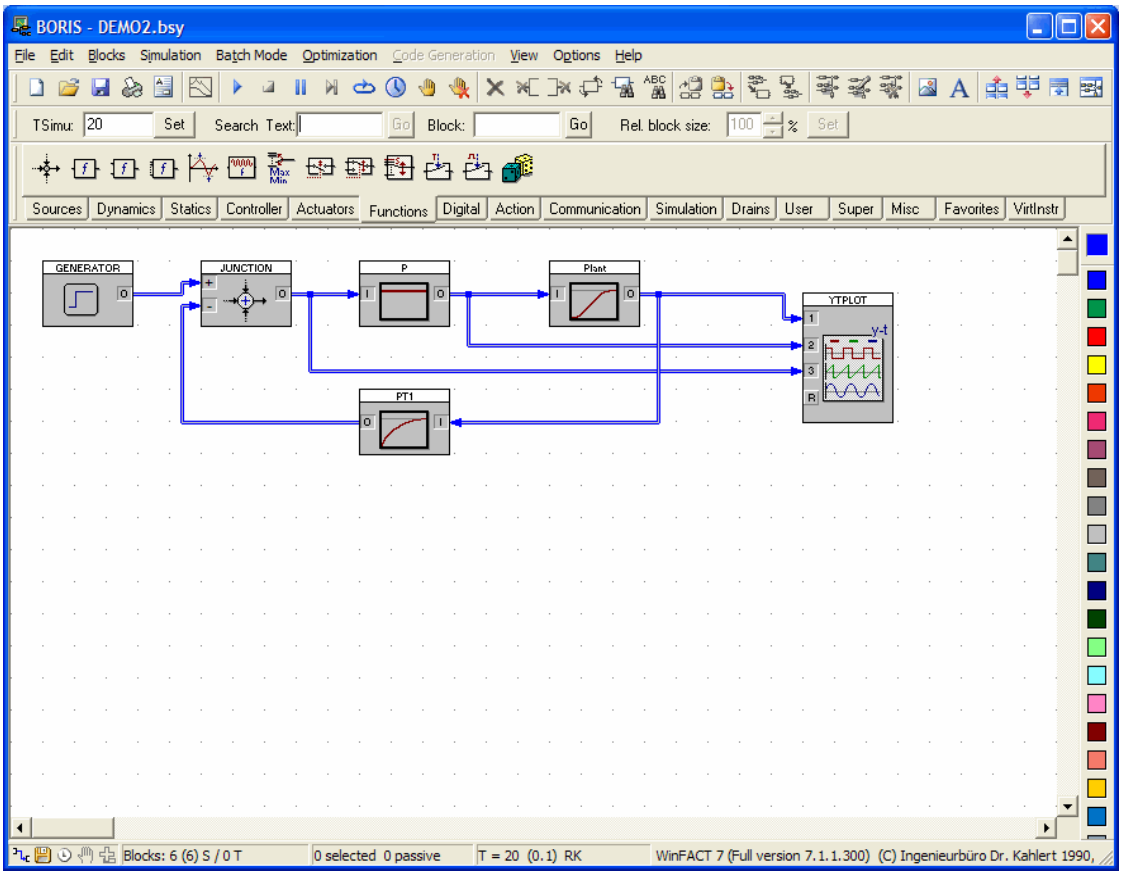
the subtractor (junction, palette *Function*)

Now we draw the corresponding connections and parameterize the single blocks. We have to assign the negative sign for the feedback for the junction. Then we set the generator to the operation *Pulse* to produce a unit step response.

The position of the measuring element in the feedback is unfavourable. This can be improved in a quite simple way by a rotation of the block by 180 degrees so that the input is on the right side. The corresponding button you also find in the system toolbar.

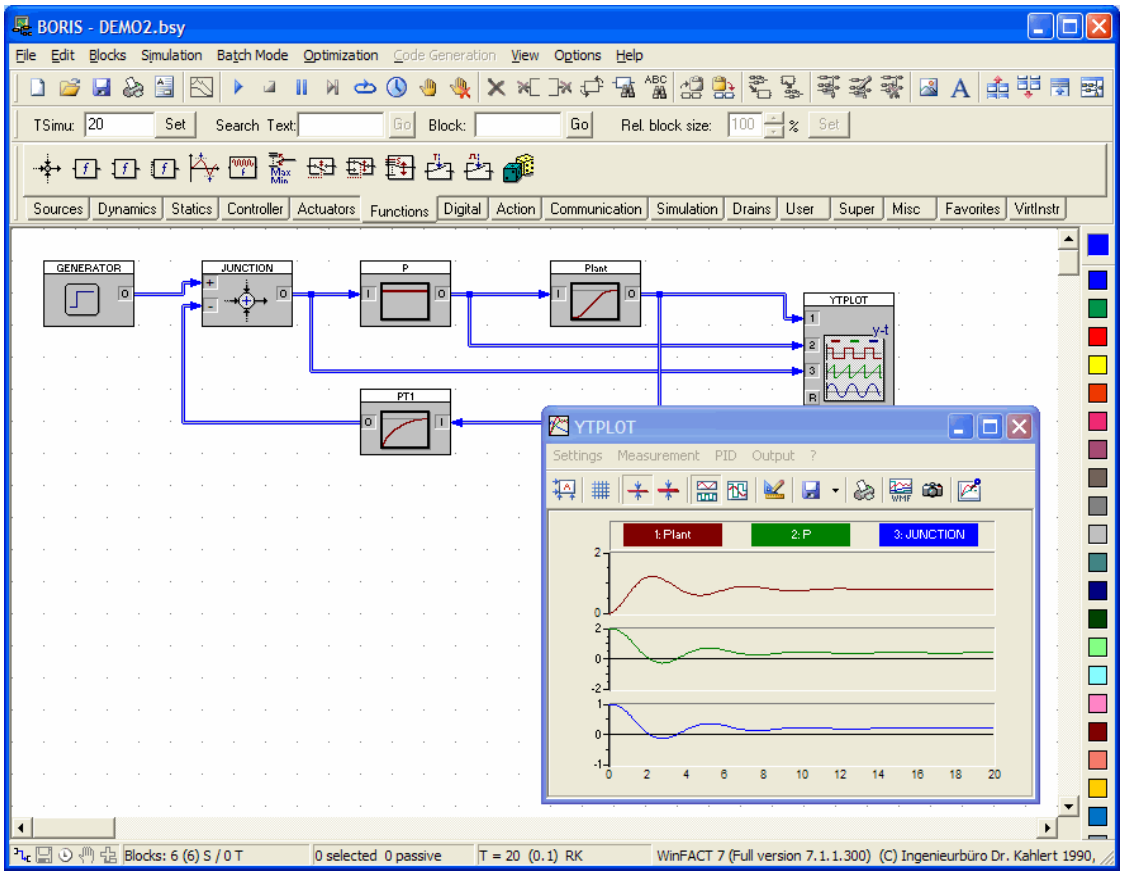
Besides the controlled variable - that is the output of the plant – we will as well connect the correcting variable to the time response block (output of the P-controller) and the control deviation (output of the subtractor).

If we have carried out the modifications correctly our application window will approximately look like this:



Complete control loop

To size all three signals which have to be presented in separate diagrams we have to deactivate the option *All curves in common diagram* of the time response dialog. This option you get in the menu SETTINGS. Otherwise you click at the corresponding toolbar button. The subsequent simulation supplies the following results:



Results of the simulation

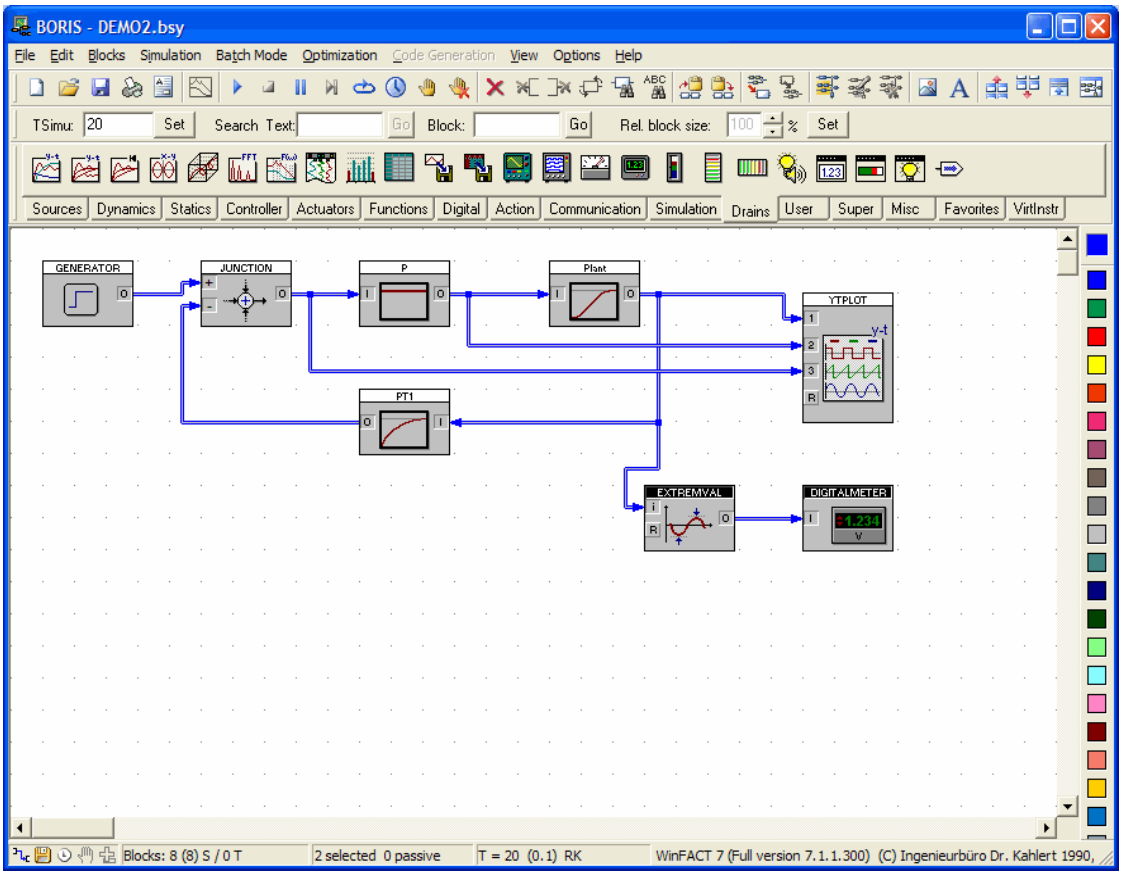
Finally we want to expand our system to get an exact value of the maximum output variable. For this purpose we insert two further system blocks:

an extremal value block (palette *Function*)

a digital display (palette *Drains*)

This example you can find within the examples directory under the name DEMO3.BSY.

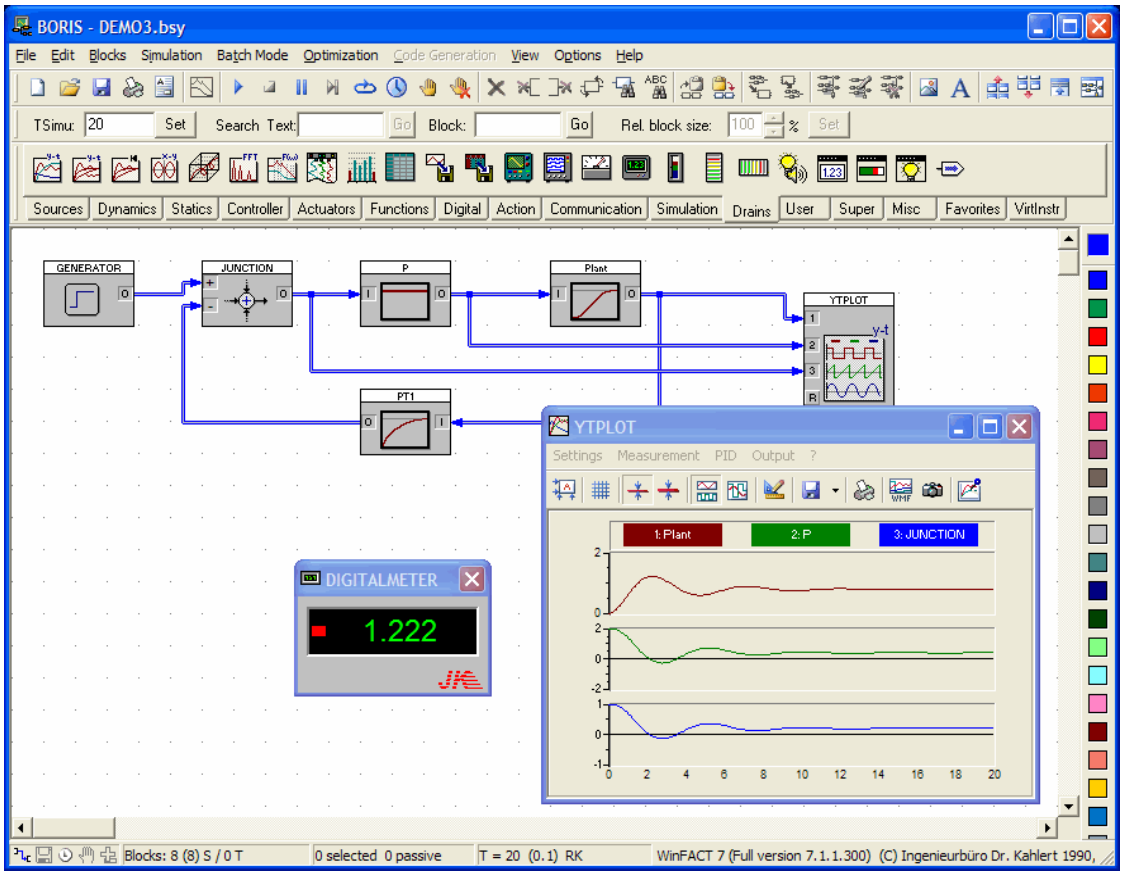
Our simulation system now has the following structure:



System with additional extremal value determination

Before we start the simulation we normalize both displays (time response and digital display) from the symbol state. The following screen graph shows the simulation result. From the digital instrument we get a maximum output variable with a value of nearly 1.22.

Now we want to finish the introducing examples. In the following chapters you find a complete and detailed description of all the features, options and available system blocks.



Simulation result

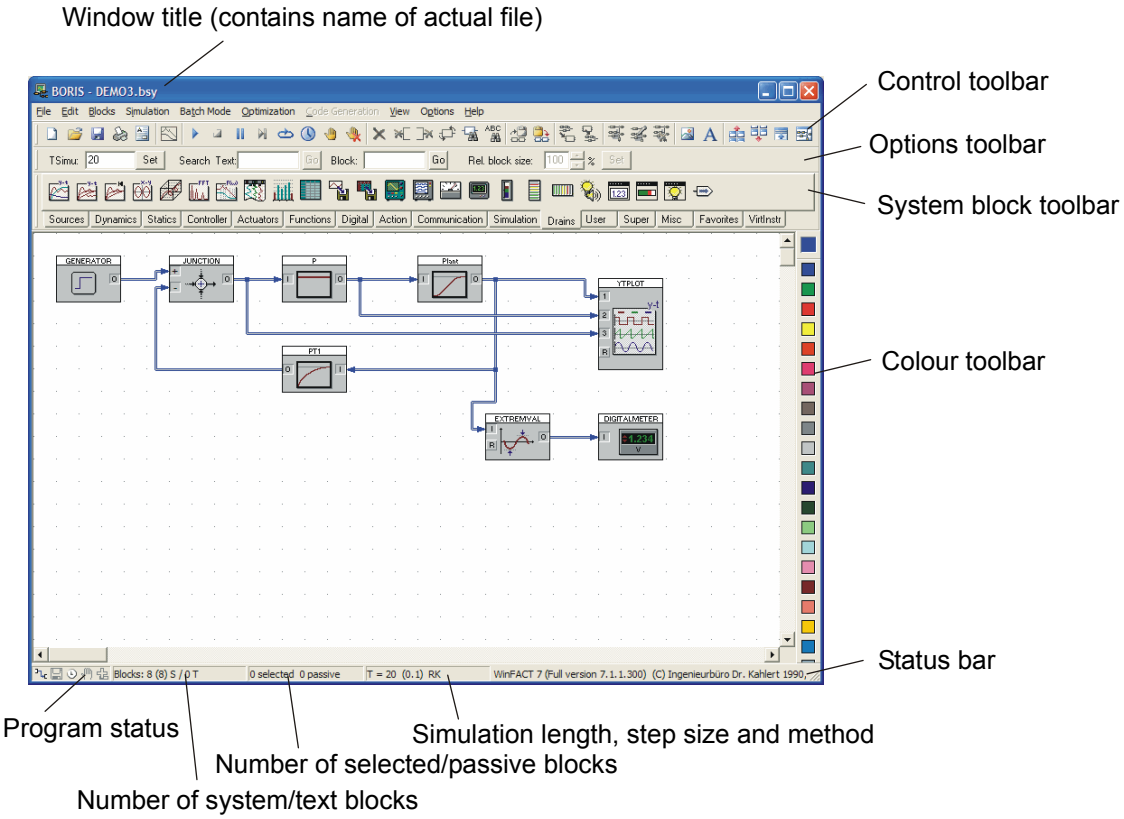
Building simulation structures

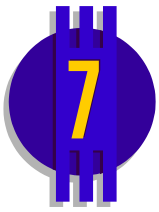
Components of the BORIS main window

The following graph shows the main window of BORIS with the single components. Besides the Windows standard elements the window contains the following components:

- A first horizontal toolbar (system toolbar) below the menu. This toolbar contains buttons for the most frequently used functions. With

OPTIONS | TOOLBARS | SYSTEM TOOLBAR VISIBLE the toolbar can alternatively be activated or deactivated





- A second horizontal toolbar (options toolbar) which allows a fast search for text or system blocks within the current system structure as well as the change of block sizes. This toolbar can be activated/deactivated via the OPTIONS | TOOLBARS | OPTIONS TOOLBAR VISIBLE menu option.
- A third horizontal toolbar which is divided into several palettes (system block toolbar). This toolbar allows a direct access to all system blocks. The palette *Favorites* can be configurated by the user in any way (s. chapter *Configuration of the system block toolbar*). With OPTIONS | TOOLBARS | SYSTEM BLOCK TOOLBAR VISIBLE the toolbar can alternatively be activated or deactivated
- A vertical toolbar at the right border (color toolbar). This toolbar allows a fast modification of colors, e. g. those of connections or text blocks
- A status bar at the lower edge of the window which shows the number of currently present system blocks, the number of selected or passivated blocks (please note that the display of the number of system

blocks only shows "real" system blocks and no text blocks whereas the display of the selected or passivated blocks shows all types of blocks!) and the current simulation parameters in the mode T = T_{Simu} (ΔT). T_{Simu} represents the simulation length and ΔT the simulation step size. The following abbreviation (*RK* above) specifies the selected integration method. If real time simulation was activated an additional *RT* is included.

If this option is not deactivated the status bar shows the progress during the simulation. On the left border of the status bar the current state of the system is shown by five icons:

 Autorouter active

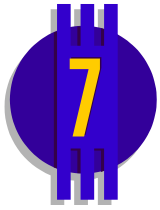
 System has not been saved since the last modification

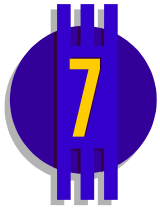
 Simulation is running

 Breakpoint active

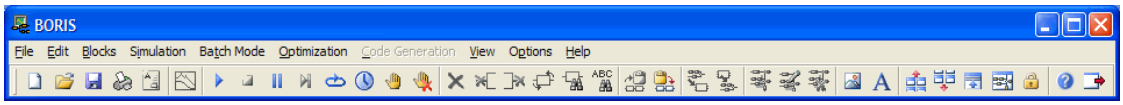
 Optimization is running

- The system window that shows the current system structure. It is scrollable by the horizontal and vertical scrollbars or the mouse wheel. At the beginning you see the visible section at the upper left corner. This state can be restored at any time with OPTIONS | VIEW SECTION TO ORIGIN. As a standard the window has a dotted grid to align all system blocks at their upper left corner. With the menu OPTIONS | SNAP TO GRID the automatical alignment can be deactivated to place the system blocks optionally. The display of the grid can be deactivated by OPTIONS | SHOW GRID.





In some cases it might be nice to hide the system strcuture, e. g. during a simulation run. This can be done via the VIEW | MINIMIZE TO TOOLBAR/RESTORE menu option or the  toolbar button. In this case the main window of BORIS is minimized to the height of the system toolbar (see screenshot below) resp. restored from this size to its original size.



Main window of BORIS minimized to toolbar height

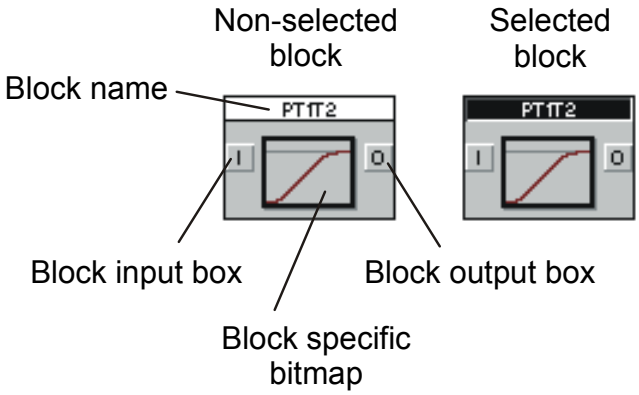
Inserting and editing system blocks

You can insert or edit the system blocks in the following ways:

- *Inserting* new blocks from the block library
- *Selecting* single blocks or block groups
- *Moving* single blocks or block groups
- *Deleting* single blocks or block groups
- *Rotating* of single blocks
- *Copying* and *Pasting* of single blocks or block groups
- *Deactivating* of single blocks or block groups
- *Parameterizing* of blocks
- Changing the *block size*

Block representation

The following graphic shows the visual representation of a system block:



It consists of the following components:

- The *caption* which contains the *block name*. By default the block name corresponds to the name of the block type but can be modified by the parameter dialog (max. 25 characters). In the selected mode the caption is inverted. If the block name is too long to be shown in the caption it will automatically be shortened.
- The *in*- and *output boxes* which are necessary for drawing connections. The input boxes of blocks with only one in- and output is named with an "I" and the output box with an "O". If the block has several in- and

outputs normally they are numbered or characterized by special terms (e. g. "R" and "S" for the RS-flip-flop)

- A *block specific bitmap* which characterizes each block type clearly.

Depending on the block type a block can have up to fifty in- and outputs.

To insert a new block...

...you can choose from three different ways:

1. Click with the left mouse button on the corresponding button of the system block toolbar. The block will automatically be placed on a vacant position of the working sheet.
2. Choose the corresponding block from the BLOCKS sub menu of the main menu. The block will be placed automatically as well.
3. Click with the left mouse button on the corresponding button of the system block toolbar, hold the mouse button pressed and draw the block by "drag and drop" to the desired position. If you release the mouse button the block will be placed in this position.

The way described under No. 3 is recommendable especially if a lot of blocks have already been placed on the working sheet.

How to select blocks

There are different ways available to select single blocks or complete block groups:

- A single block can be selected by a simple click with the mouse. If another block has been selected before, this selection will be removed.
- If more blocks should be selected you have to click on them with pressed down <Shift>- or <Ctrl>-key. The selection will be removed by a second click on a block which has already been selected.
- Alternatively you can select complete block groups by drawing a rectangle with pressed left mouse button. All blocks which are completely enclosed by the rectangle will be selected.
- With the menu option EDIT | SELECT ALL BLOCKS you can select all blocks at the same time.

All selections can be removed by a click on any vacant position of the drawing area.

Moving blocks

If you want to move a block you have to act in the following way:

1. Select the block or blocks you want to move.
2. Click with the left mouse button on the inner area of the block you want to move or – if you want to move a whole block group – the inner area of any selected block with pressed mouse button and move the blocks the way you want. The cursor changes its shape into the form of a cross. If you reach the edge of the window an automatical scrolling will take place.

After being moved the blocks will be aligned to avoid overlapping if necessary.

Moving the complete system

During the system configuration it can happen even in the case of very complex systems that further system blocks have to be inserted left of or above the system although there is no more space available. Therefore BORIS allows the movement of the whole system (all blocks and connections) in a very easy way without having selected it before. The movement can be performed in any direction by one unit of the grid for each step. These are the menu items you need:

OPTIONS | MOVE STRUCTURE RIGHT

OPTIONS | MOVE STRUCTURE LEFT

OPTIONS | MOVE STRUCTURE DOWN

OPTIONS | MOVE STRUCTURE UP

Deleting blocks

Blocks can be deleted in three different ways:

- By selection of the blocks which have to be deleted and choice of the menu option EDIT | DELETE BLOCK
- By selection of the blocks which have to be deleted and a click on the  button of the system toolbar.
- By selection of the blocks which have to be deleted and a click with the right mouse button. In the now appearing popup menu you have to choose the option DELETE. If only a single block is to be deleted this

block can be clicked at with the right mouse button without being selected before.

The in- and output connections of the deleted blocks will automatically be deleted as well.

How to rotate blocks

The *orientation* of a block can be rotated by 180° so that the inputs are on the right side of the block (e. g. for blocks in a feedback). For that the block has to be selected first and then to be rotated by choosing the menu option EDIT | ROTATE BLOCK or a click on the button. Alternatively you can click at the block with the right mouse button and then choose the option ROTATE in the now appearing popup menu. The block connections will follow automatically (this is not available for manual connections; see later!).

Copy and paste

Copying and pasting of single blocks or complete blockgroups is transacted by a temporary file named BOCOPY.TMP. Then the selected blocks, their parameters and the connections of the blocks are copied or pasted. Please act as it is described in the following:

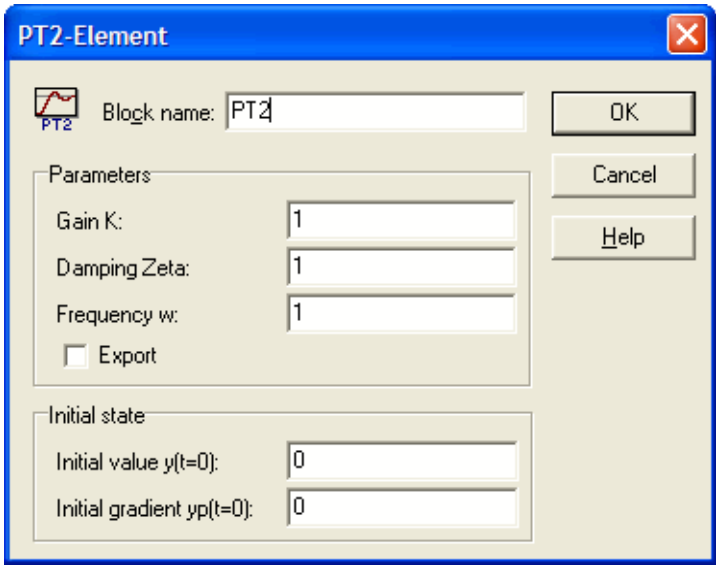
- Select the blocks which should be copied.
- Choose the menu option EDIT | COPY or click at the button of the system toolbar.
- To insert the blocks choose EDIT | PASTE or the button. The cursor thereupon changes its shape and you can move to the desired position (left upper corner) of the subsystem which should be inserted by a click on the working sheet.

Specifying the block parameters

The parameterization of a system block can be performed in the easiest way by a double click inside the block with the left mouse button. Alternatively you can call the menu sequence EDIT | EDIT BLOCK... after the selection of the block. The block typical parameter dialog then appears to specify the desired modifications. As an example the following screen shot shows the parameter dialog of an oscillating PT_2-element.

On the upper edge the parameter dialog has – independent of the block type – an edit field for the block name which in the block presentation appears in the

caption. By default this block name corresponds to the name of the block type but can be changed by the user if only the length does not exceed 25 characters. In addition to this each parameter dialog has a *Help* button to demand information of each block type.



Parameter dialog of an oscillating PT_2-element

How to set blocks passive

Sometimes you want to deactivate special parts of the simulation structure for a short time, e. g. to compare several model structures, to deactivate file output blocks or simply to save computer time in cases of very complex structures. Instead of deleting those components labouriously from the structure and inserting them again later on BORIS enables you to *set blocks passive* by a mouse click. During the simulation those blocks which have been set passive will be handled as if they don´t exist. Block inputs which are connected to a block which has been set passive therefore will be handled like open inputs (a passive block within a serial connection e. g. "interrupts" the connection!).

To set a single block passive:

➢ click at the block with the right mouse button and choose the option PASSIVE in the now appearing popup menu

or (and this is the more complicated way)

➢ select the block and choose the menu option EDIT | BLOCK PASSIVE in the main menu.

If you want to set several blocks passive at the same time

- select the blocks and
- click with the right mouse button on the working sheet and choose the option PASSIVE in the now appearing popup menu

or (also the more complicated way)

- select the blocks and choose the menu option EDIT | BLOCK PASSIVE in the main menu.

To reactivate a passive block

- click at the block with the *right* mouse button and choose the option ACTIVE in the now appearing popup menu

or (and this is the complicated way)

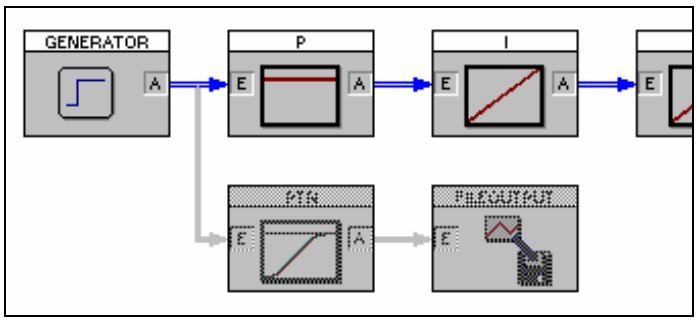
- select the block and choose the menu option EDIT | BLOCK ACTIVE in the main menu.

To reactivate several blocks which have been set passive you have to act in the same way.

To reactivate *all* the blocks

- choose the menu option EDIT | ACTIVATE ALL BLOCKS.

Blocks which have been set passive can be recognized by the grey-shaded block bitmap. The connections to or from passive blocks will also be drawn in grey color if this option has not been deactivated by the display dialog (OPTIONS | CUSTOMIZE...)



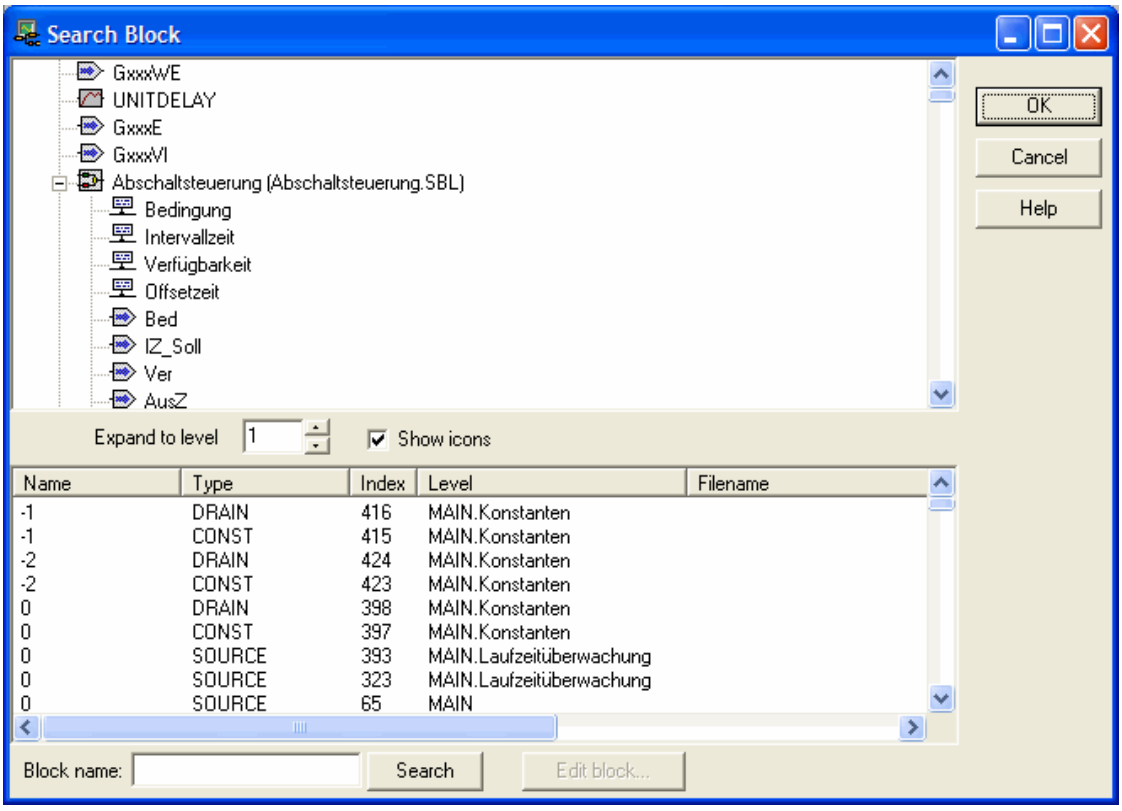
Setting blocks passive : The lower substructure leaving the generator has been deactivated.

Searching (and finding) blocks

If you work with complex simulation structures often a quick access to a special block will be necessary e. g. to modify its parameters. The search with the

scrollbars sometimes is very complicated especially if the block searched for is not present in the currently loaded system but e. g. in one of the defined superblocks (or even in a superblock inside of a superblock). Therefore BORIS has a very comfortable search function which can be called by EDIT | SEARCH BLOCK or by a click on the button of the system toolbar.

The search dialog lists up all blocks up to the lowest hierarchical level in the upper part called the tree view. The levels can be expanded step by step with the edit field *Expand up to level.* The lower view (list structure) contains all levels. The upper level (that is the currently loaded system or superblock file) is named *MAIN,* lower levels are divided by a point. For example *Main.Sub1* means a block which you can find in the superblock *Sub1* of the current system file.

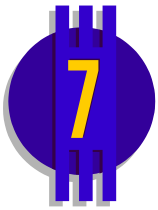


The block search dialog of BORIS

All entries are arranged in alphabetical order corresponding to the block name. By a click on the caption of the list you can change the assortion so that it follows another criterion, e. g. the block type. To find a block searched for you have two different possibilities: a doubleclick at the corresponding tree- or listview or the insertion of a block name (or a part of it, e. g. *GEN*) in the edit field *Block name* and a click at the *Search* button. The further action of BORIS now depends on the selected blocktype:

- A block which belongs to the *highest hierarchical level* will be selected automatically and the screen will be scrolled automatically in a way that the selected block is positioned exactly in the middle of the window.
- If you select a block which is *contained in a superblock* (no matter in which hierarchical level) the corresponding superblock will automatically be called in a new instance of BORIS. In this case the selected block will *not* be placed in the middle automatically so that the search function must be called in the superblock file again if necessary!

Blocks which belong to the highest hierarchical level (with the exception of superblocks) can be edited directly from the dialog by a click on the button *Edit block.*



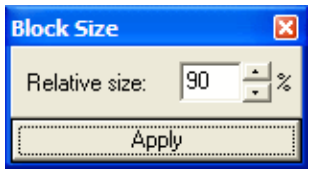
Blocks within the current system structure can also be searched directly using the options toolbar (see screenshot below). For that purpose enter the name of the block to be found or the first characters of the name (*Motor* below) into the corresponding edit field and start the search via the *Go* button. If a block which matches the name is found it appears in the upper left corner of the worksheet; if no block is found, an error message is displayed. Repeated pressing the *Go* button continues the search for further blocks matching the specified name.

Searching blocks using the options toolbar

Changing the block size

BORIS is able to present all system blocks in different sizes. A block which has been inserted newly is normally shown in the biggest size (100%). This default can be modified by the display dialog (this can be reached by OPTIONS | CUSTOMIZE..., edit field *Standard block size).* To modify the size of single or several blocks act in the following way:

1. Open the block size window by VIEW | BLOCK SIZE WINDOW
2. Select the blocks which should be modified.
3. Specify the desired size in the edit field *Relative size* and click at the *Apply* button.



Block size window

Connecting system blocks

How to connect two blocks

All connections between the blocks are drawn by mouse button. Because the integrated autorouter takes care that the connections are drawn rectangularly without any crossings you basically have to select the blocks which should be connected. If necessary the autorouter can be deactivated by OPTIONS | AUTOROUTER and the connections can be drawn manually. The current state will be shown graphically in the status bar.

To draw an automatical connection proceed as follows:

Drawing an automatical connection

1. At first click with the left mouse button at the output field of the block where the connection should start. The cursor will change its shape into a stilized solvering iron. An *A* beside the cursor indicates the activated autorouter.
2. Now the input field of the target block can be selected and confirmed by a click with the left mouse button. If you move into an allowed input field the cursor changes its color into black and a stilized reticule appears. During the movement of the mouse an "elastic" will follow the movement. When you reach the border of the window it will be scrolled automatically. If you want to delete a connection which you have started to draw, just click at any vacant position of the window.

After a connection was regularily terminated it will automatically be drawn with the corresponding arrow. The connections are drawn in a way that no other blocks will be touched if even possible. If this should happen nevertheless it can be corrected by moving the single blocks. Connections which originate in the same block output basically are summerized by the autorouter.

Manual connections are completely drawn by the user and can take nearly any course. As a maximum they can have eight base points. If you want to draw

manual connections, the autorouter has to be deactivated. Then you have to act in the following way:

Drawing a manual connection

1. Click at the output box of the block from which the connection should start with the left mouse button.
2. To insert base points click with the left mouse button. As a maximum a connection can have eight base points. If the option *Snap to grid* is activated the single parts of the connection will automatically be aligned rectangularly if they do not exceed a definite gradient.
3. To terminate the drawing of a manual connection you have to click at the input box of the block in which the connection should find its end.

A connection you have started with can be cancelled by a *double* click at any vacant position of the screen.

Manual connections can be modified at any time. Please act like this:

Modifying a manual connection

1. Click at the connection with the left mouse button. The base points will thereupon be marked.
2. Now each base point can be moved anywhere by holding the left mouse button. If you press the <Shift>- or <Ctrl>-key as well the moved base points will be aligned at a virtual 5-pixel-grid automatically.
3. To insert a new base point, double click at the position where the new point should be inserted.

Manual connections can be changed into automatical connections and vice versa at any time. To change a manual connection into an automatical connection you have to act in the following way:

Changing the connection type

1. Click at the block input resp. the block output box in which the connection ends with the right mouse button.
2. Choose the option MANUAL CONNECTION resp. AUTO CONNECTION in the now appearing popup menu.

If you want to modify all connections of a block at the same time proceed as follows:

1. Select the block.
2. Choose the option EDIT | CONNECTIONS | INPUT CONNECTIONS MANUAL resp. the corresponding option from the main menu.

Many connections can start at the same block output, but each block input can only get one connection. The connections can be presented in different ways which are offered by OPTIONS | CUSTOMIZE.

Deleting connections

To delete single connections

1. Click at the block input resp. the block output field in which the connection ends with the right mouse button.
2. Choose the option DELETE CONNECTION in the now appearing popup menu.

If you want to delete all connections of a block at the same time you can do this in the following way.

1. Select the block
2. Choose the option EDIT | CONNECTIONS | DELETE INPUT CONNECTIONS or EDIT | CONNECTIONS | DELETE OUTPUT CONNECTIONS of the main menu or click at the button resp. of the system block toolbar.

Giving connections colours

To ehance the clearness it sometimes may be desired to draw the connections in different colors. Therefore BORIS enables you to color each connection individually.

Basically new connections are colored in blue. This default can be modified by the display dialog which you get by OPTIONS | CUSTOMIZE....

To change the color of connections which have already been drawn act as described in the following. To change the color of a single input or output connection of a block

- click at the block input respective block output field to which the desired connection is drawn with the right mouse button and
- choose the option CONNECTION COLOUR.... in the now appearing popup menu.

To change the color of all in- respective output connections of a block at the same time

- select the block and

➢ choose the option Option EDIT | CONNECTIONS | COLOUR INPUT CONNECTIONS ...in the main menu resp. the corresponding option.

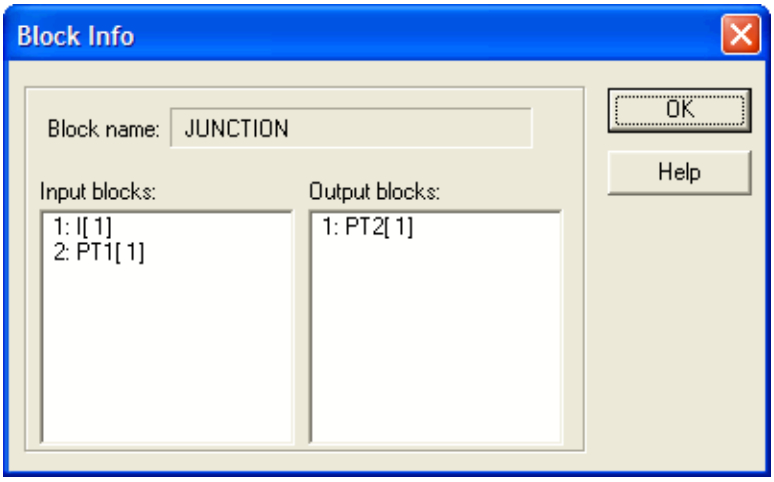
The desired color can be chosen by the corresponding Windows Color dialog.

A more comfortable way of changing colors is offered by the color toolbar at the right border of the main window:

- If no block is currently selected, a left mouse click on the toolbar changes the default color for new connections (displayed by the top color square of the toolbar).
- If one or more blocks are selected, a left mouse click changes the color of all input connections, a right mouse click the color of all output connections of the selected blocks. The default color for new connections is not changed in this case.

Displaying the block connections

With the help of the menu option EDIT | BLOCK INFO... you can keep the survey on the connections of a block in cases of very complex systems. The corresponding dialog then lists all blocks which are connected with the selected block separated into in- and outputs.



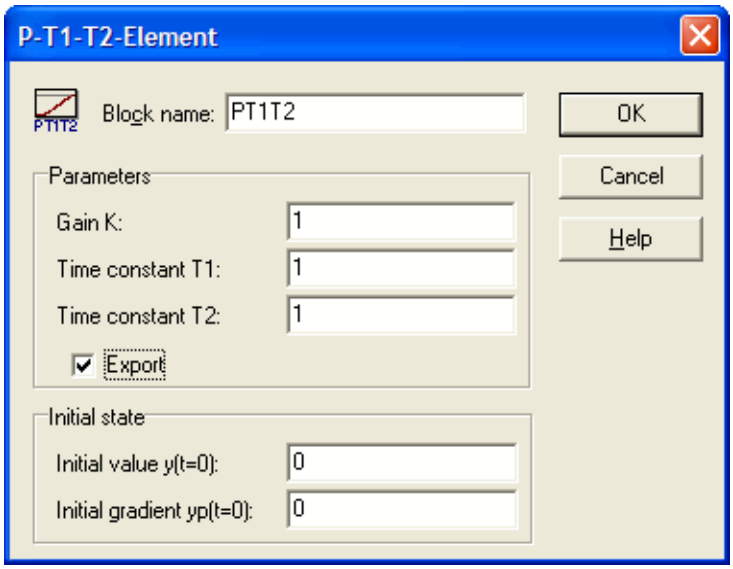
Block info dialog

Working with export parameters

A multitude of system block types feature so-called *export parameters*. These are block parameters which offer the user an *access from outside* that can be used for different purposes:

- Within the current level of the system the export parameters can be set with the help of a global dialog or can be read out of a file (see below).
- Inside of a superblock the export parameters can be used to export these from the superblock to the level above and modify them from there. Further details you find in the chapter *Working with superblocks.*
- Floating point export parameters can be optimized within the current level of the system by a numerical parameter optimization. More details about this you find in the chapter *Numerical optimization of system parameters.*
- Floating point export parameters can be modified during a simulation by PARMOD blocks (parameter modifier). Their current values can be determined via PARVAL blocks (parameter values).
- Based on floating point export parameters the so-called batch mode can be used to execute whole series of simulations automatically. Details concerning this feature are explained in the chapter *Batch mode simulation*.

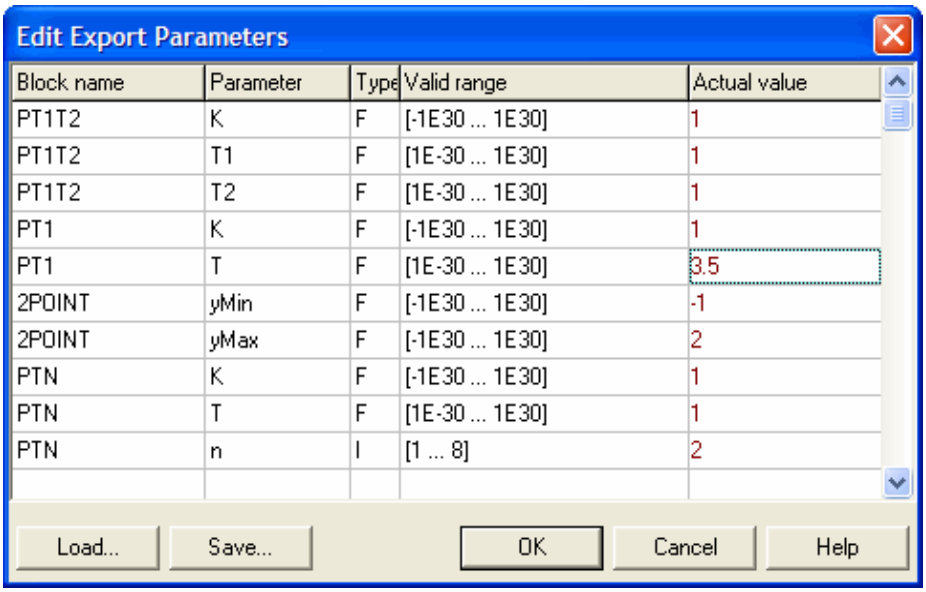
To activate an exportable block parameter as an export parameter this must be "unlocked" at first. The screenshot below shows the parameter dialog of a PT_1T_2-element. In this block type the gain K and the time constants T_1 and T_2 can be exported. For this purpose you have to activate the *Export* checkbox. You can only activate or deactivate all export parameters of a block at the same time.



Parameter dialog of a PT1T2-block with activated export parameters

If all available export parameters of the current system structure are to be activated at the same time, this can be effected by choosing EDIT | EXPORT PARAMETERS | ACTIVATE ALL. Via the menu option EDIT | EXPORT PARAMETERS | DEACTIVATE ALL all parameters can be deactivated again. Via the menu option EDIT | EXPORT PARAMETERS | EDIT... all activated export parameters of the current system structure can be modified.

The assignment of the export parameters results from the *block name*; therefore all the blocks which export parameters should have different block names to avoid errors in assignment. With the menu option FILE | CHECK FOR DUPLICATE BLOCK NAMES... the system can be controlled if necessary. In the *Type* column of the dialog the parameter type (*F* for floating point, *I* for Integer and *S* for String) is displayed. Alternatively the parameters can be read from file by the button *Load*.... This file must contain one parameter value in each row. This means that in the example below the first row must contain the parameter value of the block *PT1T2,* parameter *K,* in the second row the value of the block *PT1T2*, parameter *T1* and so on.



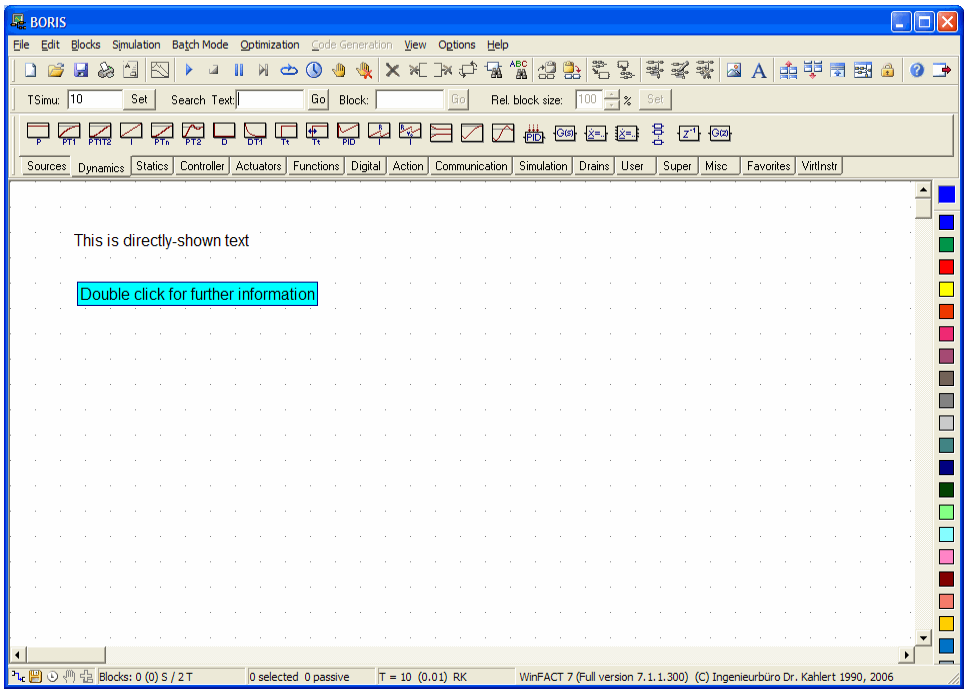
Dialog for modifying export parameters

Text blocks, bitmaps and frames

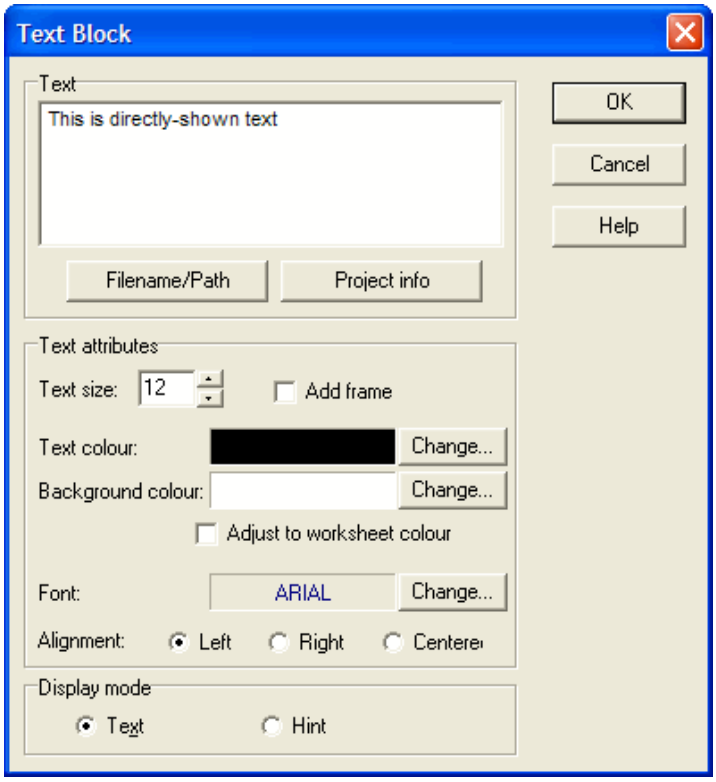
Working with text blocks

Besides the system blocks BORIS enables you to insert any comment text in the structure. You can choose different colors and fonts for the text and as well as the system blocks it can be moved and deleted. Alternatively the text itself can be shown on the working sheet or only a hint to the text on which you have to doubleclick in order to show the text completely. This is especially useful if you want to integrate long information texts into a system structure.

To insert a new text block choose the menu option EDIT | INSERT TEXT BLOCK resp. the button. The cursor thereupon changes its shape and the default text *Text* can be placed on the working sheet by drag & drop. It can be modified later on by a doubleclick.



Directly shown text (top) and hint to a hidden text (bottom)



Dialog to modify text blocks

As well as "normal" system blocks text blocks can be moved by keeping the left mouse button pressed. The text resp. background color may also be

changed more comfortable via the color toolbar at the right border of the main window: A left mouse click changes the text color of the currently selected text block, a right mouse click the background color.

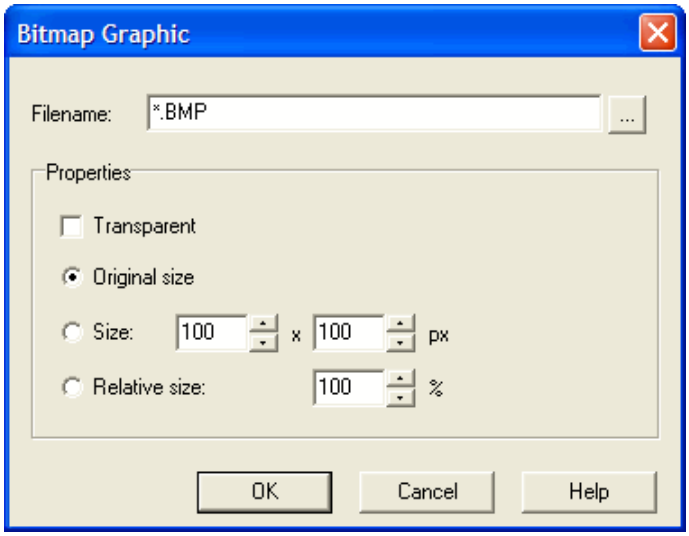
Searching text

Text blocks can also be used to find specific substructures within the current simulation structure by marking them with a describing text. For this purpose the menu option EDIT | SEARCH TEXT... (whereupon all existing text blocks of the current system are listed) or the options toolbar (see screenshot below) may be used. For that the text to be searched or its first characters (*abc* below) has to be entered into the corresponding edit field. Clicking the *Go* button starts the search. If a proper text block is found it appears in the upper left corner of the worksheet, if not, an error message is displayed.

Searching text blocks via the options toolbar

Inserting bitmaps

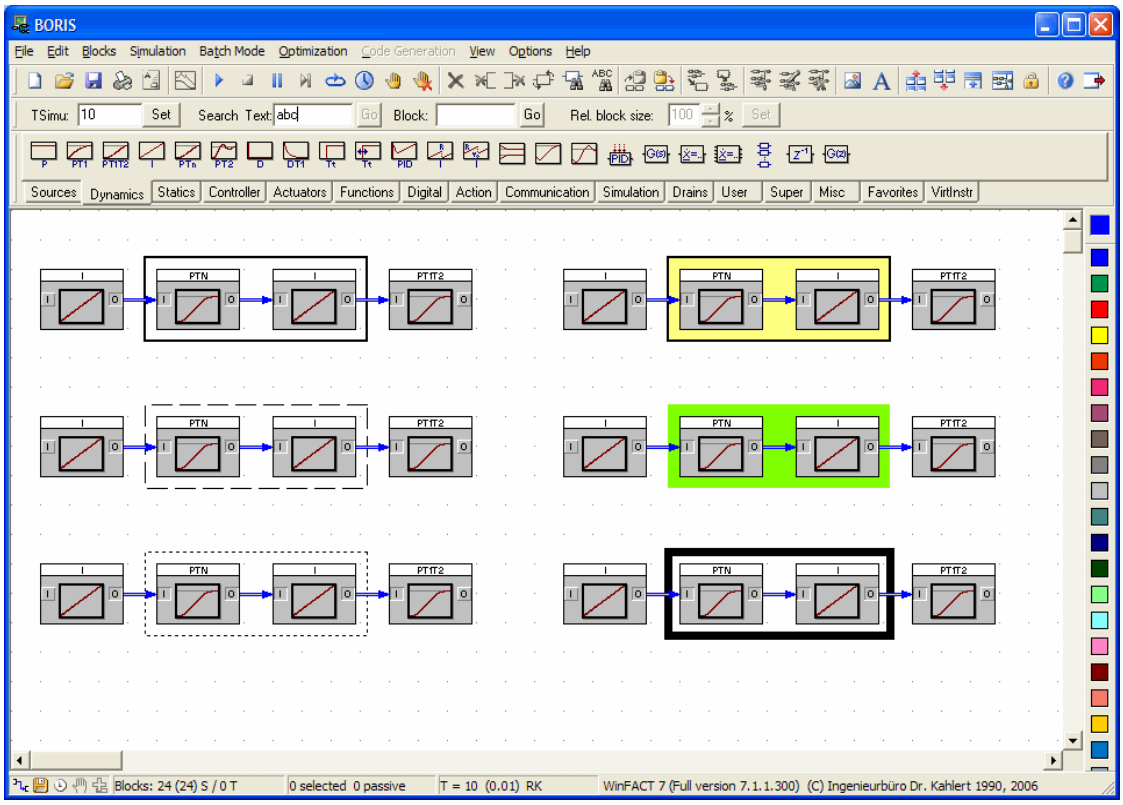
Besides text information also bitmap graphics can be used within the system structure via the EDIT | INSERT BITMAP menu option or the button. The inserted bitmap can be displayed transparent; also its size can be changed in various ways (see screenshot below). Please note that standard system blocks and text blocks always appear *above* a bitmap graphic whereas connections are drawn *below* bitmap graphics.



Dialog for modification of bitmap graphics

Working with group frames

An additional feature for a better documentation of system structures is the frame function of BORIS. This enables you to group blocks which belong together optically by an enclosing frame. Of course this function has no influence to the simulation.



Different types of group frames

To insert a frame,...

- at first select the blocks which are to be framed,
- then choose the menu option EDIT | GROUP FRAME | INSERT FRAME resp. click at the button of the system toolbar

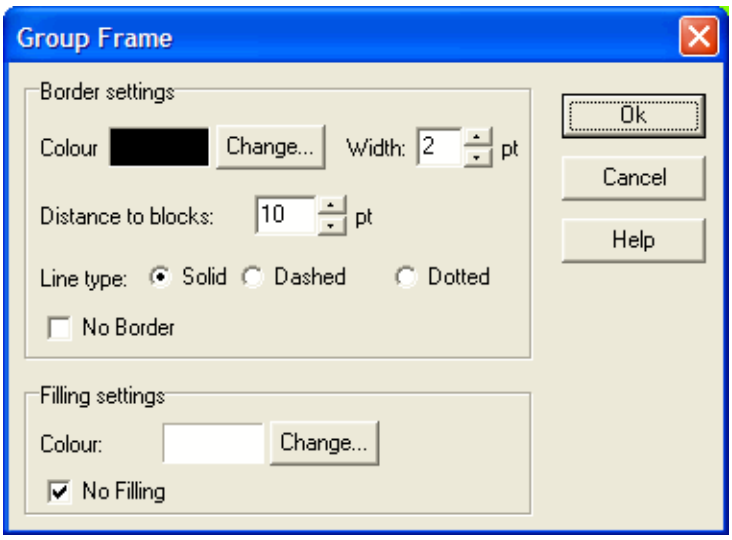
Note: Frames are *static* structures which have no direct relationship to the framed blocks. Therefore they can´t either be moved or deleted automatically when deleting the block itself!

Frame types

The inserted frame basically has a distance of eight pixels to the block enclosing rectangle, a black margin with a filled line and a width of one pixel and no filling. These parameters can be modified by the user in the following way:

1. Select any block inside the frame.
2. Choose the menu option EDIT | GROUP FRAME | EDIT FRAME... or click at the button of the system toolbar

The following dialog box will appear which allows you to modify the parameters.



Group frame dialog

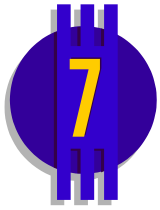
How to delete frames

If you want to delete a frame you have to act in the following way:

1. Select any block inside the frame.
2. Choose the menu option EDIT | GROUP FRAME | DELETE FRAME or click at the button of the system toolbar

Structure overview

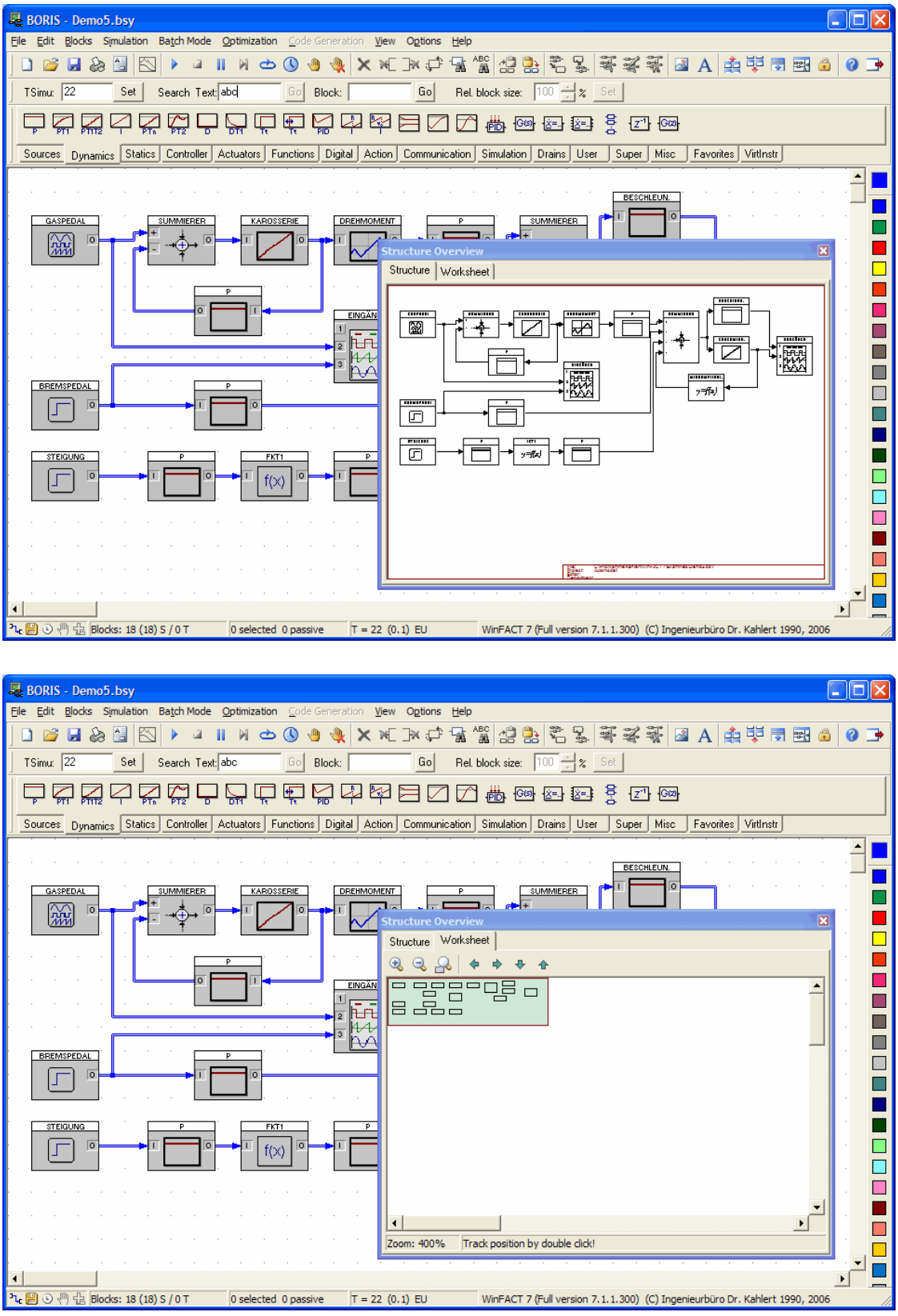
If you have more complex systems basically the visible part of the window is too small to show the complete system. To survey the complete structure it is possible to open an additional window which shows the complete structure zoomed to the window size. This window you can open with



VIEW | STRUCTURE OVERVIEW or with a click at the  button of the system toolbar. The size of the window can be enlarged or reduced optionally and will be updated automatically as long as it is visible. The structure overview window has two palettes titled *Structure* and *Worksheet* which offer different views of the system structure (see screenshots below).

- In the *Structure* mode the complete system structure including all connections is shown with the system blocks represented by b/w-bitmaps; this view corresponds to that one being used when the system structure is printed.
- In the *Worksheet* mode a zoomable and moveable section of the complete structure is shown; the current zoom factor is displayed in the status bar of the window. If the mouse is located over a block, block type, block name and block index are displayed in a small hint window. In this view no connections and block bitmaps are displayed.

In both views a double click with the left mouse button within the window moves the current structure section in that way that the specified position is located exactly in the center of the new section. Thus a fast change to specific positions within the system structure is practicable.



Strcuture overview for a simple system in both views

File operations

File types

All BORIS system files are ASCII-files which can be viewed at or modified (only exceptionally!) with a normal text editor. You have to distinguish between *regular system files* (extension BSY) and superblock files (extension SBL). The superblock files are special modifications of system files and contain a file header with special information about the structure of the superblock. Regular system files can be changed into superblock files and vice versa. In one of the following chapters you will get detailed information about the work with superblocks and their files. In this chapter we only take a look at regular system files.

Opening a file

You can open an already existing system or superblock file in this way:

- Choose the menu option FILE | OPEN... resp. click at the  button of the system toolbar and specify the filename.
- The last five files you have worked with will be shown at the bottom of the FILE submenu from where you can call them directly.

The filename of the current file is shown in the caption of the BORIS main window if it has once been saved before.

To save a file...

- ... choose the FILE | SAVE menu option or the  button of the system toolbar. If you want to save the file under a new filename or file type (e. g. a superblock file as a regular system file or vice versa) use the FILE | SAVE AS... menu option.

Creating a new file

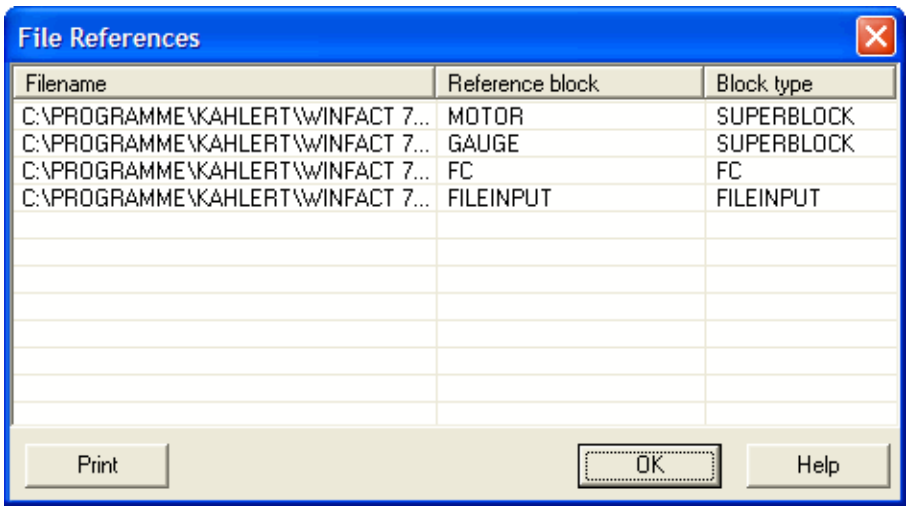
With the menu option FILE | NEW or a click at the button BORIS will be reset to the status the program had when it was started. If the current file has been modified since the last saving a confirmation dialog will appear.

Adding files

You can add further files to the currently loaded file by choosing the menu option FILE | ADD SYSTEM FILE... and inserting the name of the system file which should be added. The additionally loaded system structure will be inserted below the current system. The adding of files is limited to blocks, connections, texts and frames. By adding of the files all simulation parameters, digital levels etc. are kept unchanged.

File references

A lot of system blocks (e. g. file inputs, fuzzy controllers, superblocks ...) need specific files for the simulation. For example if a simulation structure should be passed on it is not sufficient to pass on the corresponding system file but all the files which are referenced have to be copied as well. The menu option FILE | FILE REFERENCES... gives support by listing a dialog with all referenced files. In this dialog you can see the name of the connected file as well as name and type of the corresponding system block.



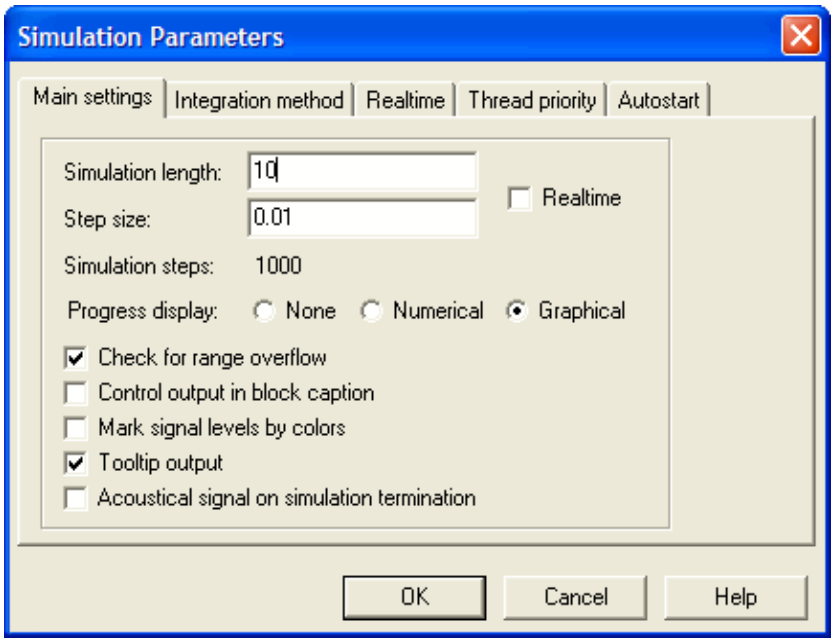
File references dialog

Controlling the simulation

Simulation parameters

Before the simulation can be started basically the simulation parameters have to be chosen. The dialog for the choice of the simulation parameters can be called up by the menu option SIMULATION | SIMULATION PARAMETERS... or a click at the button. It is separated into three sections.

Main settings



Main settings *palette*

The function of the options of the palette *Main settings* is as following:

Option	Meaning
Simulation length	The simulation length T_{Simu} specifies up to which time the simulation is executed if it is not interrupted by the

user. Default is a value of 10.

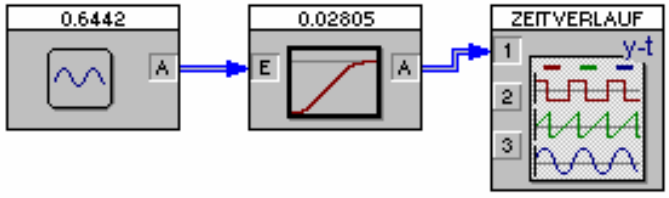
Step size — ΔT specifies the simulation step size and therefore influences the precision of the simulation results. If ΔT is too large, you will get discretisation errors which produce falsification of the results by numerical instability. A clue for the right choice is that the step size should not exceed 1/10 of the lowest time constant of the system (see also: *Integration methods*). Default is a value of 0.01.

Realtime — If this option is marked the simulation is executed in real time, i. e. the simulation step size which is set by *step size* corresponds to the real time between two simulation steps.

Progress display — This option determines the display of the simulation progress. In the setting *Graphical* the display shows a running bar in the status bar, in the setting *Numerical* it shows the actually passed time. For time critical simulations the setting *None* should be chosen.

Check for range overflow — If this option is activated all state variables of the system are checked with every step of the simulation. In the case that one of the variables overflows the range of the value 10^{20} the simulation will be stopped by giving a corresponding message. Because this check for range overflow needs some calculation time the simulation with an active check (which is default) elapses a little bit slower.

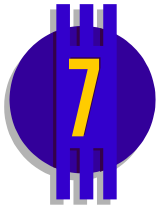
Control output in block caption — This option is helpful if you search for errors inside the system structure. If it is activated, during the simulation the current output variable of the block is shown in the block caption of each block (with the exception of superblocks). For blocks with several outputs only the first output can be shown.



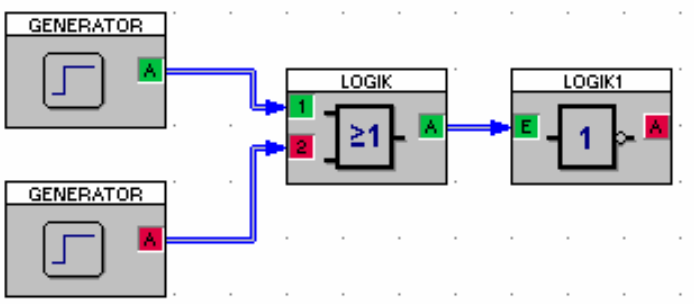
Control output in block caption

Mark signal levels — This option is recommended when analyzing systems

by colours



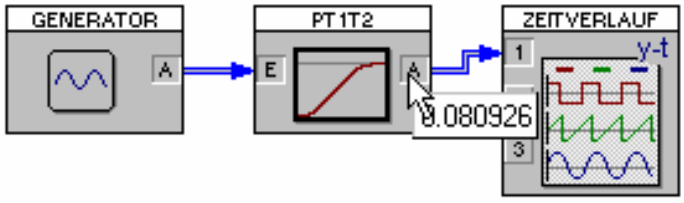
with a lot of digital blocks. If it is activated, all block outputs as well as all connected block inputs are marked by a colour depending on the current signal level of the input resp. output: If the in-/output has logical LOW level, it is displayed red, if it has HIGH level, it is displayed green. Block inputs which are not connected stay uncoloured.



Coloured block inputs and outputs

Tooltip output

If this option is activated a hint window appears while moving the mouse cursor over the block in- or outputs (*tooltip window*). It shows the corresponding signal value.

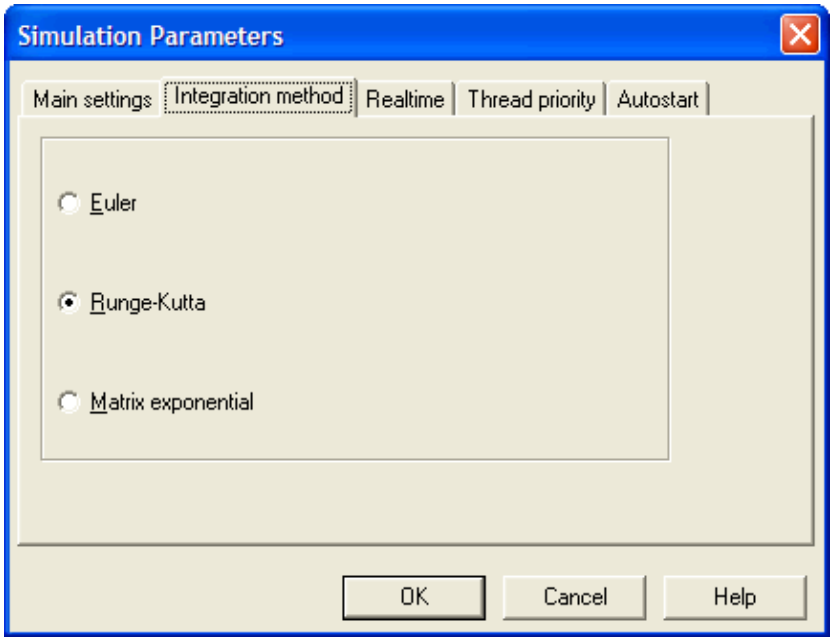


Tooltip output

Acoustical signal on simulation termination

This option creates a beep on the end or an interruption of the simulation.

Integration methods



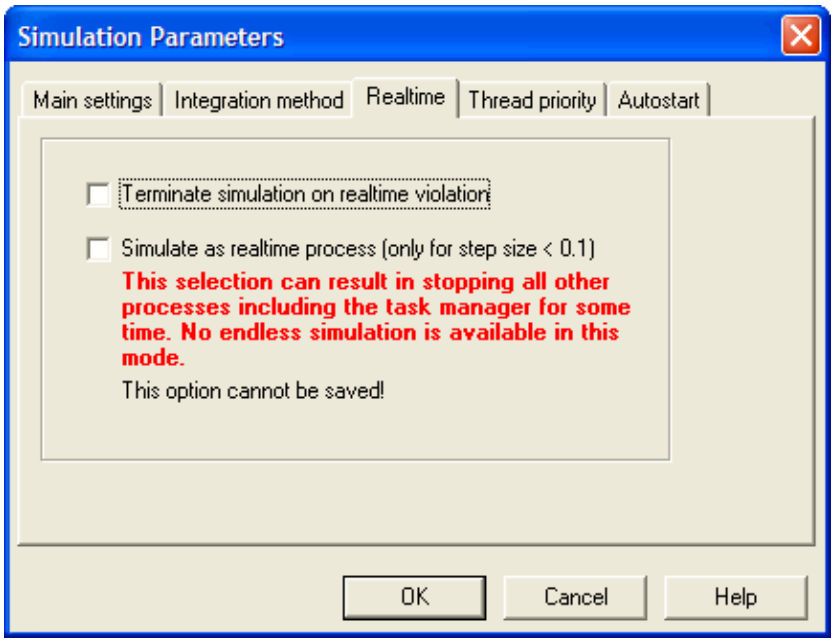
Integration methods *palette*

This palette allows the choice of the numerical integration method for the simulation of the dynamic system components. In the current version of BORIS the explicit Euler-method, the Runge-Kutta-method of the fourth order and the Matrix exponential method are available [3].

- The Euler-method is the easiest of all integration methods and needs less time because only one function value has to be calculated for each simulation step. This method is only qualified for the simulation of simple systems combined with small simulation step sizes because it only has the error order $O(\Delta T^2)$. Especially for strongly oscillating systems or systems with very different time constants ("stiff" systems) this method should not be used.
- In contrast to this the Runge-Kutta-method has much better numerical qualities; it has the local error order $O(\Delta T^5)$. Because it has to execute four function evaluations for each simulation step the calculation time is much longer than it is in using the Euler-method. Nevertheless the use of this method can produce inaccuracies in the transition area if you want to change input signals with steps (e. g. calculation of pulse responses). Therefore in these cases it is better to use the Euler-method in combination with a smaller step size.

- The matrix-exponential-method is only suitable for linear systems where it has principally an infinite high error order. It should especially be used if e. g. transfer functions of a higher order (> 3) are used as system blocks. If you have chosen this method the nonlinear dynamic system blocks (nonlinear Differential equation systems) will be simulated with the Runge-Kutta-method.

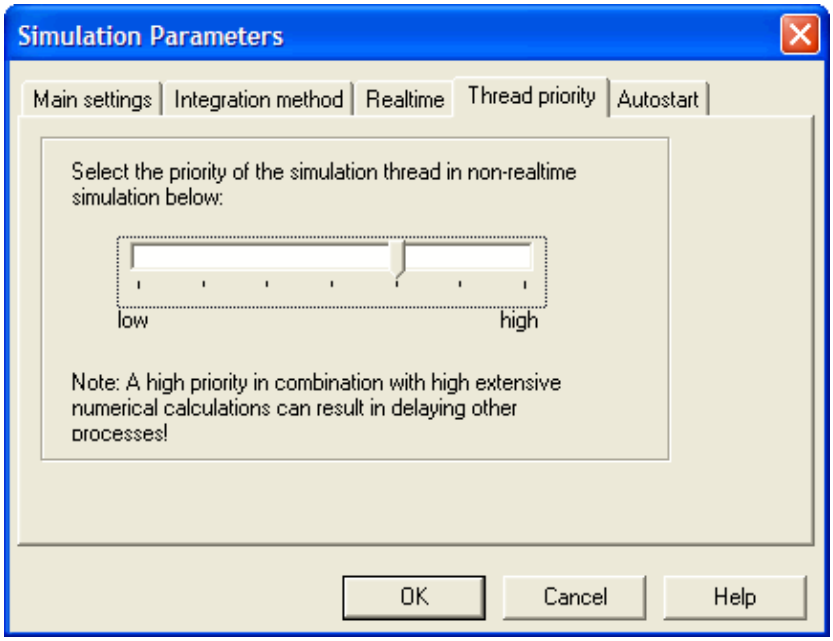
Realtime simulation



Realtime *palette*

The option *Terminate simulation on real time violation* determines the behaviour of BORIS during the real time simulation in the case that the desired simulation step size cannot be kept for any reason (e. g. because of a very complex system structure with many calculation intensive blocks). If the option is activated BORIS cancels the simulation displaying a corresponding message.

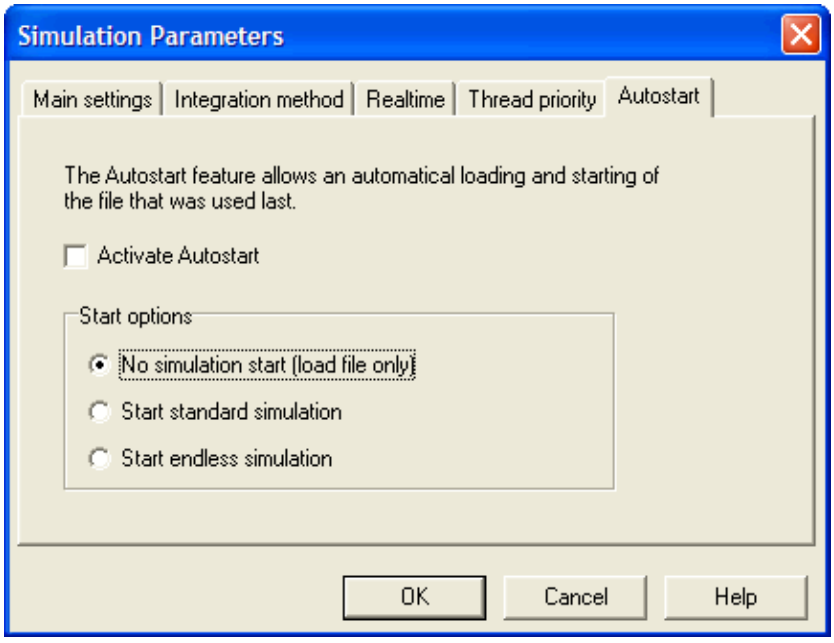
Thread priority



Thread priority *palette*

BORIS uses different threads for the simulation itself and for the graphical output (e. g. of a digital display). If non-realtime simulation is executed, the priority of the simulation thread can be selected via this palette. The higher this priority is selected, the faster is the simulation running, but the lower is the refreshment rate for all output functions (that includes graphical outputs as well as the refreshment of the simulation progress display in the status bar of BORIS). So to attain a continous refreshment rate it is recommended to decrease the priority of the simulation thread, to attain the maximum simulation speed it is recommended to increase it.

Autostart



Autostart *palette*

The autostart feature allows an automatical loading (and if need be, starting) of the last opened file after starting BORIS. Within the *Start options* group box the corresponding options can be selected. If BORIS is started with command line parameters, the autostart function is without effect.

Simulation mode control

Expediently the simulation will be controlled by the buttons of the system toolbar; if required the control can also be taken by using the options of the submenu SIMULATION. The following table gives a survey of these options.

Button	Menu option SIMULATION	Function
	START	Starts standard simulation
	STOP	Terminates simulation
	BREAK | SINGLE STEP MODE	Activates/Deactivates the single step mode

	PERFORM SINGLE STEP	Performs a single step
	START ENDLESS SIMULATION	Starts endless simulation
	SET BREAKPOINT...	Sets a breakpoint
	DELETE BREAKPOINT	Deletes a breakpoint

Starting the standard simulation

In the standard mode the simulation is started with the menu option SIMULATION | START resp the system toolbar. At first BORIS checks whether an algbraic loop exists or one of the system blocks has been incorrectly parameterized. If needed a corresponding message will appear.

For the simulation the inputs of the system blocks can stay open, basically they are set to zero resp. in special cases they are set to other defaults. Therefore it is possible e. g. for special simulation tasks (e. g. simulation of a free system with default start values) to start the simulation without using a generator with the output value 0 additionally.

If the check has been negative the simulation will be started. If this option is activated the simulation time which has already been passed is recorded in the status bar of the main window during the simulation.

In the standard mode the simulation will be executed up to the preset simulation time if no breakpoint is reached or the simulation is cancelled by the user before.

Endless simulation

In contrast to the standard simulation the simulation time will be ignored during an endless simulation. Therefore the simulation can only be cancelled by the user (e. g. if needed by a SIMCANCEL-block).

Single step simulation

The user can turn to the so-called *single step mode* at any time during or before starting the simulation. If the single step mode is activated the simulation will be interrupted (pause) and each simulation step can be executed singly until leaving the single step mode again. Therefore this mode is especially suitable for a detailed investigation of special time ranges.

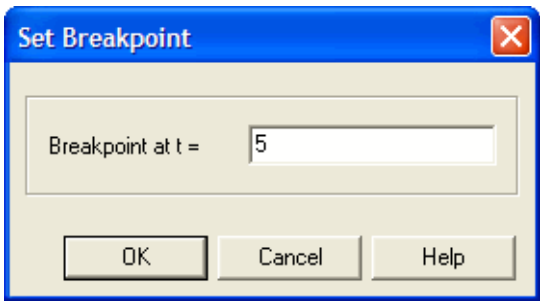
The single step mode can also be activated by setting a breakpoint or using a SIMCANCEL-block.

Terminating the simulation

The simulation can be terminated by the user at any time before its regular end by using the menu option SIMULATION | STOP. If the single step mode is activated the simulation will stay in this mode even after being cancelled.

Using breakpoints

By setting so-called *breakpoints* the simulation can be switched into the single step mode automatically at any determined time.



Setting a breakpoint (for t = 5 here)

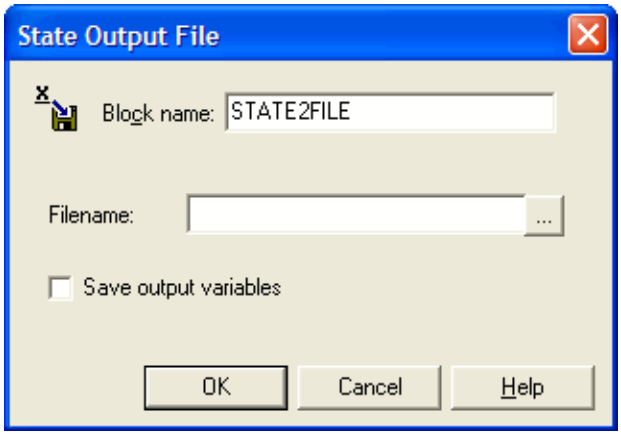
If the simulation reaches the specified time, it will automatically be switched into the single step mode. With the help of this function a default time can exactly be reached without interrupting the simulation "manually" (which basically means unexactly) at this point.

Transferring the system state to initial values

The option SIMULATION | STATE TO INITIAL VALUES enables you to take over the current state of all dynamic system blocks to their initial values for example to continue a new simulation later on from this state. In doing so the current state variables of the blocks are copied into the initial values. However in the current version of BORIS this option is only available for system blocks of the highest hierarchical level (i. e. the currently loaded system file), the initial values of dynamic superblocks cannot be set in this way. Therefore if necessary the corresponding superblocks have to be splitted before the start of the simulation.

Saving and loading system states

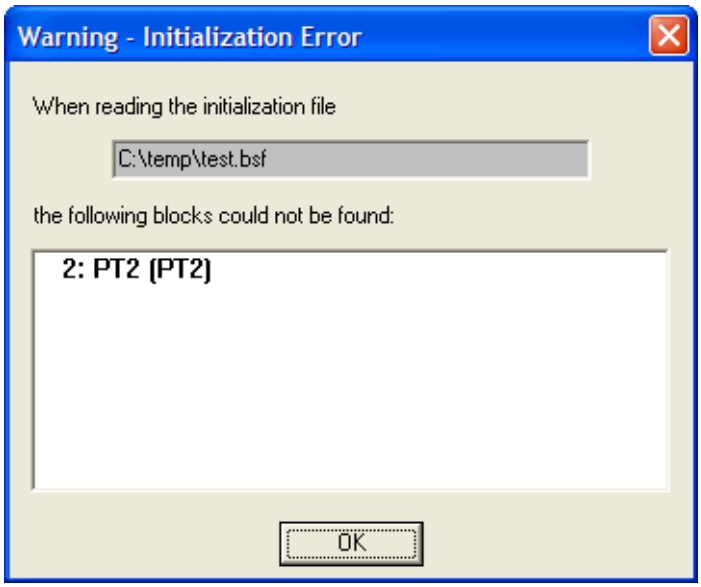
If the state of a complex structure (e. g. with a lot of superblock levels) should be saved, the STATE2FILE block type (state output file) can be used. This block has no in- or outputs but effects that the state variables of all system blocks (also those within superblocks) are saved in a user-defined BSF file (*BORIS State File*) after the simulation has terminated. In addition to that the current output values of the blocks can be saved if desired.



Parameter dialog of the STATE2FILE *block*

A BSF file created in this way can later be used for the initialization of a simulation by using a *FILE2STATE* block. In this case the initial values specified in the block parameter dialog are overwritten by the corresponding values found in the BSF file.

The identification of each system block within the BSF file is realized by comparing block index and block type. So a BSF file created by a STATE2FILE block can later be used for the simulation initialization by a FILE2STATE block only if the system structures are identical when creating resp. loading the BSF file. If BORIS detects inconsistencies when reading a BSF file, the user is informed by a corresponding warning dialog (see following screenshot).

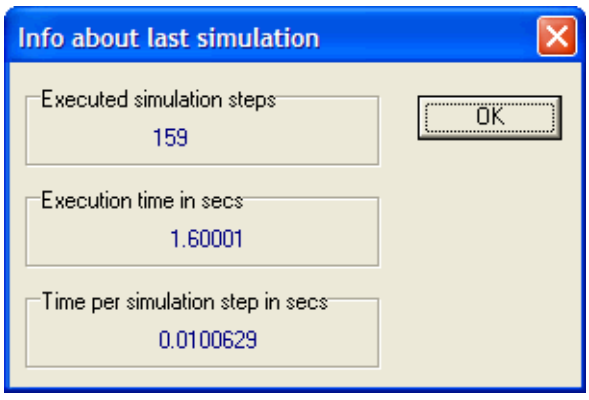


Warning caused by inconsistencies between system strcuture and BSF-file. Here a block of PT2 type was added after the BSF-file was saved.

Therefore the STATE2FILE block for creating the BSF file always should be added to the system structure as the *last* block; only in such way it can be guaranteed that a later deletion of this block does not change the order (and therewith the block index) of the other blocks. Alternatively the STATE2FILE block can be set passive instead of deleting it.

Information about last simulation run

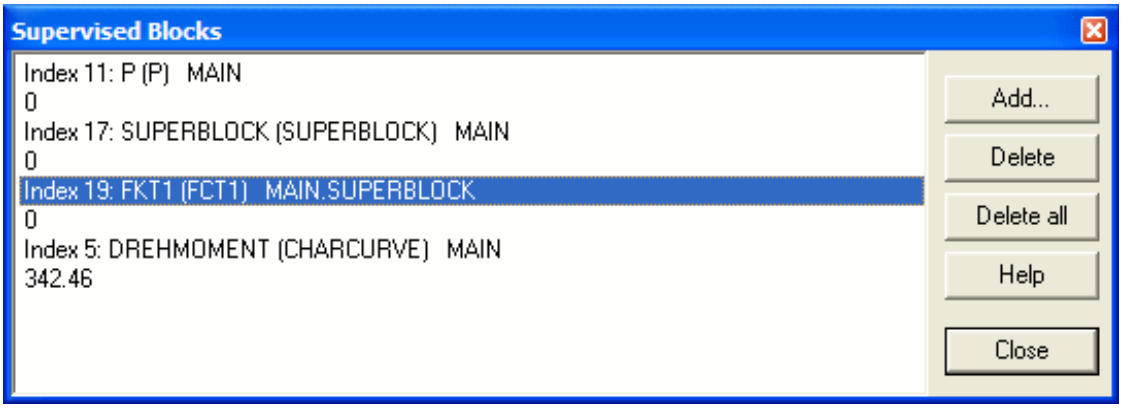
Via the option SIMULATION | INFO ABOUT LAST RUN... you can get information about a preceding simulation. This information is especially interesting in context with real time simulation because it give hints for the smallest valid simulation step size.



Info dialog for latest simulation run

Supervising blocks

The option VIEW | SUPERVISED BLOCKS lets BORIS show you a window in which all output signals of any blocks are listed during the simulation. The advantage of this window is that you can supervise blocks inside of superblocks in an easy way.



Supervising blocks

Each block is presented in two rows. The upper row contains the block spezification, consisting of block index, block name, block title and block level, the lower one contains all current output signals of the block, separated by commata. Using the button *Add* you will get a dialog which enables you to choose the block which should be supervised. With the button *Delete* resp. *Delete all* you can delete all or single items.

Remark: Supervised blocks are identified within BORIS via their block index. The consequence of this fact is that after deleting blocks from the system strcuture some entries of the supervised block list may become invalid. Those entries are deleted automatically when the next simulation is started.

Online parameter modifications

Basically you are enabled to modify the parameters of the system blocks during the simulation (with the exception of system blocks which are parameterized by files as e. g. *File Input*). For that you have to select the desired system block by a doubleclick and then modify it afterwards. After quitting the parameter dialog all modifications become efficient and the simulation will continue. Modifications of the system structure (e. g. deletion of blocks or connections) are not possible during the simulation.

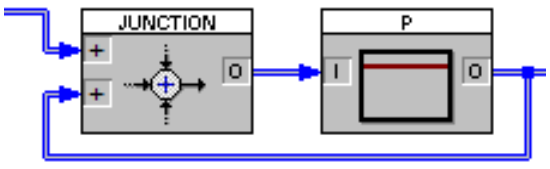
Dissolving algebraic loops

After you have started the simulation at first BORIS sorts all system blocks internally in a sequence which is valid for the simulation. This is necessary to get reproducible correct results, independent of the sequence in which the blocks have been inserted.

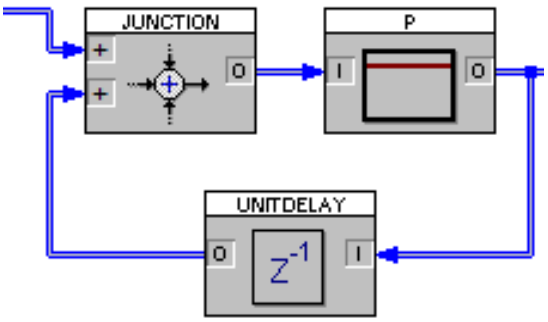
However this sorting fails if there is a so-called *algebraic loop* inside the system structure. This is a closed structure which means a structure with a feedback (loop). Within this loop you only find system blocks without any delay, i. e. system blocks in which a modification of the input is effective at the output within the same simulation step. To enable the sorting BORIS therefore always needs at least one block with a delay within a loop. Among others the following block types are blocks with a delay:

- PT_1, PT_2, PT_1T_2, PT_n - elements
- Integrators and dead time elements
- All-pass-elements
- Transfer functions with numerator order < denominator order
- Unit delays

If you have an algebraic loop within a structure BORIS gives a corresponding warning. In this case you should control the system at first because algebraic loops basically give a hint at modelling errors (in reality algebraic loops do not exist!). If you nevertheless want to simulate the structure you can eliminate the loop by inserting one of the blocks with delay into the loop. Most qualified is a unit delay which delays the input signal exactly one step or a PT_1-element with a small time constant. The following graphics elucidate the principle:



Example for an algebraic loop...



...and how to dissolve it with a unit delay

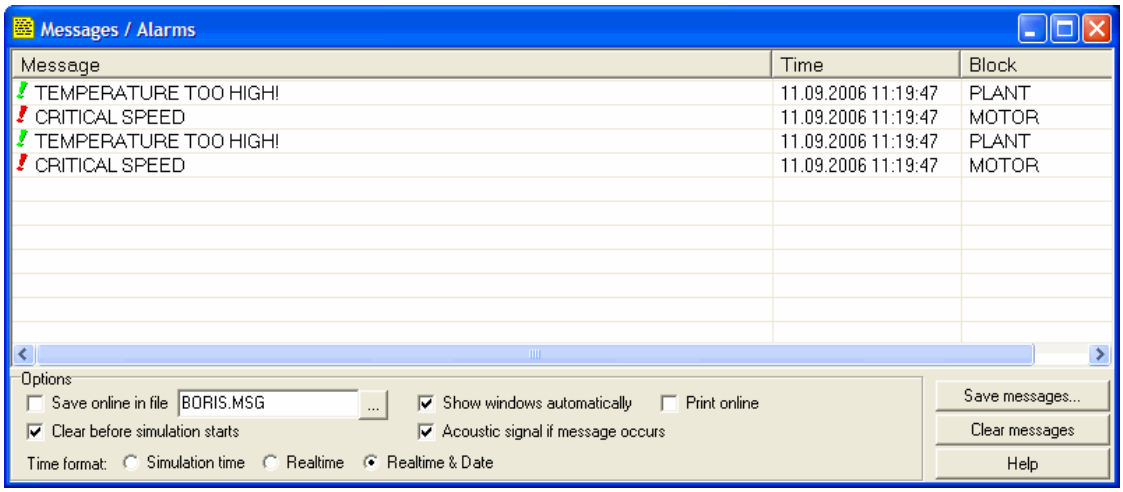
What else you should know

Administration of alarms and messages

BORIS has a powerful administration of warnings and messages which enables you to record and display warnings/messages centralized during the simulation. For that purpose BORIS offers a window which can be called at any time using the menu option VIEW | MESSAGES/ALARMS. For the generation of messages you simply have to insert a message block at the corresponding position. Each message will be displayed in clear text with the name of the corresponding message block as well as the simulation time or real time/date and can be represented acoustically by a WAV-file. In addition to the display function the window offers the following options:

- With the activation of the option *Save online in file* all warnings/messages can be saved automatically online in the file specified in the corresponding edit field.
- Before you start the simulation the messages in the display can be deleted automatically (option *Clear before simulation starts*).

- The window can be displayed automatically at the start of the simulation (option *Show window automatically*; this option is only effective if the system contains at least one message block!)
- At the arrival of a message a beep can be produced (option *Acoustic signal if message occurs*).
- All messages in the display can be deleted manually and saved in any ASCII-file (button *Clear messages* resp. *Save messages.....*).

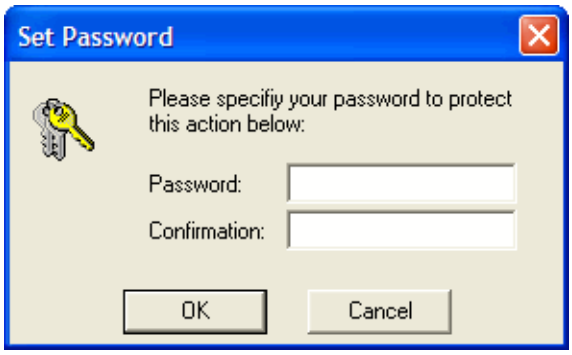


Messages and alarms window

The priority of the message is marked by the colour of the exclamation mark in front of the message text. "Yellow" shows messages of low priority, "green" messages of middle priority and "red" messages of high priority. More details you can find in the description of the message block in the chapter *The BORIS system block library.*

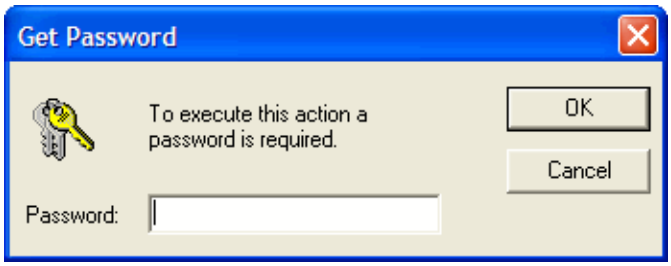
Locking the main window

To avoid unauthorized access – e. g. during a long running simulation or measurement – the main window of BORIS can be locked using the menu option VIEW | LOCK... or the button. Afterwards BORIS will ask for a password which allows the unlocking of the main window later on.



Setting the password for locking the main window

To avoid typing errors the password has to be confirmed in a second edit field. Subsequently the main window will be minimized and can only be normalized by entering the password.



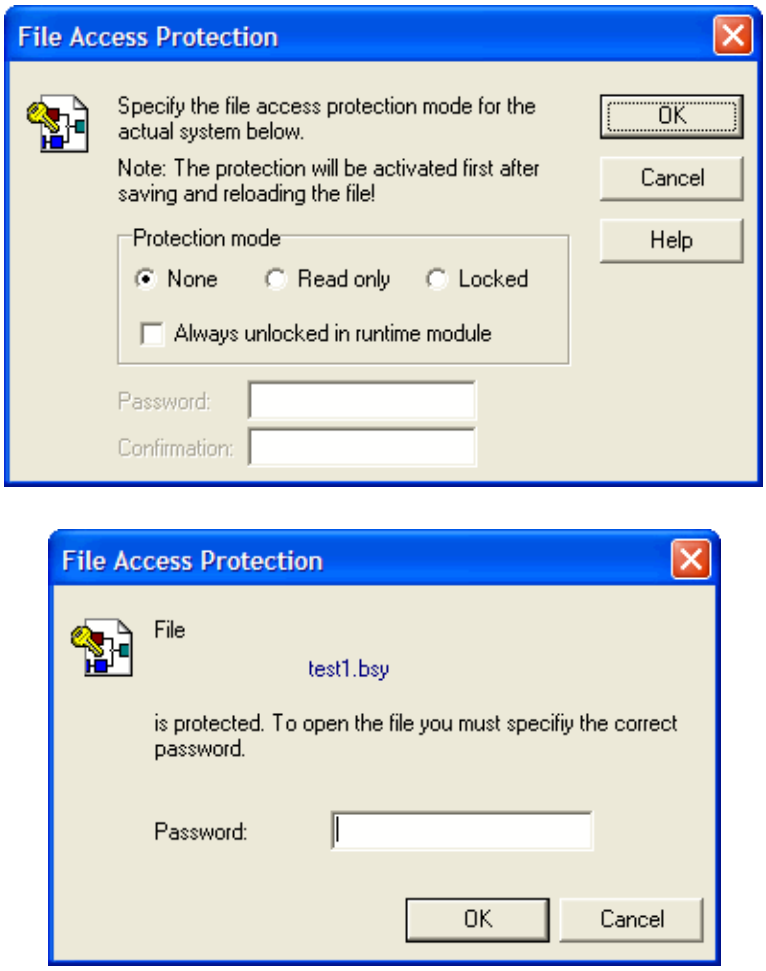
Entering the password to unlock the main window

Access protection for system files

BORIS system and superblock files can be supplied with an access protection which avoids unauthorized modification or loading of a file by using the option FILE | FILE ACCESS PROTECTION... (see screenshot below).

None	No restriction of file access
Readonly	A modification of the system is only allowed after entering the correct password. Without a password or with a wrong password the system can only be read and simulated.
Locked	Loading the system is only allowed after entering the correct password.

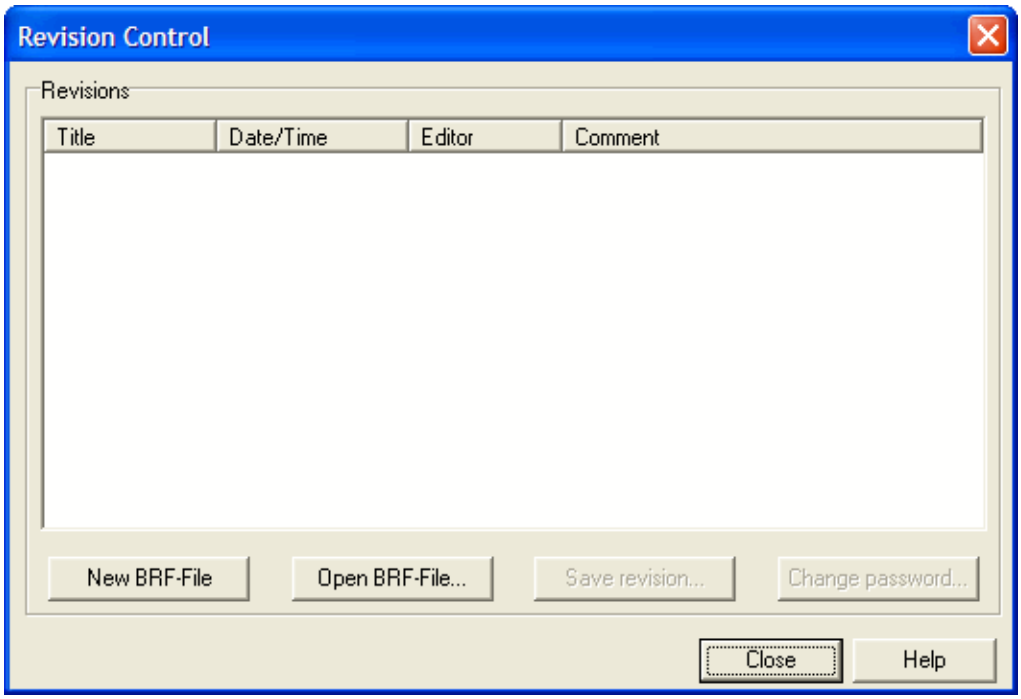
To avoid typing errors the password has to be confirmed in the lower edit field. Please consider that the access protection will be activated not until you load the system again after you have saved it.



Specifying the access protection for system resp. superblock files (top) and password request before loading a file with access protection (bottom)

The revision management system of BORIS

For complex problems BORIS includes a powerful and comfortable revision management system which allows the fusion of different development stages of a project in a single revision file and a later extraction of each version if necessary. The revsion management system is started via the FILE | REVISION CONTROL... menu option.



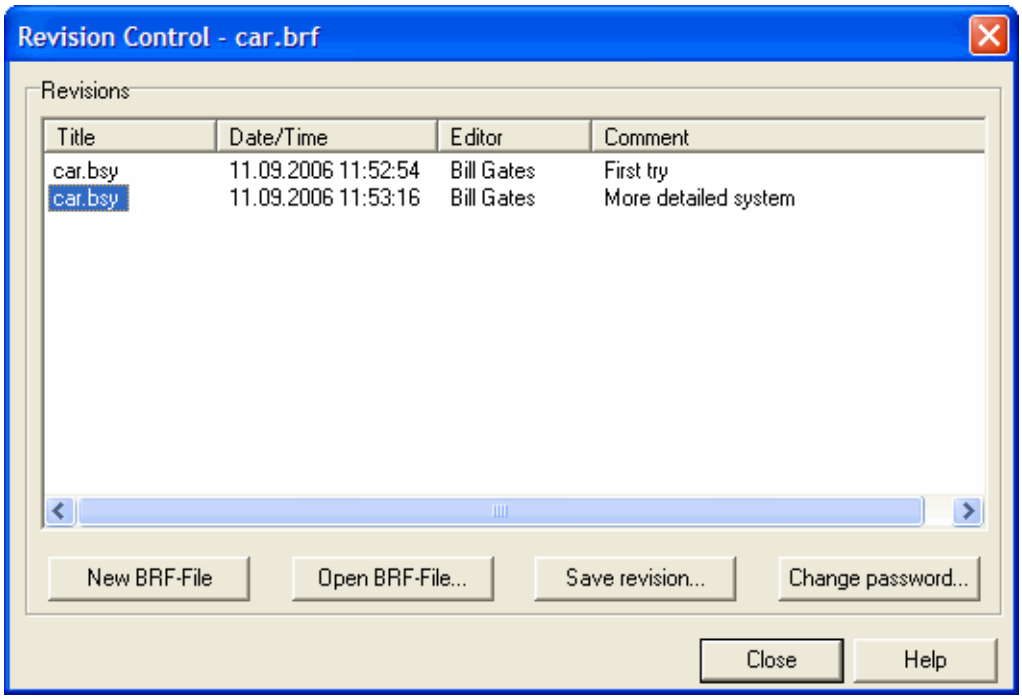
Revision control dialog

Base of the revision management system are *BORIS revision files* (File extension BRF). Such a revision file contains any number of system or superblock files in an encrypted and password protected format. The files contained in the revision file normally represent different stages of a project which can be extracted from the BRF file if desired. Each included file contains information about the file date, the file editor and a comment for a better file specification.

If a current project is to be managed via the revision management system, first a new BRF file has to be created via the *New BRF file*... button. The *Open BRF file*... button loads an existing revision file. If a revision file has been opened once it stays open even after closing the dialog so that it is displayed automatically if the dialog is called for the next time.

If a new revision file is created automatically a password is required which has to be used for the file protection. If no file protection is desired, an empty string can be used for the password. The *Change password*... button may be used to change the password at any time.

After creating a new or loading an existing revision file the current BORIS system structure can be included in the revision file by using the *Save revision*... button. The following screenshot shows the dialog after including two versions of the *car.bsy* file.

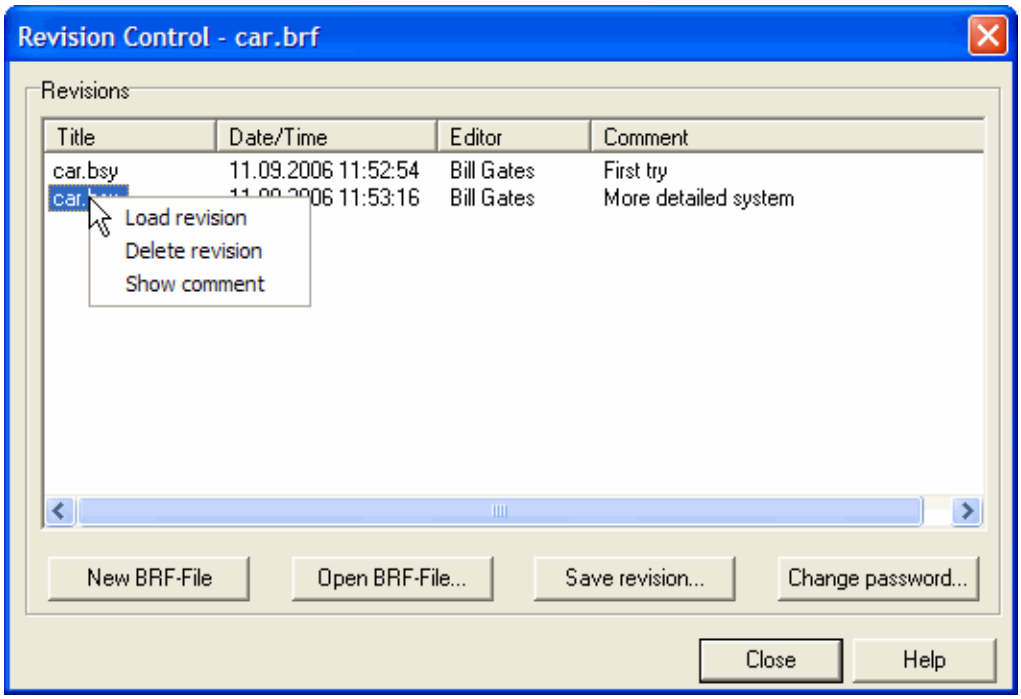


Dialog with two versions of the file car.bsy

By a right mouse click on a file name (left column of the list) a context menu appears which contains further options:

- Loading a revision from the revision file to BORIS
- Deleting a revision from the revision file
- Showing the complete comment of a revision

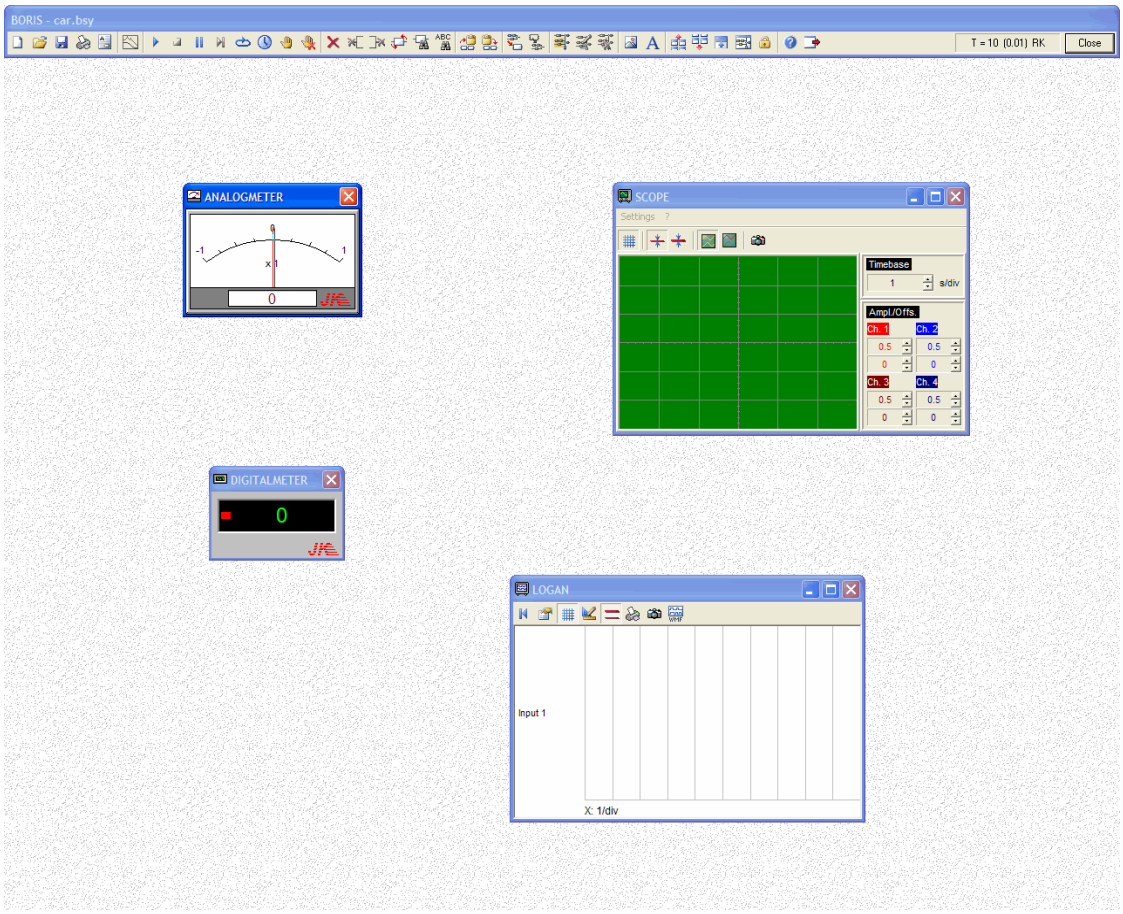
An alternative way to load a revision is a double click on the filename, an alternative way to delete a revision the usage of the <Del> key.



Revision control context menu

Changing the display mode

For some applications of BORIS it can be helpful to hide the main window of the system structure and show only the action resp. display blocks on the screen. For this purpose BORIS offers an alternative display mode which can be activated with the menu option VIEW | CHANGE DISPLAY MODE. The main window of BORIS will be turned into a modified system toolbar and supplied with a neutral background. With the button *Close* you can turn back to the standard mode.

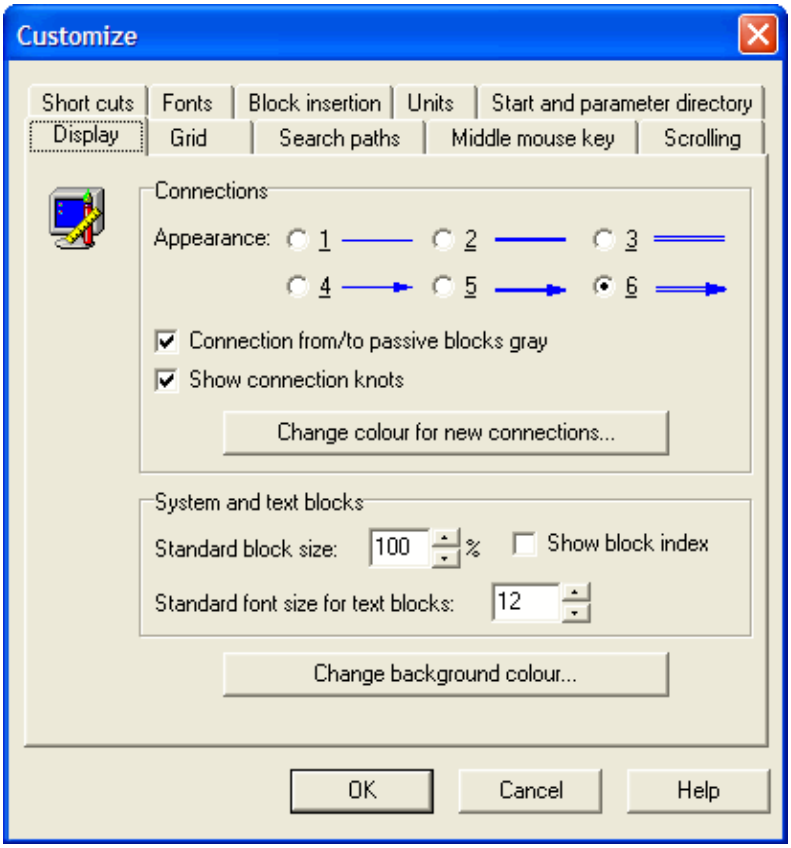


Alternative display mode

User-defined settings

Via the menu option OPTIONS | CUSTOMIZE... BORIS allows the default setting of a lot of user-defined options which are separated into different groups (dialog palettes).

Display

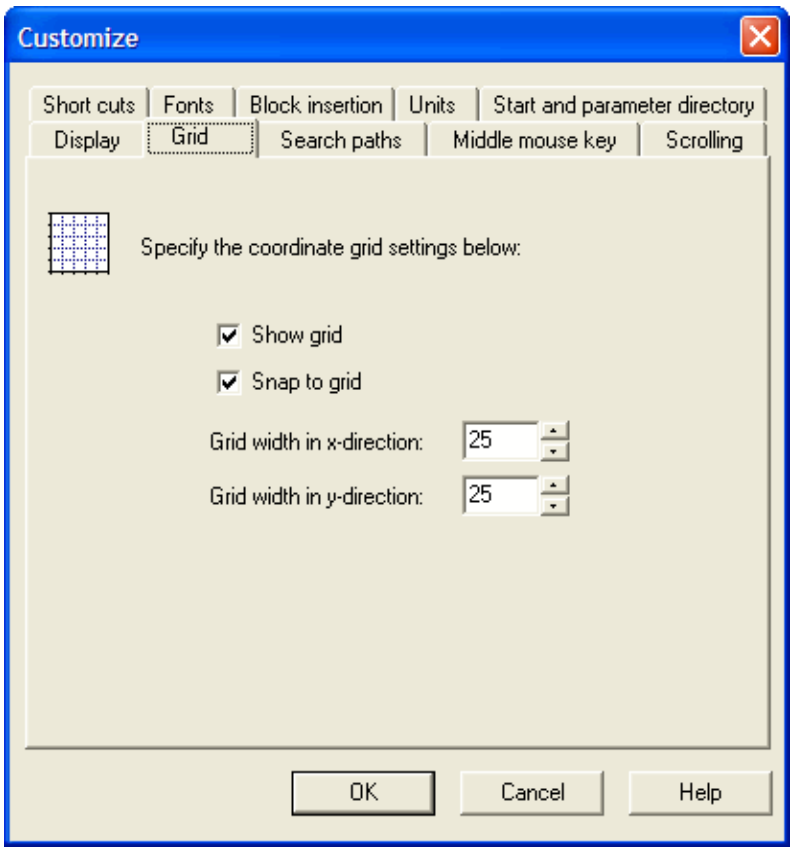


With the palette *Display* you get some of the options which concern the screen display. In detail the following options are available:

Option	Meaning
Appearance	Display mode of connections
Connections from/to passive blocks gray	Display mode of connections from/to passive blocks
Show connection knots	Activates the drawing of connections knots
Change colour for new connections...	Allows the choice of a new default colour for blocks which will be inserted in future. This option has no influence on connections which already exist!
Standard block size	Allows the choice of a new default size for blocks which will be inserted in future. This option has no influence on blocks which already exist!
Show block index	If this option is activated the block index will be

	shown in the block caption beside the block name.
Standard font size for text blocks	Allows the choice of a new default font size for text blocks which will be inserted in future. This option has no influence on text blocks which already exist!
Change background colour...	With this button the colour of the background of the BORIS paint window can be changed.

Grid

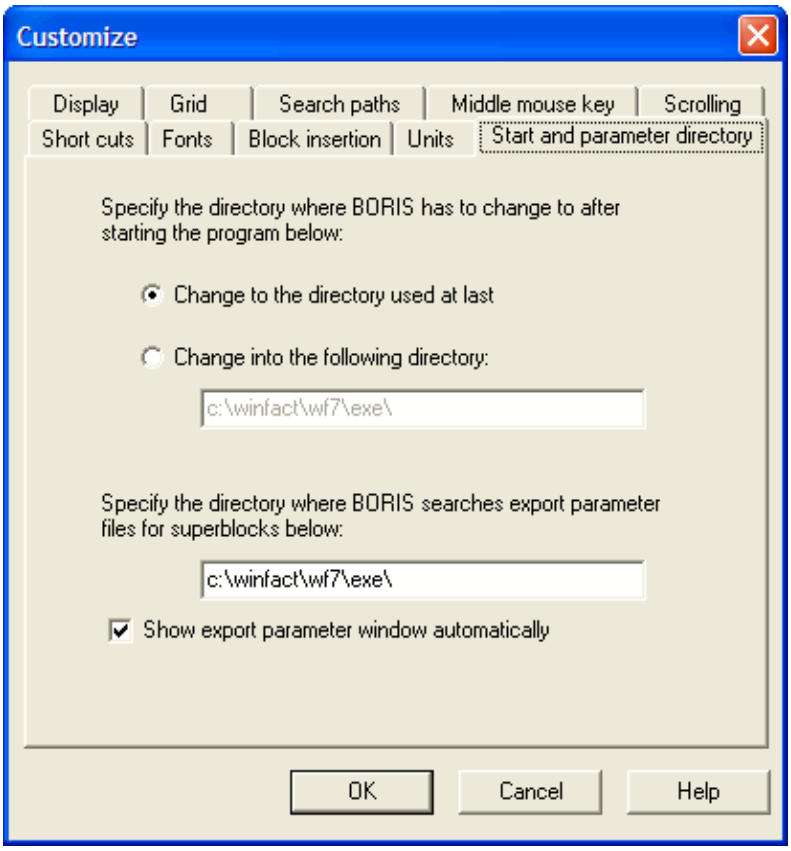


In the following you find a description of the options of this palette:

Option	Meaning
Show grid	Switches off resp. on the display of the screen grid
Snap to grid	Activates resp. deactivates the alignment of the system blocks to the grid. The alignment is independent of the visibility of the grid.
Grid width in x-direction	Specifies the horizontal distance between two points of the grid in pixels

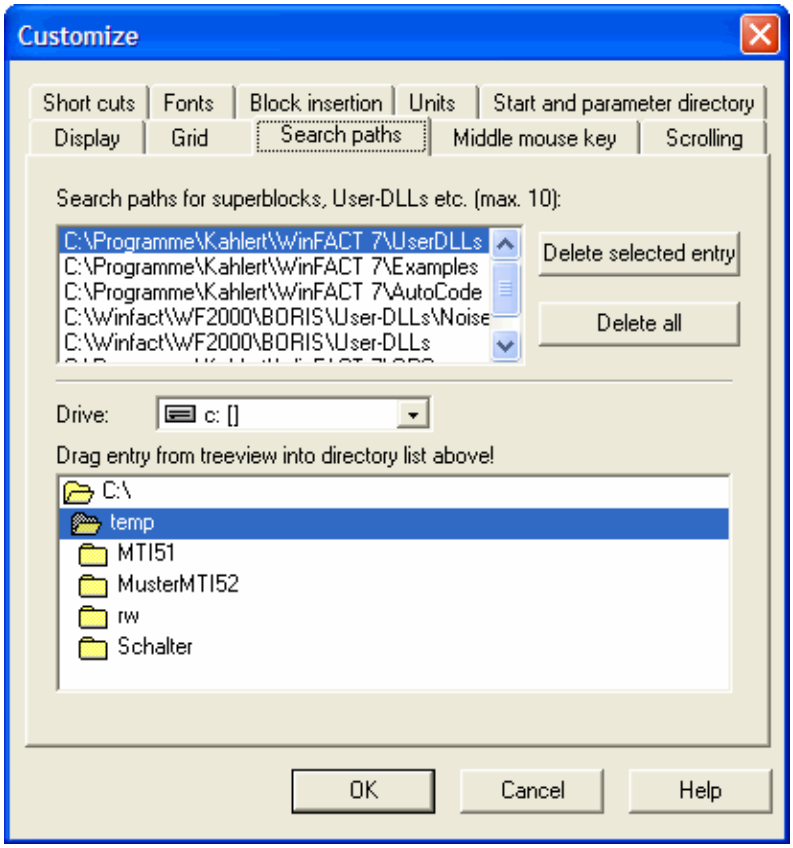
Grid width in y-direction	Specifies the vertical distance between two points of the grid in pixels

Start directory



This palette specifies the directory BORIS changes to automatically after it was started (that means e. g. the directory that is used as the default directory in file open resp. file save dialogs). On one hand this might be the directory used last before BORIS was closed, on the other hand it might be any independent directory specified by the user.

Search paths



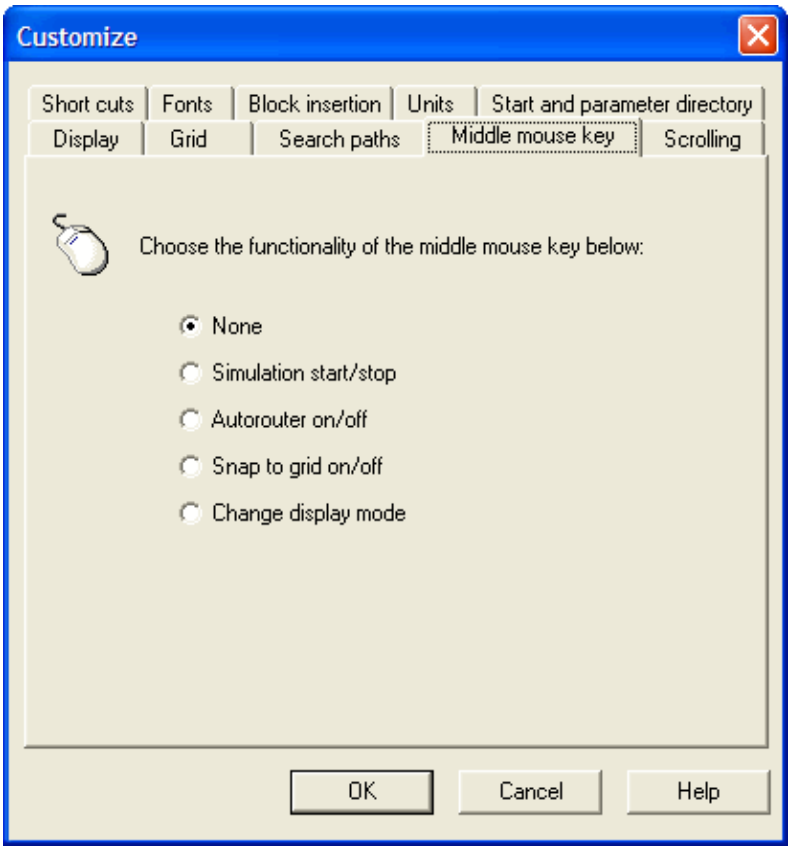
BORIS allows the specification of up to ten search paths using the menu option OPTIONS | CUSTOMIZE..., palette *Search paths*. All file-based blocks (e. g. superblocks or user blocks) will be searched in these paths (in the sequence of their position) if the names specified by the user contain no or no existing path. You insert a new entry by moving the desired path from the directory tree in the lower section of the dialog via drag & drop into the upper list. With the button *Delete selected entry* the entry which has just been selected can be deleted, the button *Delete all* deletes all entries.

The specification of search paths is especially helpful if complex system structures with many superblocks, User-DLLs, etc. should be copied to another computer with a different directory tree. It is not necessary to change all path specifications troublesome manually but the definition of a single or only a few search paths is sufficient.

Example: On computer A a BSY-file which contains several superblocks in the subdirectory C:\WINFACT\SB is located in the path C:\WINFACT. This simulation structure shall now be copied to computer B which basically uses the path C:\WF for BSY-files and the subpath C:\WF\SUPER for superblock files. Now it is sufficient to insert these path names as search paths on the

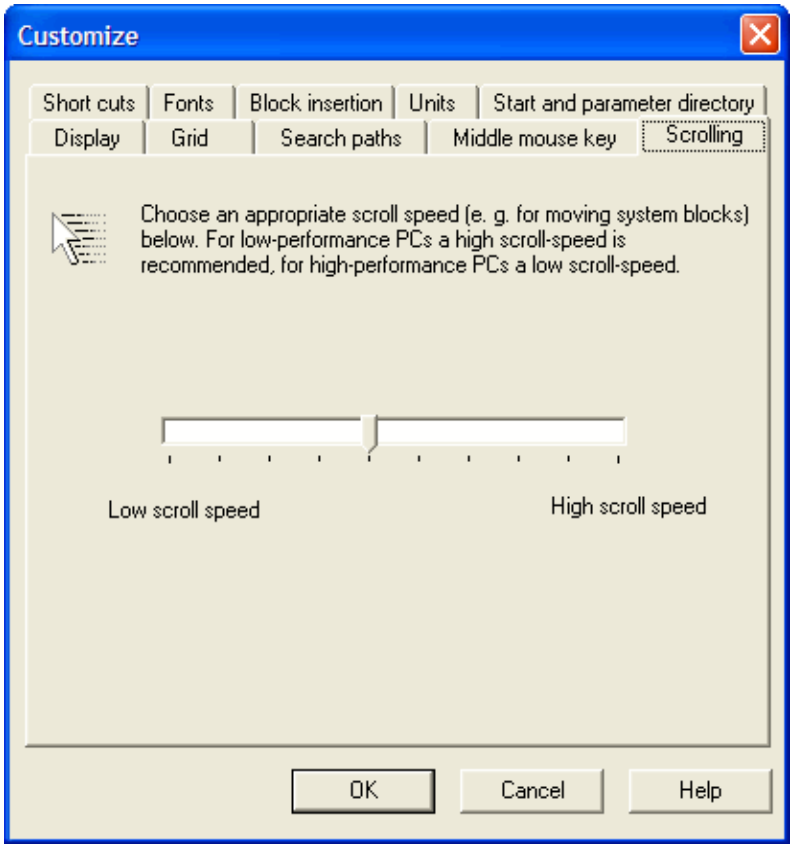
computer B and the simulation can be started. If the system is saved on computer B all filenames will be adjusted automatically.

Middle mouse button



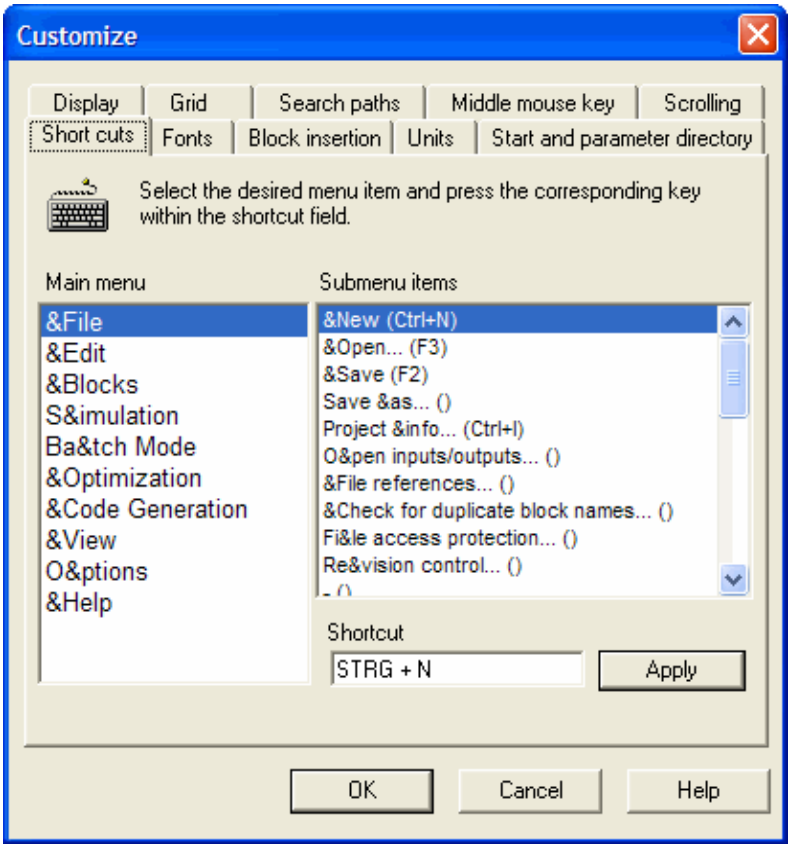
This palette enables the user of a three-button-mouse to configurate a special function for the middle mouse button.

Scrolling



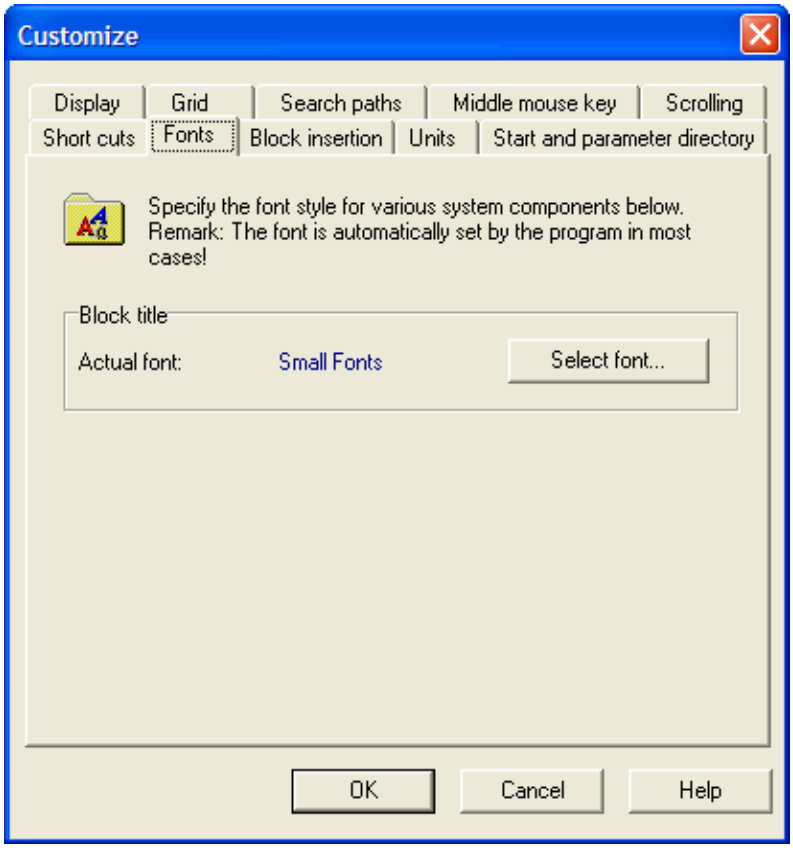
This palette enables the user to adjust the scrolling speed to the used computer configuration, especially to the used graphic card while moving blocks etc. If the <Shift>-key is pressed additionally while moving the blocks etc. the scrolling has double speed.

Shortcuts



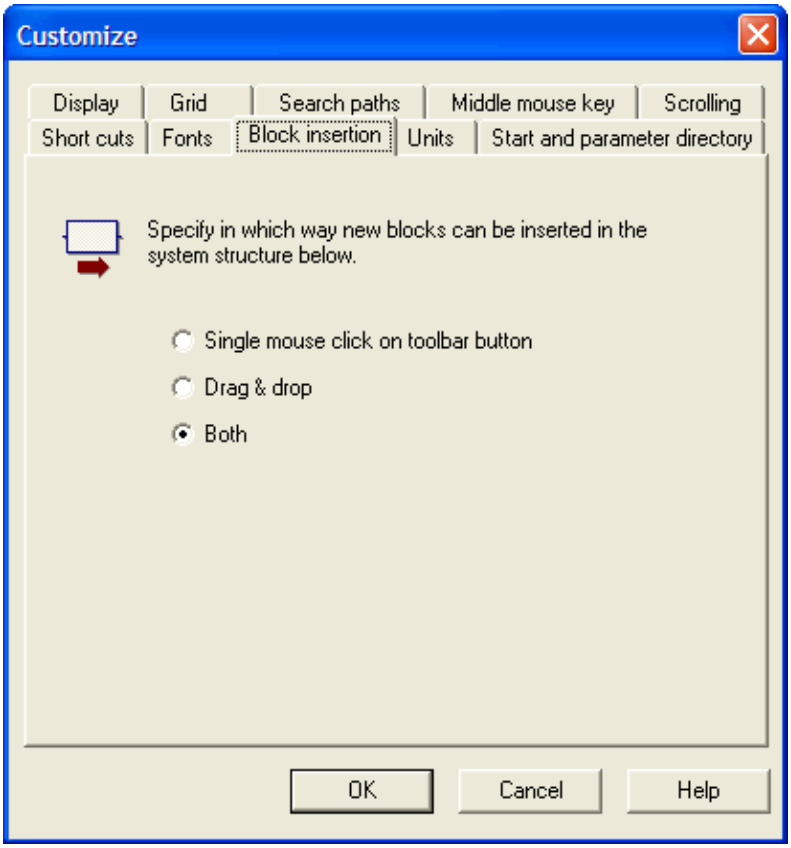
This palette enables the user to adjust the shortcuts of all options of the BORIS main menu to his specific habits. For that the desired menu option has to be chosen by using the lists *Main menu* and *Submenu items*. In the edit field *Shortcuts* the assignment can be made by a direct entry of the desired key or key combination (e. g. <Ctrl> <A>) and a click at the button *Apply*.

Fonts



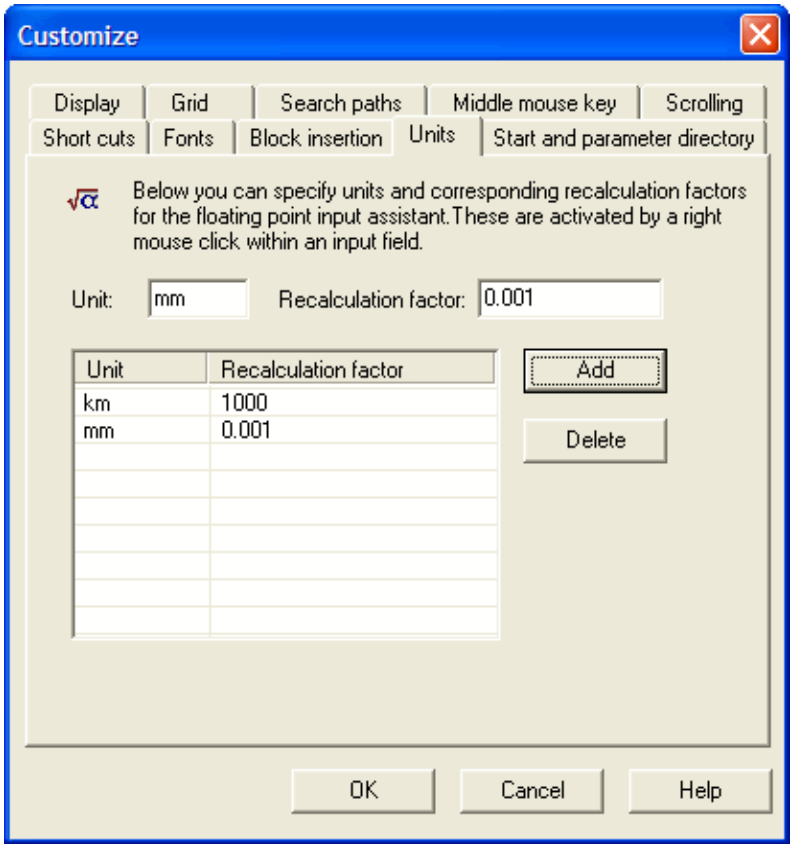
Depending on the operating system of the computer and the installed fonts the block captions which are shown in the size of 6 pixels may be hardly readable. Therefore the most suitable font for the block captions can be chosen using this palette.

Block insertion

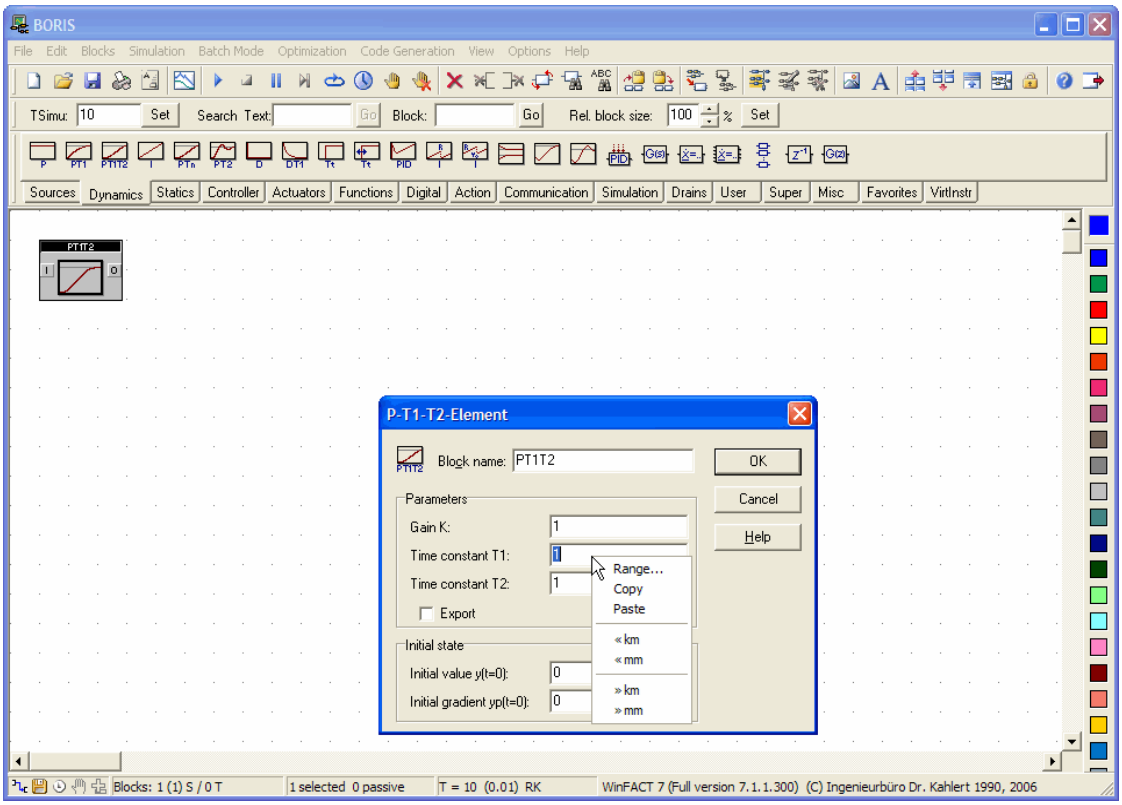


BORIS enables you to insert new blocks either by a click on the corresponding button of the system block toolbar (the inserted block will be placed automatically) or by drag & drop. Using this palette both options can be activated resp. deactivated separately.

Units



To convert variables which exist in different units (e. g. m^3/s and l/h) BORIS has an integrated input assistant which can be activated automatically by a click with the right mouse button inside the edit field for a floating point number. In the now appearing popup menu you can apply a unit specific converting coefficient or its reciprocal value to the inserted number. Using the palette *Units* you can configurate these units together with their converting coefficients. For that you have to specify the unit in the edit field *Unit* and the converting coefficient in the edit field *Recalculation factor*. With the button *Add* the unit is inserted in the list. The corresponding reciprocal will be produced automatically by BORIS.

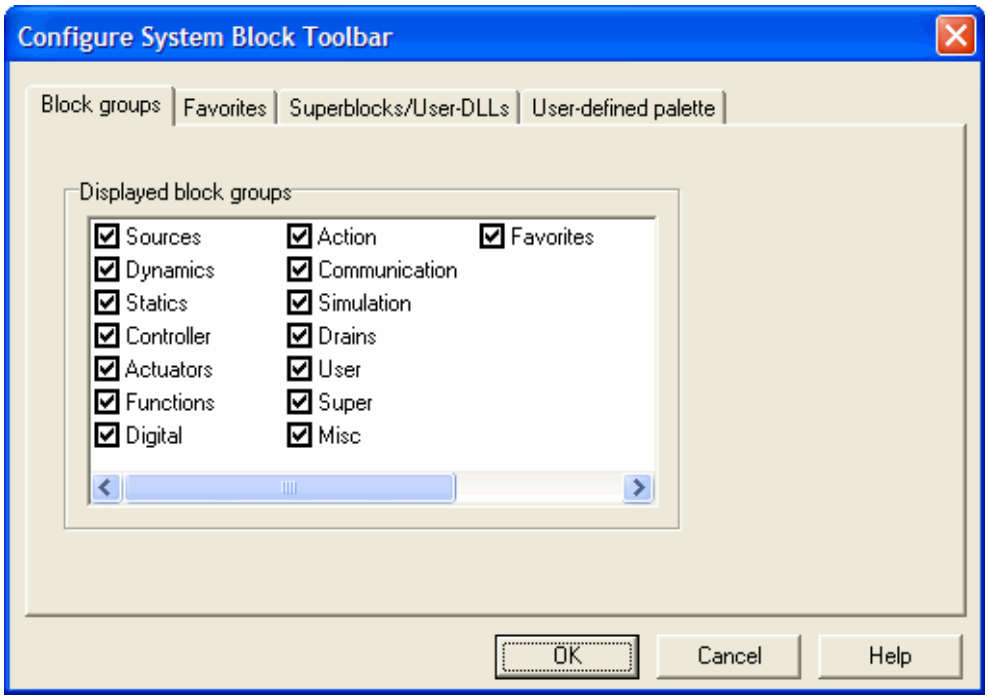


Calling the input assistant for floating point numbers

Configuring the system block toolbar

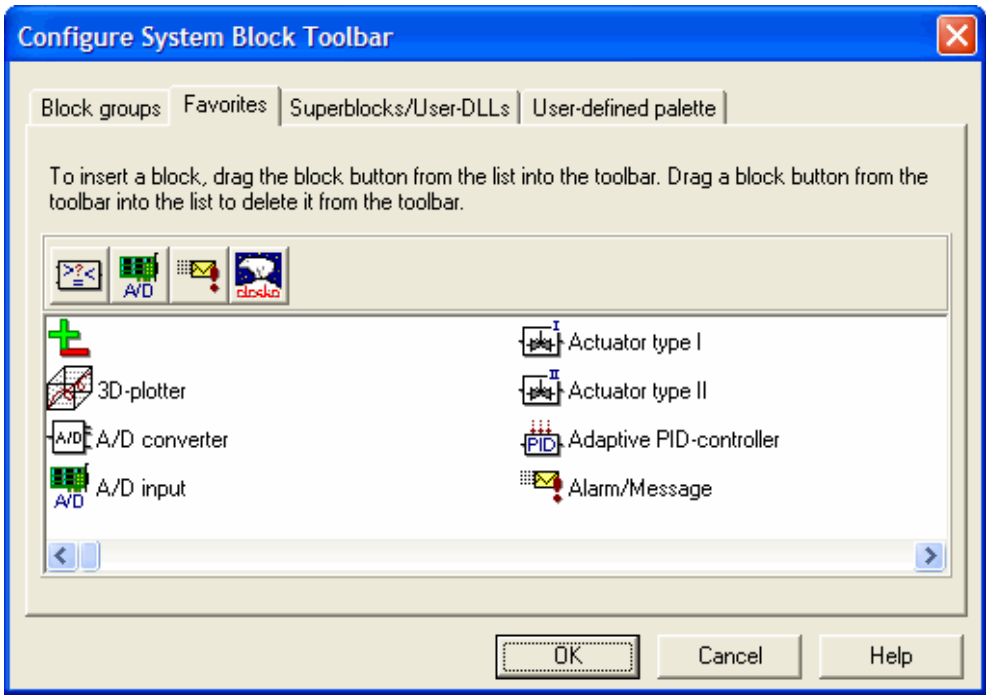
The system block toolbar can be customized by the user. The corresponding dialog which is divided into the following three palettes is available using the menu option OPTIONS | TOOLBARS | CONFIGURE SYSTEM BLOCK TOOLBAR...

Block groups



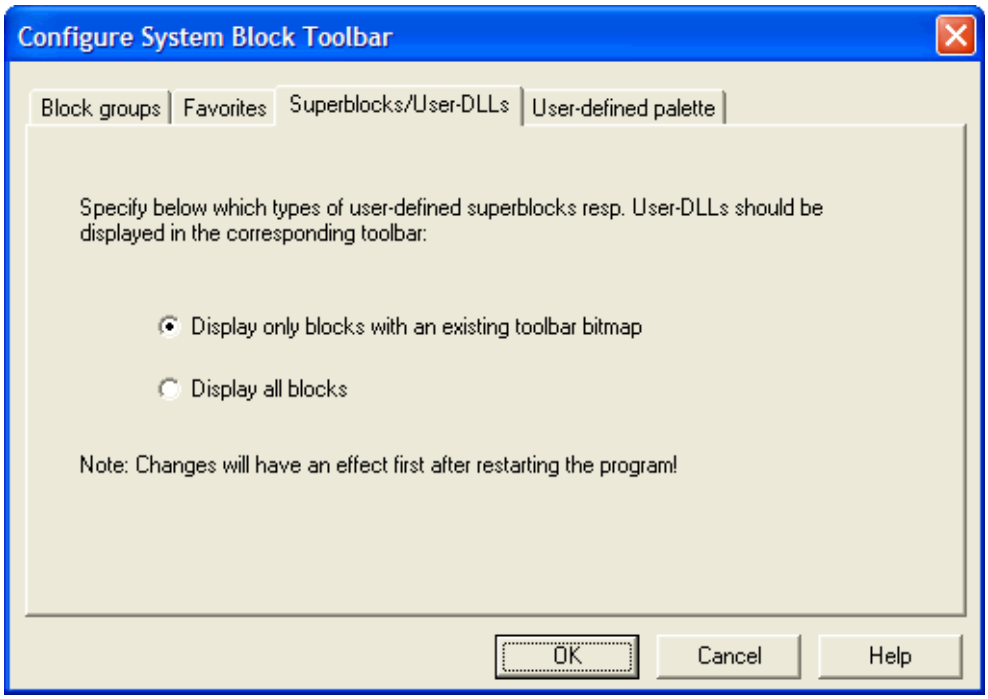
The system block toolbar is divided into several palettes which contain block types belonging together. With the palette *Block groups* single palettes of the system block toolbar can be hidden if necessary. By default all palettes are visible.

Favorites



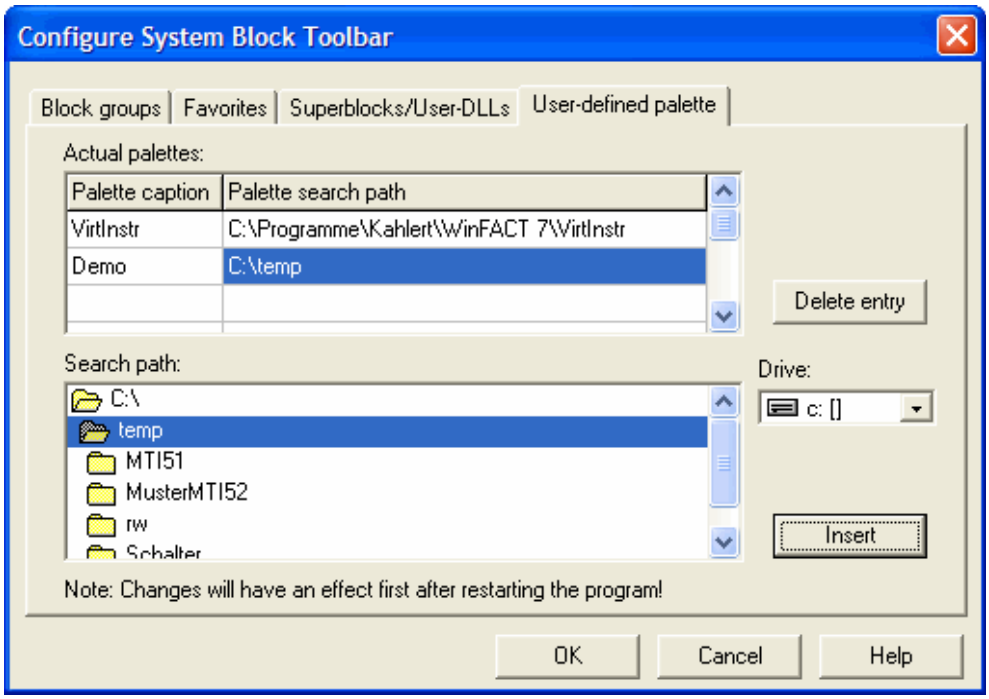
The palette *Favorites* of the system block toolbar can be configurated by the user with the dialog palette. For that the desired blocktypes have simply to be moved from the lower list window to the toolbar located above by drag & drop. By default the palette *Favourites* is empty.

Superblocks and User-DLLs



The palettes *User* resp. *Super* of the system block toolbar contain – besides "empty" blocks of the corresponding type – as well all user-defined superblocks resp. User-DLLs which can be found by BORIS in the WinFACT program directory or in one of the search directories. This dialog palette determines whether all user-defined blocks will be inserted or only those ones for which a user-defined toolbar bitmap has been found. Please note that a modification of this setting will not become active before the next call of BORIS.

User-defined palettes



In addition to the standard block type palettes the system block toolbar can be expanded by user-defined palettes. The blocks on these palettes are all User-DLLs resp. superblocks BORIS finds within a palette-specific directory. To insert a new palette just do the following:

- Within the *Palette caption* column of the list in the upper part of the dialog specifiy the title of the new palette (*Demo* above)
- Within the tree view below select the directory your palette shall base upon and insert the directory (*c:\temp* above) into the list.

After restarting BORIS the new palette appeares within the system block toolbar.

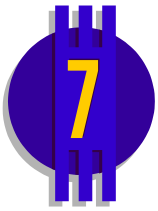
Starting BORIS with command line parameters

Optionally BORIS can be started with command line parameters which effect an automatical reading of a file and if necessary an automatical start of the simulation.

BORIS *Filename*	Starts BORIS and loads the file *Filename* automatically.

BORIS *Filename* /S	Starts BORIS, loads the file *Filename* automatically and starts the simulation. Alternatively the switch */S* can be placed in front of the filename.
BORIS *Filename* /E	Starts BORIS, loads the file *Filename* automatically and starts an endless simulation. Alternatively the switch */E* can be placed in front of the filename.
BORIS *Filename* /O	Starts BORIS, loads the file Filename automatically and starts a parameter optimization. Alternatively the switch */O* can be placed in front of the filename.
BORIS *Filename...* /C	Terminates BORIS automatically after the simulation resp. optimization. This switch only makes sense combined with */S* or */O*..

Using BORIS as a COM automation server



For a lot of functions and parameters BORIS offers a so-called *IDispatch*-interface which can be used by other Windows programs for a remote control of BORIS. In this mode BORIS works as a so-called *COM automation server*. On one hand the interface consists of *methods* which correspond to specific functions of BORIS, on the other hand it offers *properties* to read or set BORIS settings (parameters). The name of the automation server is *BORIS.BoAutoObject*. The tables below give an overview of all available methods and properties of the interface.

Method	Parameters	Return value	Function
LoadFile	*Filename* (BSTR)	-	Loads the file named *Filename*
AddFile	*Filename* (BSTR)	-	Adds the file named *Filename* to the current structure
Clear	-	-	Clears the current worksheet
StartSim	-	-	Starts a standard simulation
StartEndlessSim	-	-	Starts an endless simulation

StopSim	-	-	Terminates the simulation
Break	-	-	Switches simulation to the single step mode
SingleStep	-	-	Performs a single step
ShowWindows	-	-	Shows all output windows
HideWindows	-	-	Minimizes all output windows
MinimizeRestore	-	-	Minimizes the main window to toolbar height resp. restores it from that state
GetBlockOutput	*BlockIndex* (LONG), *OutIndex* (LONG)	VARIANT	Returns the value of output *OutIndex* of the block *BlockIndex*
SetConstValue	*BlockIndex* (LONG), *Value* (DOUBLE)	-	Sets the parameter value of a CONST-block with *BlockIndex* to *Value*

Methods of the IDispatch-*interface of BORIS*

Property	Type	Meaning
Filename	BSTR	Name of current file
Time	DOUBLE	Current simulation time
DeltaT	DOUBLE	Simulation step size
TSimu	DOUBLE	Simulation length
IsRealTimeSim	BOOL	Realtime simulation yes/no
IsSimulating	BOOL	Standard simulation running yes/no
IsEndlessSimulating	BOOL	Endless simulation running yes/no
IsBreak	BOOL	Single step mode active yes/no

Properties of the IDispatch-*interface of BORIS*

The source code fragment below demonstrates the usage of the COM interface from DELPHI; in this sample BORIS is started via button click, the file *c:\temp\test.bsy* is loaded and an endless simulation is executed.

```
procedure TForm1.Button1Click(Sender: TObject);
var V: Variant;
begin
  // Objekt anlegen, d. h. BORIS öffnen
  V := CreateOLEObject('BORIS.BoAutoObject');
  // Datei laden
  V.LoadFile('c:\temp\test.bsy');
  // Endlossimulation starten
  V.StartEndlessSim;
end;
```

Using the COM interface from DELPHI

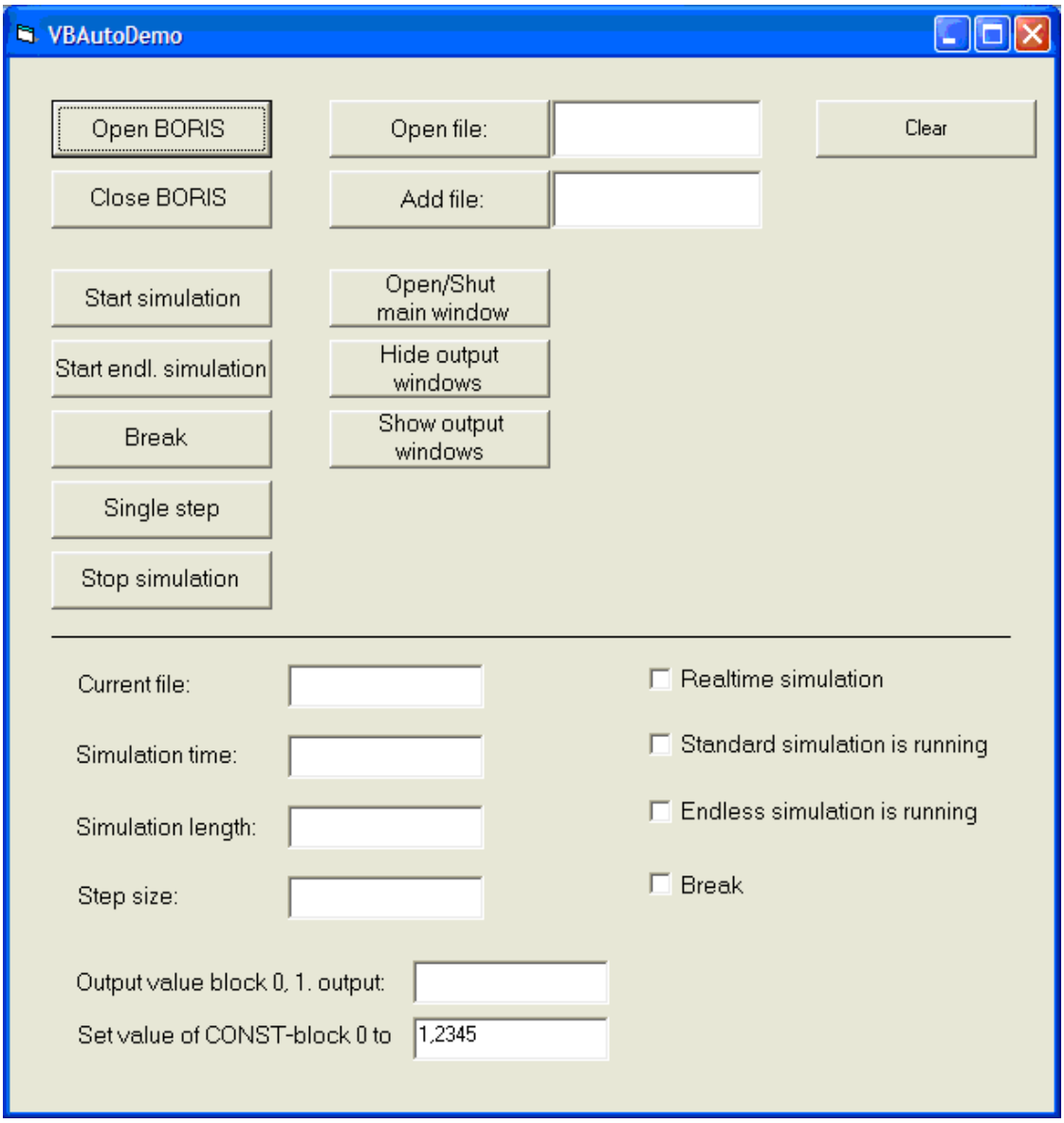
The following listing shows the equivalent source code for Visual Basic:

```
Private Sub Command1_Click()
Dim V As Object
Set V = CreateObject("BORIS.BoAutoObject")
V.LoadFile ("c:\temp\test.bsy")
V.StartEndlessSim
End Sub
```

Using the COM interface from Visual Basic

The *\VBAutoDemo* directory of your WinFACT installation contains a complete Visual Basic project (source codes included) that demonstrates the usage of all functions of the automation server. The screenshot below shows the main window of this program.

Remark: To enable BORIS to work as an automation server BORIS has to be registered in the Windows registry once. This is done *automatically* when BORIS is started for the first time.



Visual Basic test program VBAutoDemo

The system block library of BORIS

Types of system blocks

In the following chapters you find a description of all modules of the BORIS system block library. The system block types are divided into the following type classes, each has its own palette in the system block toolbar:

Sources

This group contains those system block types which only have an output and therefore produce input signals for the system (e. g. generator, file input). An exception to this is the VCO which has a control input but should belong to the group of the input blocks because it can be handled like a signal generator.

Dynamics

To this group belong all linear and nonlinear dynamic systems from the simple P-element up to the transfer functions which can be parameterized in any way and the differential equation systems.

Statics

Basically these are the nonlinear characteristic curve elements resp. 3D-characteristics (e. g. limiter, dead zone).

Controller

This block group includes various linear and nonlinear controller types.

Actuators

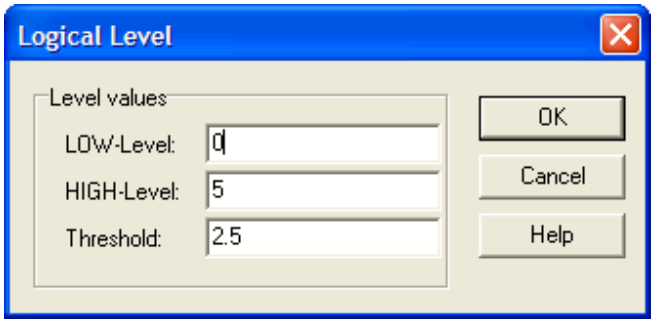
This class contains blocks which imitate the behaviour of real industrial final controlling elements.

Function

Those block types which are no conventional transfer systems are named function blocks. Especially these are the adder and other junctions of several inputs variables.

Digital

This class contains those blocks which output variables can only take the binary states HIGH (logical 1) resp. LOW (logical 0). The corresponding levels can be specified by BLOCKS | DIGITAL | LEVELS....



Specification of logical levels

The value for the *Threshold* defines the level from which the digital input value can be seen as logical 1. By default the value of low is 0, the value of high is 5 and the value of the switching value is 2.5 (corresponding to TTL).

Action

These are blocks which allow an interactive access by the user (e. g. push button).

Communication

This group includes blocks for the communication with DDE or the TCP/IP-protocol.

Simulation

These are all blocks concerning the simulation control itself.

Drains

This type class includes all blocks which only have one resp. several inputs. Basically these are block types for the visualization or processing of simulation results (e. g. oscilloscope, file output). Besides the system block in the paint window additionally the blocks have a display window which shows the simulation results. If you insert such a block, this display window is presented symbolized by a type-specific icon. For the simulation all display windows can be set in normal size using the menu option VIEW | SHOW ALL OUTPUT WINDOWS, otherwise they can be minimized at any time by the menu option VIEW | HIDE ALL OUTPUT WINDOWS. In each case the display shows the same caption as the system block itself. Alternatively the system blocks can be parameterized

by a double click at the system block or inside the display. For some of the output blocks the display windows are resizable (e. g. oscilloscope).

Miscellaneous

This class includes all blocks which can not be classified in one of the groups named before.

Note: The blocks which are marked with an asterisk (*) during the following chapters are only available in combination with the corresponding BORIS-add-ons resp. hardware components and drivers!

Sources

Generator

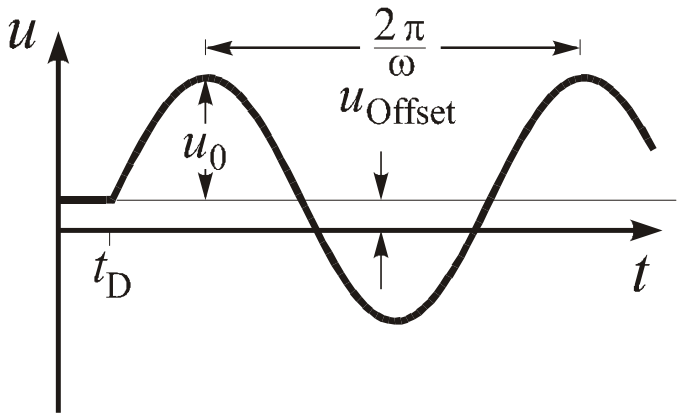
Typename: GENERATOR

Function: Produces different kinds of test signals for the stimulation of systems. The generator has different modes which are marked within the system structure by different block bitmaps:

- Sinus generator

  In this case the generator produces a sinusoid output signal of the form

$$u(t) = \begin{cases} u_{\text{Offset}} & for\ \ t < t_{\text{D}} \\ u_0 \sin(\omega(t - t_{\text{D}}) + \varphi) + u_{\text{Offset}} & otherwise \end{cases}$$

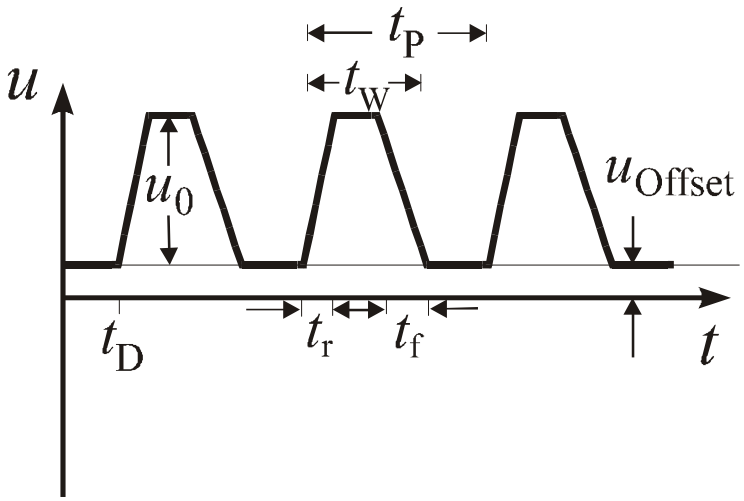


Parameters:

u_0 :	Amplitude
u_{Offset} :	Offset
t_D :	Delay time
ω :	Angular frequency
φ :	Phase (in degree)

- Pulse generator

  Depending on the choice of parameters you will get rectangular, triangular or saw tooth pulses:



Parameters:

t_r :	Rise time
t_f :	Fall time
t_w :	Pulse width
t_p :	Periodic time

other parameters see sinus generator!

Single pulses can be produced by a special choice of parameters. If you choose for example pulse width and period time bigger than the simulation time and a rise resp. fall time of 0 you will get a step function (default).

- Noise generator

  In this case the generator produces a uniformly distributed random number from the interval $[-u_0 + u_{\mathrm{Offset}},\ u_0 + u_{\mathrm{Offset}}]$ at each simulation step.

- *Programmable function generator*

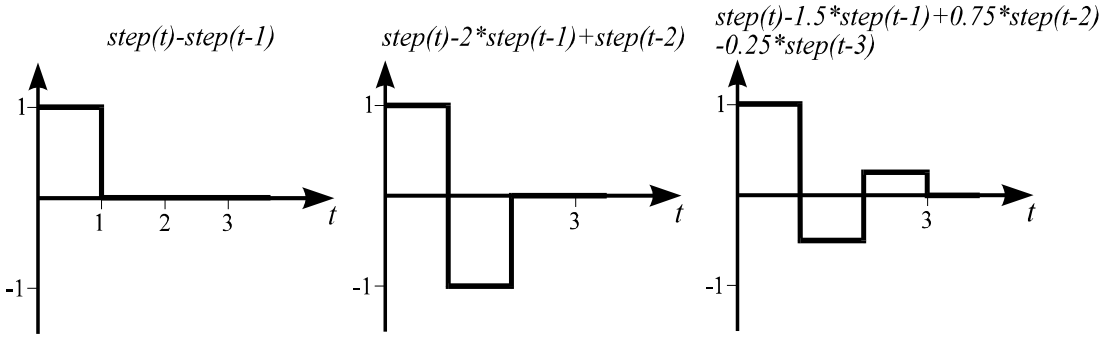
  For special applications the generator can be programmed by the user with nearly any function which is interpreted by a function parser. This mode, however, needs some calculation time. The parser allows the interpretation of the following symbols resp. functions:

Syntax	**Function**
t	Independent variable (time)
+	Plus (addition)
-	Minus and unary minus
*	Multiplication
/	Division
^	Power operator
( )	Brackets (max. depth 20)
ln	Natural logarithm
log	Decadic logarithm
sqr	Square
sqrt	Square root
exp	Exponential function
sin	Sinus
cos	Cosinus
tan	Tangens
asin	Arcussinus
acos	Arcuscosinus
atan	Arcustangens
sinh	Sinus hyperbolicus
cosh	Cosinus hyperbolicus
tanh	Tangens hyperbolicus

abs	Absolute value
deg	Conversion Radiant → Degree
rad	Conversion Degree → Radiant
int	Integer part
frac	Fractional part
round	Round to next integer value
sign	Signum function
step	Step function

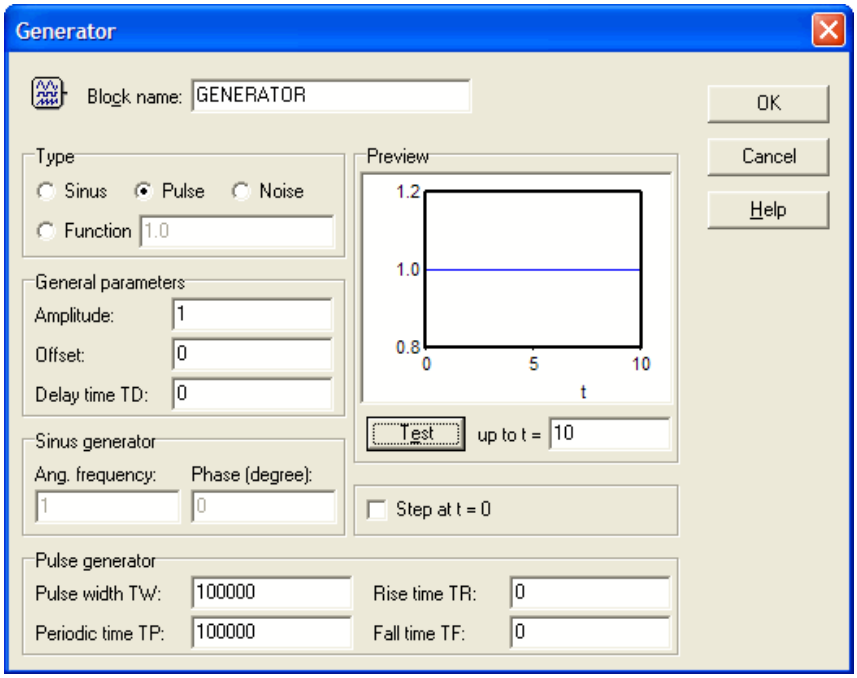
The function string may have a maximum length of 255 characters. The parser works case-insensitive.

By using the step function the function generator can be used to create nearly any form of a pulse signal. The diagram below shows some examples with the corresponding function strings.



Various pulse signals created by using the step function

Parameter dialog:



The *Type* button group allows the selection of the operating mode. The current settings can be tested via the *Test* button at any time.

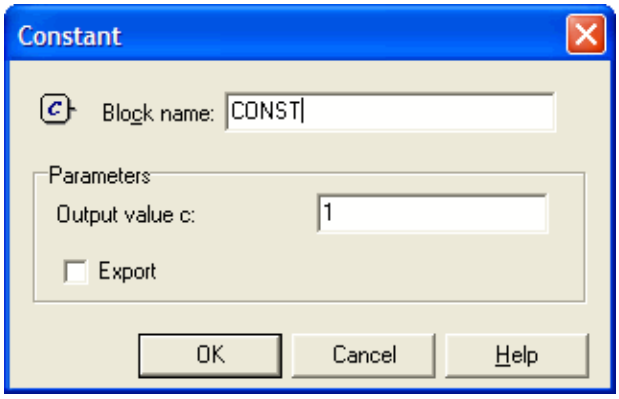
If the checkbox *Step at t = 0* is activated the output signal of the generator is set to zero at the simulation start independent of other settings. In this case the actual output is set at the first simulation step, i. e. after a simulation time ΔT .

Constant

Typename: CONST

Function: Delivers a constant value.

Parameter dialog:

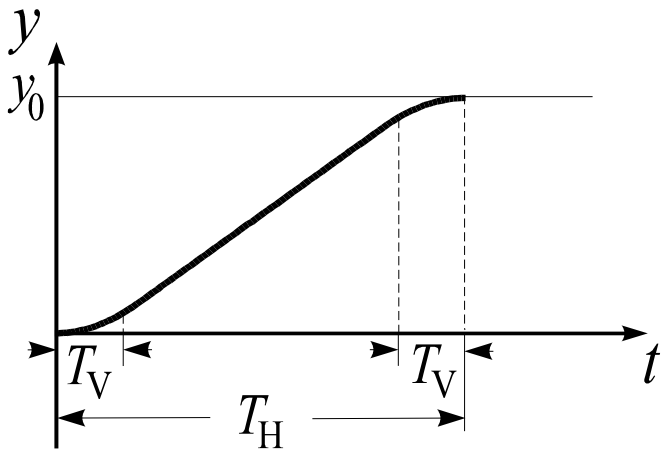


Output value specifies the value which is delivered by the block.

Driving curve

Typename: DRIVCURVE

Function: The input block produces a continuous, from 0 to a user-defined amplitude increasing signal, as it is needed, for instance, for the controlled running up of engines. The following graphic shows the signal progression.



The output $y(t)$ results from the following equation

$$y(t) = \frac{y_0 t^2}{2(T_H - T_V)T_V}\Bigg|_{t=0}^{t=T_V} + \frac{y_0}{T_H - T_V}(t - T_V)\Bigg|_{t=T_V}^{t=T_H - T_V} + \ldots$$

$$\ldots + \frac{y_0 T_V}{2(T_H - T_V)}\left(1 - \frac{(t - T_H)^2}{T_V^2}\right)\Bigg|_{t=T_H - T_V}^{t=T_H}$$

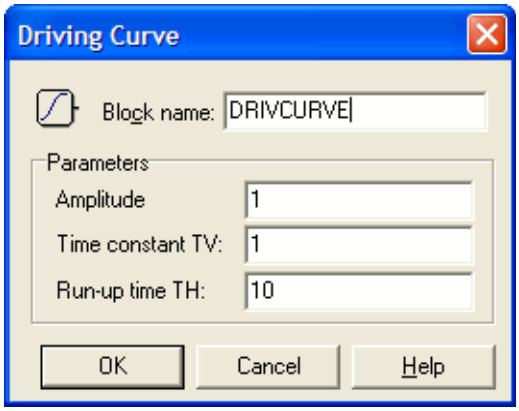
with

y_0: Signal amplitude

T_V: Transition time

T_H: Run-up time

Parameter dialog:

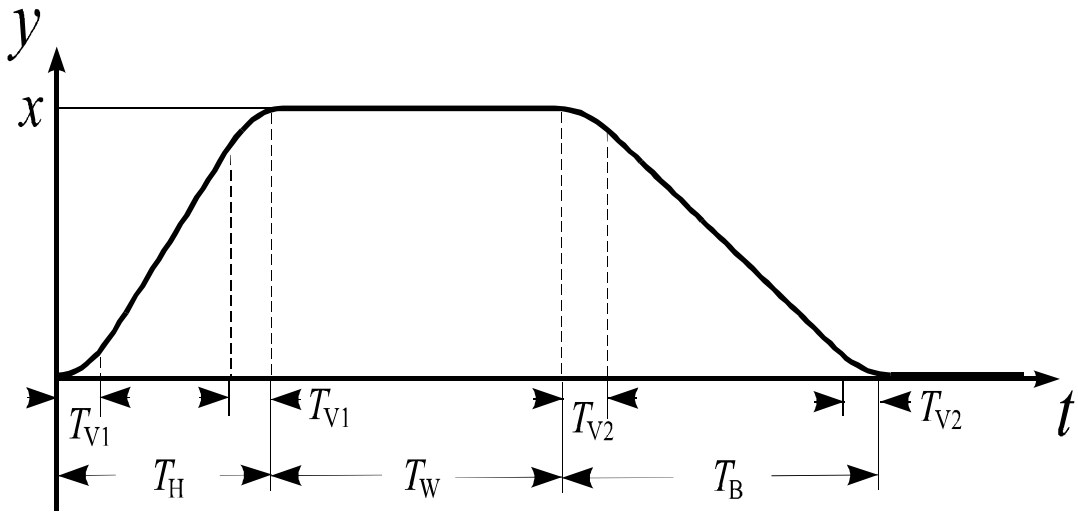


Restrictions: $T_H \geq 2T_V$

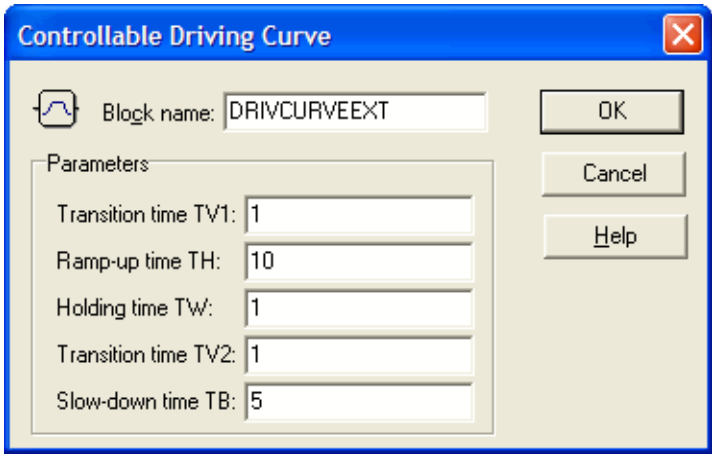
Controllable driving curve

Typename: DRIVCURVEEXT

Function: This block type represents an extension of the DRIVCURVE block type. The output signal of the block first increases within a ramp-up time $T_H \geq 2T_V$ to the (set point) value x delivered by the block input and then after a holding time T_W decreases to zero again within a slow-down time T_B. By a positive edge at the reset input R the output can be reset at any time. The graphic below shows the principal form of the signal.



Parameter dialog:



Restrictions: $T_H \geq 2T_{V1}, T_B \geq 2T_{V2}$

Controllable sinus generator (VCO)

Typename: VCO

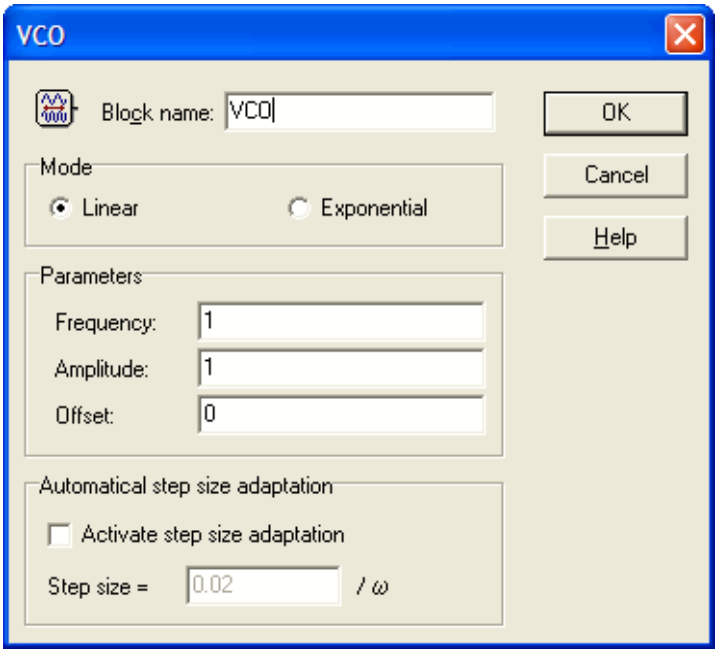
Function: This block represents a sinus generator which frequency is controllable via the block input. The dependence can be linear or exponential.

The basic values correspond to the settings of the block type *Generator* in the operating mode of a sinus generator. If ω_0 is the basic frequency of the VCO you will get the current frequency ω depending on the input variable x as follows:

Operating mode *linear*: $\omega = x\omega_0$

Operating mode *exponential*: $\omega = 10^x \omega_0$

Parameter dialog:



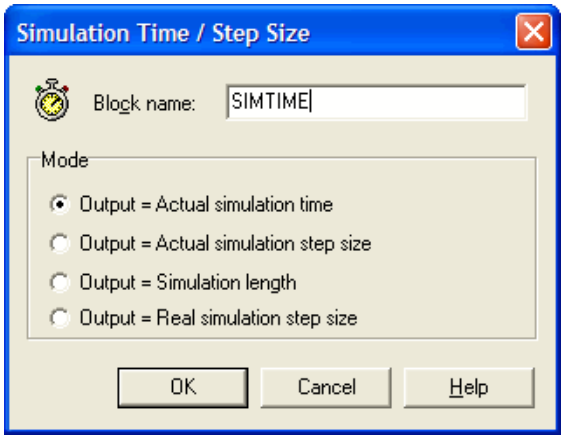
The settings within the *Automatical Step Size Adaptation* group box can be used in the batch mode for an automatical adaptation of the simulation step size dependent on the current frequency. For hints concerning this adaptation see chapter *Batch mode simulation*.

Simulation time/Simulation step size

Typename: SIMTIME

Function: This block delivers the current simulation time, the simulation step size or the total simulation time (simulation length).

Parameter dialog:

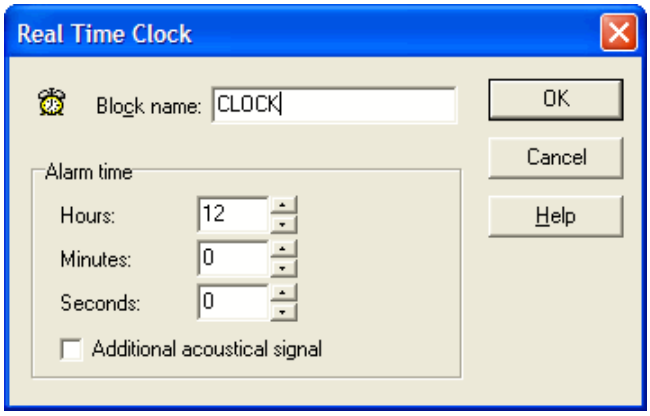


Real time clock with alarm function

Typename: CLOCK

Function: Real time clock with alarm function. At the outputs the current time in hours (H), minutes (M) and seconds (S) appears. When reaching the specified alarm time the alarm output A takes High-level for one simulation step. Additionally if desired an acoustical signal can sound.

Parameter dialog:



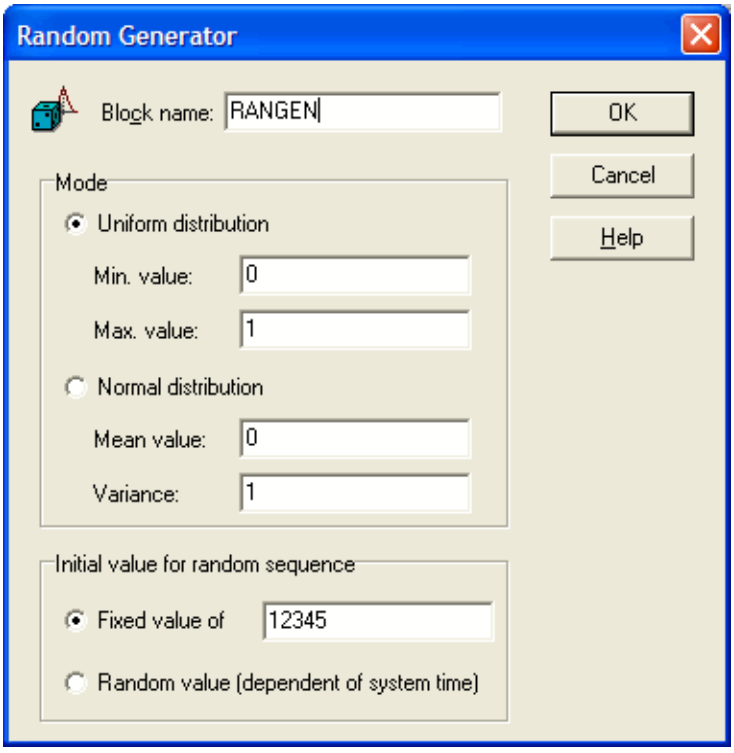
Random number generator

Typename: RANGEN

Function: This block type allows the generation of uniformly or normally distributed random numbers. In case of uniform distribution the lower and upper limit of the numbers have to be specified, in case of a normal distribution mean value and standard deviation.

The initial value for the random sequence can be specified; for a fixed initial value each simulation run generates an *identical* sequence of random numbers. If a different sequence is requested for each simulation, the initial value can be generated from the current system time.

Parameter dialog:



File input

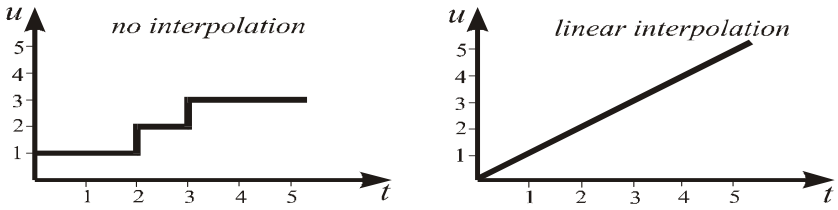
Typename: FILEINPUT

Function: Allows the reading of of a signal $u(t)$ in the form of pairs of values (t_i, u_i) from a file of the type SIM. The time values {bmc image1819.bmp} can be arbitrary but in ascending order.They do not have to be equidistant. Depending on the settings a linear interpolation between the values will be executed or not. In the last case you will get a stepwise constant progression of input values.

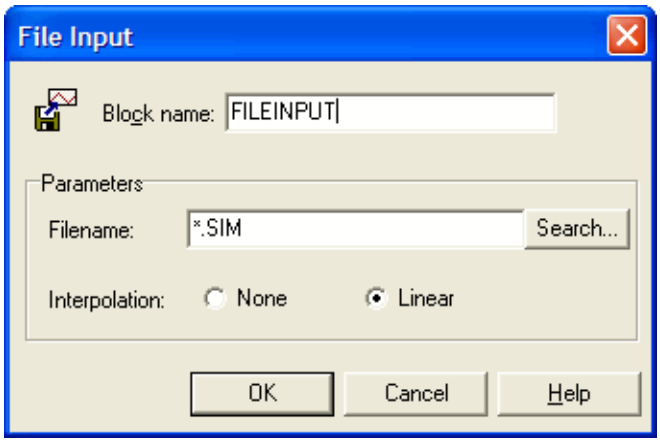
Example: The file which has been read contains the values

1	1
2	2
3	3.

If you simulate up to $t = 5$ you will get the following results:



Parameter dialog:



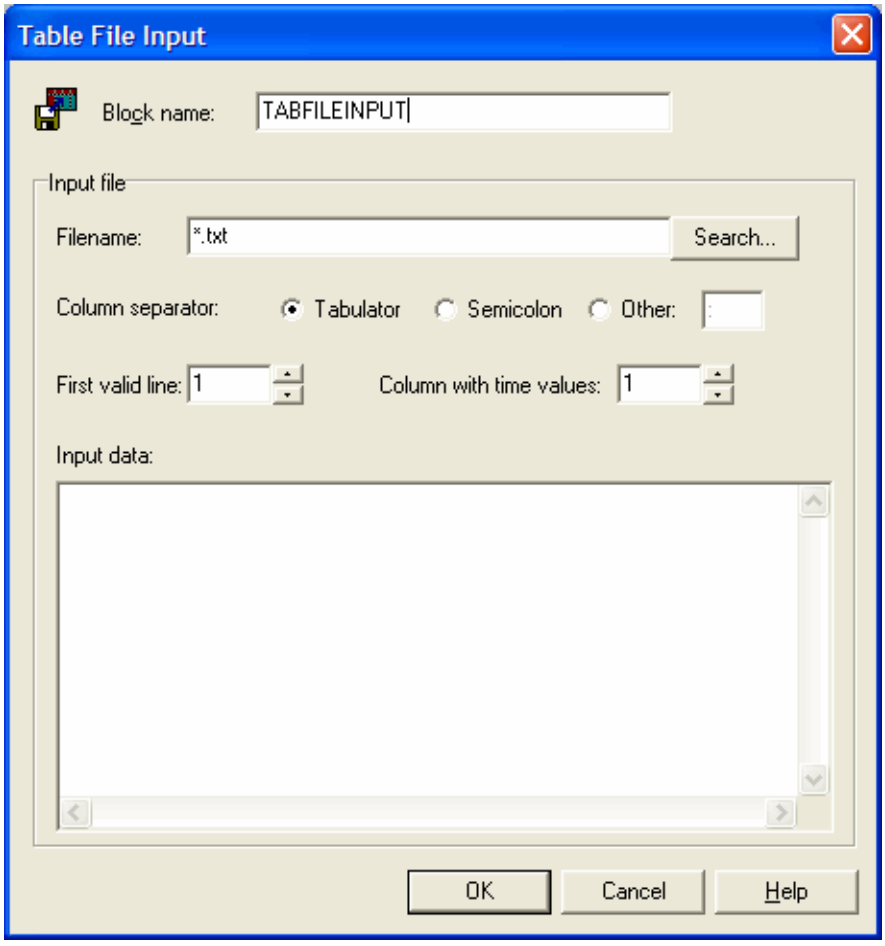
If you don´t specify an extension for the *input file* the extension SIM will be used. With the *Search* button you can call a file input dialog.

Table file input

Typename: TABFILEINPUT

Function: This block type allows the input of time progresses from a structured ASCII file whereby each line of the file contains the time value and the corresponding amplitude values. The file header may contain any number of comment lines (e. g. column header). The column separator can be any character.

Parameter dialog:



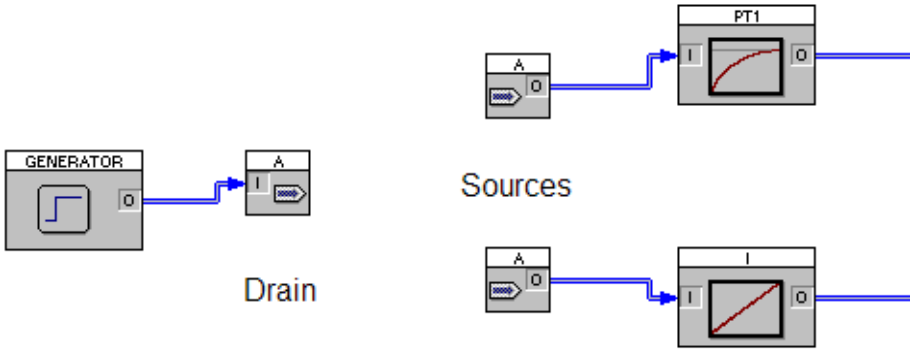
The *First valid line* edit field specifies the first line containing real data. The *Column with time values* edit field specifies the column containing the time values of each line. The time values must be sorted in increasing order, but the difference between two time values has *not* to be equivalent to the simulation step size; if necessary, a linear interpolation between two subsequent values is executed. If no extension for *Filename* is specified, the extension TXT is used. The *Search*... button can be used to select the file via a file open dialog.

Signal source

Typename: SOURCE

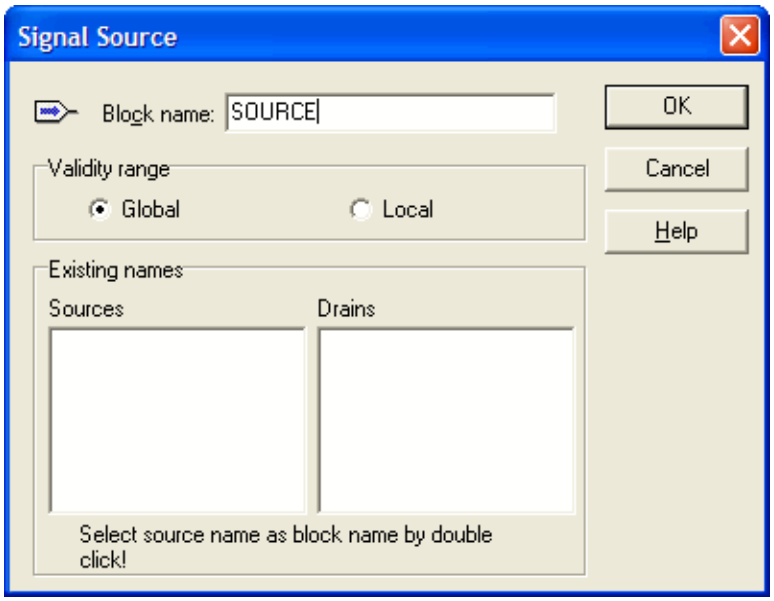
Function:

In combination with the output block type *Signal drain* this block enables you to realize "wireless" connections between blocks. In doing so the signal drain ”sends” its input signal with a specific name (the name of the block). This signal can then be received at any position of the system structure by a signal source of the same name. The block names are not case-sensitive. Signal sources can be of *local* or *global* validity. In the first case they are only known within the corresponding system or superblock file, in the second case they are valid outside the file as well. The use of both block types is especially recommended for complex, strongly branched system structures to decrease the number of visible connections.



The drain/source concept

Parameter dialog:



In the left listbox of the dialog all existing sources are listed in alphabetical order, in the right listbox all drains. With a double click at the name of a drain in the right listbox this can be adopted directly as the name of a source.

Dynamic blocks

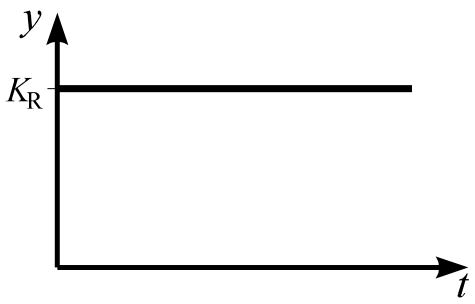
P-Element

Typename: P

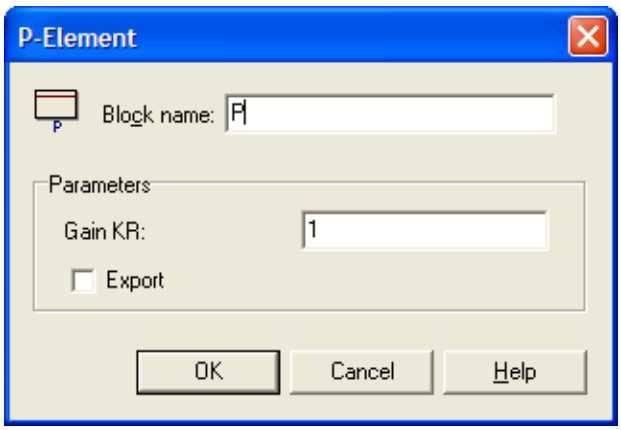
Function: The proportional element (P-element) transfers the input variable $x(t)$ into an output variable $y(t)$ according to the relationship $y(t) = K_R x(t)$. Thus it has the transfer function

$$G(s) = K_R$$

and the following step response:



Parameter dialog:



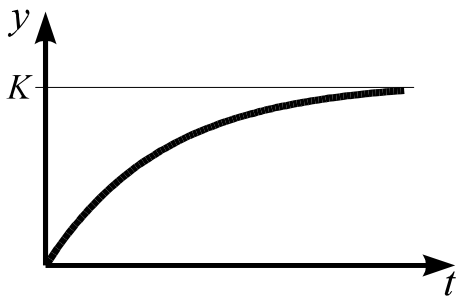
PT_1-Element

Typename: PT1

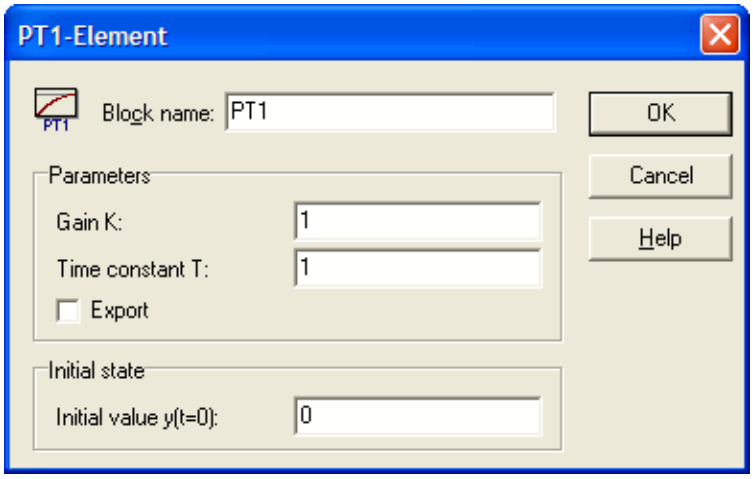
Function: Delay element of first order with the gain K and the time constant T. The PT_1-element thus has the transfer function

$$G(s) = \frac{K}{1 + Ts}$$

and the following step response:



Parameter dialog:



For the simulation the initial value $y(t = 0)$ has to be specified.

Restrictions: $T > 0$

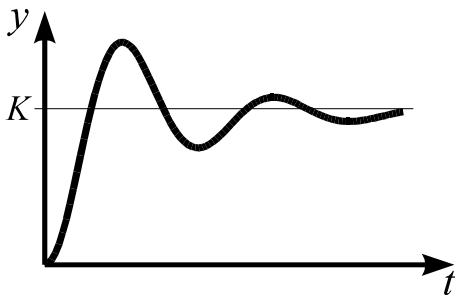
PT_2-Element (oscillating)

Typename: PT2

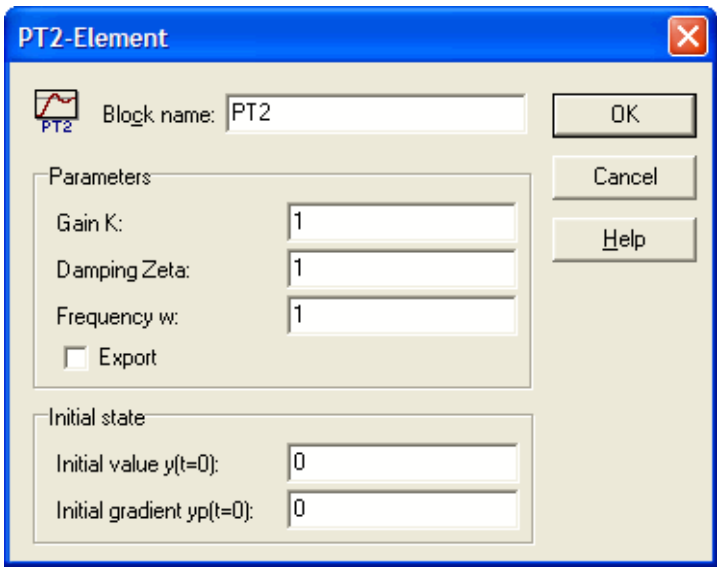
Function: Oscillating delay element of second order with the gain K, the eigenfrequency ω and the damping ζ. The system has the transfer function

$$G(s) = \frac{K}{\left(\frac{s}{\omega}\right)^2 + 2\frac{\zeta}{\omega}s + 1}$$

and the following step response:



Parameter dialog:



For the simulation the initial value $y(t=0)$ and the initial gradient $\dot{y}(t=0)$ have to be specified.

Restrictions: $\zeta \geq 0,\ \omega > 0$

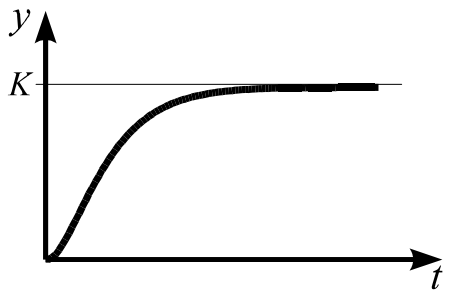
PT_1T_2-Element

Typename: PT1T2

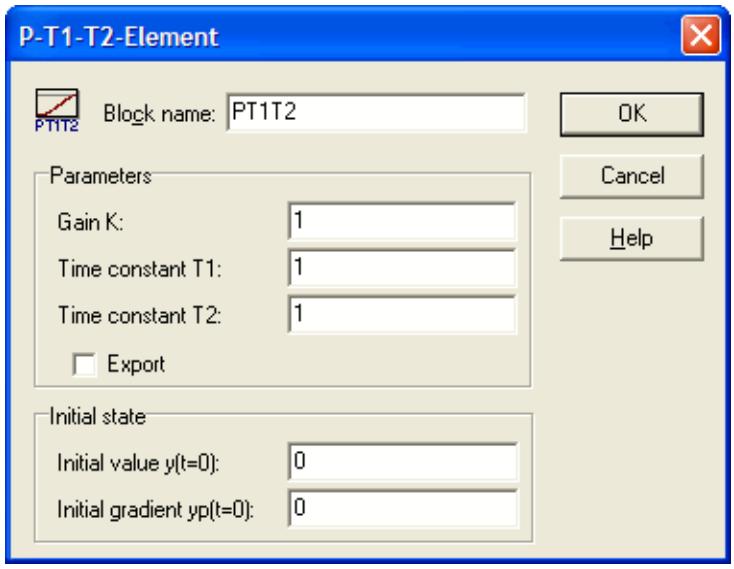
Function: Non-oscillating delay element of second order with the gain K and the two time constants T_1 and T_2. Thus it has the following transfer function:

$$G(s) = \frac{K}{(1 + T_1 s)(1 + T_2 s)}.$$

The system has the following step response:



Parameter dialog:



For the simulation the initial value $y(t = 0)$ and the initial gradient $\dot{y}(t = 0)$ have to be specified.

Restrictions: $T_1, T_2 > 0$

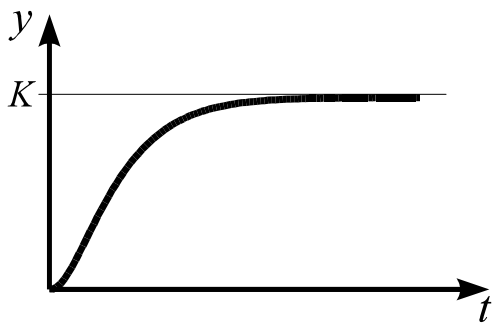
PT_n-Element

Typename: PTN

Function: Delay element of n-th order with the gain K and n equal time constants T. It has the transfer function

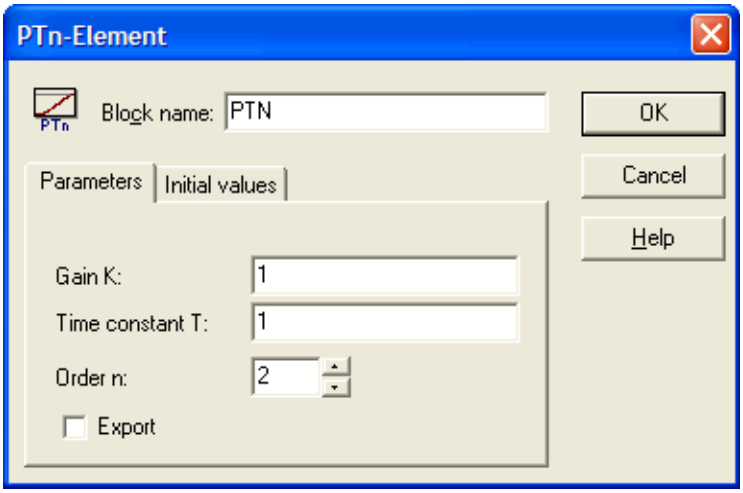
$$G(s) = \frac{K}{(1+Ts)^n}.$$

and the following step response:



For constant values of T the step response progresses increasingly flatter with rising value of n.

Parameter dialog:



Restrictions: $T > 0, \quad 1 \le n \le 8$

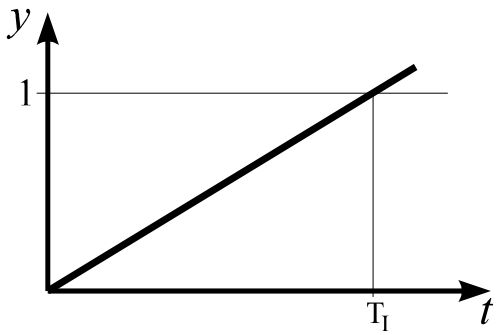
Limited integrator

Typename: I

Function: Limited integrator with the integration time T_I. Within the linear working range the system has the following transfer function

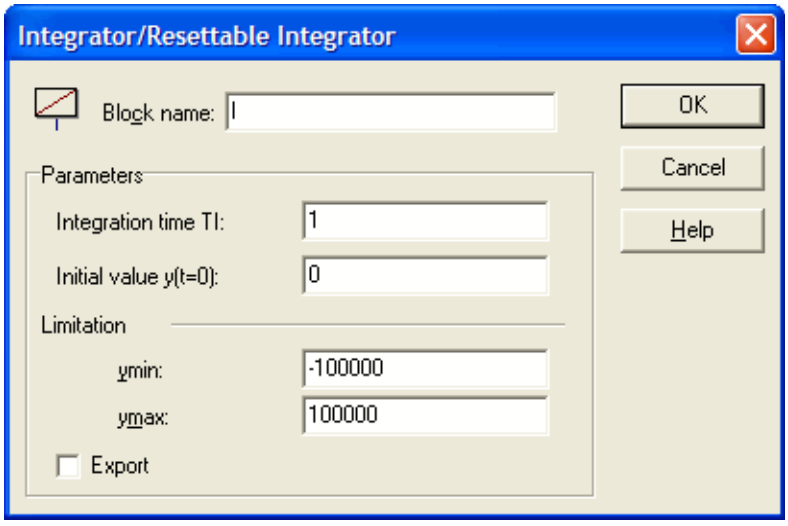
$$G(s) = \frac{1}{T_I s}$$

and the following step response:



The output $y(t)$ of the integrator is limited to the range of values $y_{min} \le y(t) \le y_{max}$ by an anti-windup-mode. This stops the integration as long as the output variable is at the upper limit y_{max} and the input variable of the integrator is positive resp. the output variable is at the lower limit y_{min} and the input variable is negative. At the change of sign of the input variable the output variable will leave the limit directly.

Parameter dialog:



For the simulation the initial value $y(t=0)$ has to be specified.

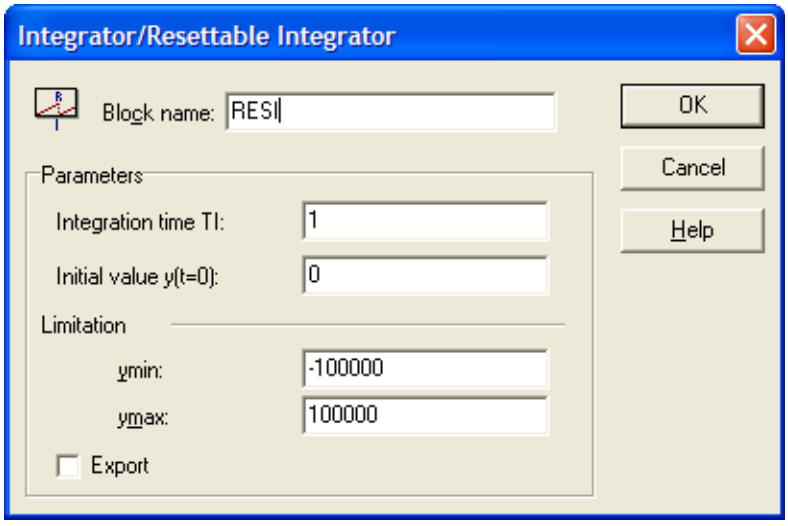
Restrictions: $T_I > 0$

Resettable integrator

Typename: RESI

Function: This block corresponds to the standard integrator but the output variable can be reset to the initial value by a positive edge at the control input R at any time during the simulation.

Parameter dialog:



Restrictions:

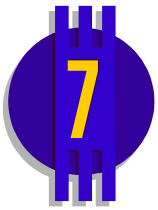
$T_I > 0$

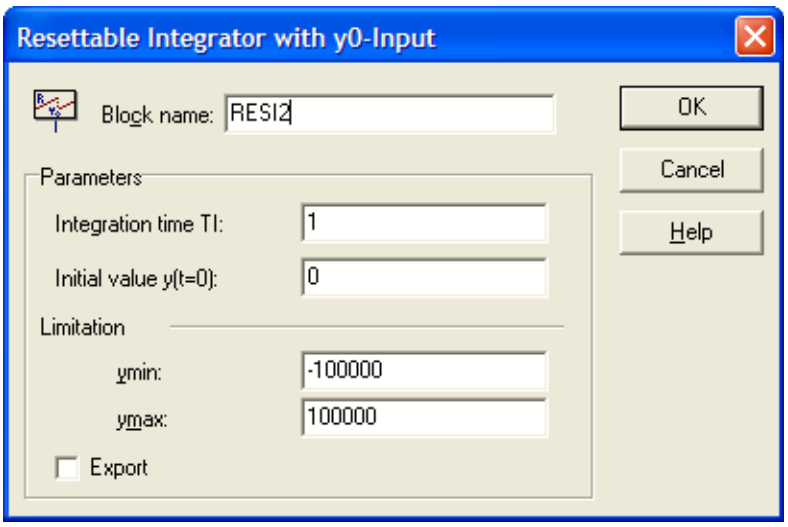
Resettable integrator with y_0-input

Typename: RESI2

Function: This block type corresponds to the resettable integrator, but by a positive edge at block input R the output is reset to the current value of input y0.

Parameter dialog:





Restrictions:

$T_I > 0$

Differentiator

Typename: D

Function: Ideal differentiator with the differentiatial time T_D , the transfer function

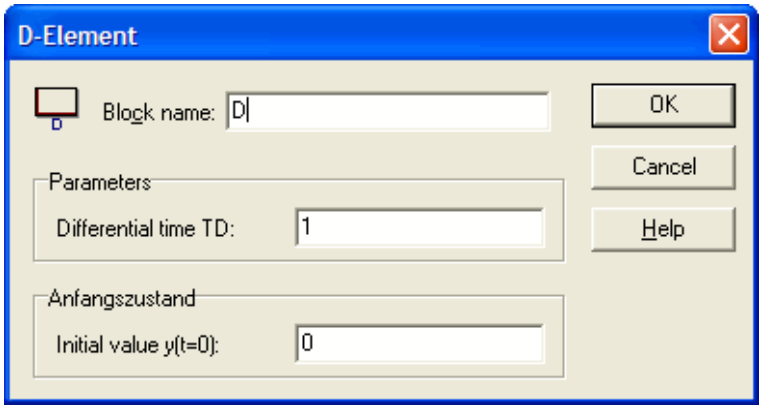
$$G(s) = T_D s$$

and the following step response:

At t=0 the output variable theoretically has an infinite value. Effected by the finite step size ΔT an input step of amplitude Δx leads to an output value limited to

$$y_{\max} = \frac{\Delta x}{\Delta T}.$$

Parameter dialog:



For the simulation the initial value $y(t=0)$ of the differentiator has to be specified.

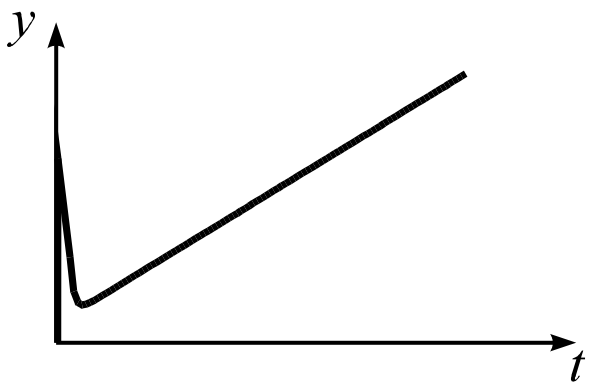
PID-Controller

Typename: PID

Function: PID-controller with the gain K_R, the reset time T_N and the rate time T_V. The D-component of the controller has an additional PT_1-element (delayed differentiation). The output variable $y(t)$ is limited to the range of value $y_{\min} \le y(t) \le y_{\max}$ by an anti-windup-mode. Within the linear working range the PID-controller has the transfer function

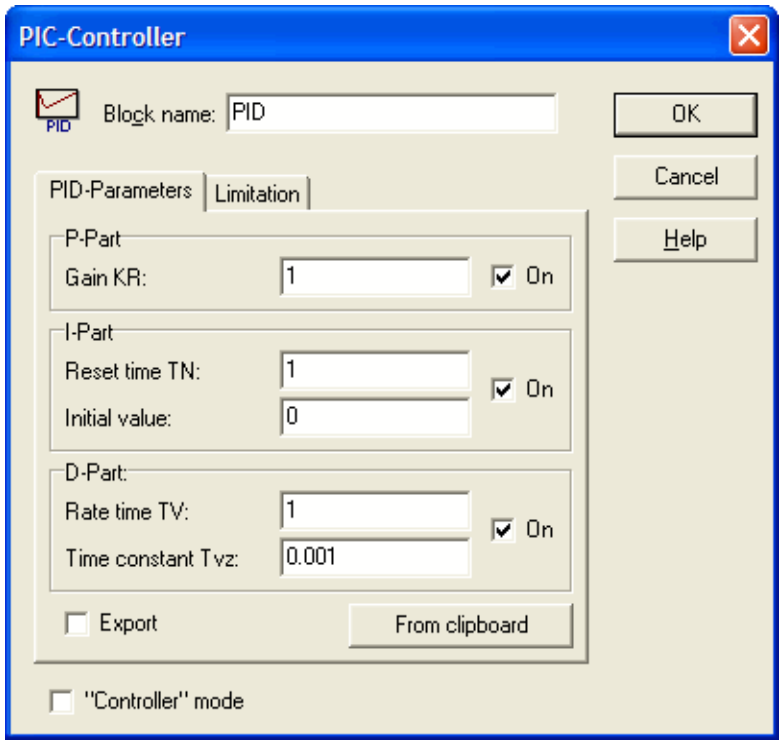
$$G(s) = K_R\left(1 + \frac{1}{T_N s} + \frac{T_V s}{1 + T_{Vz} s}\right)$$

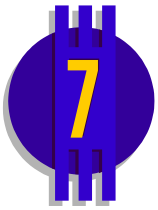
and the following step response:



P-, I- and D- components can be deactivated separately just as the limitation. For the simulation the initial value of the integrator has to be specified.

Parameter dialog:





If the *Controller mode* is activated, the block has two inputs (set point *w* and feedback variable *r*) instead of the error variable *e*. In this case the error variable is calculated internally as $e = w - r$.

Restrictions:

$$T_N,\ T_{Vz} > 0,\quad T_V \geq 0$$

Adaptive PID-controller with controllable limitation

Typename: ADAPID

Function: In its way of operation the block corresponds to the standard PID-controller but the parameters K_R, T_N and T_V can be modified during the simulation by the control inputs P, I and D. The modification can be multiplicative or additive. The current values for the parameters result from the basic values K_{R0}, T_{N0} and T_{V0} as follows

Operating mode *multiplicative*:

$$K_R = K_{R0} x_P(t)$$
$$T_N = T_{N0} x_I(t)$$
$$T_V = T_{V0} x_D(t)$$

Operating mode *additive*:

$$K_R = K_{R0} + x_P(t)$$
$$T_N = T_{N0} + x_I(t)$$
$$T_V = T_{V0} + x_D(t)$$

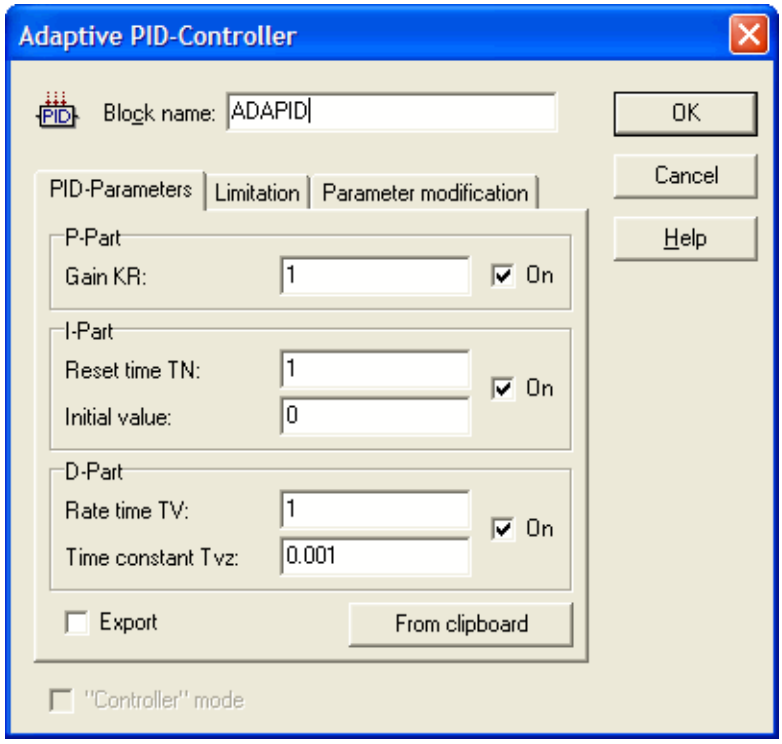
$x_P(t), x_I(t)$ and $x_D(t)$ are the signals at the control inputs P, I and D. In the operating mode 'multiplicative' open control inputs will be set to 1, in the operating mode 'additive' to 0.

With the control input S additionally the limitation of the controller can be influenced from outside. If the control input is open, the controller works with the fixed limitation of the output variable to the range $y_{min} \leq y(t) \leq y_{max}$. If the control input is connected, the real limitation results from the multiplication of the values which are set by the dialog and the signal $x_S(t)$ at the control input:

$$y_{min,\,akt}(t) = y_{min} \cdot x_S(t)$$
$$y_{max,\,akt}(t) = y_{max} \cdot x_S(t)$$

Of course the limitation has to be activated before!

Parameter dialog:



Restrictions:

$$T_N,\ T_{Vz} > 0,\ \ T_V \geq 0$$

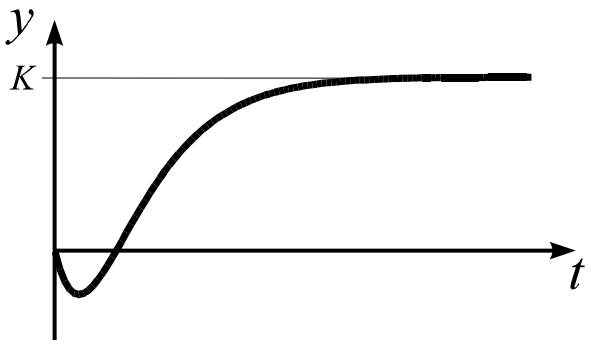
All-pass element type I

Typename: ALLPASS_1

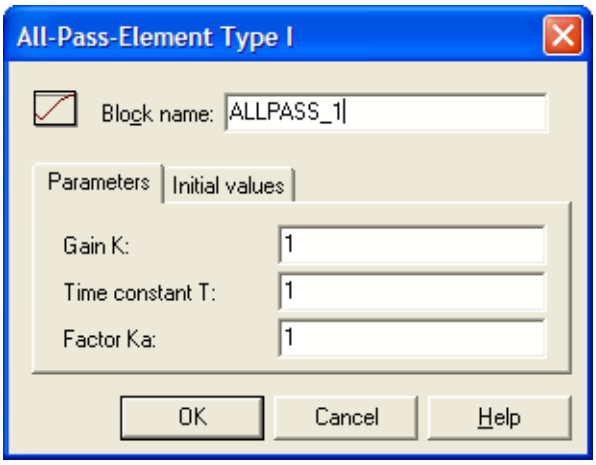
Function: All-pass system of second order (non-oscillating) with the transfer function

$$G(s) = K\frac{1 - K_a Ts}{(1 + K_a Ts)(1 + Ts)}$$

and the following step response:



Parameter dialog:



For the simulation the initial values of the system have to be specified.

Restrictions:

$T,\ K_a > 0$

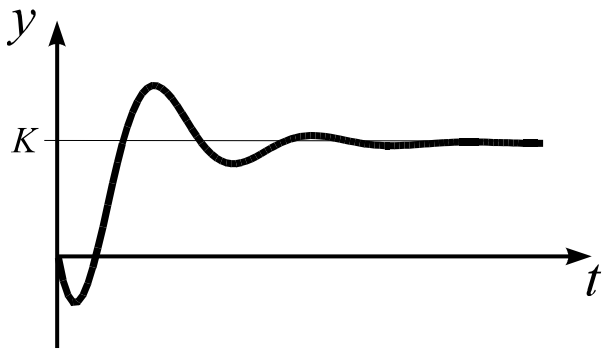
All-pass element type II

Typename: ALLPASS_2

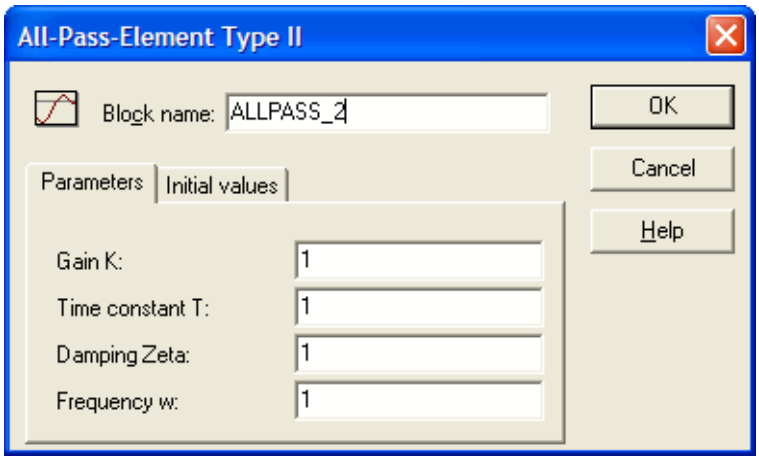
Function: All-pass system of second order (oscillating) with the transfer function

$$G(s) = K\frac{1 - Ts}{\left(\frac{s}{\omega}\right)^2 + 2\frac{\zeta}{\omega}s + 1}$$

and the following step response:



Parameter dialog:



For the simulation the initial values of the system have to be specified.

Restrictions:

$T,\ \omega > 0,\ \zeta \geq 0$

Dead time element

Typename: DEADTIME

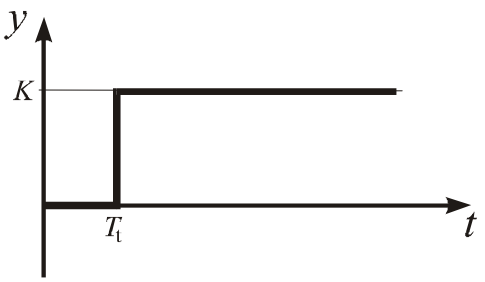
Function: The dead time element delays the input variable $x(t)$ by the dead time T_t according to

$$y(t) = \begin{cases} y_0 & \text{für } t < T_t \\ K\,x(t - T_t) & \text{für } t \geq T_t \end{cases}$$

and has the transfer function

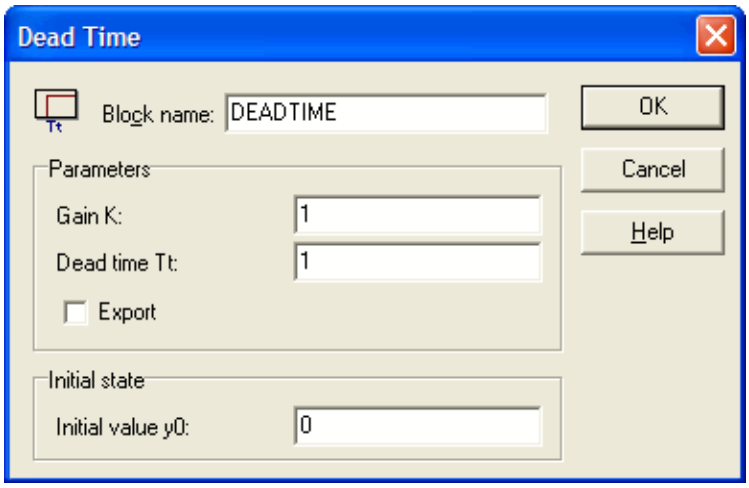
$$G(s) = Ke^{-T_t s}$$

and the following step response:



The internal buffer of the dead time element can contain as many elements as desired. The dead time should be a multiple of the simulation step size ΔT.

Parameter dialog:



Restrictions:

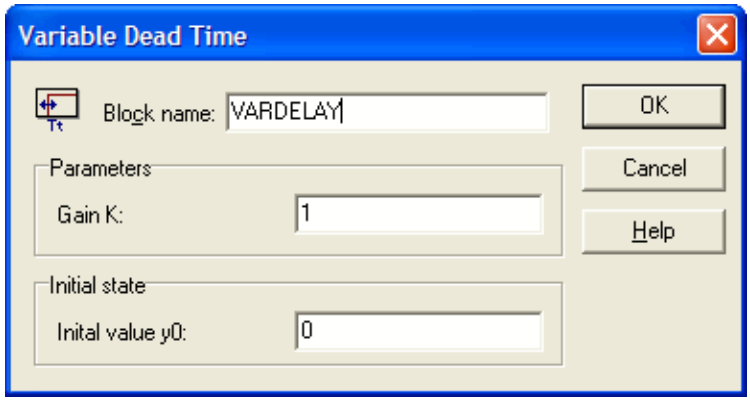
$T_t \geq 0$

Variable dead time

Typename: VARDELAY

Function: This block type realizes a dead time element with variable dead time; it can be used for modelling systems with a variable delay (e. g. in process technology). The signal at the block input I is delayed by the dead time delivered by block input Tt (see also block type DEADTIME). The internal buffer of the block can contain any number of elements.

Parameter dialog:



Restrictions: The value of the dead time is internally limited to the value ΔT of the simulation step size.

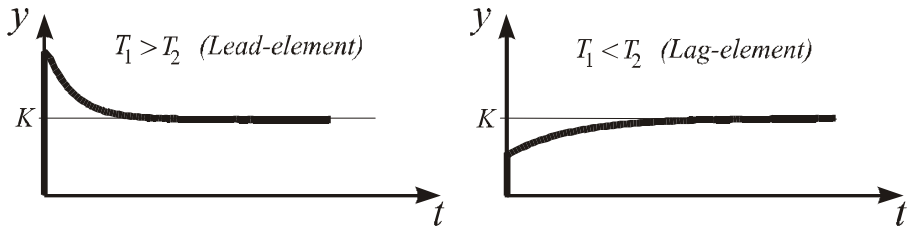
Lead/Lag-element

Typename: LEADLAG

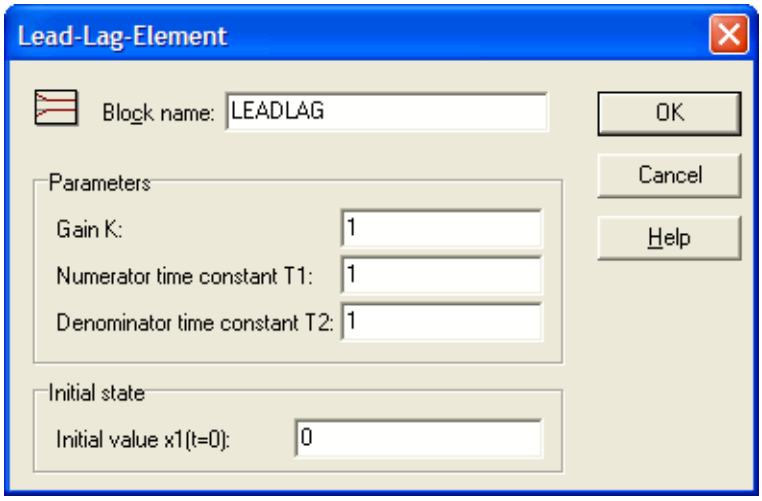
Function: Rational element of first order with the transfer function

$$G(s) = K\frac{1+T_1 s}{1+T_2 s}$$

and the following step responses:



Parameter dialog:



For the simulation the initial value $x_1(t=0)$ has to be specified.

Restrictions:

$T_1 \geq 0, \quad T_2 > 0$

Rate element (DT1-Element)

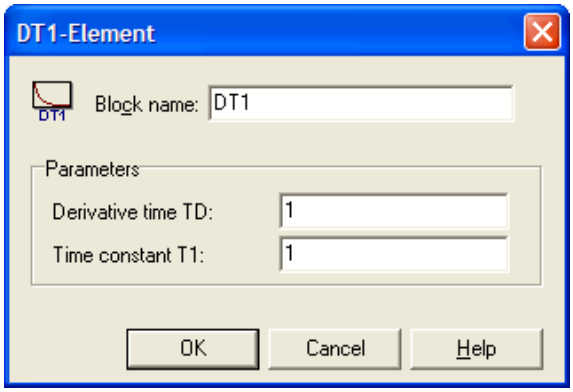
Typename: DT1

Function: Rational element of first order (delayed differentiator) with the transfer function

$$G(s) = \frac{T_D s}{1 + T_1 s}$$

and the following step response:

Parameter dialog:



Restrictions:

T_D, $T_1 > 0$

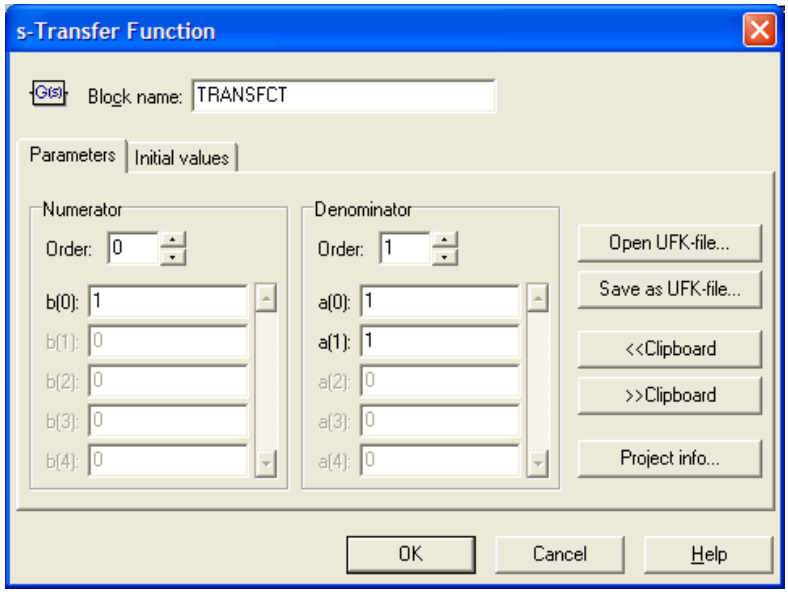
s-Transfer function

Typename: TRANSFCT

Function: Parametrizable s-transfer function of the form

$$G(s) = \frac{b_m s^m + b_{m-1} s^{m-1} + \ldots + b_1 s + b_0}{a_n s^n + a_{n-1} s^{n-1} + \ldots + a_1 s + a_0}, \quad m \le n \ .$$

Parameter dialog:



Alternatively the transfer function can be entered by the keyboard or from the clipboard resp. a UFK-file.

Restrictions: $m \le n,\ n \le 8$

$a_n \neq 0$

Differential equation system

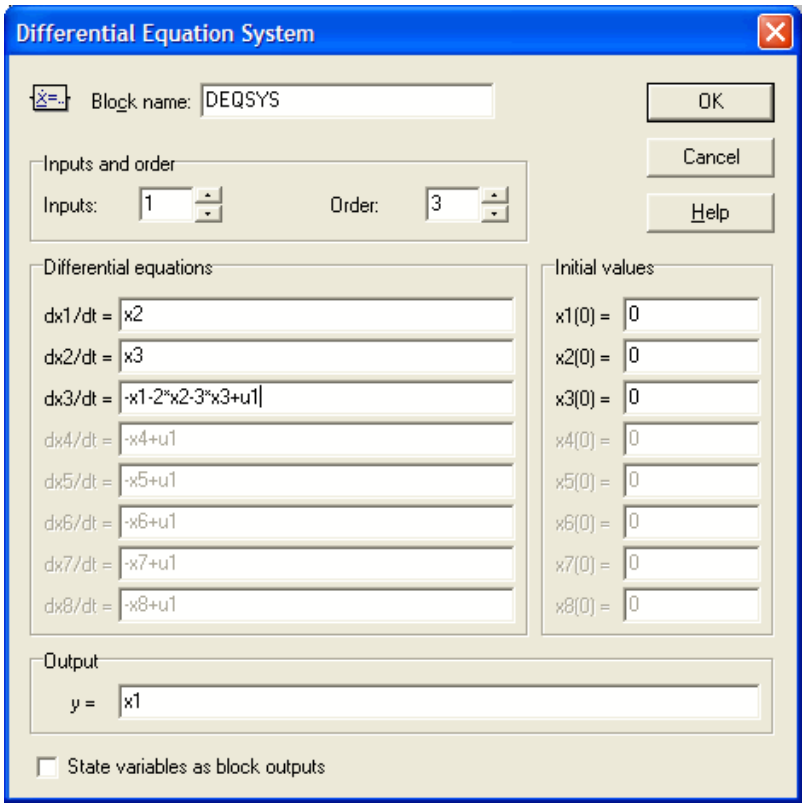
Typename: DEQSYS

Function: Parametrizable differential equation system of the form

$$\begin{aligned} \dot{\underline{x}} &= F(\underline{x}, \underline{u}) \\ y &= g(\underline{x}, \underline{u}) \end{aligned} .$$

The differential equation system can be linear or nonlinear and is interpreted by a function parser. $\underline{x}(t)$ is the state vector of the system (max. order: 8), $\underline{u}(t)$ is the input vector and $y(t)$ the output variable. The initial state $\underline{x}(t = 0)$ can also be specified.

Parameter dialog:



The state variables are named *x1*, *x2*, ..., the input variables *u1*, *u2*,...The dialog above contains the – in this case linear – differential equation system

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = x_3$$

$$\dot{x}_3 = -x_1 - 2x_2 - 3x_3 + u_1$$

$$y = x_1$$

with the initial state $\underline{x}(0) = \underline{0}$.

MIMO-State space model

Typename: SSM

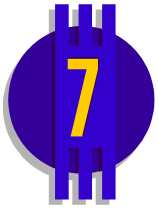
Function: Linear multi-input multi-output state space model represented as

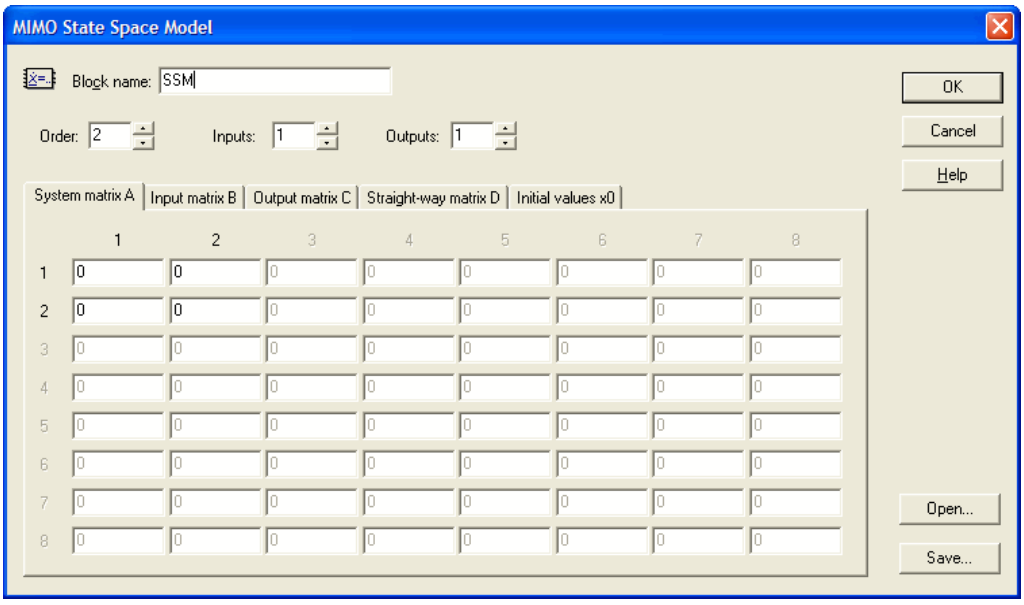
$$\dot{\underline{x}} = \underline{A} \cdot \underline{x} + \underline{B} \cdot \underline{u}$$

$$\underline{y} = \underline{C} \cdot \underline{x} + \underline{D} \cdot \underline{u}$$

$\underline{x}(t)$ is the state vector of the system (maximum order: 8), $\underline{u}(t)$ the input vector and $\underline{y}(t)$ the output vector. The initial state vector $\underline{x}(t=0)$ can be specified. $\underline{A}$ is the system matrix, $\underline{B}$ the input matrix, $\underline{C}$ the output matrix and $\underline{D}$ the straight-way matrix.

Parameter dialog:



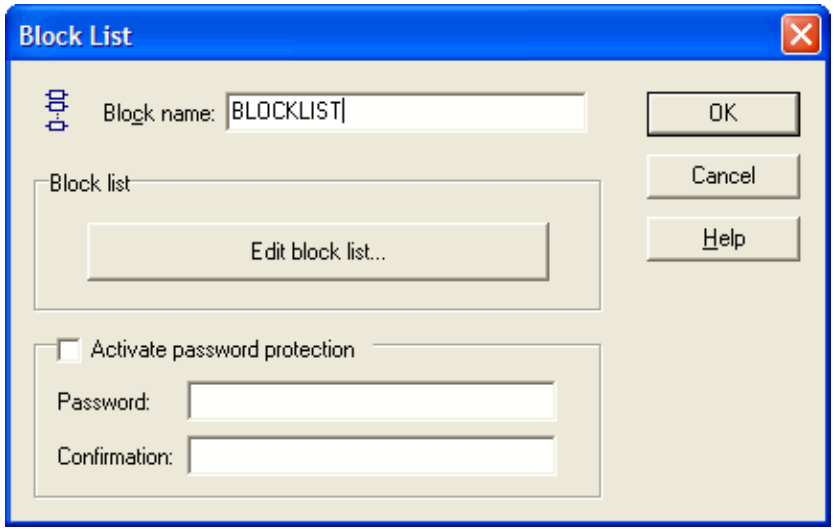


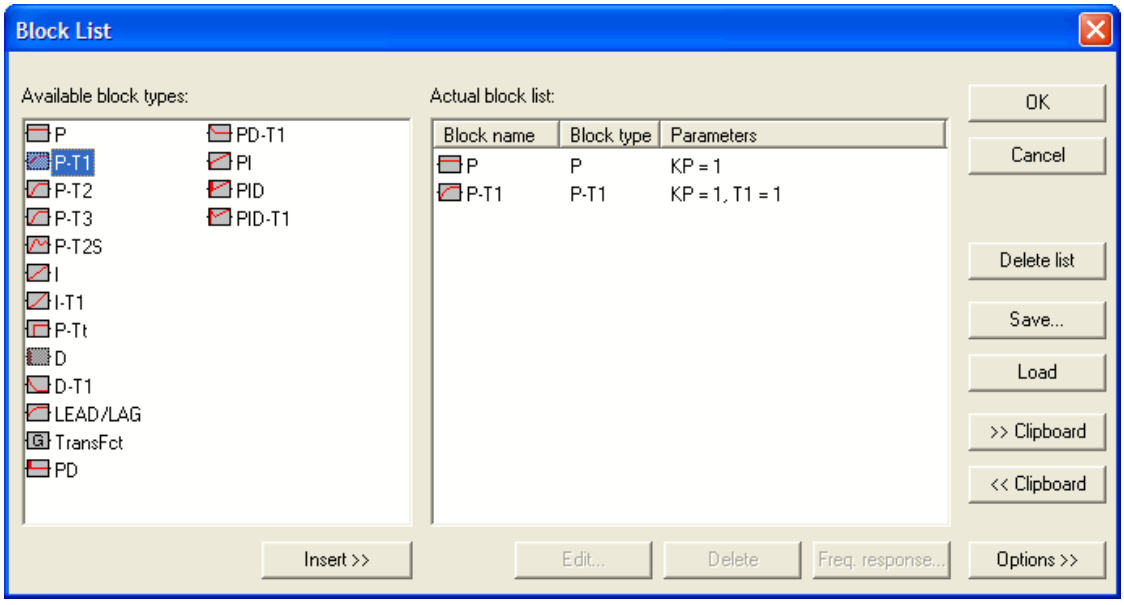
Block list

Typename: BLOCKLIST

Function: This block type represents a list of linear standard transfer elements in serial connection. The list can be protected via a password; in this case the internal structure of the list can also be displayed and modified after entering the valid password. For details concerning the configuration of the block list refer to chaper 2 *Basics*.

Parameter dialog:





Unit delay

Typename: UNITDELAY

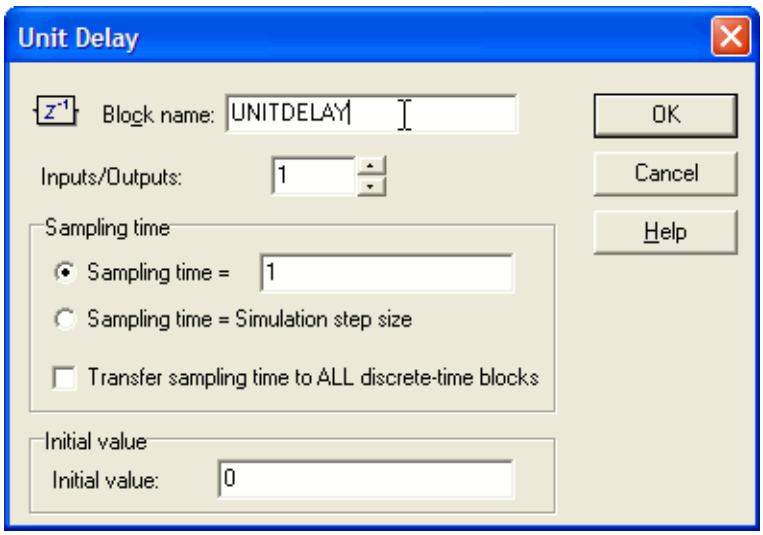
Function: This block represents a sample and hold element with a delay of a sampling period *T*. Thus it has the z-transfer function

$$G(z) = \frac{1}{z}.$$

The sampling time *T* and the initial value can be defined by the user. The block can have up to 50 in- resp. outputs.

This block type can efficiently be used to avoid an *algebraic loop* and get a delay of always accurately one step in the feedback. The simulation step size can be linked to the sampling time so that it is not necessary to adjust the sampling time for each modification of the simulation step size.

Parameter dialog:



If the option *Transfer sampling time to ALL discrete-time blocks* is activated, the specified sampling time of the block is automatically applied to all other discrete-time blocks (i. e. UNITDELAY or ZTRANSFCT blocks) after the dialog was closed.

The sampling time should be a multiple of the simulation step size, otherwise you will get a warning at the start of the simulation.

Restrictions:

$T > 0$

z-Transfer function

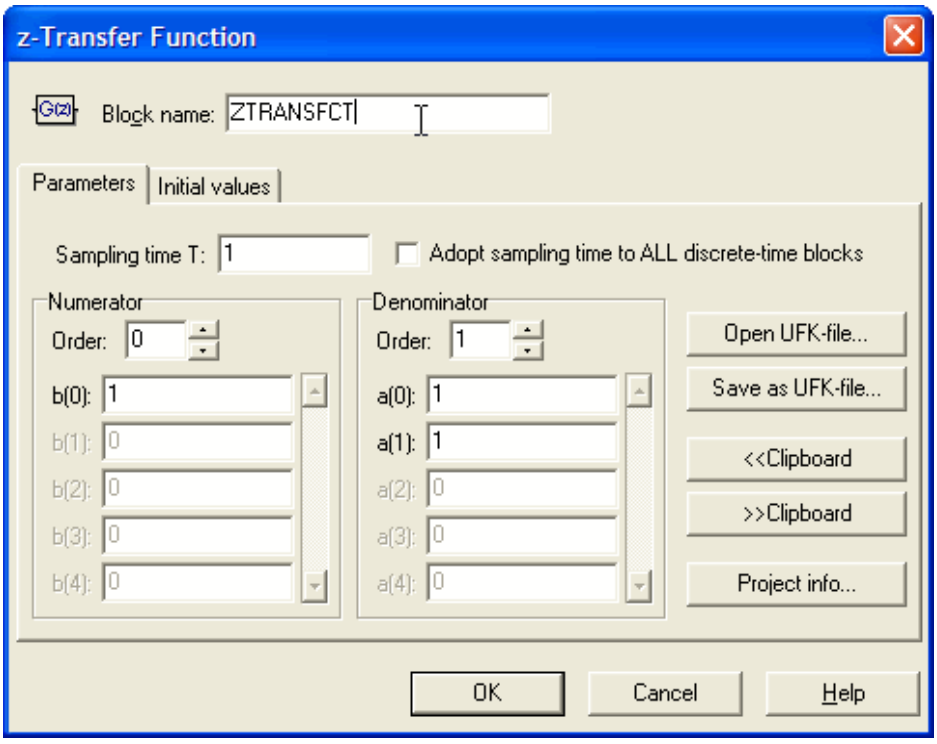
Typename: ZTRANSFCT

Function: Parametrizable z-transfer function of the form

$$H(z) = \frac{b_m z^m + b_{m-1} z^{m-1} + \ldots + b_1 z + b_0}{a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0}, \quad m \le n \ .$$

The sampling time T can be specified.

Parameter dialog:



The transfer function can alternatively be entered by the keyboard or the clipboard resp. a UFK-file. The file format corresponds to the *s*-transfer function, but instead of the dead time you have to specify the sampling time.

If the option *Transfer sampling time to ALL discrete-time blocks* is activated, the specified sampling time of the block is automatically applied to all other discrete-time blocks (i. e. UNITDELAY or ZTRANSFCT blocks) after the dialog was closed.

The sampling time should be a multiple of the simulation step size, otherwise you will get a warning at the start of the simulation.

Restrictions:

$$m \le n,\ n \le 8,\ a_n \neq 0$$

Static blocks

Linear characteristic with offset (variable scaler)

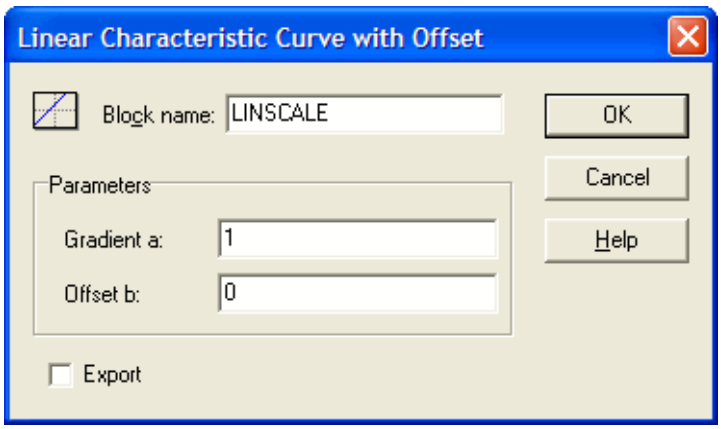
Typename: LINSCALE

Function: This block represents a linear characteristic curve with offset of the form

$$y(t) = ax(t) + b\,.$$

x is the input variable of the block, y the output variable, a the gain and b the offset. This block can therefore be used e. g. to scale a variable.

Parameter dialog:



Restrictions: none

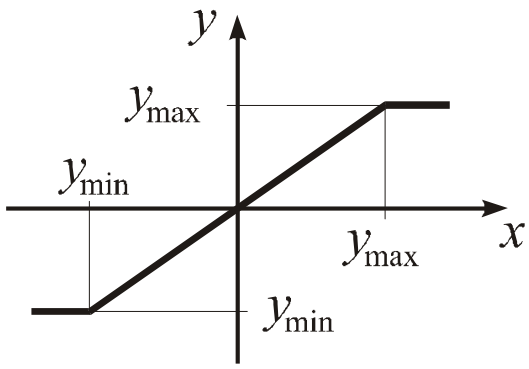
Limiter

Typename: LIMITER

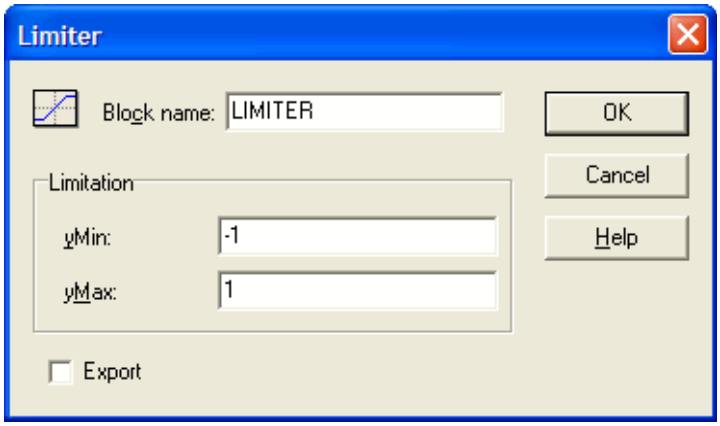
Function: The limiting characteristic curve (saturation characteristic curve) limits the output variable to the range of value $y_{\min} \le y(t) \le y_{\max}$ according to the instruction

$$y(t) = \begin{cases} y_{\min} & for \quad x(t) < y_{\min} \\ x(t) & for \quad y_{\min} \le x(t) \le y_{\max} \\ y_{\max} & for \quad x(t) > y_{\max} \end{cases}.$$

Within the linear working range the characteristic curve element has the gain 1. Thus the characteristic curve has the following shape:



Parameter dialog:



Restrictions:

$y_{\min} \le y_{\max}$

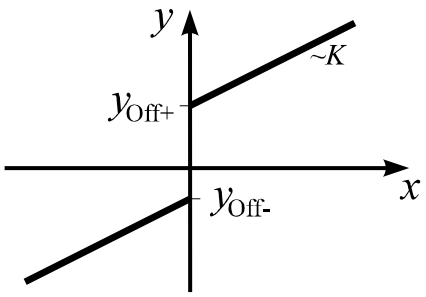
Preload characteristic curve

Typename: PRELOAD

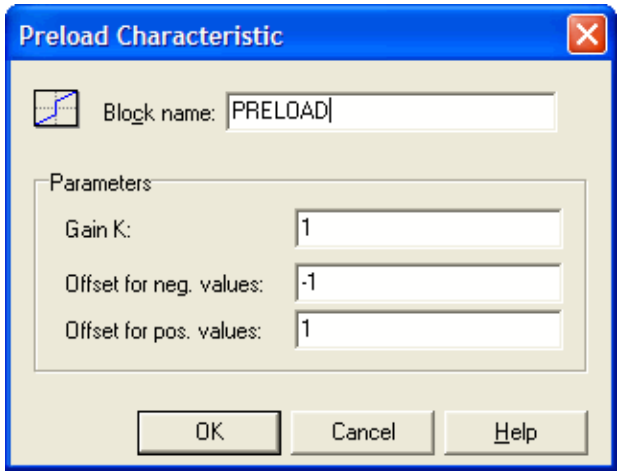
Function: The preload characteristic curve is piecewise linear with an offset at the position $x = 0$. It is characterized by the relation

$$y(t) = \begin{cases} y_{Off-} + K x(t) & for \quad x(t) \le 0 \\ y_{Off+} + K x(t) & for \quad x(t) > 0 \end{cases}$$

and the following curve:



Parameter dialog:



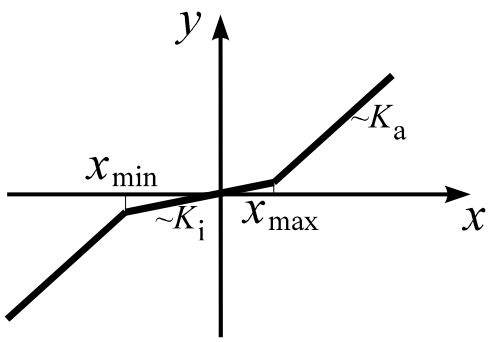
Dead zone

Typename: DEADZONE

Function: This system type represents a reduced gain from K_a to K_i within a dead zone $[\, x_{min}, x_{max} \,]$ of the input variable $x(t)$ according to the equation

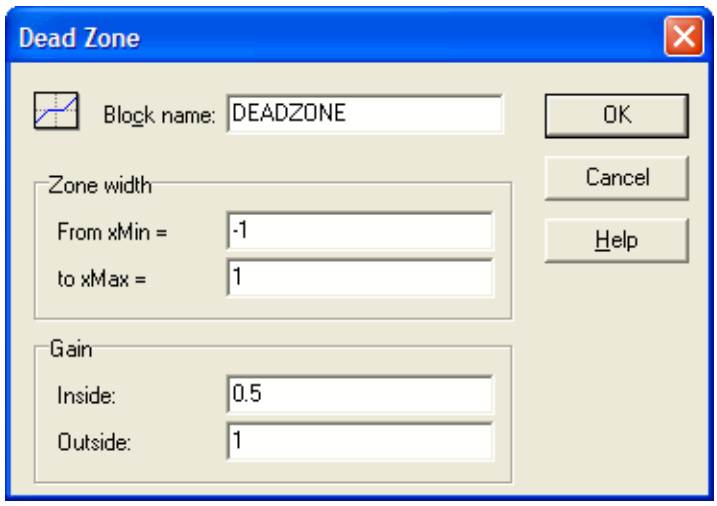
$$y(t) = \begin{cases} K_a x(t) + x_{min}(K_i - K_a) & for \quad x(t) < x_{min} \\ K_i x(t) & for \quad x_{min} \le x(t) \le x_{max} \\ K_a x(t) + x_{max}(K_i - K_a) & for \quad x(t) > x_{max} \end{cases} .$$

The element has the following characteristic curve:



For $K_i = 0$ you will get a characteristic curve with a real dead zone.

Parameter dialog:



Restrictions:

$$x_{min} \le x_{max}$$

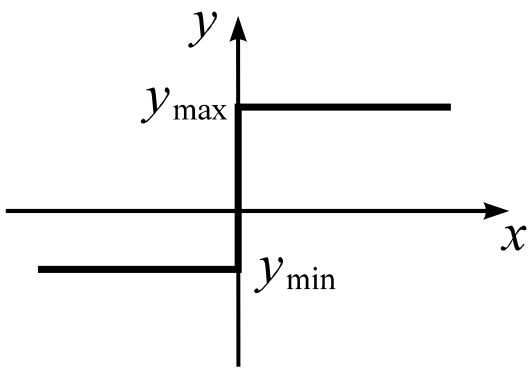
Two-point characteristic curve

Typename: 2POINT

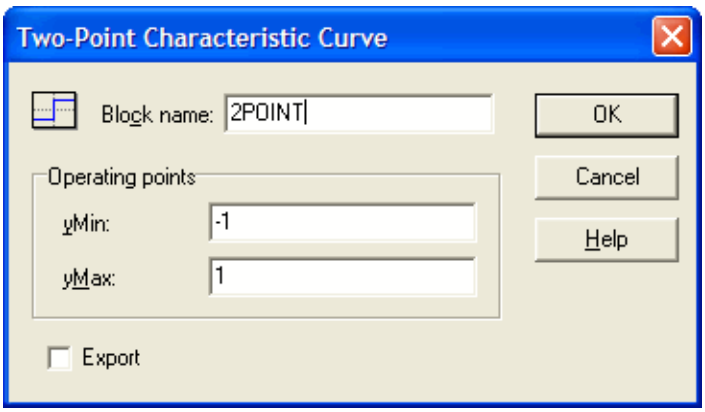
Function: The characteristic curve follows the relation

$$y(t) = \begin{cases} y_{\min} \; for \; x(t) < 0 \\ y_{\max} \; for \; x(t) \geq 0 \end{cases}$$

and has the following shape:



Parameter dialog:



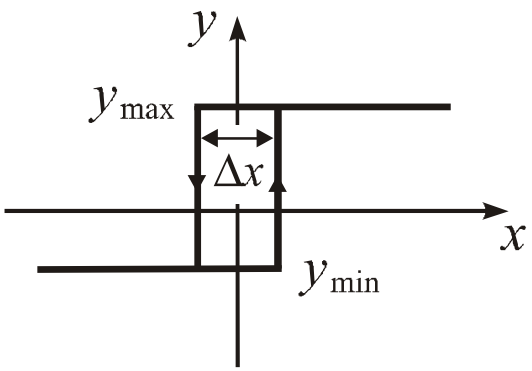
Restrictions:

$y_{\min} \leq y_{\max}$

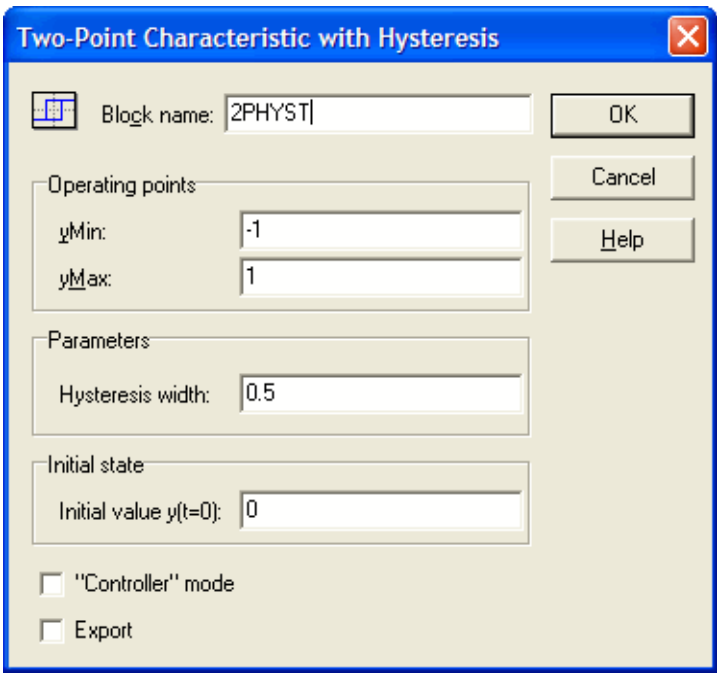
Two-point characteristic curve with hysteresis

Typename: 2PHYST

Function: This block represents a combination of a two-point element with the operating points $y_{\min}$ and $y_{\max}$ and a hysteresis element with a hysteresis width of Δx. The diagram below shows the corresponding curve.



Parameter dialog:



If the *Controller mode* is activated, the block has two inputs (set point *w* and feedback variable *r*) instead of the error variable *e*. In this case the error variable is calculated internally as $e = w - r$.

Restrictions:

$$y_{min} \leq y_{max}$$

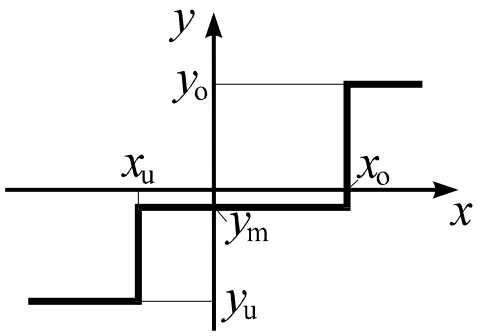
Three-point characteristic curve

Typename: 3POINT

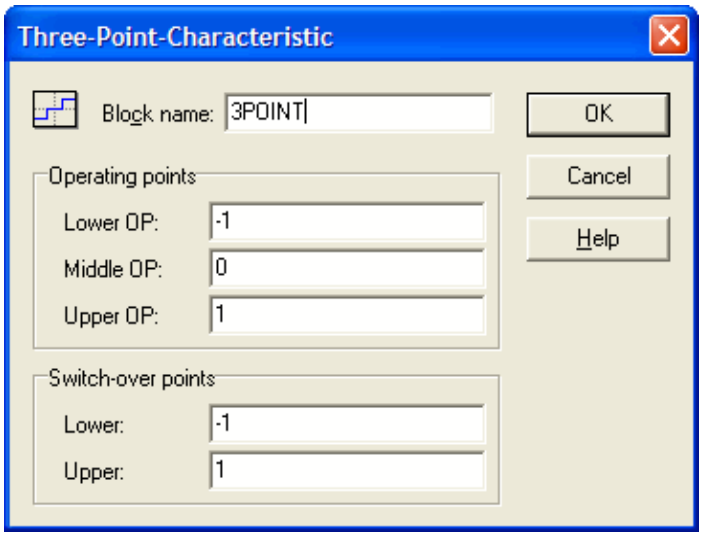
Function: This characteristic curve follows the relation

$$y(t) = \begin{cases} y_u & for \quad x(t) < x_u \\ y_m & for \quad x_u \le x(t) \le x_o \\ y_o & for \quad x(t) > x_o \end{cases}$$

and has the following shape:



Parameter dialog:



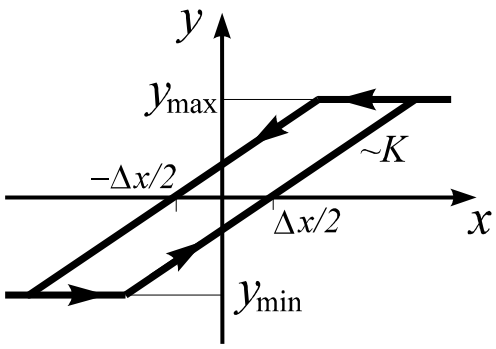
Restrictions:

$$x_u \le x_o, y_u \le y_m \le y_o$$

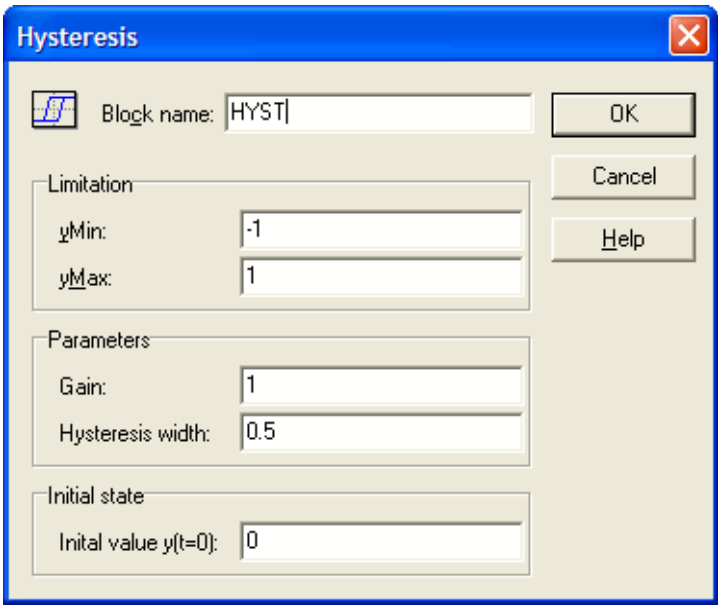
Hysteresis characteristic curve

Typename: HYST

Function: The hysteresis-characteristic curve is characterized by the gain K, the hysteresis width Δx and the limitation y_{min} resp. y_{max} and has the following shape:



Parameter dialog:



The initial value $y(t=0)$ has to be specified.

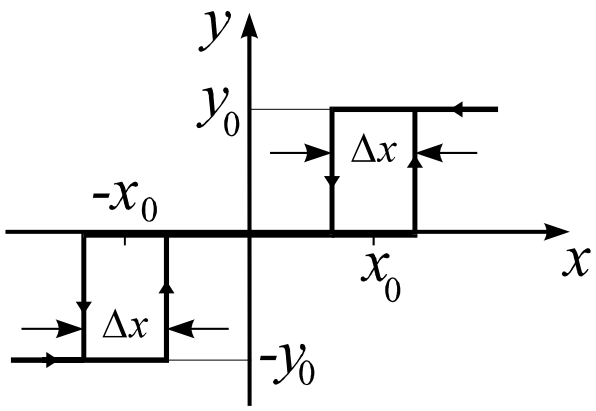
Restrictions:

$K, \Delta x > 0,\ y_{\min} \le y_{\max}$

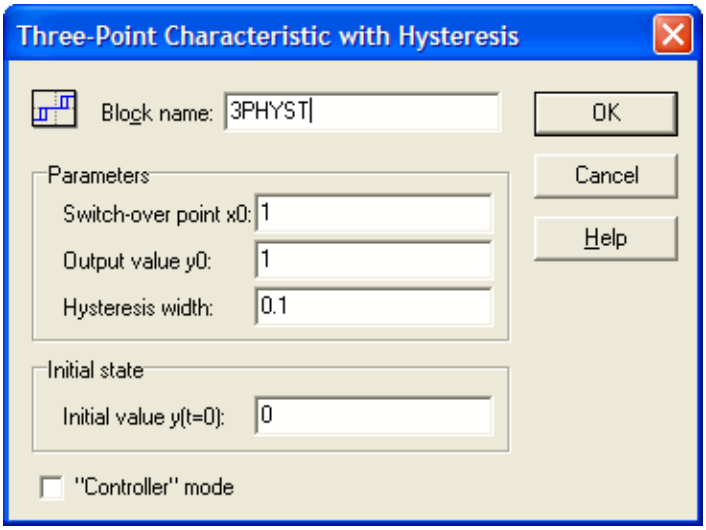
Three-point element with hysteresis

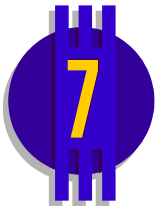
Typename: 3PHYST

Function: Realizes a combination of a symmetrical three-point-element with the switching points x_0 and $-x_0$ and the operating points y_0 and $-y_0$ and a hysteresis element with the hysteresis width Δx. The characteristic curve has the following shape:



Parameter dialog:





If the *Controller mode* is activated, the block has two inputs (set point *w* and feedback variable *r*) instead of the error variable *e*. In this case the error variable is calculated internally as $e = w - r$.

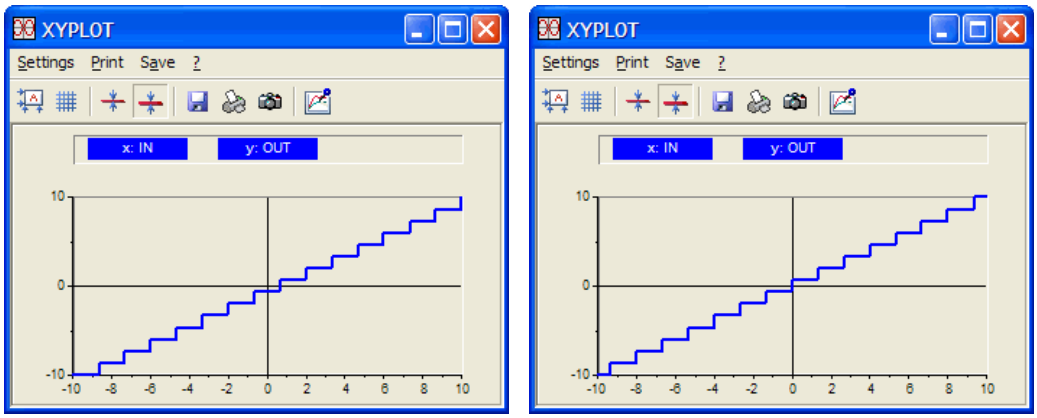
Restrictions:

$$x_0, y_0, \Delta x > 0$$

Quantizer

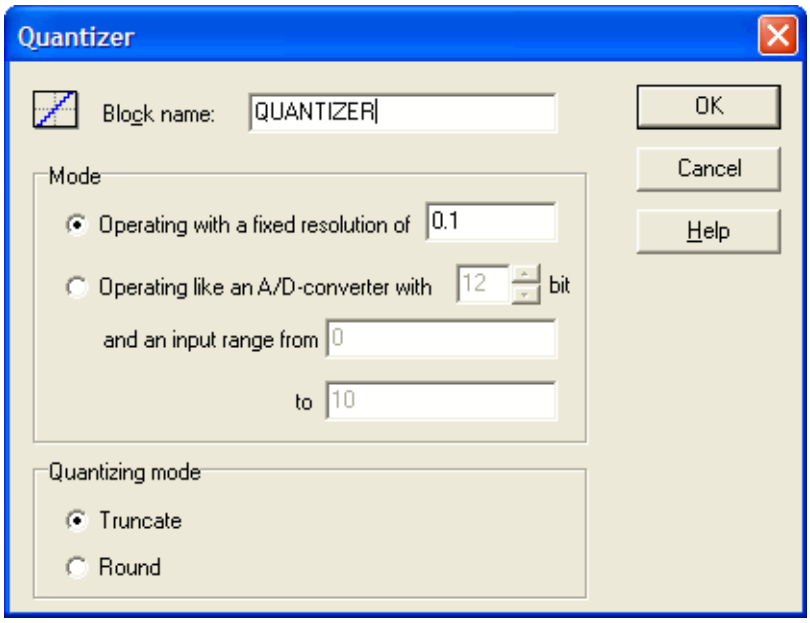
Typename: QUANTIZER

Function: This block type realizes a quantizer which can work with a fixed resolution or like an *n* Bit A/D converter. The quantization can be executed by truncating the decimal places or by rounding.



Example: Characteristic curve of a quantizer working like a 4 Bit A/D-converter (range -10 ... +10) by truncating (left) resp. rounding (right)

Parameter dialog:



User-defined characteristic curve

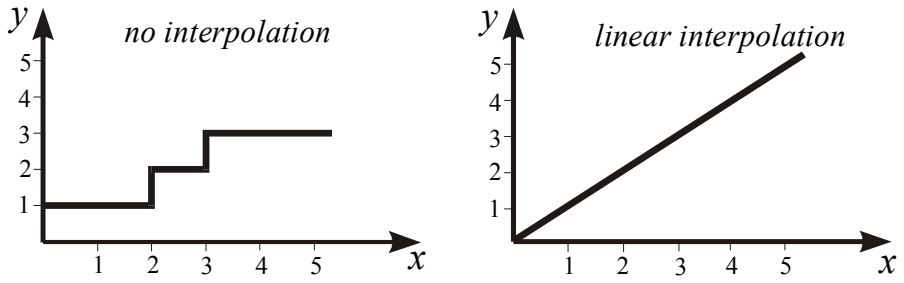
Typename: CHARCURVE

Function: Allows the definition of a characteristic curve $y(x)$ by n pairs of values (x_i, y_i). The base points x_i can be sorted in any way (i. e. not necessarily equidistant) but in ascending order. Between the values a linear interpolation can be executed. Otherwise the last valid value will be kept until a new value appears. In this case you will get a stepwise constant characteristic curve.

Example: The characteristic curve is defined by the three pairs of values:

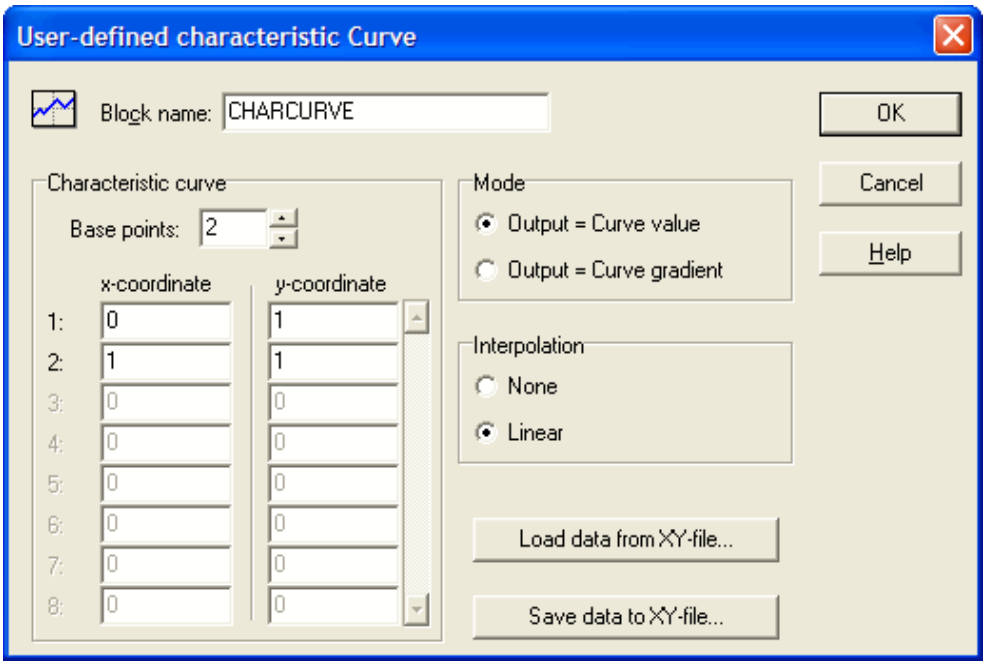
$$(1, 1), (2, 2), (3, 3).$$

Then you will get the following characteristic curve:



Instead of a *y*-coordinate value itself also the *Gradient* $\mathrm{d}y / \mathrm{d}x$ of the characteristic curve for the value at the input can be generated. In this case an additional interpolation will be without effect. For the characteristic curve in the example you will get e. g. a constant gradient of 1.

Parameter dialog:



Restrictions:

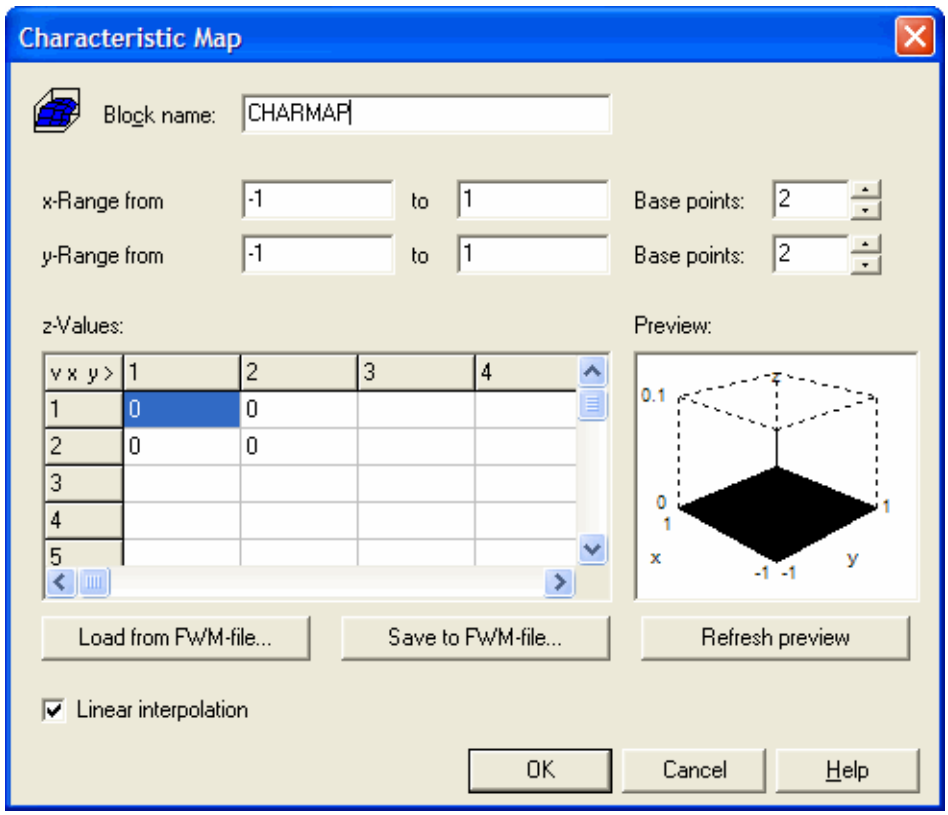
$$2 \le n \le 1000$$

User-defined characteristic map

Typename: CHARMAP

Function: This block type allow the specification of a characteristic map $z(x, y)$ by a $n \times m$ base point matrix ($n, m \leq 100$). The matrix can be entered via keyboard or read from a FWM file. By default a linear interpolation between the base points is executed; this option can be deactivated if desired. If the interpolation is activated, values out of the base point matrix range are calculated by extrapolation. If the current input values are out of range, block output *Ex* gets HIGH level.

Parameter dialog:



Controllers

PID-Controller

see block group *Dynamic blocks*

Adaptive PID-Controller with controllable limitation

see block group *Dynamic blocks*

Industry-PID-Controller

see block group *Action blocks*

Two-point element

see block group *Static blocks*

Two-point element with hysteresis

see block group *Static blocks*

Three-point element

see block group *Static blocks*

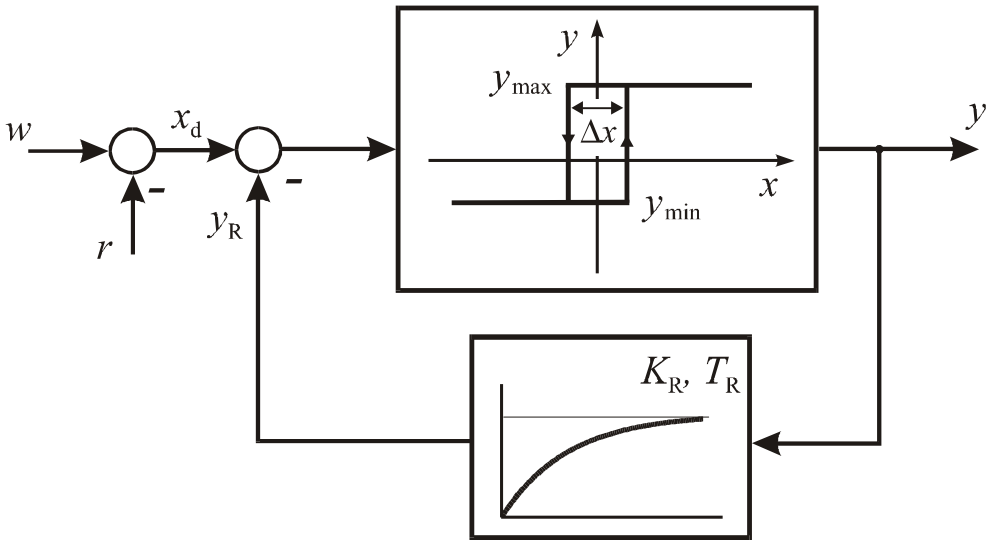
Three-point element with hysteresis

see block group *Static blocks*

Two-point controller with PT_1-feedback

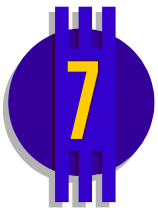
Typename: ZP-PD

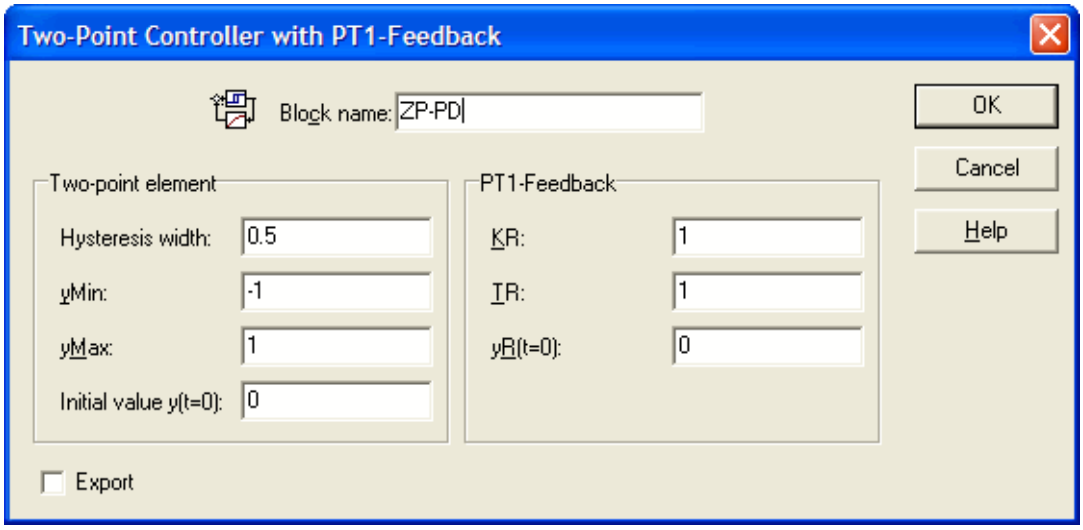
Function: This block type represents a two-point controller with the operating points y_{min} and y_{max} and a hysteresis width of Δx which has an internal PT_1-feedback (gain K_R, time constant T_R) of the manipulated variable. If the controller is parameterized properly it works similar to a continuous PD-controller. The graphic below shows the structure of the controller.

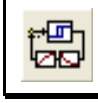


The block outputs *y*+ and *y*- (second resp. third output) can be used as switching outputs; they get HIGH-level if the two-point element is at the upper operating point (*y*+) resp. at the lower operating point (*y*-).

Parameter dialog:



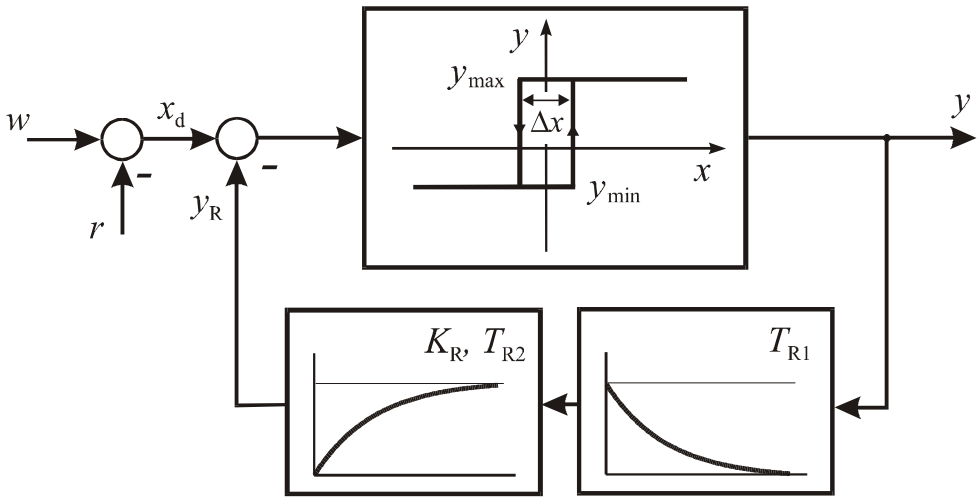




Two-point controller with delayed feedback

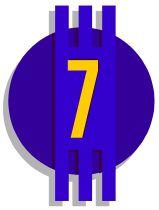
Typename: ZP-PID

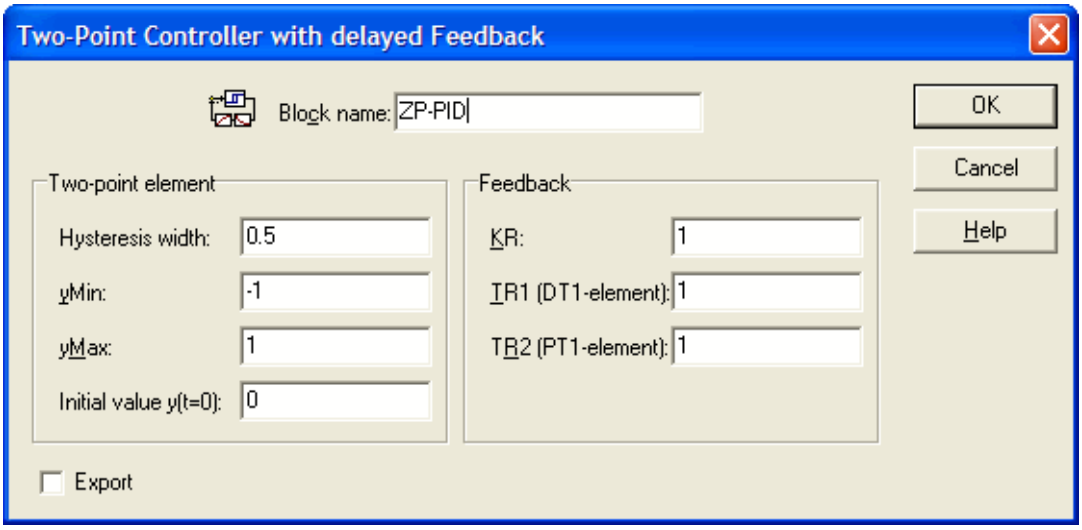
Function: This block type represents a two-point controller with the operating points y_{min} and y_{max} and a hysteresis width of Δx which has an internal feedback of the manipulated variable. This feedback is a serial connection of a DT_1-element (time constant T_{R1}) and a PT_1-element (gain K_R, time constant T_{R2}). If the controller is parameterized properly it works similar to a continuous PID-controller. The graphic below shows the structure of the controller.



The block outputs *y*+ and *y*- (second resp. third output) can be used as switching outputs; they get HIGH-level if the two-point element is at the upper operating point (*y*+) resp. at the lower operating point (*y*-).

Parameter dialog:

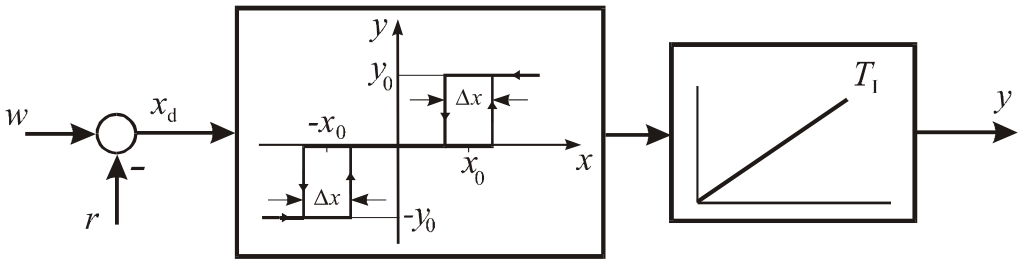




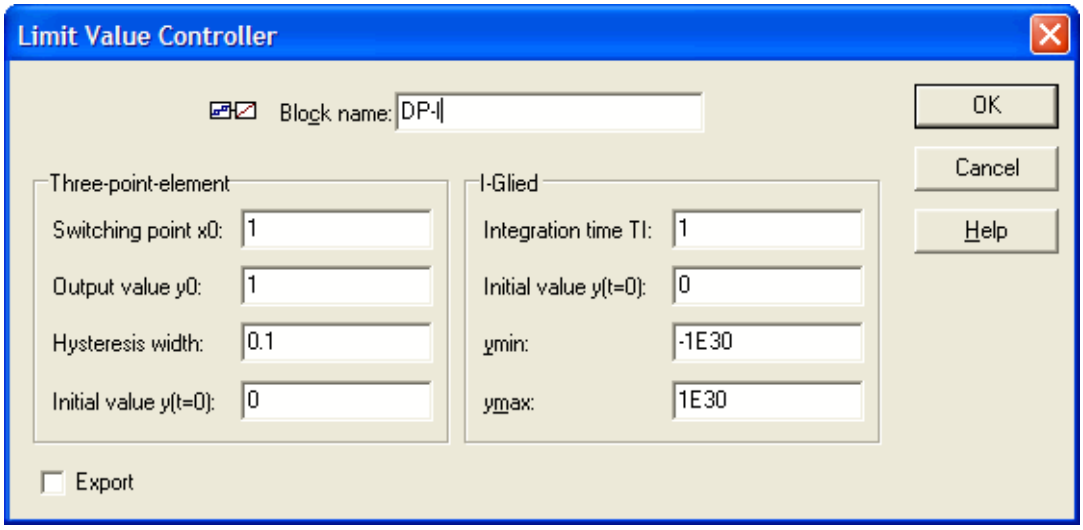
Limit value controller

Typename: DP-I

Function: This block represents a serial connection of a symmetrical three-point controller with the switching points x_0 and $-x_0$, the operating points y_0 and $-y_0$ and the hysteresis width Δx and an integrator (integration time T_I) limited to a range of $[y_{min}, y_{max}]$. Due to this integrator the controller is able to generate a continuous manipulated variable. The graphic below shows the structure of the controller.



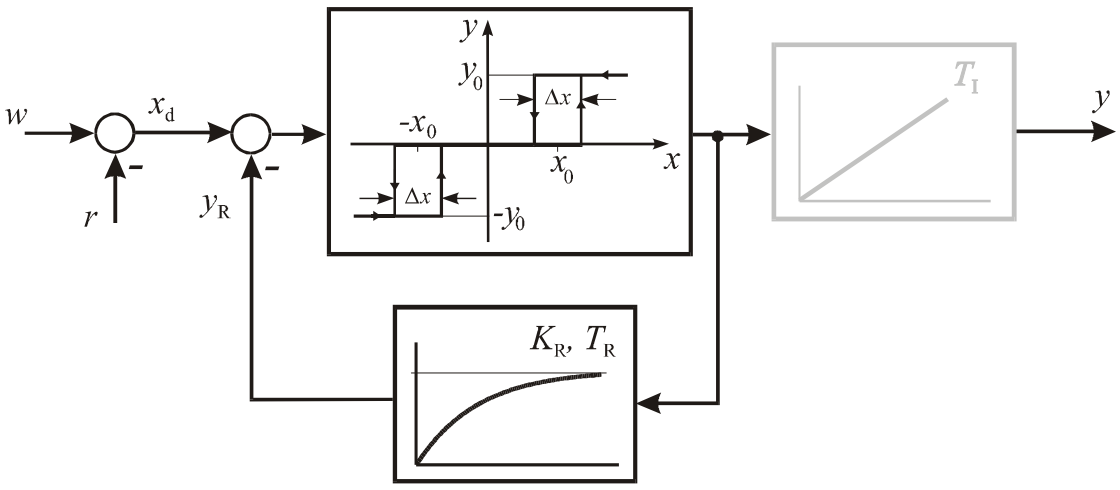
Parameter dialog:



Three-point step controller

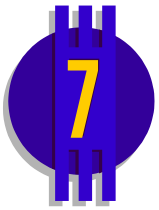
Typename: DP-PI

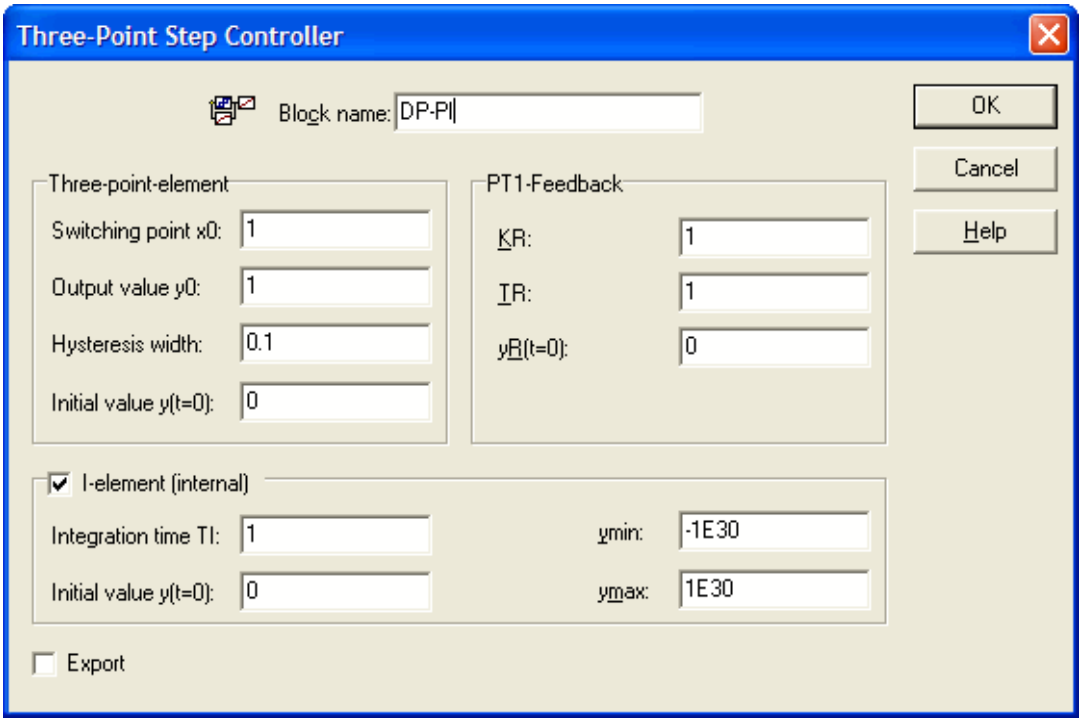
Function: This block represents a symmetrical three-point controller with the switching points x_0 and $-x_0$, the operating points y_0 and $-y_0$ and the hysteresis width Δx with a PT_1-feedback (gain K_R, time constant T_R) of its output. If desired the controller can be used in serial connection with an integrator (integration time T_I) limited to a range of $[y_{min}, y_{max}]$. If the controller is parameterized properly it works similar to a continuous PI-controller. The graphic below shows the structure of the controller.



The block outputs *y*+ and *y*- (second resp. third output) can be used as switching outputs; they get HIGH-level if the three-point element is at the upper operating point (*y*+) resp. at the lower operating point (*y*-).

Parameter dialog:





Fuzzy Controller

see block group *Miscellaneous*

Online-Fuzzy Controller

see block group *Miscellaneous*

Actuators

Actuator type I

Typename: ACTUATOR_1

Function: This block represents a final controlling element which has besides the typical output limitation

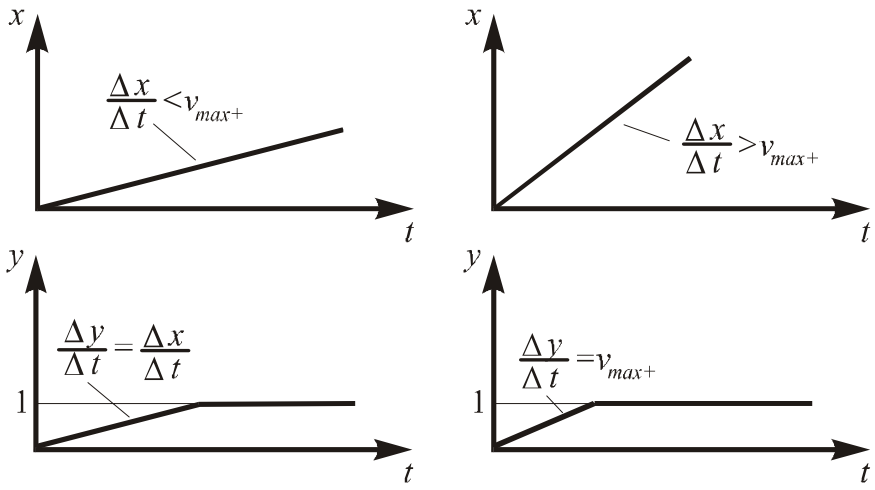
$$y(t) = \begin{cases} y_{\min} & for \quad x(t) < y_{\min} \\ x(t) & for \quad y_{\min} \le x(t) \le y_{\max} \\ y_{\max} & for \quad x(t) > y_{\max} \end{cases}$$

(see. limiter block) a limited actuating speed so that it can only react to changes of input variables with a limited speed. The gradient of the output variable of the final controlling element will thereby be limited to v_{max-} for negative and to v_{max+} for positive input variable gradients:

$$v_{\max -} \le \frac{\mathrm{d}y}{\mathrm{d}t} \le v_{\max +}$$

The following graphic shows the response of the output variable for different input ramps in the case

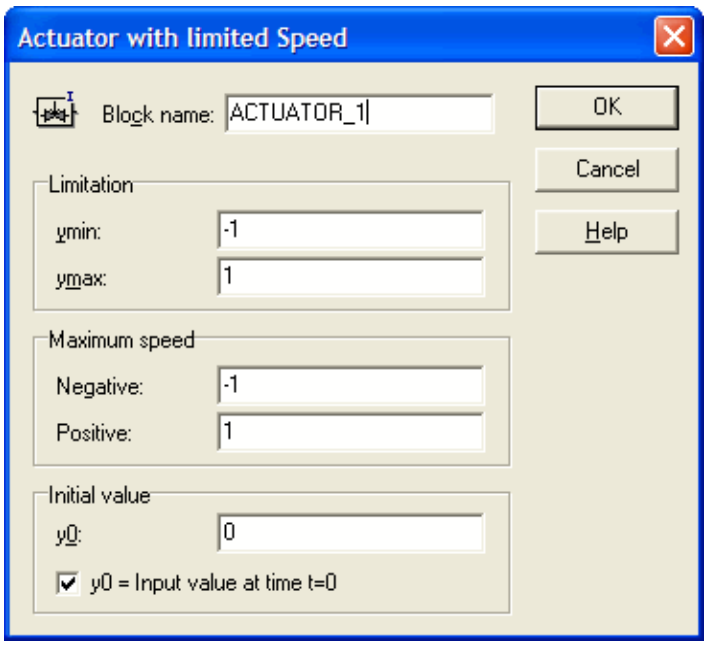
$$y_{\min} = -1, y_{\max} = 1, \;\; v_{\max -} = -1, v_{\max +} = 1 .$$



Behaviour of the final controlling element for an input ramp with small (left) and big (right) gradient, i. e. changing speed (right)

The initial value y_0 of the final controlling element can alternatively be preset or set to the input value of the block at $t = 0$. In the last case the value specified in the dialog has no effect.

Parameter dialog:



Restrictions:

$$y_{\min} < y_{\max}\,,\; v_{\max-} < 0, v_{\max+} > 0$$

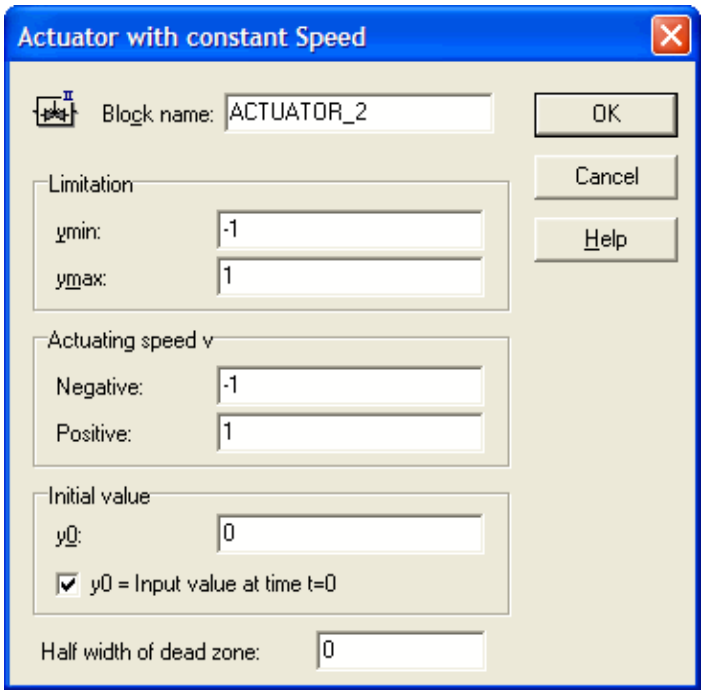
Actuator type II

Typename: ACTUATOR_2

Function: This type of a final controlling element has an output variable limitation as well as a constant ramp speed. For negative input variables the output variable changes independently of the value of the input variable with v_-, for positive input variables with v_+. Further on the final controlling element has a dead zone of width Δx. Input variables which have a value smaller than $\Delta x / 2$ will be ignored, i. e. the current output value will be kept.

The initial value y_0 of the final controlling element can alternatively be specified or set to the input value of the block at the point $t = 0$. In the last case the value specified in the dialog has no effect.

Parameter dialog:



Restrictions: $y_{\min} < y_{\max}$

$v_- < 0, \quad v_+ > 0$

$\Delta x / 2 \geq 0$

Function blocks

Junction

Typename: JUNCTION

Function: This block allows the connection of two or more input variables by the operations addition, multiplication or division. For each input variable x_i a positive or negative sign can be specified:

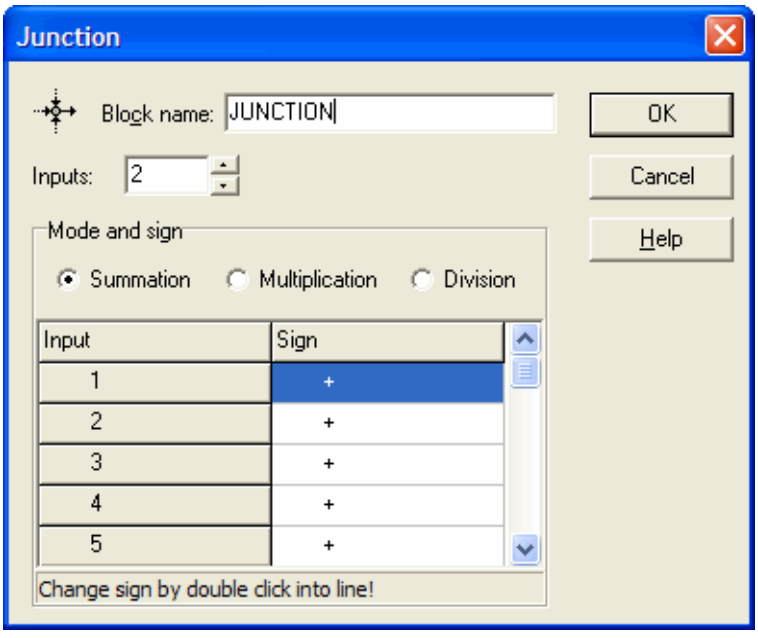
Operating mode *Summation*: $y = \pm x_1 \pm x_2 \pm x_3 \pm \ldots$

Operating mode *Multiplication*: $y = (\pm x_1) \cdot (\pm x_2) \cdot (\pm x_3) \cdot \ldots$

Operating mode *Division*: $y = (\pm x_1) / (\pm x_2) / (\pm x_3) / \ldots$

In the operating modes *Multiplication* and *Division* open inputs are set to 1. In the operating mode *Addition* the inputs of the blocks are marked by the sign of the input variable.

Parameter dialog:



Function of one variable

Typename: FCT1

Function: This block allows the specification of a function $y = f(x)$. You can choose the following functions:

$y = \sin(x)$ $y = \cos(x)$ $y = \tan(x)$ $y = \exp(x)$ $y = \ln(x)$

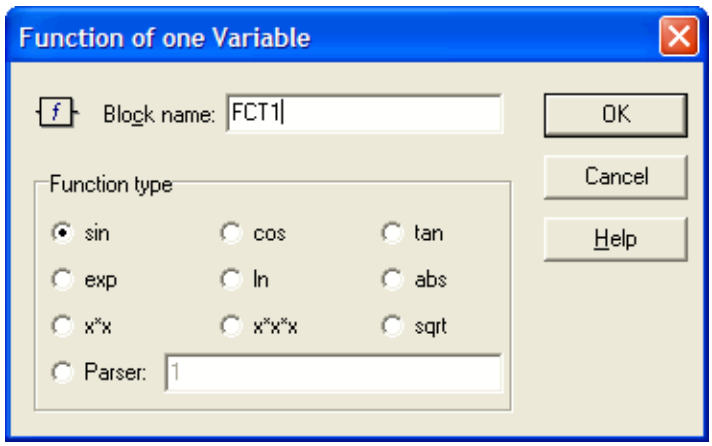
$y = |x|$ $y = x^2$ $y = x^3$ $y = \sqrt{x}$

Alternatively the function can be defined by the user via a function parser (see block description *Generator*). In this case you have to choose "x" as the independent variable.

Example: *atan(x)+asin(x)+5*

Depending on the complexity of the function the interpretation of the function by the parser could need some calculation time.

Parameter dialog:



Function of two variables

Typename: FCT2

Function: This block allows the specification of a function $z = f(x, y)$ with the input variables x and y and the output variable z. You can choose the following functions:

$z = x + y$ $z = x - y$ $z = xy$ $z = x / y$ $z = x^y$ $z = x^{1/y}$

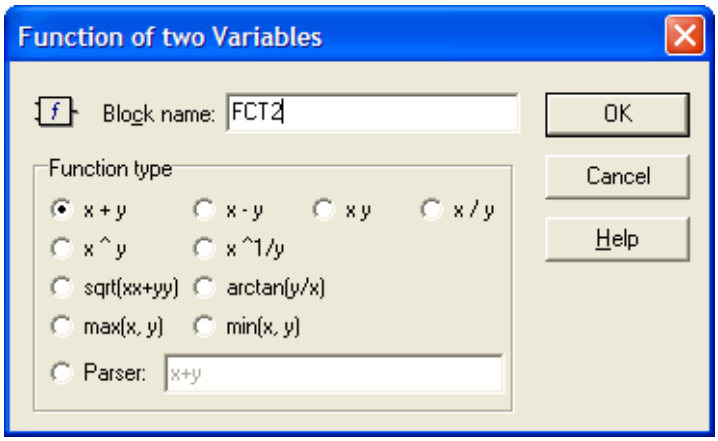
$z = \sqrt{x^2 + y^2}$ $z = \arctan(y / x)$ $z = \max(x, y)$ $z = \min(x, y)$

Alternatively the function can be defined by the user via a function parser (see block description *Generator*). In this case you have to choose "x" and "y" as independent variables.

Example: *sin(x) + cos(y)*

Depending on the complexity of the function the interpretation of the function by the parser can need some calculation time.

Parameter dialog:



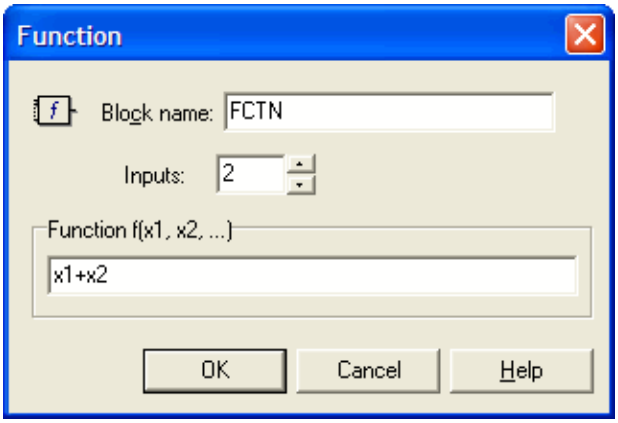
Function of several variables

Typename: FCTN

Function: This block allows the specification of a function with up to 50 variables via a function parser (see block description *Generator*). In this case you have to choose "x1" up to "x50" as independent variables.

Example: $x1 + x2 + x3 - x4 * x5$

Parameter dialog:



Extremal value detection

Typename: EXTREMVAL

Function: This block allows the determination of the extreme value of the input $x(t)$. You can choose four different operating modes:

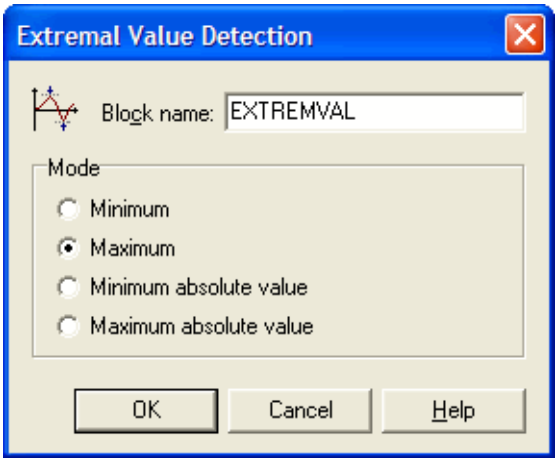
Operating mode *Minimum*: $y(t) = \min_{0<t<T_{Simu}} x(t)$

Operating mode *Maximum*: $y(t) = \max_{0<t<T_{Simu}} x(t)$

Operating mode *Min abs. value*: $y(t) = \min_{0<t<T_{Simu}} |x(t)|$

Operating mode *Max. abs. value*: $y(t) = \max_{0<t<T_{Simu}} |x(t)|$

Parameter dialog:

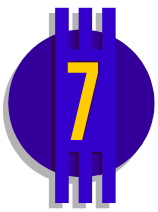


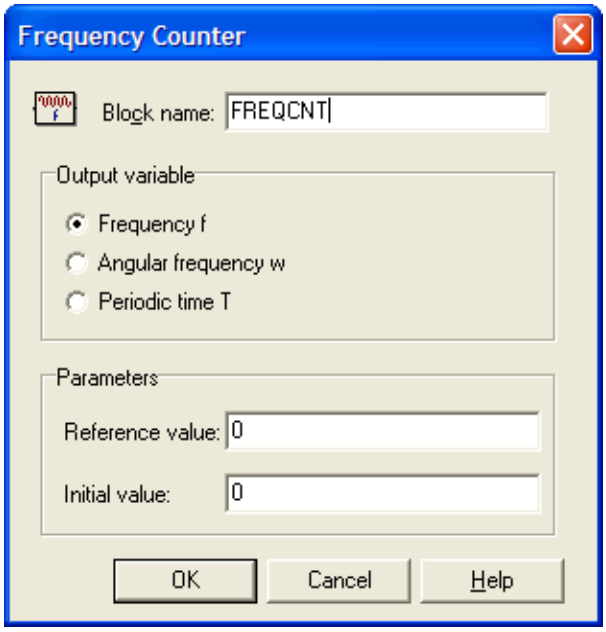
Frequency counter

Typename: FREQCNT

Function: This block allows the determination of frequency, angular frequency or periodic time of a periodic signal based on a sequence of zero-crossings (need to be related to a reference value). As long as no complete period of the input signal was detected the block output is set to the value specified as *Initial value*.

Parameter dialog:



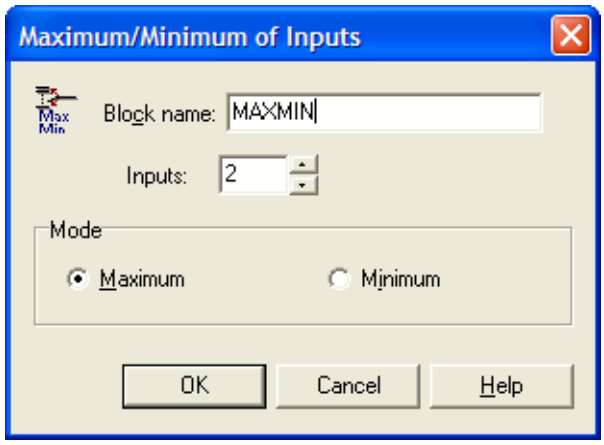


Minimum/Maximum determination

Typename: MAXMIN

Function: This block either switches the minimum or the maximum of all input signals to the output.

Parameter dialog:



Statistic functions

Typename: STATISTIC

Function: This block allows the calculation of statistic parameters of the input variable $x(t)$. The following statistic values concerning all values up to the *k*-th simulation step are available:

Mean value: $$y(t_k) = \frac{1}{k}\sum_{i=1}^{k} x(t_i)$$

Mean absolute value: $$y(t_k) = \frac{1}{k}\sum_{i=1}^{k} \left|x(t_i)\right|$$

Effective value (RMS): $$y(t_k) = \sqrt{\frac{1}{k}\sum_{i=1}^{k} x^2(t_i)}$$

Standard deviation: $$y(t_k) = \sqrt{\frac{1}{k}\left(\sum_{i=1}^{k} x^2(t_i) - \frac{1}{k}\left(\sum_{i=1}^{k} x(t_i)\right)^2\right)}$$

Furthermore different *floating* functions are available concerning a specified *time window* (period of time) which includes the last *n* values. If e. g. a period of time of 5 with a simulation step size of 0.1 is specified, the floating values result from the last 5/0.1 = 50 input values. The following functions are available:

Floating mean value: $$y(t_k) = \frac{1}{n}\sum_{i=k-n+1}^{k} x(t_i)$$

Floating effective value: $$y(t_k) = \sqrt{\frac{1}{n}\sum_{i=k-n+1}^{k} x^2(t_i)}$$

Floating standard deviation:

$$y(t_k) = \sqrt{\frac{1}{n}\left(\sum_{i=k-n+1}^{k} x^2(t_i) - \frac{1}{n}\left(\sum_{i=k-n+1}^{k} x(t_i)\right)^2\right)}$$

Floating sum: $$y(t_k) = \sum_{i=k-n+1}^{k} x(t_i)$$

Floating sum of squares: $$y(t_k) = \sum_{i=k-n+1}^{k} x^2(t_i)$$

Apart from that the statistic block can operate as a *sampling counter*.

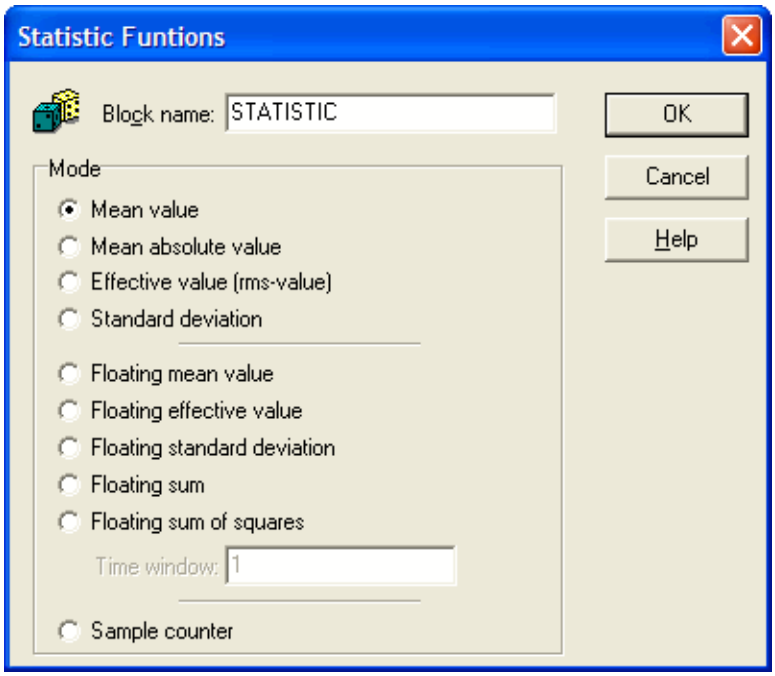
Sampling counter: $y(t_k) = k$

Note: The equations for the floating functions are only valid in the "oscillated" state. At the start of the simulation – as long as not *n* values are available, i. e. during the first steps – all existing input values will be used for the calculation of the floating function! Please note furthermore

that in all operating modes the input value at the time $t = 0$ is added. After the first "real" simulation step you will get the result for $k = 2$!

The block can be reset with a positive edge at the reset input R at any time. After the reset the calculation of the statistic parameters starts again at that time.

Parameter dialog:

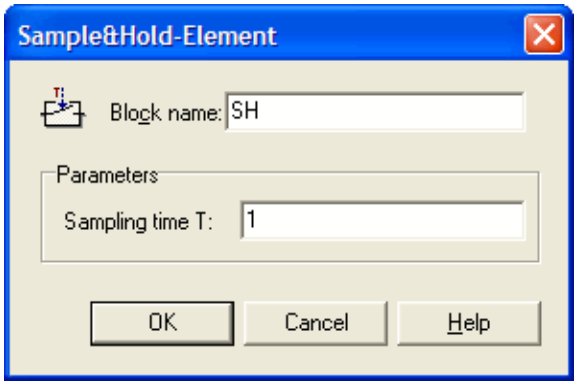


Sample & Hold element

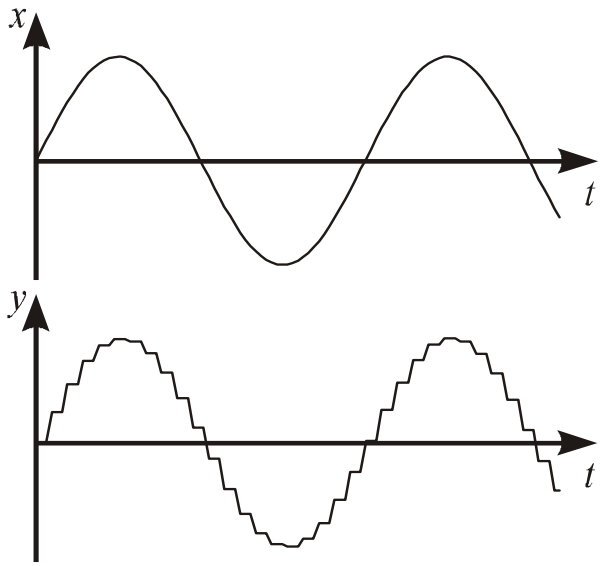
Typename: SH

Function: This function block realizes a sample and hold element of 0-th order. The input variable is sampled at a specified sampling time T, switched to the output and kept constant up to the next sampling time. The sampling time T should be an integer multiple of the simulation step size ΔT.

Parameter dialog:



Beispiel: The following graphic shows the progress course of the output variable of a sample and hold element with a sinusoid input variable.



Restrictions:

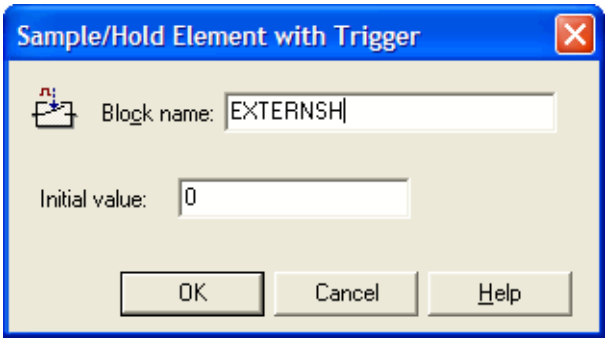
$T > 0$

Controllable sample and hold element

Typename: EXTERNSH

Function: This function block realizes a sample and hold element of 0-th order (analog buffer) with a sampling time which is triggered by a positive edge at the clock input C.

Parameter dialog:

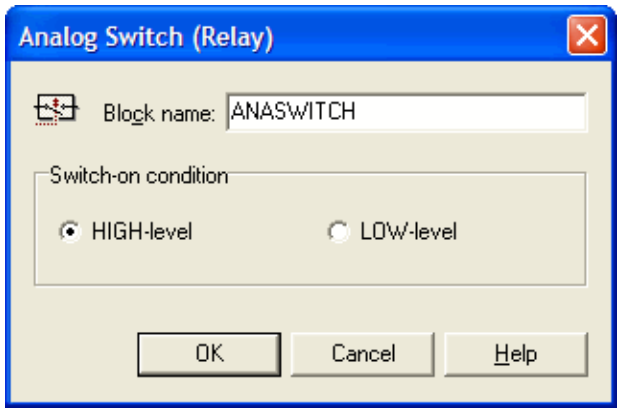


Analog switch (Relay)

Typename: ANASWITCH

Function: This function block realizes an analog switch (relay). The input variable $x(t)$ is switched to the output if High-level resp. Low-level (selectable) occurs at the control input S. Otherwise the output variable will be set to 0.

Parameter dialog:

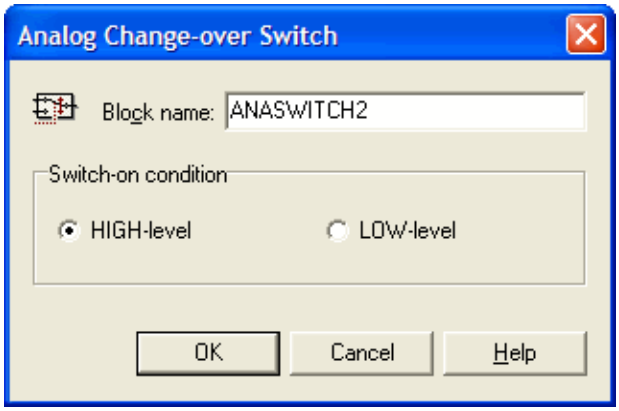


Analog change-over switch

Typename: ANASWITCH2

Function: This function block realizes an analog change-over switch. The input variable $x_1(t)$ is switched to the output if High-level resp. Low-level (selectable) occurs at the control input S. Otherwise the output variable will be set to $x_2(t)$.

Parameter dialog:



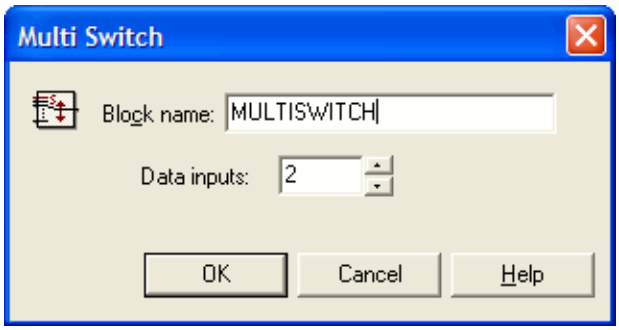
Multi-switch

Typename: MULTISWITCH

Function: This function block realizes a multiple switch controlled by the control input S. The signal value (rounded if necessary) lying at S determines the data input which is swítched to the output. E. g. if S has a value of 2, the second data input will be switched to the output.

If S is not in a valid range, the first data input will be switched to the output.

Parameter dialog:



Digital blocks

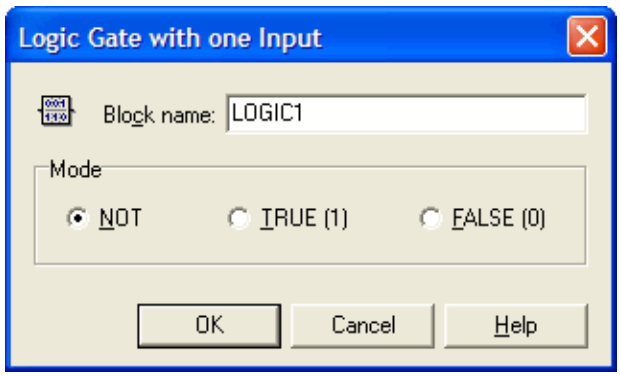
Logic gate with one input

Typename: LOGIC1

Function: This block represents a logical gate with one input. You can choose three different operating modes:

Negation:	$y = \bar{x}$	(input is negated)
TRUE:	$y = 1$	(output always set to HIGH-level)
FALSE:	$y = 0$	(output always set to LOW-level)

Parameter dialog:



Logic gate with two inputs

Typename: LOGIC2

Function: Logical gate with two inputs x and y and one output z. The following operations are available:

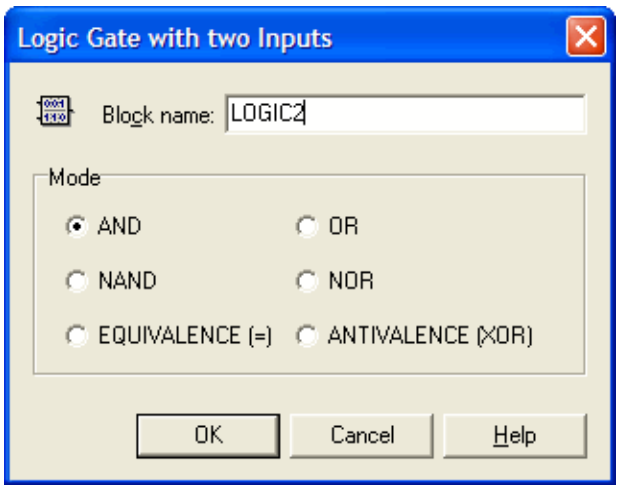
AND:	$z = x \wedge y$
OR:	$z = x \vee y$
NAND:	$z = \overline{x \wedge y}$

NOR: $z = \overline{x \vee y}$

Equivalence: $z = x \equiv y$

Antivalence: $z = x \neq y$

Parameter dialog:

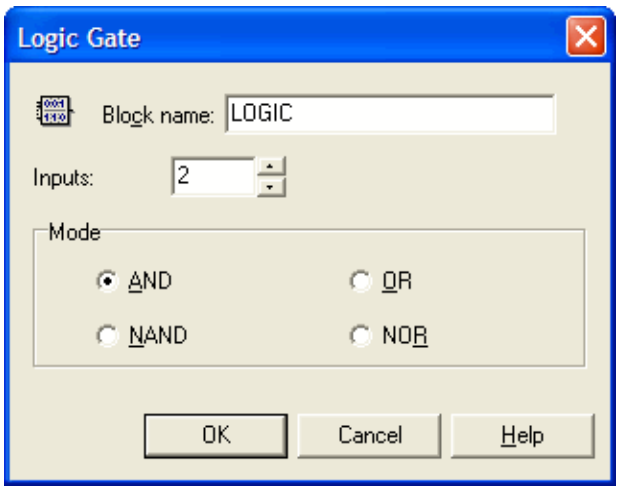


Logic gate with several inputs

Typename: LOGIC

Function: Logical gate with up to 50 inputs and the operating modes AND, NAND, OR and NOR.

Parameter dialog:



RS-Flip-Flop

Typename: RSFLIPFLOP

Function: This block realizes a RS-flip-flop. The output is set by input S and reset by input R. The flip-flop can operate in two modes:

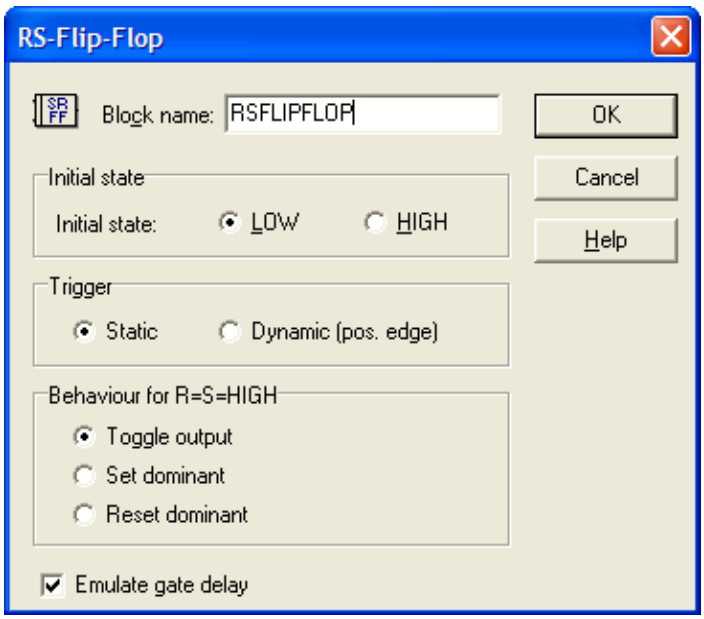
- In the mode *static* the static input level determines the output value.
- In the mode *dynamic* the flip-flop is positive edge triggered, i. e. the output only changes with input edges 0 (Low) → 1 (High).

Below you find the operating table for the RS-flip-flop.

R	S	y_{new}
0	0	y_{old}
0	1	1
1	0	0
1	1	$\overline{y}_{old}$

For the simulation the output level of the flip flop at the beginning of the simulation has to be specified.

Parameter dialog:



If the option *Emulate gate delay* is activated, the calculated output value will be switched to the output at the next simulation step. This option should

therefore be activated if several flip-flops are connected in a row, e. g. to create shift or ring registers or to build up frequency dividers (see the examples SCH_REG.BSY, RING_REG.BSY and TEILER2.BSY in the examples directory).

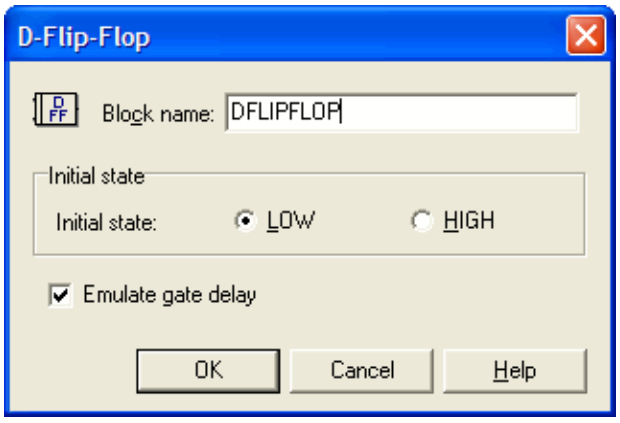
D-Flip-Flop

Typename: DFLIPFLOP

Function: This block represents a D-flip-flop and can therefore be used as a 1-bit storage unit or for the realization of shift registers or frequency dividers. The input value at the data input is saved and switched to the output with a positive edge at the clock input C.

For the simulation the output level of the flip-flop at the beginning of the simulation has to be specified.

Parameter dialog:



If the option *Emulate gate delay* is activated, the calculated output value will be switched to the output at the next simulation step. This option should therefore be activated if several flip-flops are connected in a row, e. g. to create shift or ring registers or to build up frequency dividers (see the examples SCH_REG.BSY, RING_REG.BSY and TEILER2.BSY in the examples directory).

JK-Flip-Flop

Typename: JKFLIPFLOP

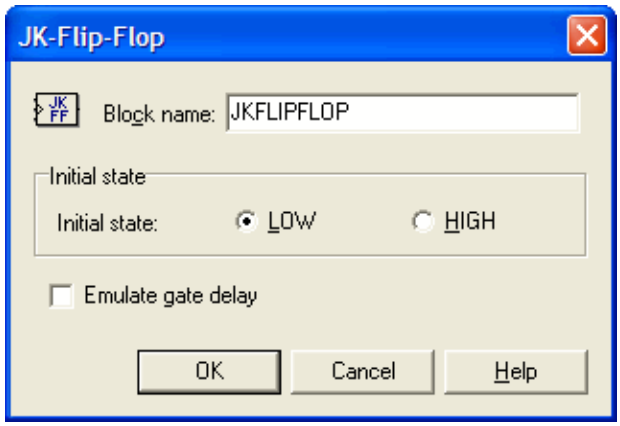
Function: This block realizes a positive edge-triggered JK-flip-flop. With a positive edge at the clock input C the output of the flip-flop is set via the set-input J and reset via the reset-input K. Below you see the corresponding operating table.

J	K	y_{new}
0	0	y_{old}
0	1	0
1	0	1
1	1	$\overline{y}_{old}$

The indefinite state J = K = 1 (last row of the table) produces a change of the output state with each simulation step.

For the simulation the output level of the flip-flop at the beginning of the simulation has to be specified.

Parameter dialog:



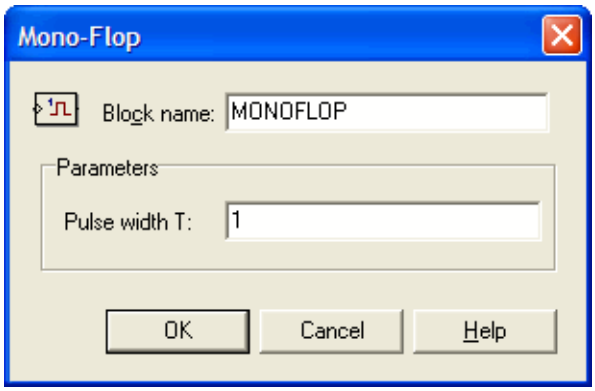
If the option *Emulate gate delay* is activated, the calculated output value will be switched to the output at the next simulation step. This option should therefore be activated if several flip-flops are connected in a row, e. g. to create shift or ring registers or to build up frequency dividers (see the examples SCH_REG.BSY, RING_REG.BSY and TEILER2.BSY in the examples directory).

Mono-Flop

Typename: MONOFLOP

Function: This block represents a monostable flip-flop (univibrator). With a positive edge at the input it produces a pulse of a user-definable length T at the output. The mono-flop cannot be retriggered. The pulse width should be an integer multiple of the simulation step size ΔT.

Parameter dialog:



Restrictions:

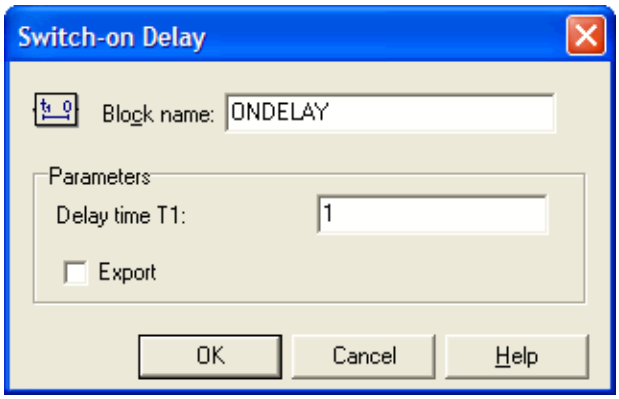
$T > 0$

Switch-on delay

Typename: ONDELAY

Function: This block type delays a switch-on event (positive edge at the block input) by the time T_1.

Parameter dialog:



Restrictions:

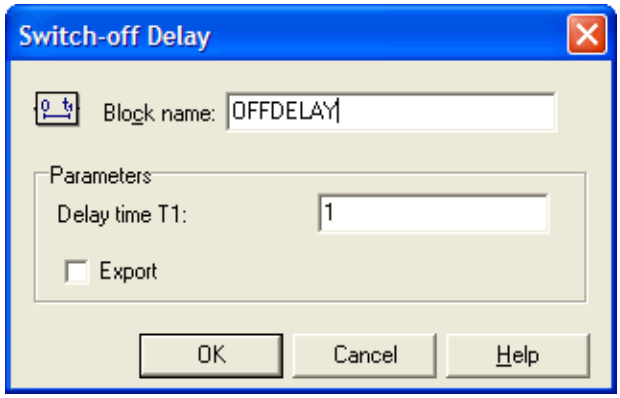
$T_1 \geq 0$

Switch-off delay

Typename: OFFDELAY

Function: This block type delays a switch-off event (negative edge at the block input) by the time T_1.

Parameter dialog:



Restrictions:

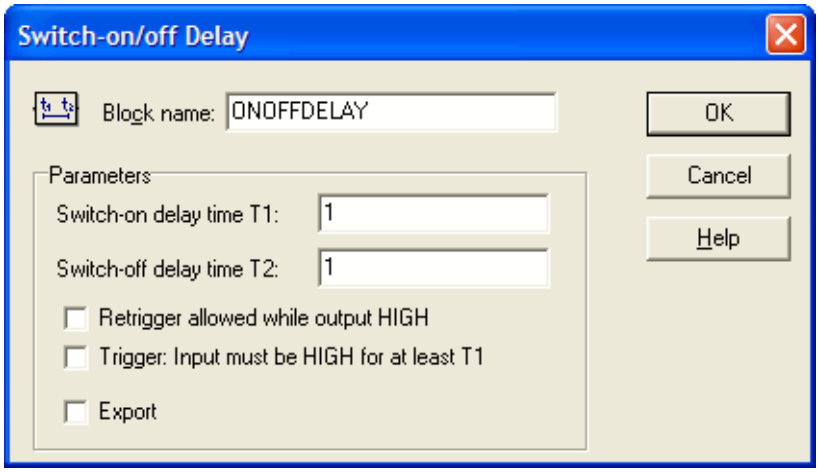
$T_1 \geq 0$

Switch-on/off delay

Typename: ONOFFDELAY

Function: This block type delays a switch-on event (positive edge at the block input) by the time T_1 and a switch-off event (negative edge at the block input) by the time T_2.

Parameter dialog:



If the option *Retrigger allowed while...* is activated a positive edge in case of HIGH-state of the output effects a new triggering of the block; if the option is deactivated, the edge is ignored. If the option *Trigger: Input must be HIGH...* is activated, a positive edge is only transferred to the block output after T_1 if the input still has HIGH level at this time.

Restrictions:

$T_1, T_2 \geq 0$

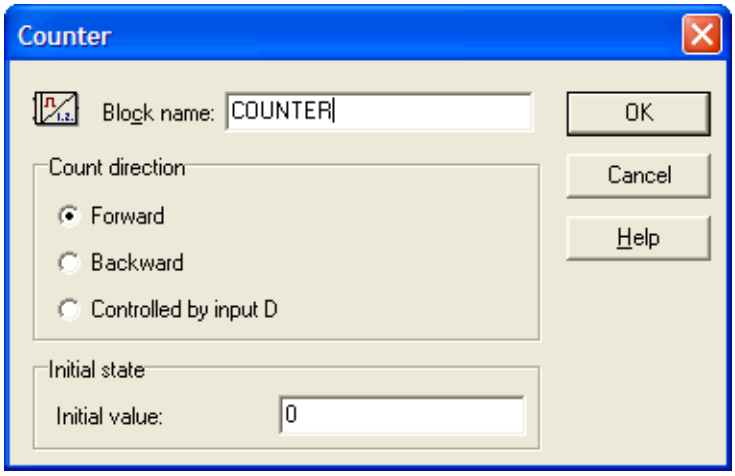
Forward/Backward counter

Typename: COUNTER

Function: This system block represents a positive edge-triggered counter module with a user-definable initial value. The counting direction is switchable internally or externally. The block can be reset to its inital value at any time during the simulation by a positive edge at the reset input R.

The counter supports negative counter states.

Parameter dialog:



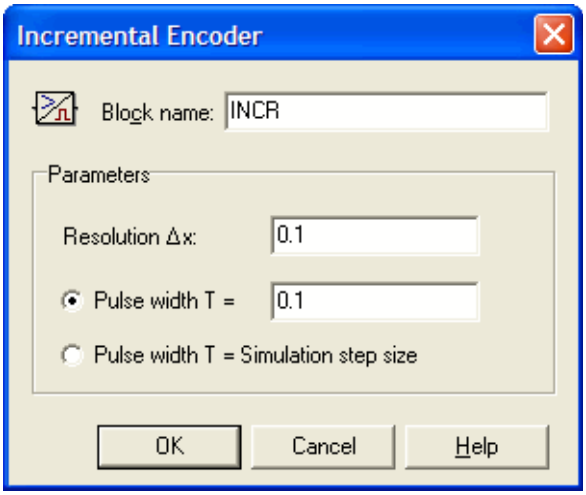
If the option *Controlled by input D* is activated, the module counts forward if there is a HIGH-level at input D, backward if there is a LOW-level at input D.

Incremental encoder

Typename: INCR

Function: This system block realizes an incremental encoder, i. e. an encoder for measuring position or angle changes. If the block input changes for an absolute value of Δx, the block generates a pulse of width T.

Parameter dialog:

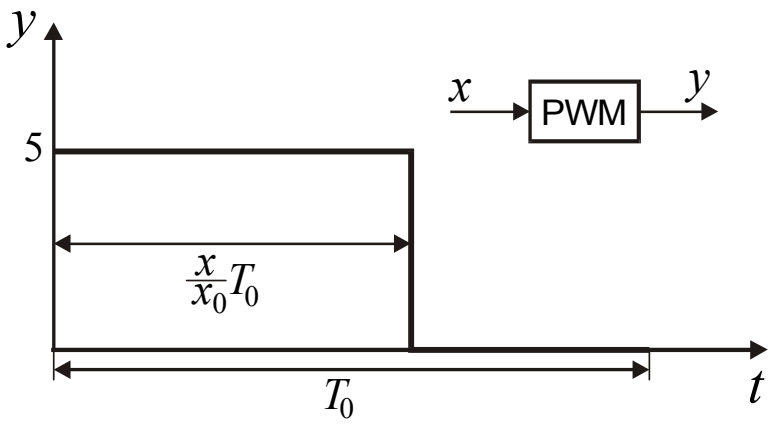


Pulse width modulator (PWM)

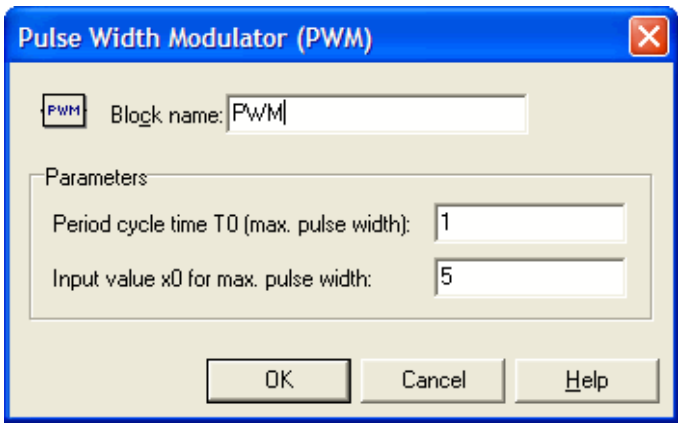
Typename: PWM

Function: This block type generates a pulse width modulated output signal y dependent on an analog input signal x. For that a HIGH pulse of constant frequency f_0 but with variable pulse width T is generated at the block output. The pulse width is proportional to the amplitude of the input signal. If $T_0 = 1/f_0$ is the period time of the output signal and x_0 the maximum allowed input signal of the block the current pulse width is calculated as

$$T = \frac{x}{x_0} T_0 .$$



Parameter dialog:



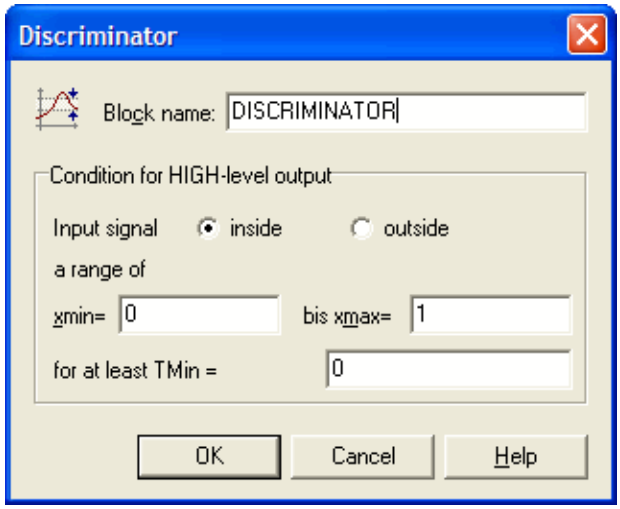
For a correct function of the block the simulation step size should be a multiple (factor 10 or greater) smaller than the periodic time T_0 .

Discriminator

Typename: DISCRIMINATOR

Function: This block realizes a window comparator (window discriminator). It delivers HIGH-level at the output if the input variable x for a user-definable minimum time T_{min} lies in/out of a special range of values $[x_{min}, x_{max}]$. Therefore it is possible to ignore short violations of the range.

Parameter dialog:



Restrictions:

$$x_{min} < x_{max}, \quad T_{min} \geq 0$$

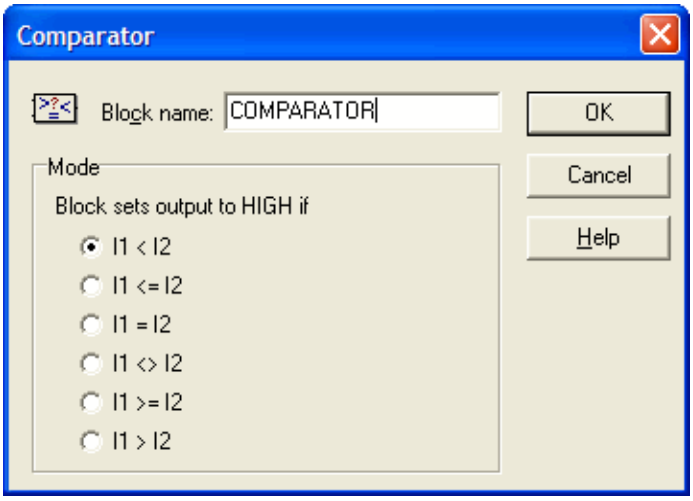
Comparator

Typename: COMPARATOR

Function: This block compares the analog input variables I_1 and I_2 and delivers HIGH- resp. Low-level at the output depending on the result of the comparison. You can choose the following operators for the comparison:

$$I_1 < I_2 \quad I_1 \leq I_2 \quad I_1 = I_2 \quad I_1 \geq I_2 \quad I_1 > I_2 \quad I_1 \neq I_2$$

Parameter dialog:

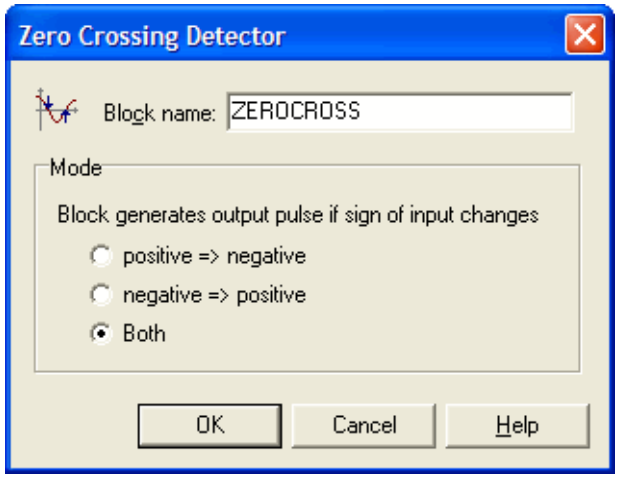


Zero-axis crossing detector

Typename: ZEROCROSS

Function: This block produces an output pulse in the case of a change of sign of the input variable. The length of the output pulse corresponds to the simulation step size ΔT. The direction of the zero-axis crossing which has to be detected can be specified.

Parameter dialog:

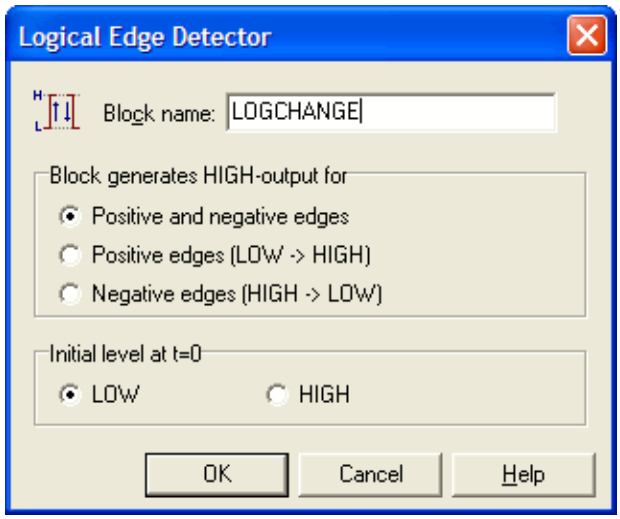


Logical edge detector

Typename: LOGCHANGE

Function: This block produces an output pulse if a logical change of level occurs at the input. You can choose the detection of positive edges (change from LOW- to HIGH-level) and/or the detection of negative edges (change from HIGH- to LOW-level). The output level of the block at the beginning of the simulation can be specified.

Parameter dialog:

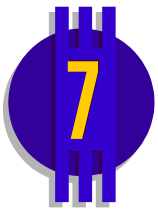


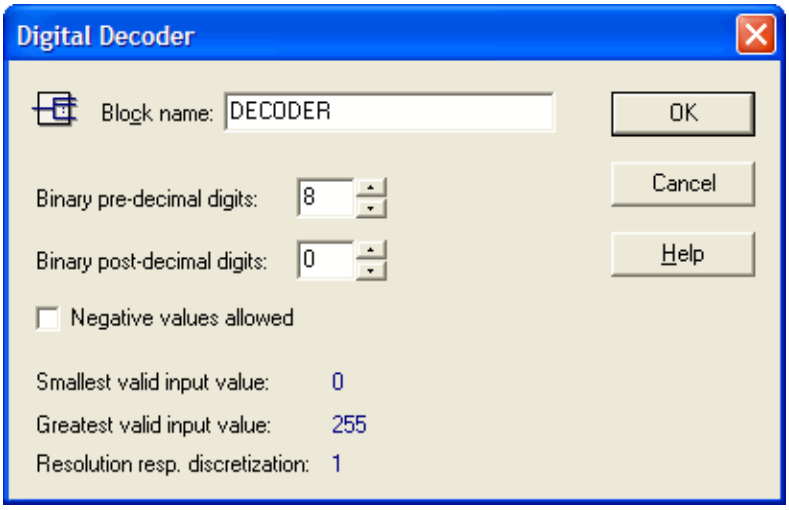
Digital decoder

Typename: DECODER

Function: This block converts an analog value at the block input to a digital value which is set to the block outputs in binary code. The number of binary pre- and post-decimal digits and thus the discretization can be specified as well as the validity of negative input values.

Parameter dialog:



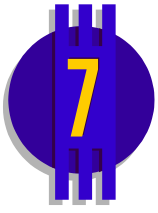


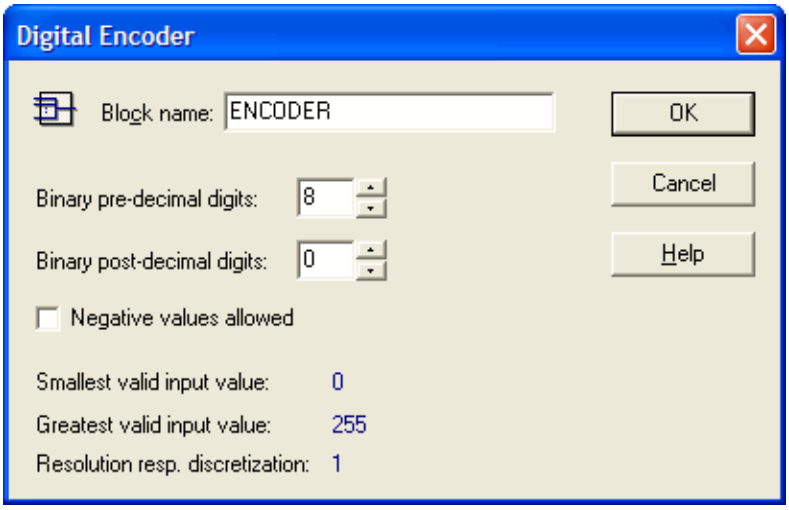
Digital encoder

Typename: ENCODER

Function: This block converts a binary coded digital value at the block inputs to an analog output value. The number of binary pre- and post-decimal digits and thus the discretization can be specified as well as the validity of negative input values. This block represents the counterpart to the digital decoder.

Parameter dialog:



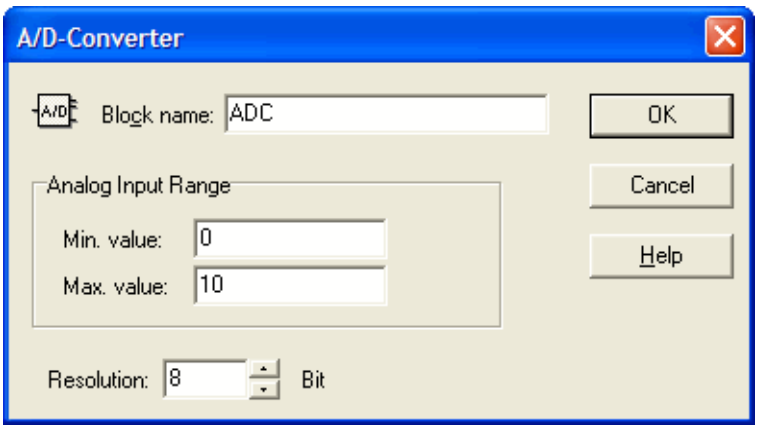


A/D-Converter

Typename: ADC

Function: This block converts an analog input value to a binary coded digital output value. The converter resolution can be specified. The block is similar to the *DECODER* block.

Parameter dialog:

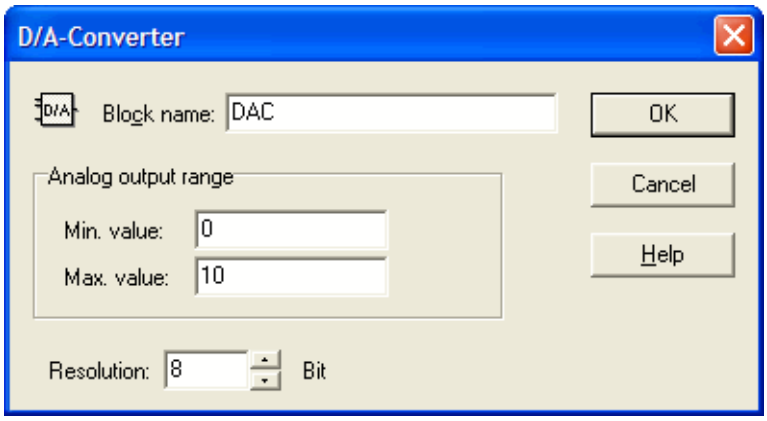


D/A-Converter

Typename: DAC

Function: This block converts a binary coded digital input value to an analog output value. The converter resolution can be specified. This block is similar to the *ENCODER* block.

Parameter dialog:



Action blocks

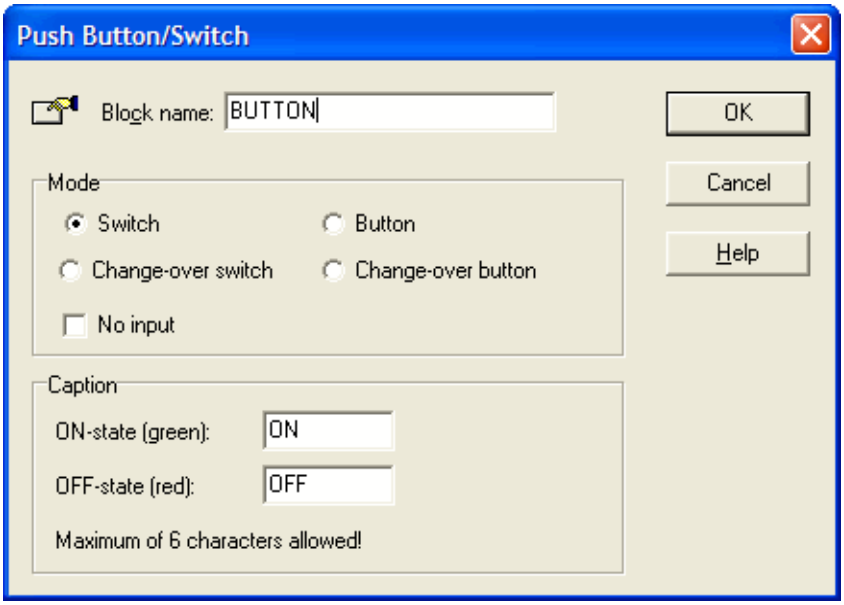
Push button

Typename: BUTTON

Function: This block realizes a push button with on/off-function which has to be pressed with the left mouse button. The push button can be labeled with a user-defined text.

Control window of the push button in On- resp. Off-state

Parameter dialog:



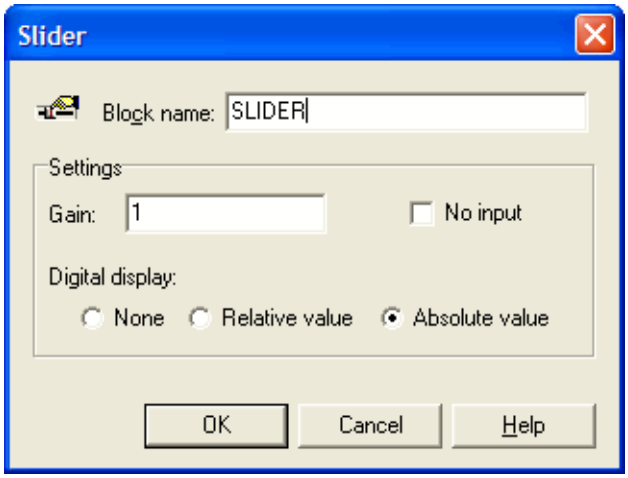
Slider

Typename: SLIDER

Function: This block realizes a slide potentiometer which has to be moved with the left mouse button. Alternatively the block can operate with (by default) or without an input. In the mode *With input* the input value will be multiplied with the *gain* of the slide potentiometer. In the mode *No input* you will get an output value between 0 and the gain depending on the position of the slide potentiometer. Additionally you can activate a digital display of the gain applied to the position of the potentiometer in absolute resp. relative value (in %).

Control window of the slider with absolute resp. relative digital display

Parameter dialog:



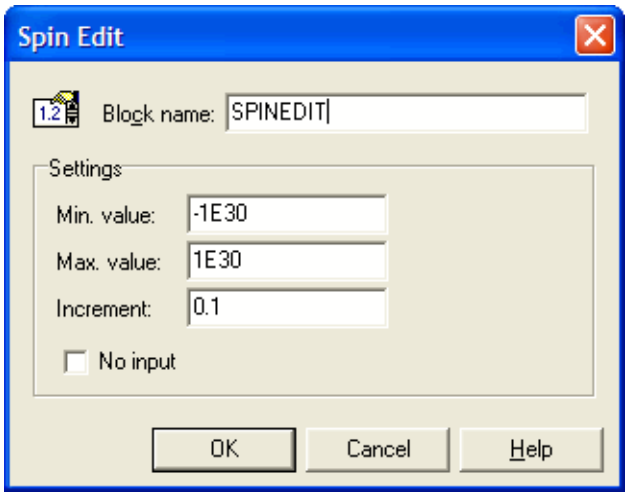
Spin edit field

Typename: SPINEDIT

Function: This block type realizes a numerical input field with a spin element. Minimum, maximum and increment of the element can be specified. The block can be used as a gain (with block input) or as a fixed value generator (no block input).

Control window of the spin edit field

Parameter dialog:



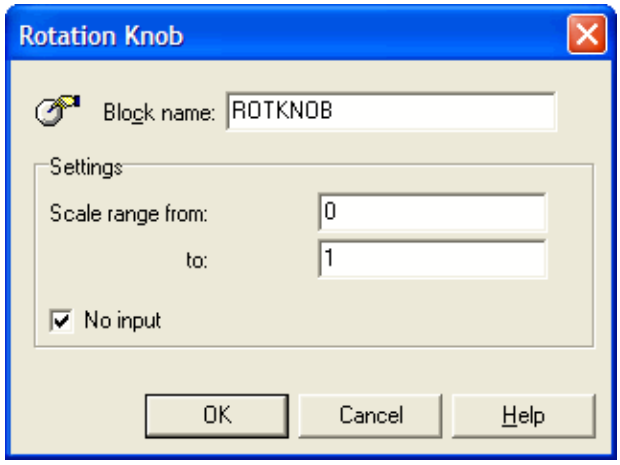
Rotation knob

Typename: ROTKNOB

Function: This block realizes a rotation knob which has to be moved with the left mouse button. Alternatively the block can operate with or without an input. In the mode *With input* the input value will be multiplied with the current position of the rotation knob. In the mode *No input* you will get the current scaling value directly.

Control window of the rotation knob

Parameter dialog:

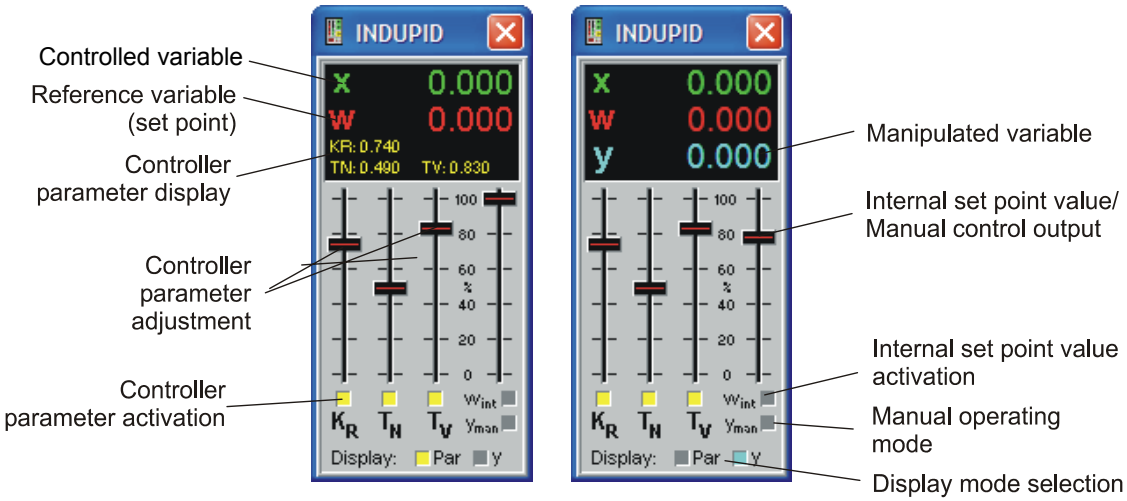


Industrial PID-controller

Typename: INDUPID

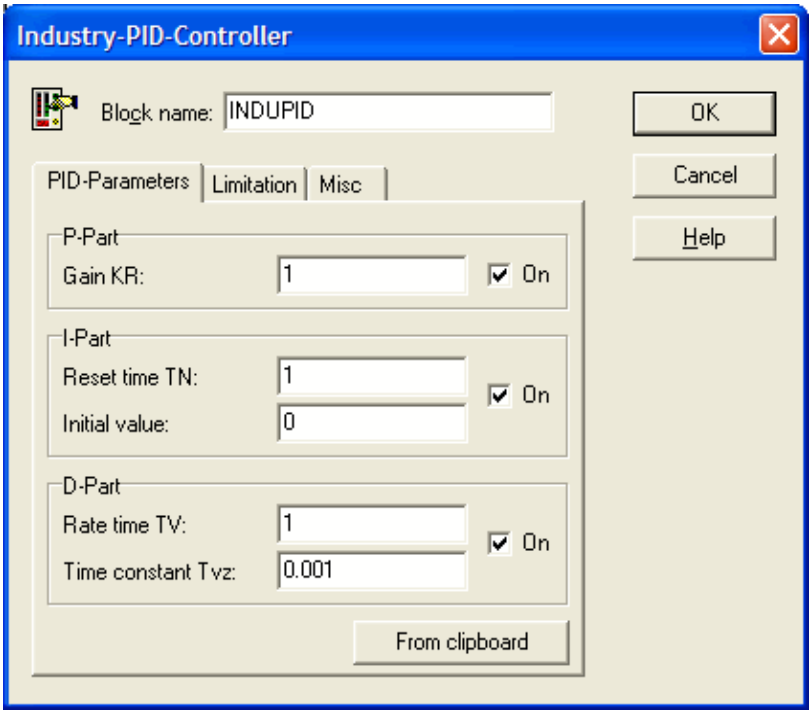
Function: This block realizes a PID-controller with a control and visualization window which corresponds to an industrial controller. In its numerical function this block type corresponds to the standard PID-controller of BORIS, but it has the nominal and the current value (reference variable *w* resp. controlled variable *x*) as separated inputs. Furthermore the reference variable can optionally be generated internally so that the first input of the block is deactivated. If necessary the controller can be used manually (manual setting of the manipulating variable).

The controller parameters and the internal nominal value (if activated) can be varied by slide controllers. The controller components (P-,I- and D-component) can be activated by mouse click. The display of the controller shows in addition to the nominal and the current value as well the current control parameters resp. alternatively the current manipulating variable *y*.



Control and visualization window of the industrial PID-controller with the display of the control parameters (left) resp. the manipulating variable (right)

Parameter dialog:

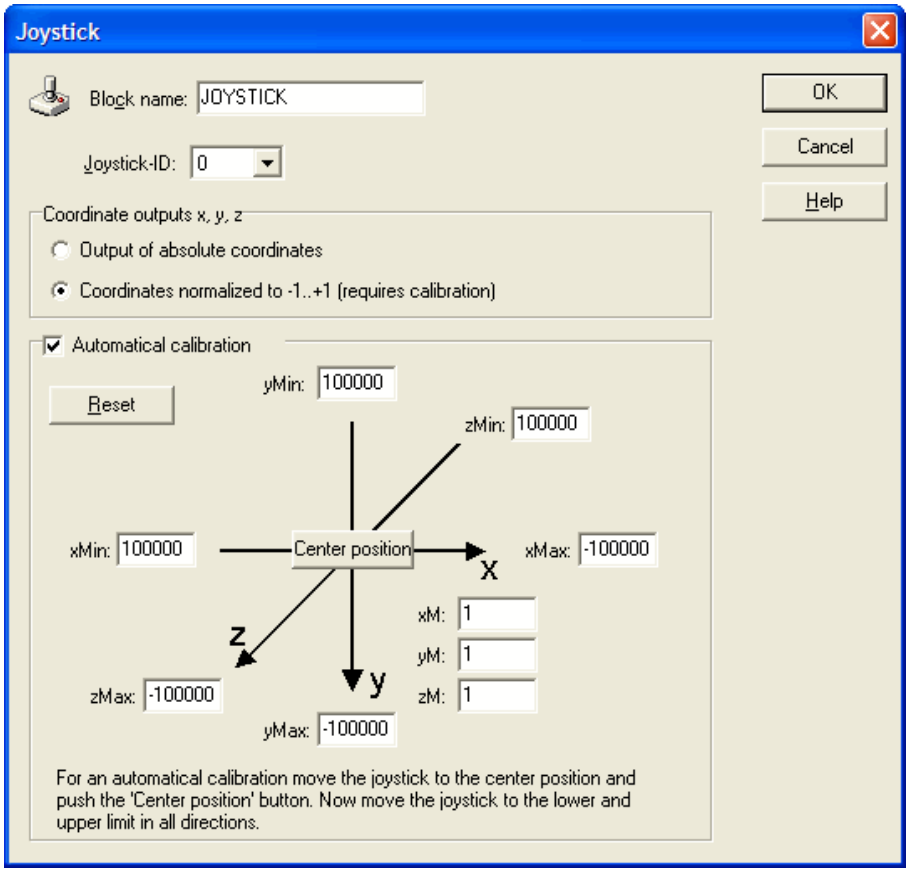


Joystick

Typename: JOYSTICK

Function: This block type allows the usage of up to two joysticks at the PC gameport. The block outputs x, y and z represent the coordinates (absolute or normalized), the outputs B1 ... B4 the state of up to four joystick fire buttons (LOW resp. HIGH level). The joystick can be calibrated automatically via the parameter dialog. The *Joystick ID* (0 resp. 1) specifies the joystick port to be used.

Parameter dialog:

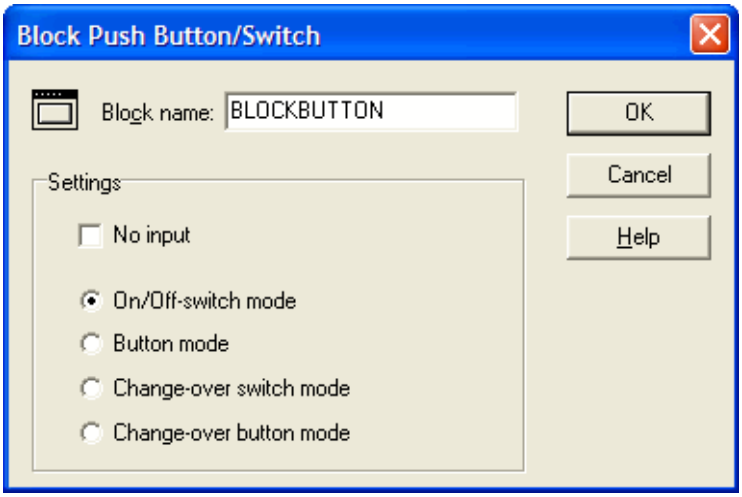


Block push button/switch

Typename: BLOCKBUTTON

Function: This block realizes a push button resp. switch which can be controlled with the left mouse button. In contrast to the standard push button (BUTTON block type) this block does not possess a separate control window but the button is located within the block itself.

Parameter dialog:



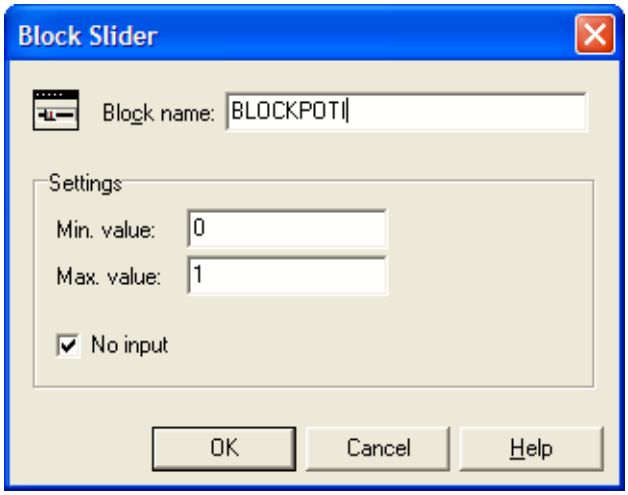
If the *No input* option is activated the block output is set dependent on the current button position to LOW- resp. HIGH-level.

Block slider

Typename: BLOCKPOTI

Function: This block realizes a slider which can be controlled with the left mouse button. In contrast to the standard slider (SLIDER block type) this block does not possess a separate control window but the slider is located within the block itself.

Parameter dialog:



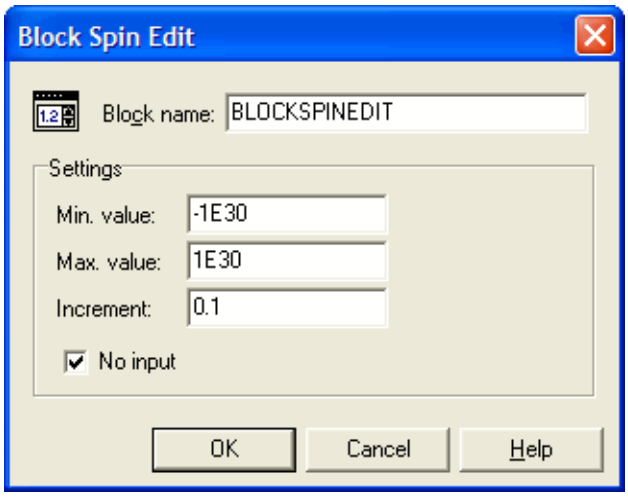
If the *No input* mode is activated, the block output is set to the current slider value, otherwise it is set to the current slider value multiplied with the current block input value.

Block spin edit field

Typename: BLOCKSPINEDIT

Function: This block realizes a spin edit field which can be controlled with the left mouse button. In contrast to the standard edit field (SPINEDIT block type) this block does not possess a separate control window but the edit field is located within the block itself.

Parameter dialog:



If the *No input* mode is activated, the block output is set to the current edit field value, otherwise it is set to the current edit field value multiplied with the current block input value.

Communication

DDE-Input

Typename: DDEIN

Function: This block allows the reading of values from other Windows applications (e. g. EXCEL or LabView) by *Dynamic Data Exchange* (DDE).

The data supplier (*server*) is specified by the following parameters:

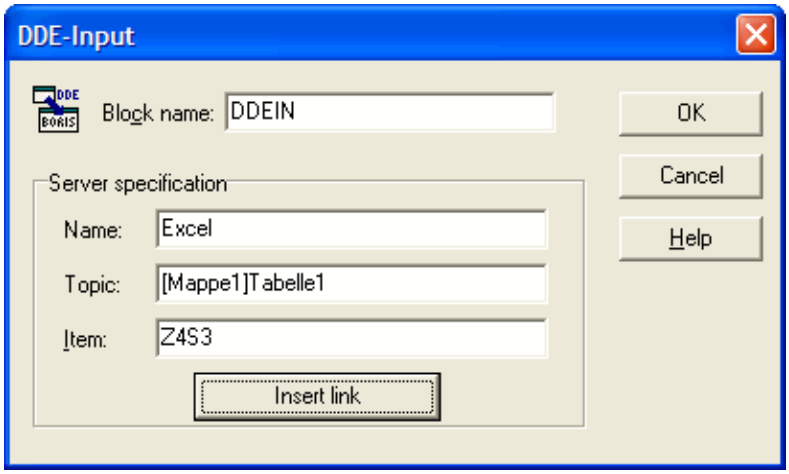
Name Name of the server application (e. g. EXCEL)

Topic Topic of the DDE conversation (e. g. name of the EXCEL table, TAB1)

Item Specification of the target for the input value (e. g. R1C1 for the first row, first column of the EXCEL table)

During the specification of the item a counter variable of the form *{#n}* can also be inserted in order e. g. to write all values consecutively to a column of an EXCEL table. If e. g. the first column should be read whereat the first value will be read from the first row, the second from the second line etc. the input has to be *R{#1}C1*.If you want to start in the second row the input is *R{#2}C1*. If the values should for example be read line-by-line from the first row the input is *R1C{#1}* etc. Please note that on principle the first value is read at the initialisation of the simulation, i. e. in the first simulation step the second value is read etc.

Parameter dialog:



With the button *Insert link* a connection which has been copied to the clipboard from another application can be inserted automatically (in the dialog above for example a reference to a cell of an EXCEL working sheet).

Note: In combination with WinFACT we deliver a demo program named DDEDEMO.EXE to test the DDE capabilities of BORIS. Start the program, then start BORIS and call the example file DDEDEMO.BSY. Then start the simulation. Now you can edit items in DDEDEMO which will be shown in BORIS, reverse the items which are generated by BORIS will be shown in DDEDEMO.

DDE-Output

Typename: DDEOUT

Function: This block allows the output of values to other Windows applications (e. g. EXCEL or LabView) by *Dynamic Data Exchange* (DDE).

The data supplier (*server*) will be specified by the following parameters:

Name Name of the server application (e. g. EXCEL)

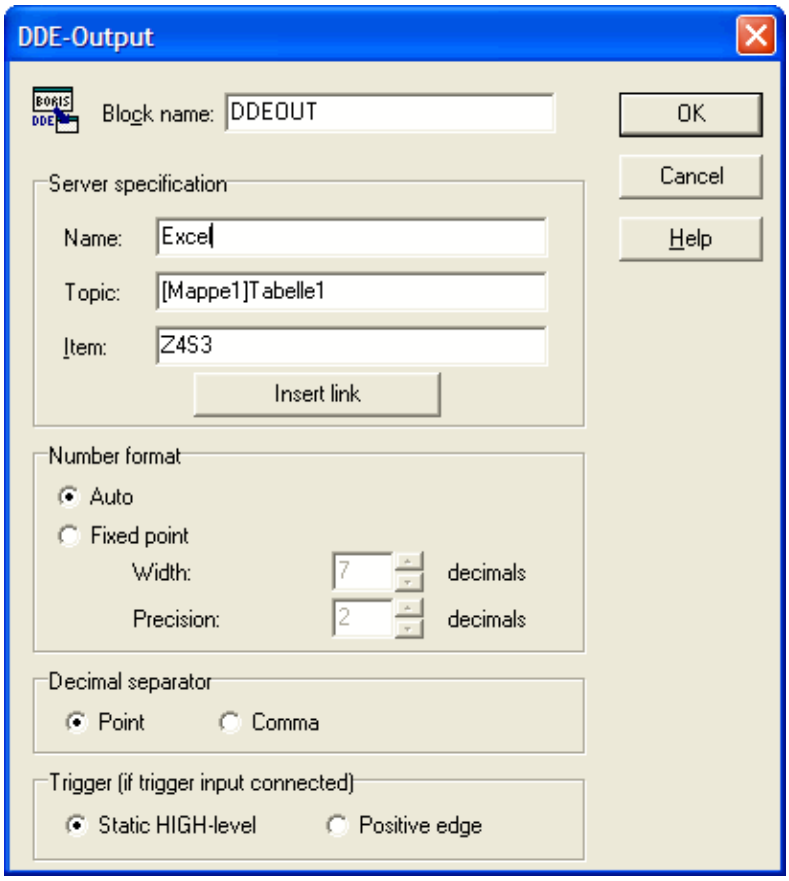
Topic Topic of the DDE conversation (e. g. name of the EXCEL table, TAB1)

Item Spezification of the target for the output value (e. g. R1C1 for the first row, first column of the EXCEL table)

During the specification of the item a counter variable of the form *{#n}* can be inserted for example to write all values consecutively to a column of an EXCEL table. If for example the first column is to be filled in which the first value shall be written into the first line, the second into the second line etc., the input has to be *R{#1}C1*. If you want to start in the second row the input correspondently has to be *R{#2}C1*. If the values should be filled for example column-by-column into the first row, the input has to be *R1C{#1}* etc. Please note that on principle the first value is written at the initialisation of the simulation, i. e. in the first simulation step the second value is written etc.

If trigger input C is connected, the output will be active only in cases of a positive edge at the trigger input or of a static High-level. The decimal field separator (point resp. comma) can be chosen with the corresponding radio button.

Parameter dialog:



With the button *Insert link* a connection which has been copied to the clipboard from another application can be inserted automatically (in the dialog above for example a reference to a cell of an EXCEL working sheet).

TCP-Client (Input)

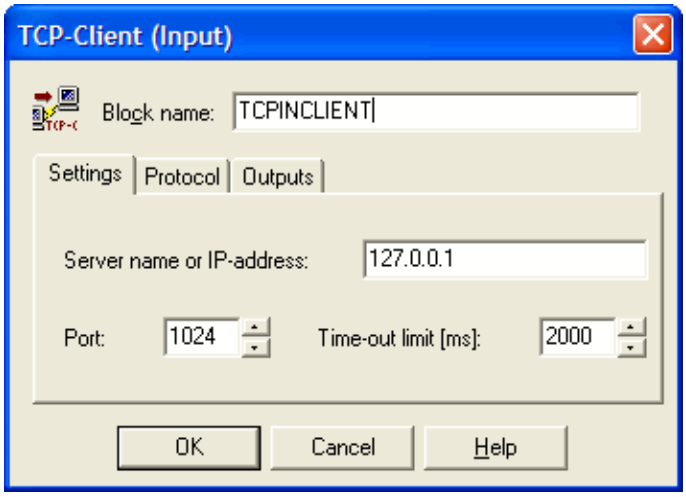
Typename: TCPINCLIENT

Function: This input block realizes a client which receives data from the TCP/IP-protocol. The current state of the connection is shown by a separate status window.

At each simulation step the block first sends the *Data request command* specified on the *Protocol* palette to the server and then waits for the requested data for a maximum time specified by the *Time-out limit* on the *Settings* palette. The data sent by the server are interpreted as a string and converted to a floating point number. If the block has more than one output, the received string has to separate the values by a semicolon.

The exact functionality is demonstrated in detail by the file TCPDEMO.BSY resp. the files TCPCLIENT.BSY/TCPSERVER.BSY (please load both files into separate BORIS instances and start both simulations!) in the examples directory.

Parameter dialog:



If the option *Use trigger input* is activated the block uses a trigger input. In this case the data transmission is only active if the trigger input has HIGH level.

TCP-Server (Input)

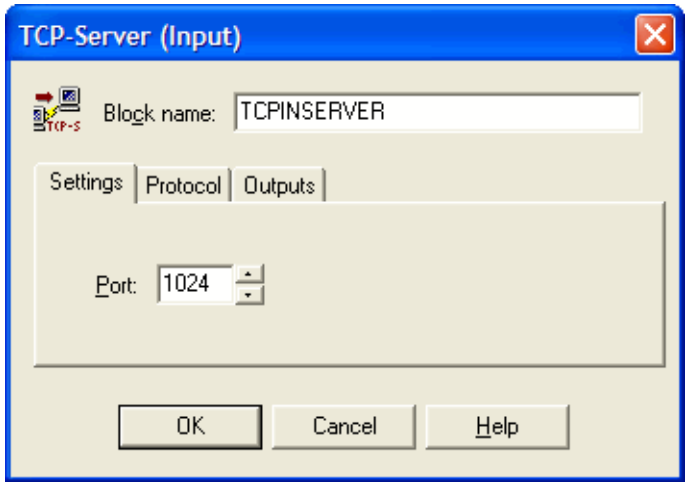
Typename: TCPINSERVER

Function: This input block realizes a server which receives data from the TCP/IP-protocol. The current state of the connection will be shown by a separate status window.

To receive data the block first waits for the client to send the command specified under *Command for data request* on the *Protocol* palette. If the block has received this command it sends the command specified under *Command for server confirmation* back to the client. for confirmation. In the following the block awaits the data sent by the client. The data are interpreted as a text string and converted to a floating point number. If the block has more than one output the received string has to separate the values by semicola. After receiving the data the block again sends the server confirmation command to the client.

The exact functionality is demonstrated in detail by the file TCPDEMO.BSY resp. the files TCPCLIENT.BSY/TCPSERVER.BSY (please load both files in separate BORIS instances and then start both simulations!) in the examples directory.

Parameter dialog:



TCP-Client (Output)

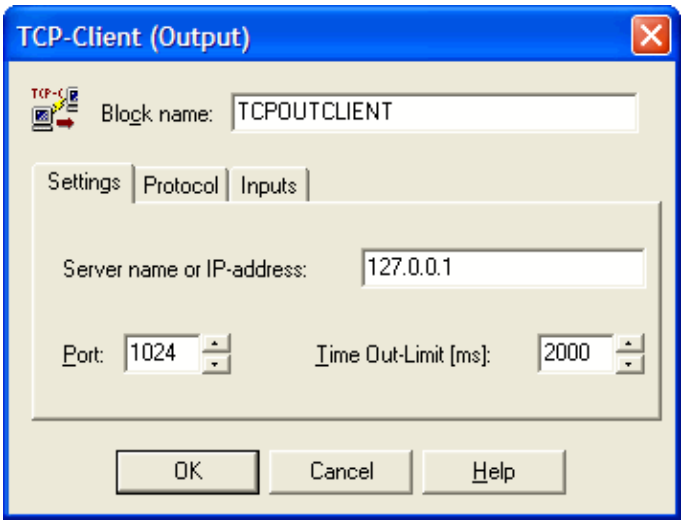
Typename: TCPOUTCLIENT

Function: This output block realizes a client which sends data corresponding to the TCP/IP-protocol. The current state of the connection is shown by a separate status window.

At each simulation step the block first sends the command specified under *Command for data send request* on the *Protocol* palette to the server and then waits for the server confirmation by the command specified under *Command for server confirmation* for a maximum time specified under *Time-out limit* on the *Settings* palette. After receiving this confirmation the block sends the current input data as a string to the server whereat in case of more than one block input the single data are separated by semicola.

The exact functionality is demonstrated in detail in the file TCPDEMO.BSY resp. the files TCPCLIENT.BSY/TCPSERVER.BSY (please load both files in separate BORIS Instances and then start both simulations!) in the examples directory.

Parameter dialog:



If the option *Use last input as trigger input* is activated, the last block input is used as a trigger input. In this case the data transmission is only active if the trigger input has HIGH level.

If the option *Send only if values change* is activated, data transmission is only performed if the value of at least one block input had changed compared to the previous simulation step.

TCP-Server (Output)

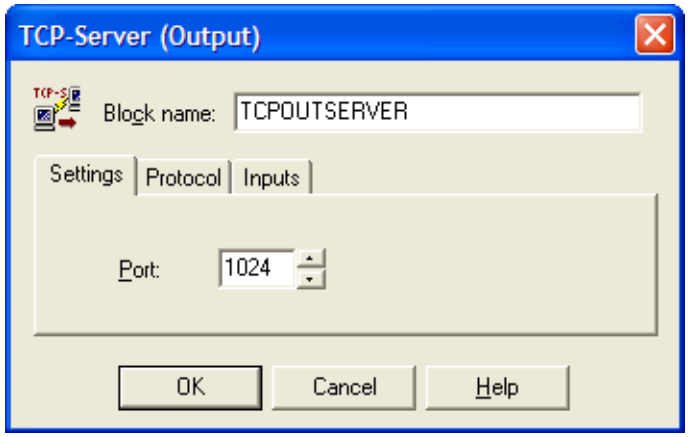
Typename: TCPOUTSERVER

Function: This output block realizes a server which sends data corresponding to the TCP/IP-protocol. The current state of the connection is shown by a separate status window.

Before sending data the block first waits for the command specified under *Command for data request* on the *Protocol* palette. After receiving this command the blocks sends the current input data as a string to the server whereat in case of more than one block input the single values are separated by semicola.

The exact functionality is demonstrated in detail by the file TCPDEMO.BSY resp. the files TCPCLIENT.BSY/TCPSERVER.BSY (please load both files into separate BORIS-instances and the start both simulations!) in the examples directory.

Parameter dialog:

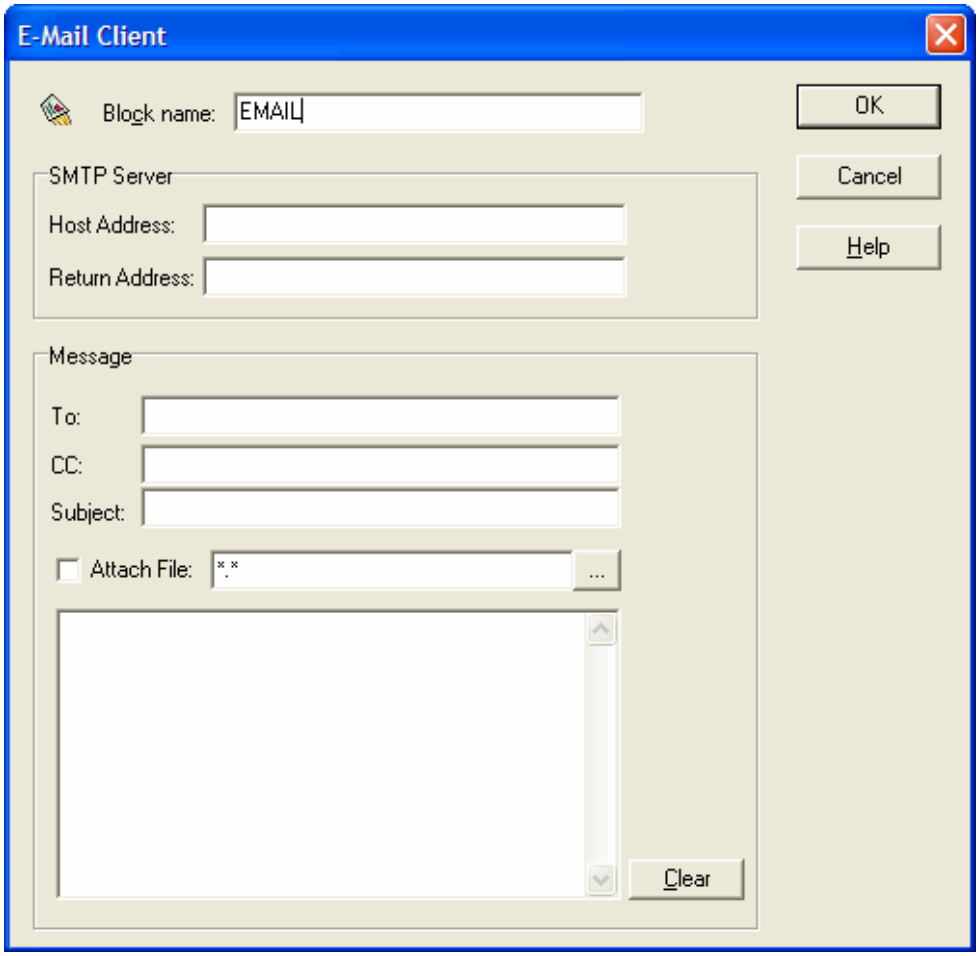


E-Mail client

Typename: EMAIL

Function: This block sends an e-mail via an active online connection if it detects a positive edge at the block input. The status of the connection is displayed in a separate control window.

Parameter dialog:



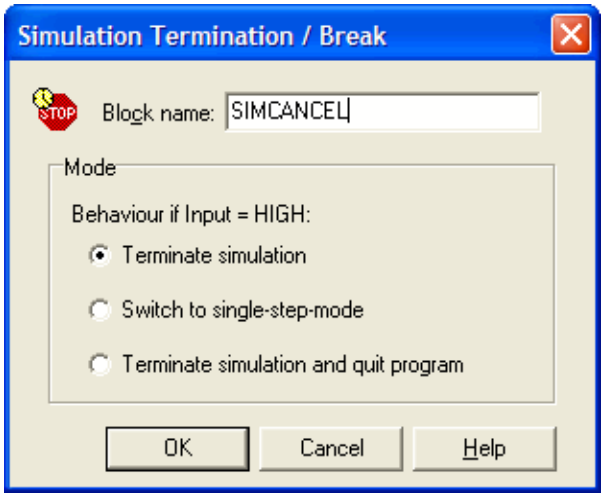
Simulation control blocks

Simulation termination/Break

Typename: SIMCANCEL

Function: This block cancels the simulation resp. switches to the single step mode as soon as High-level occurs at its input.

Parameter dialog:

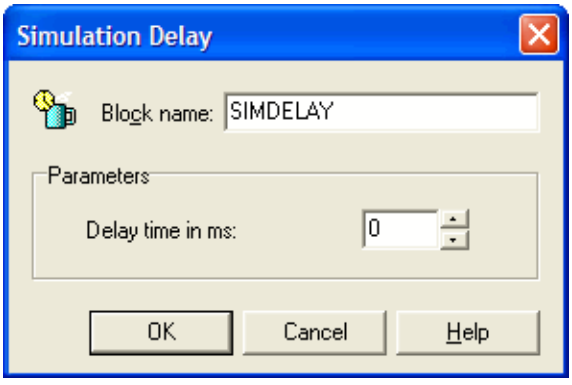


Simulation delay

Typename: SIMDELAY

Function: This block delays the simulation at each simulation step for a user-definable time in msec. It can be used to force a delay of the simulation on very fast computers.

Parameter dialog:

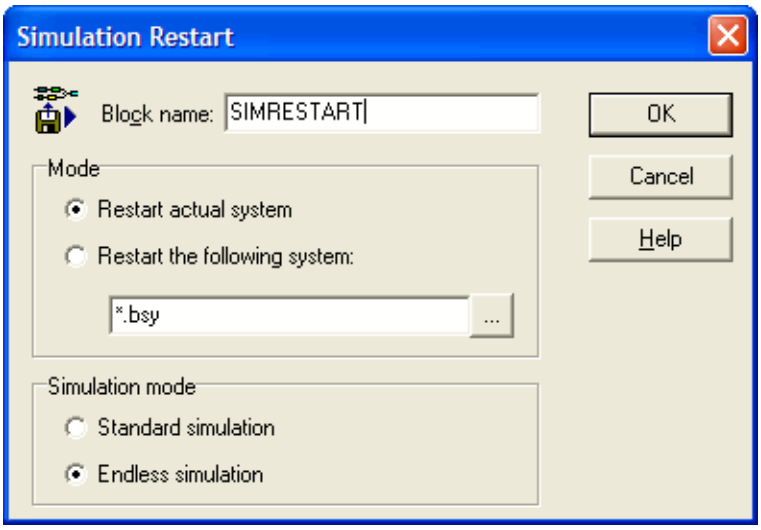


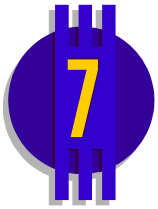
Simulation restart

Typename: SIMRESTART

Function: This block can be used to terminate the current simulation run and start a new run if a positive edge at the block input occurs. If desired the new simulation run can be executed based on another system structure than the currenty loaded in standard or endless mode.

Parameter dialog:



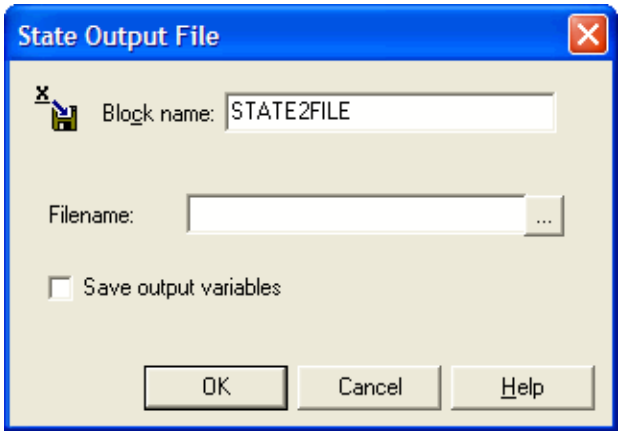


State output file

Typename: STATE2FILE

Function: This block type allows the storage of the current system state in a BSF file. Please find detailed information in the chapter *Saving and loading system states.*

Parameter dialog:

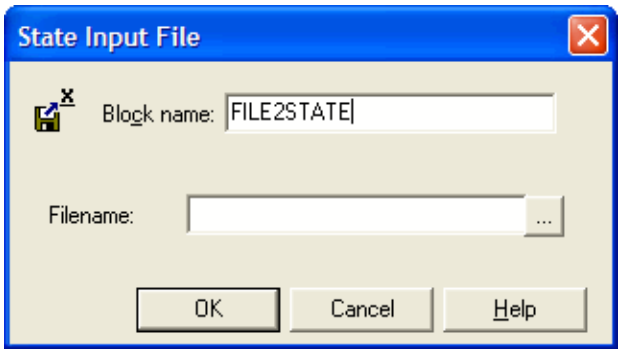


State input file

Typename: FILE2STATE

Function: This block type allows the loading of system states from a BSF file. Please find detailed information in the chapter *Saving and loading system states*.

Parameter dialog:



Drains

Time response

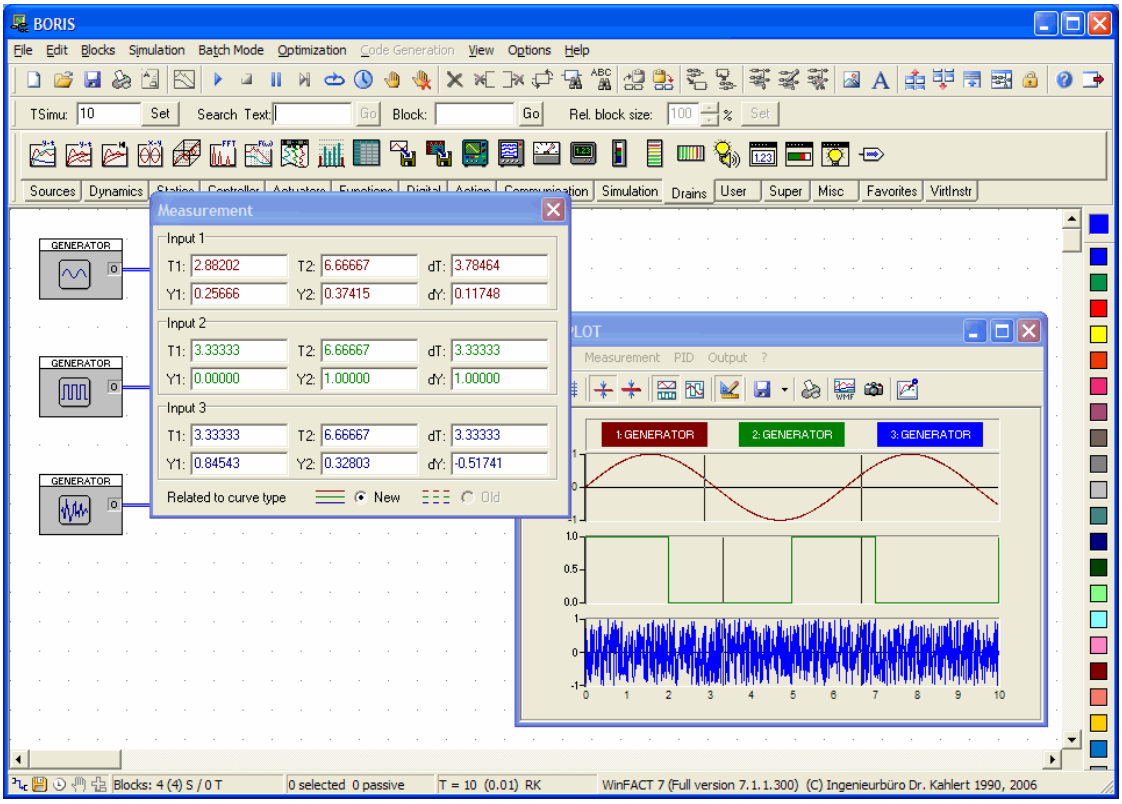
Typename: YTPLOT

Function: This output block allows the simultaneous graphical presentation of the time response of up to three variables. The curves can be drawn in a common or in sparate diagrams (with or without grid). The axes of coordinates can be scaled manually or automatically. The amplitude values can be represented linear or logarithmical. Additionally you can save the curves directly in SIM-files.

If the reset input R is connected, the response can be reset via a positive edge at this input, i. e. the response restarts from this point of time.

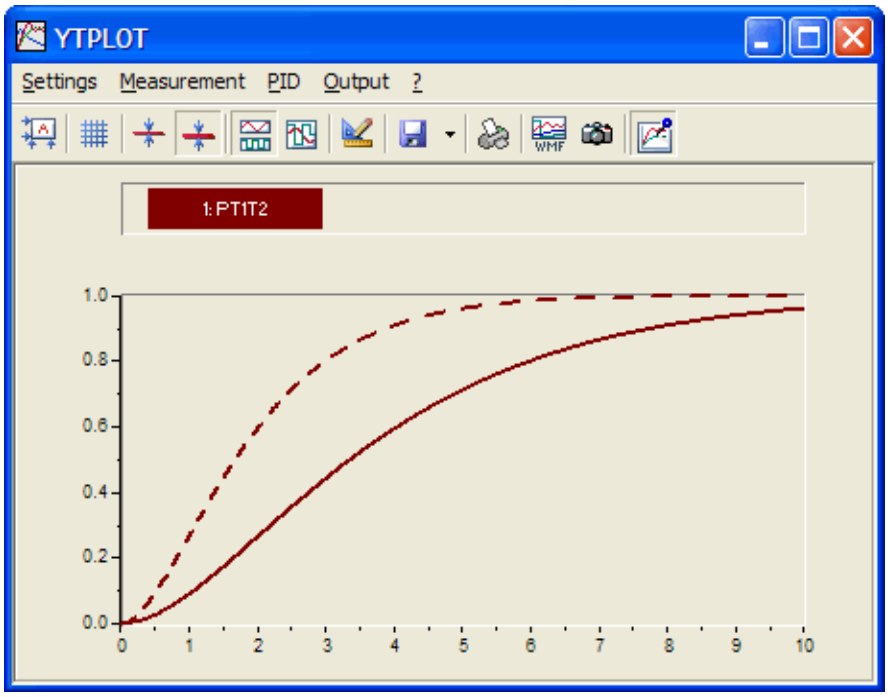
With a comfortable *measurement function* characteristic values can be determined from the diagrams. Because the time responses of the preceded simulation can be ”frozen” even tendencies of parameter variations – e. g. of control parameters – can be presented. If necessary the measurement function can be applied to the "frozen" curves. Especially for control engineering applications

(e. g. the determination of settling values) a *tolerance band* with a user-definable width can be inserted to the diagrams.



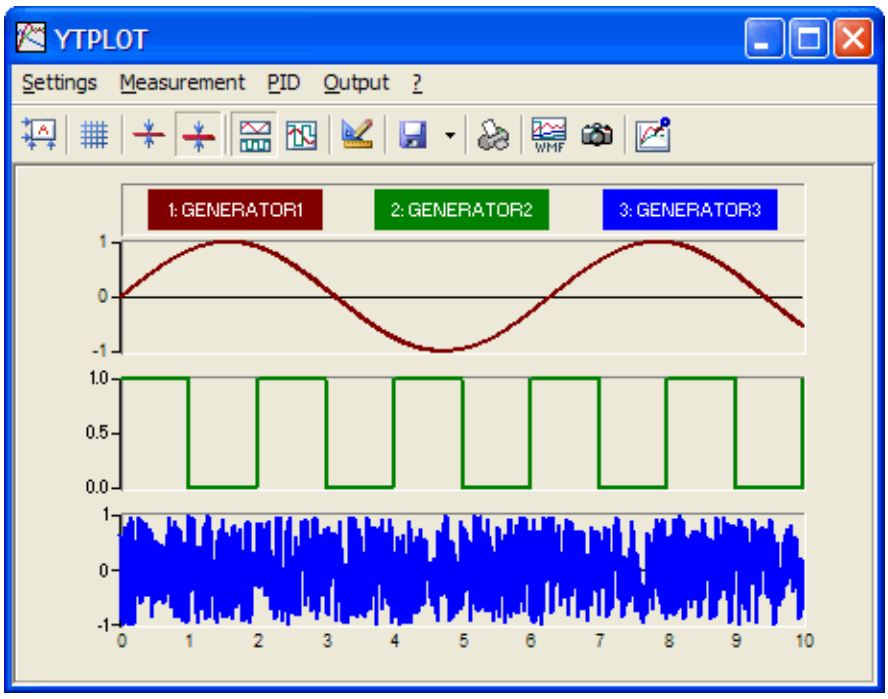
Display with activated measurement function

Furthermore PID-controllers can be designed directly from the time response display window according to the so-called *rules of thumb*. This function is described in detail in the chapter *Design of PID-controllers.*



Time response window with activated storage function:
The curve determined in the previous simulation run is drawn dashed

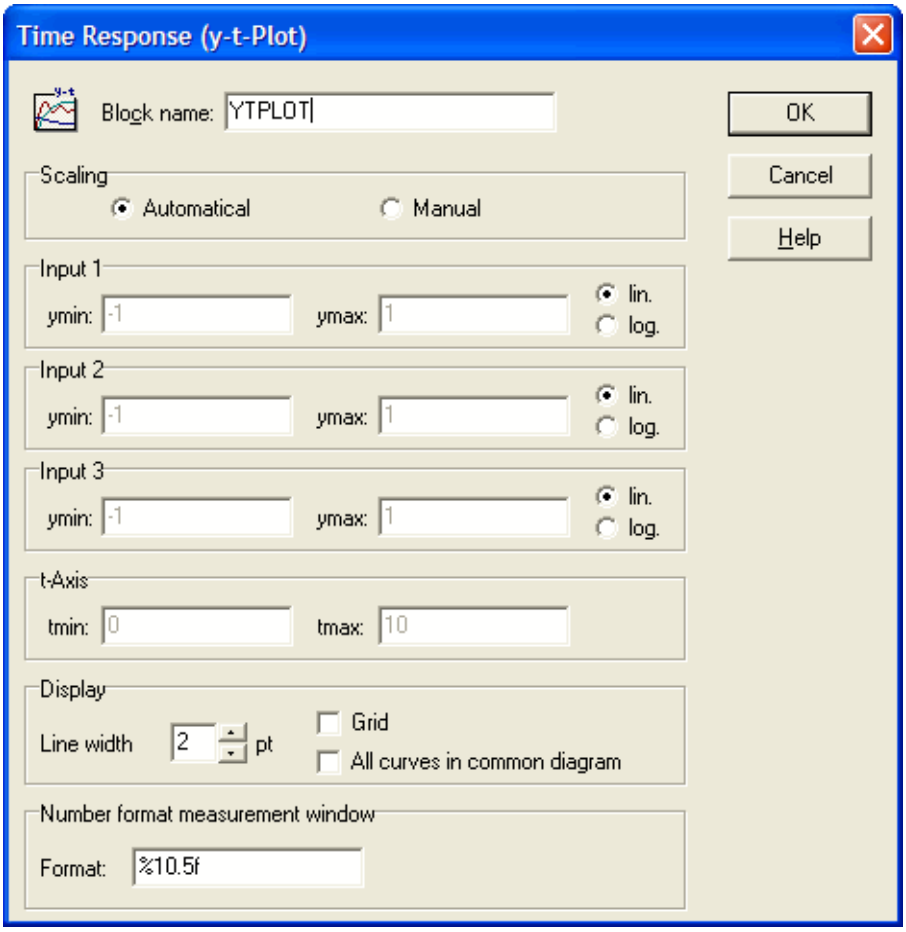
Display window:



Above the diagrams the names of the corresponding system blocks (in this case: *Generator 1*, *Generator 2* and *Generator 3*) are shown. The toolbar below the window menu allows the direct access to the most important menu options resp. settings of the parameter dialog which can also be reached with the menu option SETTINGS.

Remark: If the <Shift>-key is pressed while copying the screen to the clipboard via the camera symbol of the toolbar, the gray window background is replaced by a white one. This may be sensible if the screenshot is to be inserted into a text document that is to be printed later.

Parameter dialog:



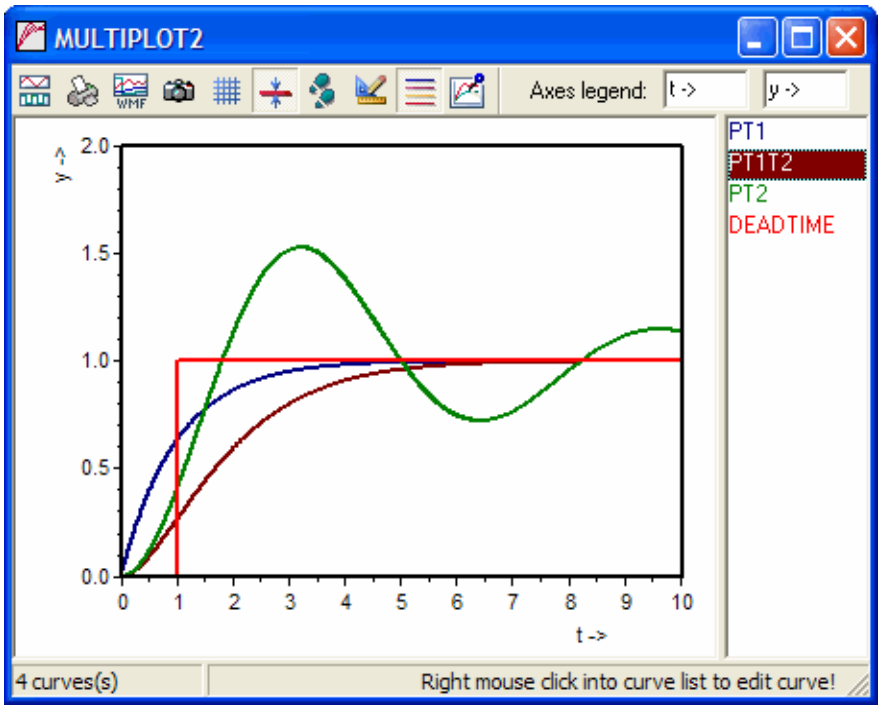
Restrictions: For each curve internally a maximum of 32000 points is available. If the number of simulation steps is higher, the values will be compressed internally ("compressed" presentation). If the number of simulation steps e. g. is 64000, only each second value will be presented graphically. Please consider this if you want to present very narrow pulses. In order to call the user´s attention to the compressed presentation in this case the message *COMPR!* appears in the upper left corner of the drawing window in red color on a yellow background.

Multi-Time response

Typename: MULTIPLOT2

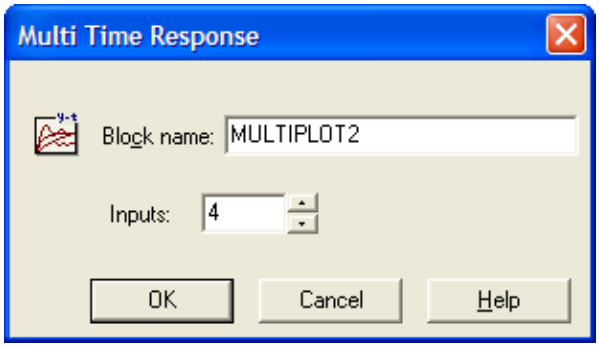
Function: This block allows the simultaneous display of up to 50 signals within a common or separate coordinate systems.

Display window:



The toolbar at the upper border of the window offers various functions which are self-explanatory. By a right mouse click within the listbox at the right window border single curves can be edited (e. g. scaled) or saved. If a manual scaling is selected for a common coordinate system, the minimum and maximum scale values specified for the *first* curve are used.

Parameter dialog:

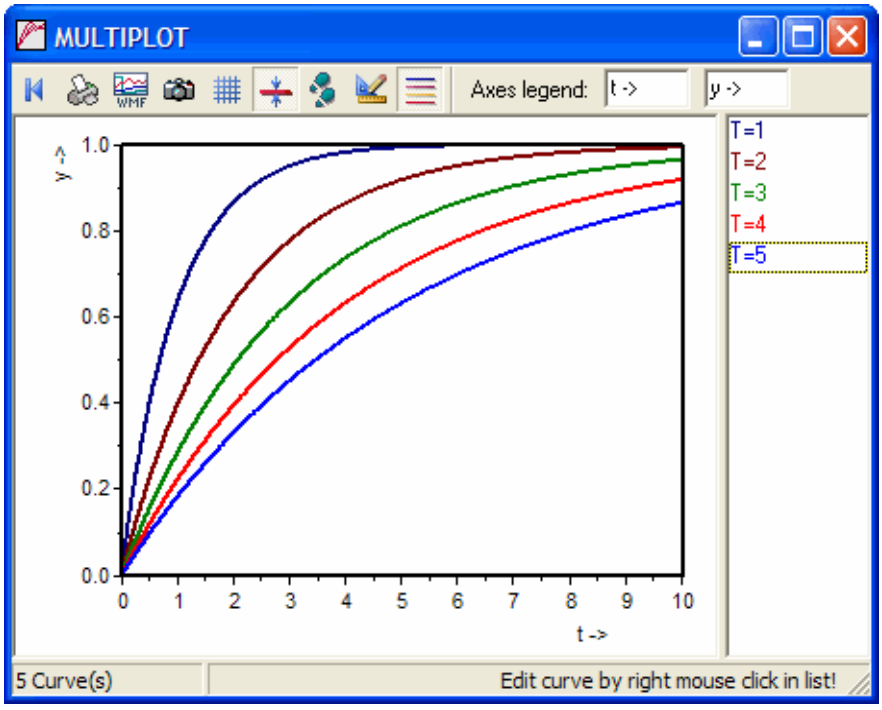


Multi-Plotter

Typename: MULTIPLOT

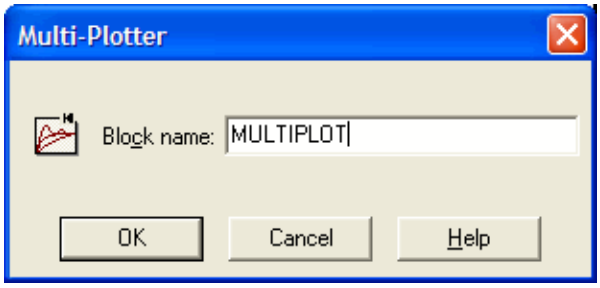
Function: This block type allows the recording of simulation results over any number of simulations and therefore is especially interesting for examinations concerning parameter variations (e. g. time constants) and batch runs. All recorded curves are listed at the right window border. By a right mouse click on a list item a context menu with further options can be opened. In the batch mode (see chapter *Batch mode simulation*) the curves are automatically labelled dependent on the current block parameter values that are modified.

Display window:



The toolbar at the upper border of the window offers some further options. Of special importance is the button which resets the plotter, i. e. deletes all recorded curves.

Parameter dialog:

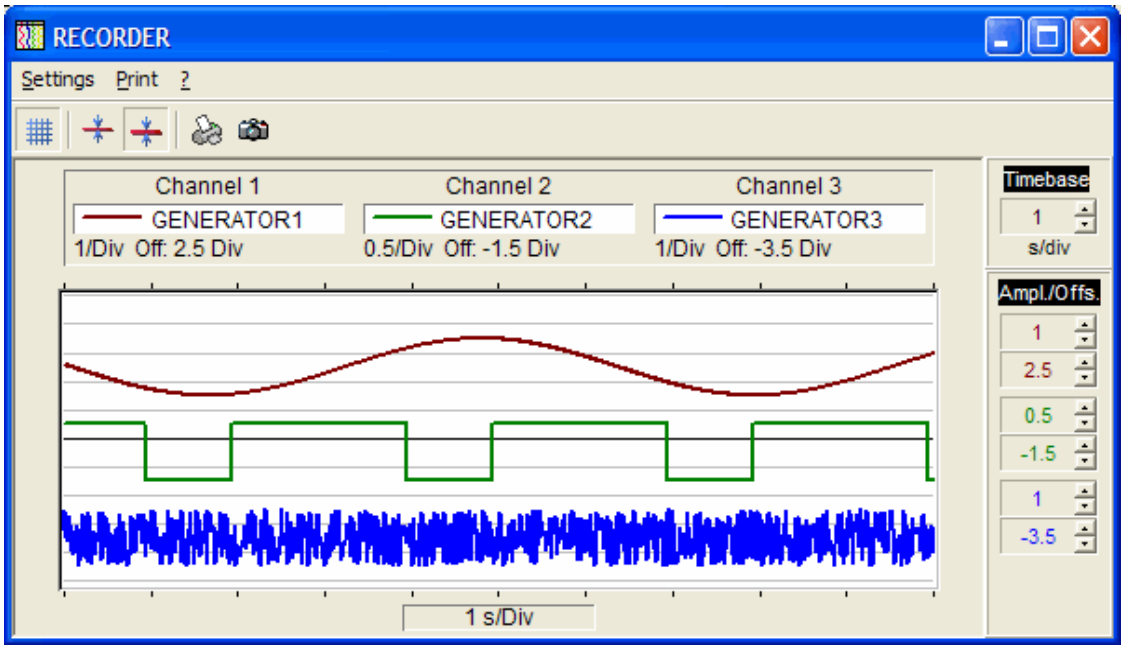


y-t-Recorder

Typename: RECORDER

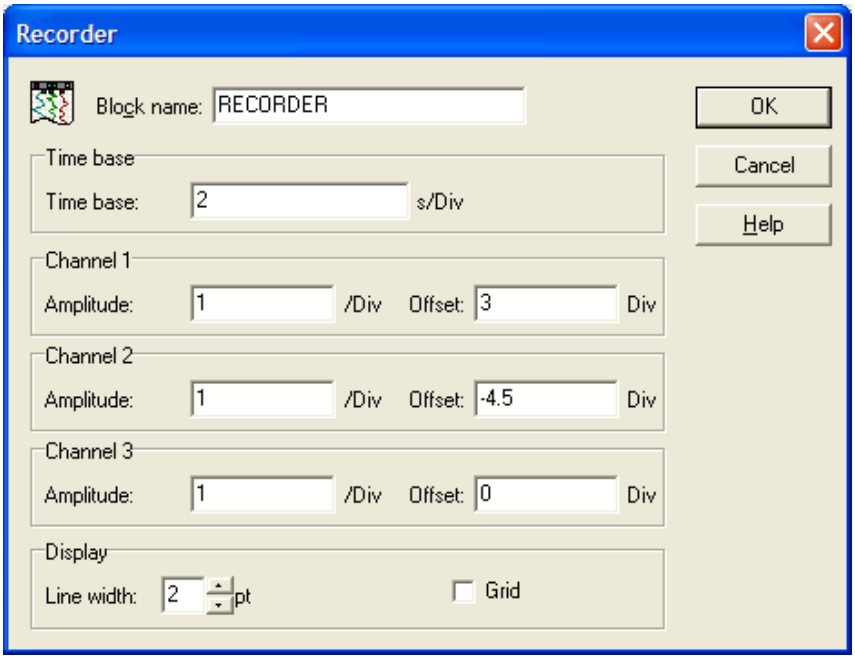
Function: This block emulates a y-t-recorder and is therefore especially suitable for long lasting recordings. The block can handle up to three input channels. Time base, sensitivity and offset of the single channels can be chosen separately.

Display window:



Above the diagrams the names of the corresponding system blocks (in this case *Generator 1*, *Generator* 2 and *Generator 3*) are shown. The toolbar below the window menu allows a direct access to the most important menu options resp. settings of the parameter dialog which can also be reached with the menu option SETTINGS.

Parameter dialog:

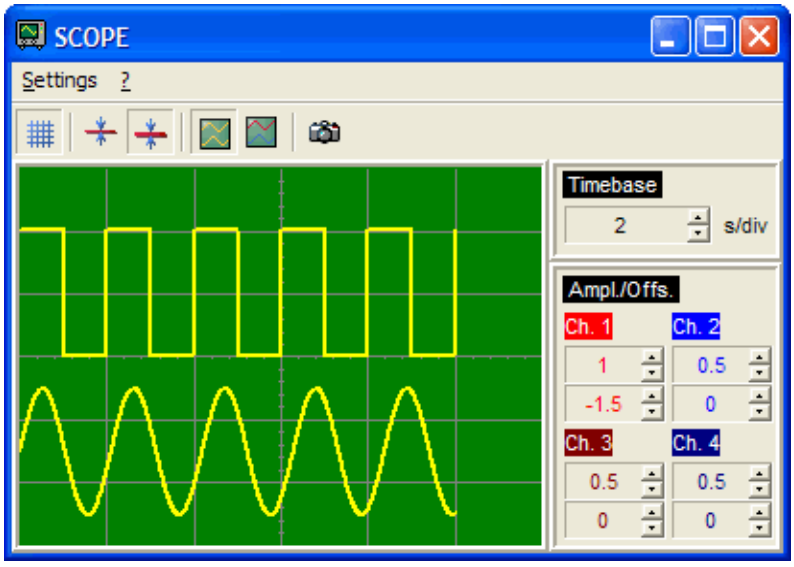


Oscilloscope

Typename: SCOPE

Function: This block emulates a four channel oscilloscope on the screen which can be resized in any way. Time base and sensitivity resp. offset of the channels are user-definable separately. All channels can be displayed multicolored.

Display window:



The channel sensitivity is displayed in the upper right corner of the display window. The toolbar below the window menu allows the direct access to the most important settings of the parameter dialog which can also be reached with the menu option SETTINGS.

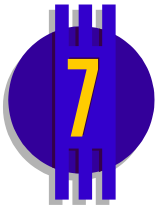
Restrictions: For each channel a maximum of 32000 points is available. If the number of simulation steps for the width of the oscilloscope is higher, the values will be compressed internally ("compressed" presentation). If the number of steps e. g. is 64000, only each second simulation step will be presented graphically. Please note this if you want to present very narrow pulses. In order to call the user´s attention to the compressed presentation in this case the message *COMPR!* appears in the upper left corner of the drawing window in red color on a yellow background (see also *Time response-block*).

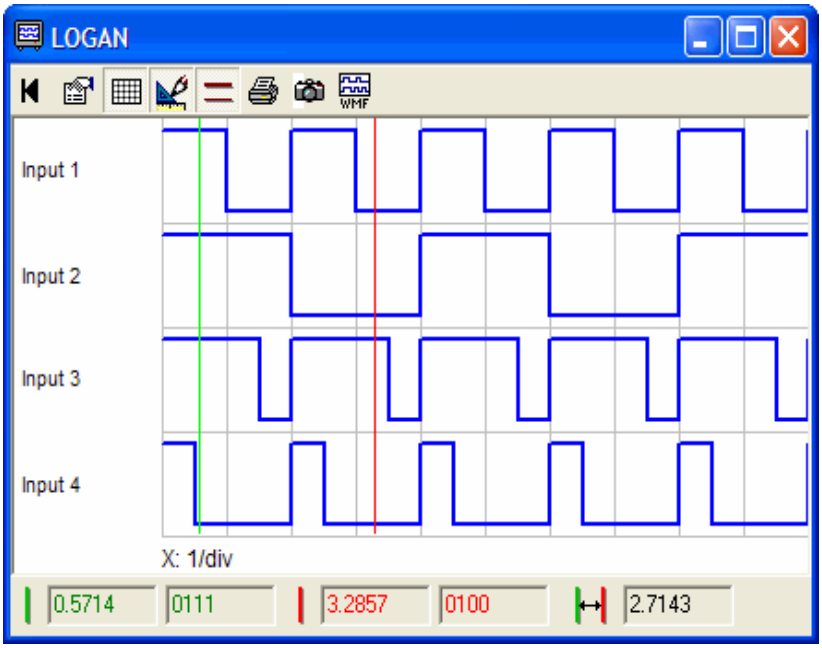
Logic analyzer

Typename: LOGAN

Function: This block realizes a logic analyzer with up to 49 data inputs and a reset input. The logic analyzer offers a comfortable measurement mode.

Display window:





The toolbar at the upper window borders offers access to all functions of the logic analyzer:

Activates the auto-reset mode. If this mode is active the display is cleared automatically if the right window border is reached and the recording is restarted.

Allows the specification of time base and names of data inputs.

Grid on/off

Measurement mode on/off.

Line mode 1 pt/2 pt

Prints window contents

Copies window contents to clipboard

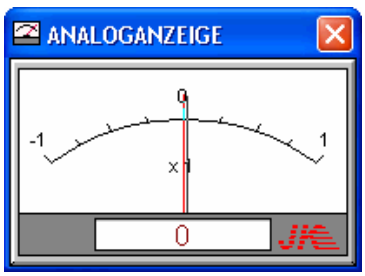
Saves window contents to a WMF-file

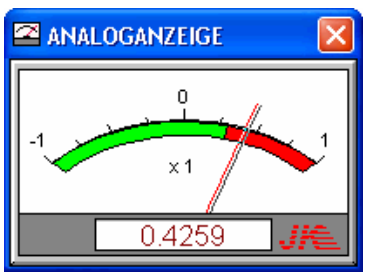
Analog meter

Typename: ANALOGMETER

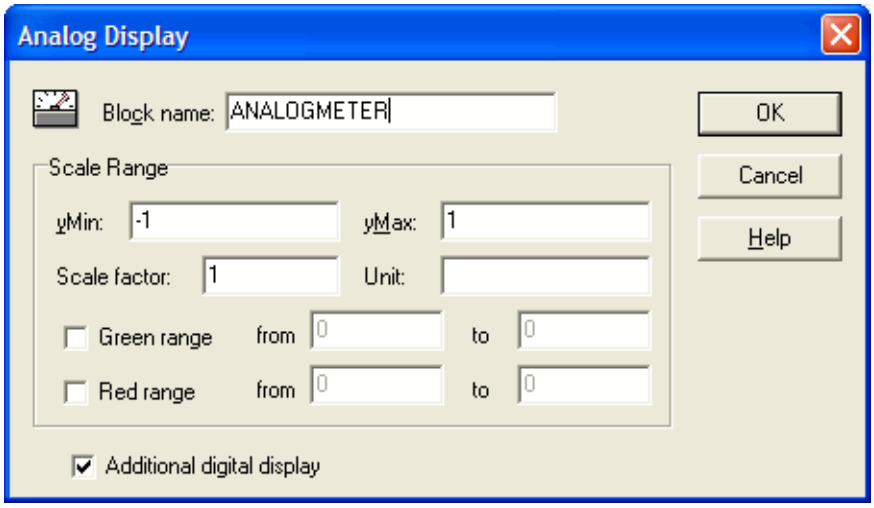
Function: This block emulates an analog instrument – optionally with a digital display. The corresponding display window is not resizable. Scale range and scale factor are user-definable. Furthermore special scale ranges can be presented in red or green color and the display can be combined with a unit (e. g. "V").

Display window:





Parameter dialog:



For the scale range $y_{\min}$, $y_{\max}$ it is recommended to specify integer values.

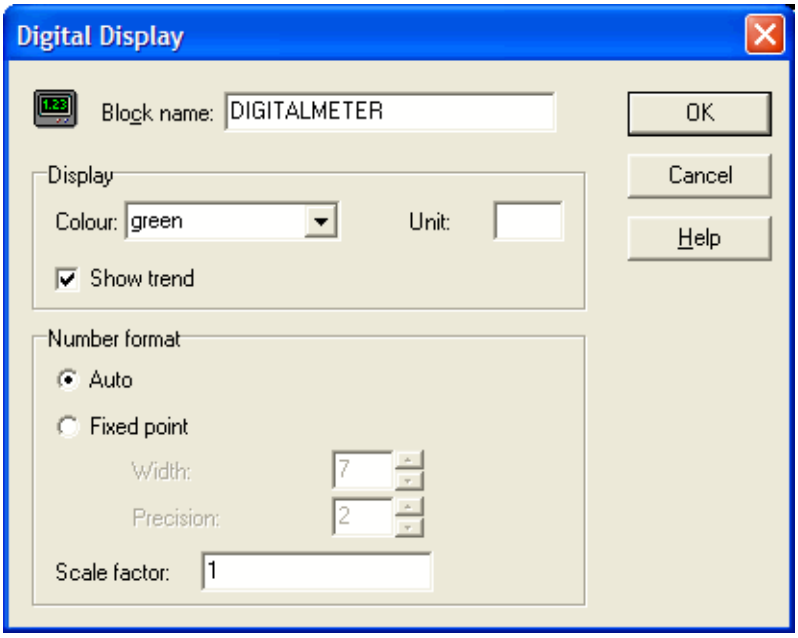
Digital meter

Typename: DIGITALMETER

Function: This block represents a digital instrument. The corresponding display window is not resizable. Optionally an additional scaling factor, a unit (e. g. "V") and a tendency display (input value increasing/decreasing) can be specified. Alternatively the numerical value is presented with a minimum of numbers or a specified number of digits.

Display window:

Parameter dialog:



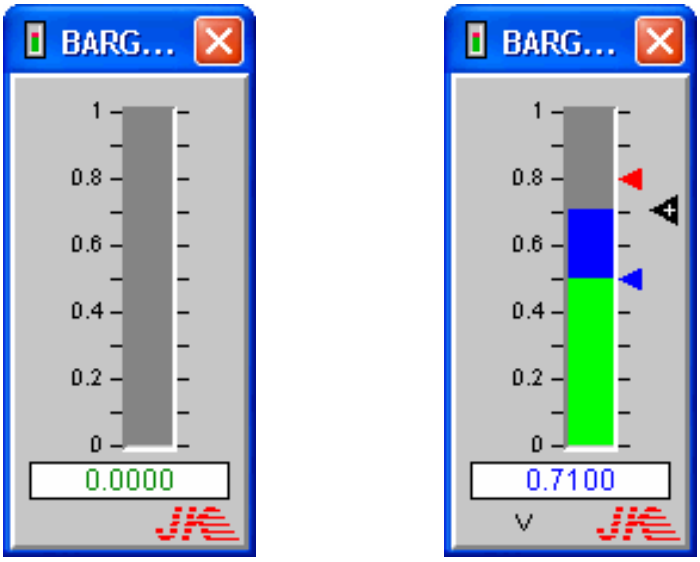
Bar graph

Typename: BARGRAPH

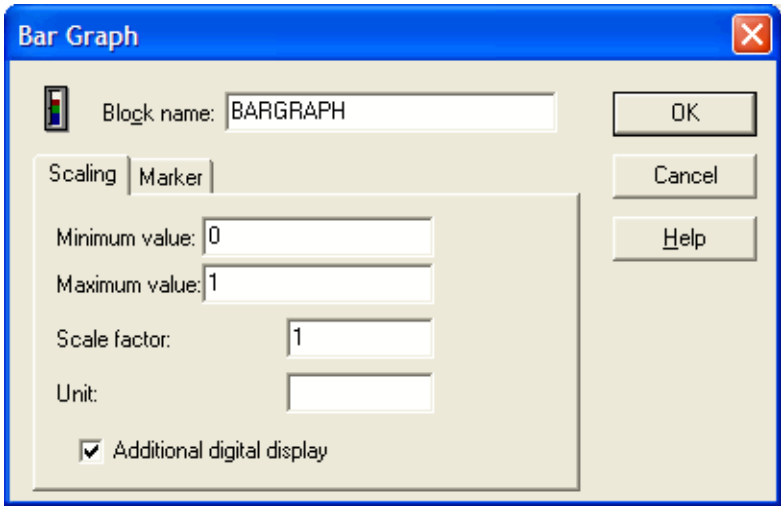
Function: This output block represents a bar graph which can work in different modes. The following options can be chosen:

- Setting of up to two markers which let the display change its color when they are exceeded.
- Setting of markers for the minimally resp. maximally reached value.
- Additional digital display , scale factor and unit (e. g. "V")

Display window:



Parameter dialog:



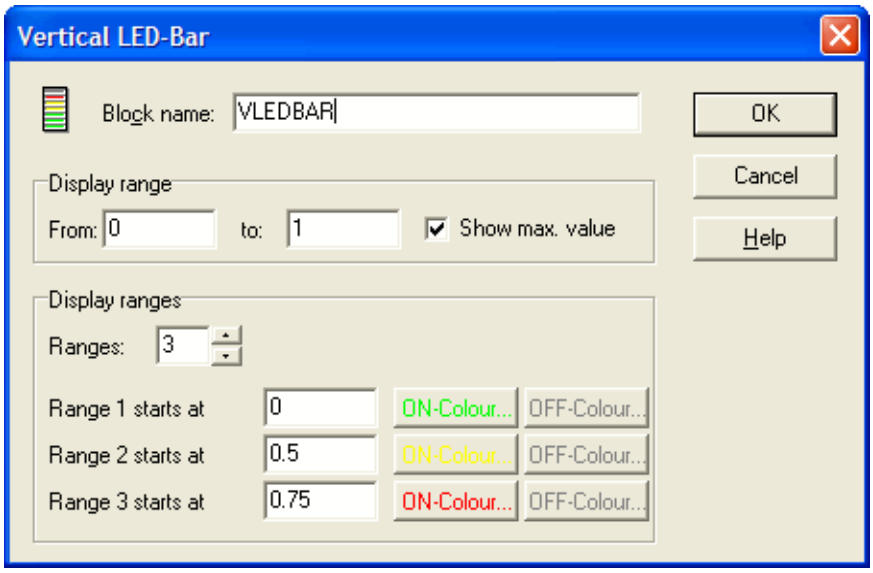
Vertical LED bar

Typename: VLEDBAR

Function: This block represents a multi-color vertical LED bar with an optional peak value storage.

Display window:

Parameter dialog:



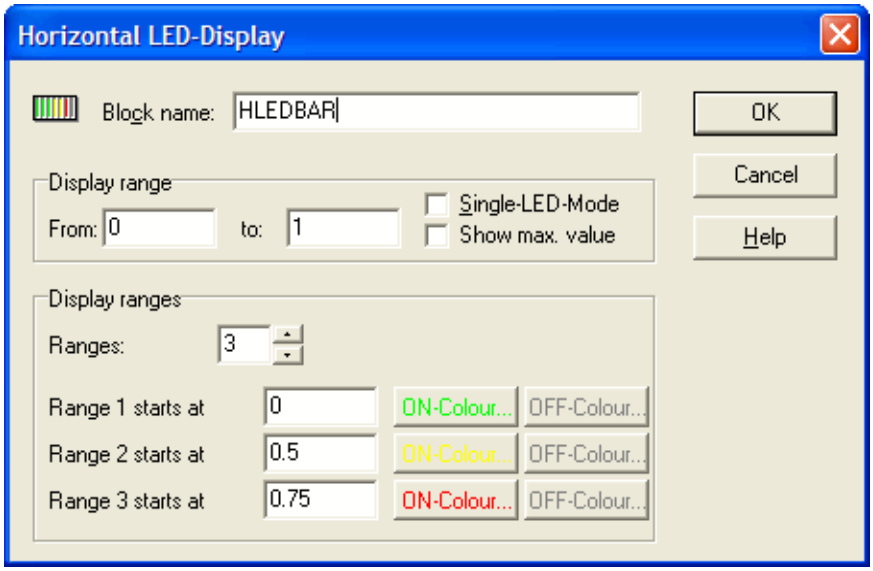
Horizontal LED bar

Typename: HLEDBAR

Function: This block represents a multi-color horizontal LED bar with an optional peak value storage.

Display window:

Parameter dialog:

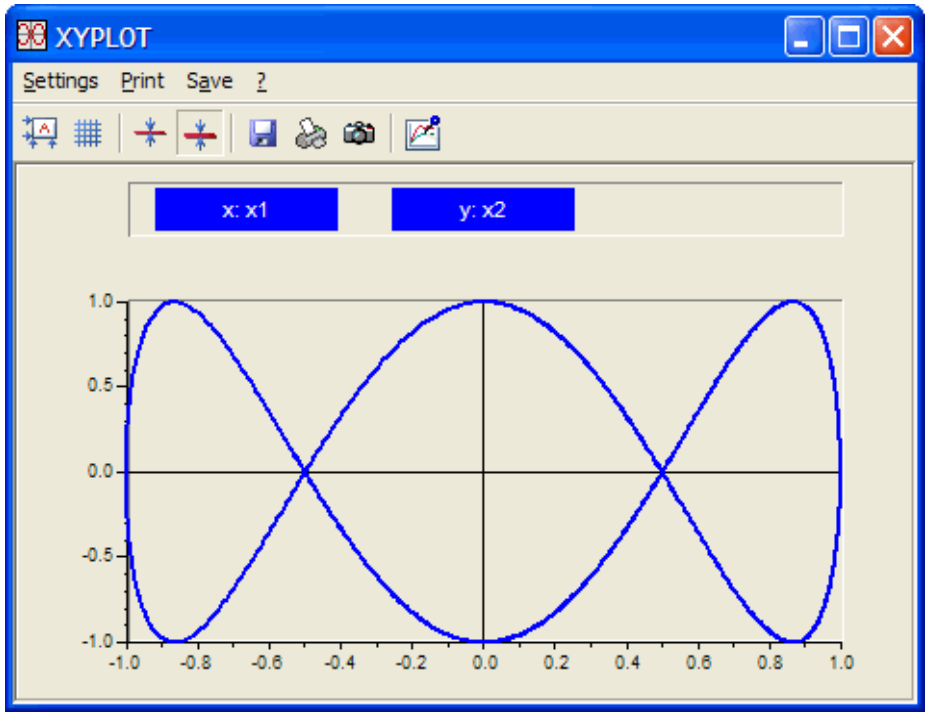


Trajectory display (x-y-plot)

Typename: XYPLOT

Function: The trajectory display allows the presentation of two input variables $x(t)$ and $y(t)$ in the *x*-*y*-domain with the time *t* as the curve parameter. The corresponding display window is resizable in any direction. The axes of coordinates can be scaled manually or automatically (with or without grid). A storage function is available for the previous simulation (as you get it for the time response block).

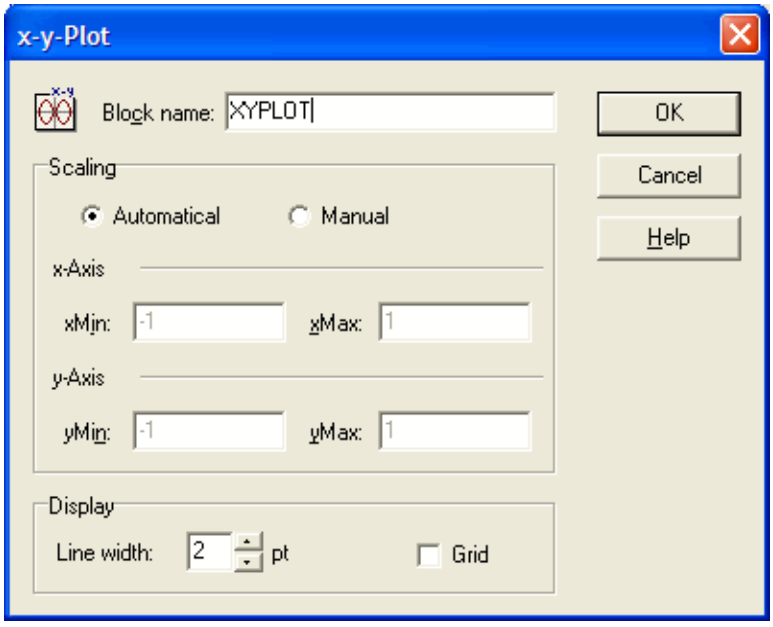
Display window:



Above the diagrams the names of the corresponding system blocks (in this case *x1* and *x2*) are displayed. The toolbar below the window menu allows a direct access to the most important settings of the parameter dialog which can also be reached with the menu option SETTINGS.

Remark: If the <Shift>-key is pressed while copying the screen content to the clipboard via the camera symbol of the toolbar, the gray window background is replaced by a white one. This may be sensible if the screenshot is to be inserted into a text document that is to be printed later.

Parameter dialog:



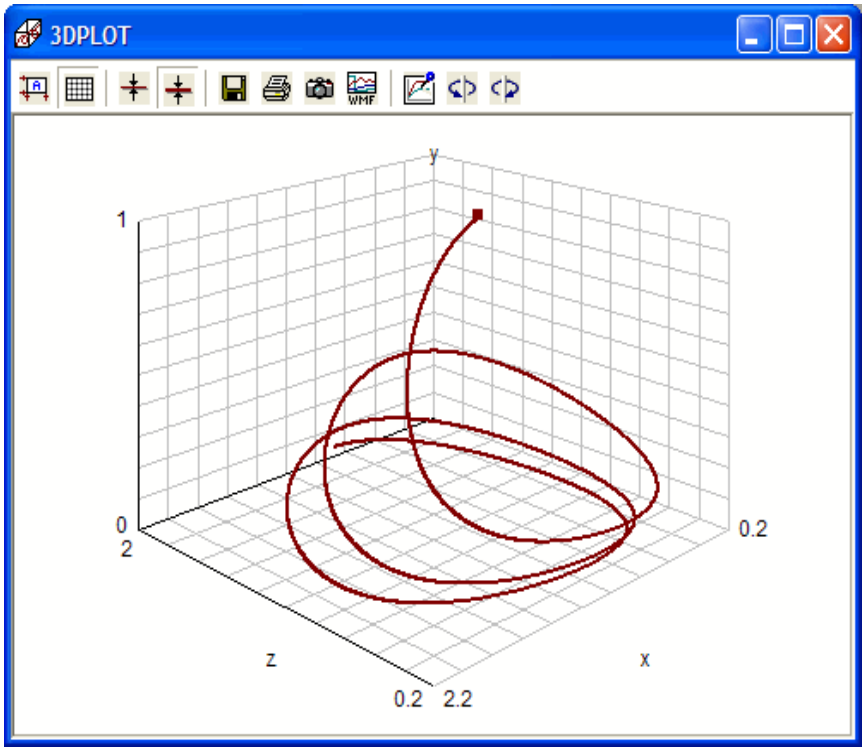
Restrictions: For each curve internally a maximum of 32000 points is available. If the number of simulation steps is higher the values will be compressed internally (”compressed” presentation). If the number of simulation steps e. g. is 64000, only each second simulation step will be presented graphically. Please note this if you want to present very narrow pulses. In order to call the user´s attention to the compressed presentation, in this case the message *COMPR!* appears in the upper left corner of the drawing window in red color on a yellow background (see *Time response-block*).

3D-Plotter

Typename: 3DPLOT

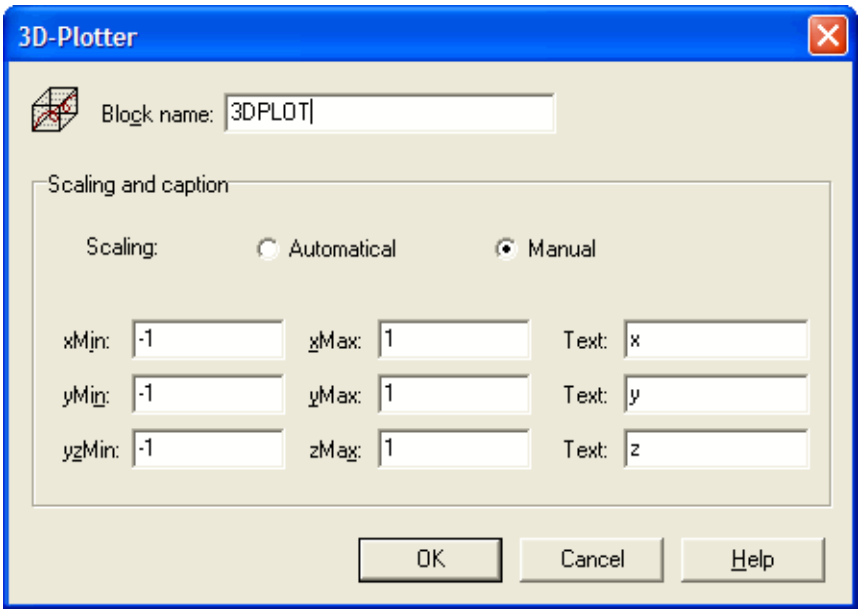
Function: This block allows the representation of three input variables $x(t)$, $y(t)$ and $z(t)$ in the *x*-*y*-*z*-domain with the time *t* as curve parameter. The diagram can be scaled automatically or manually and rotated in two directions.

Display window:



The toolbar at the upper border of the window offers various functions which are self-explanatory.

Parameter dialog:

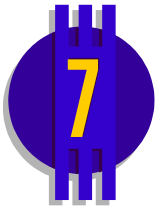


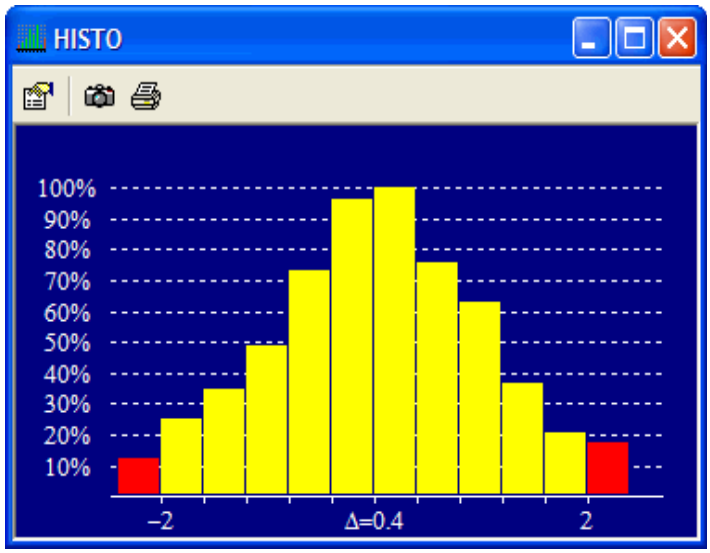
Histogram

Typename: HISTO

Function: This block allows the calculation and graphical representation of frequency distributions (histograms). The display range as well as the input variable intervals can be specified.

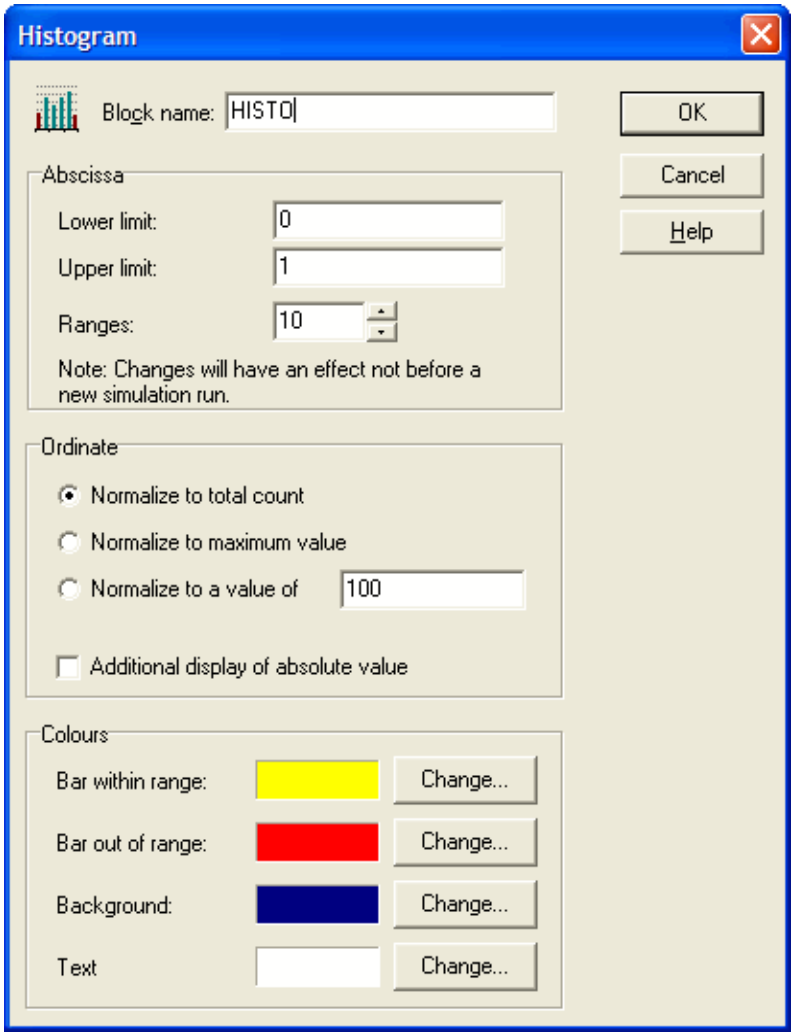
Display window:





The toolbar at the upper border of the window offers various functions which are self-explanatory.

Parameter dialog:

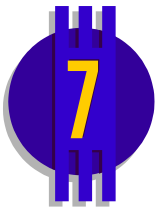


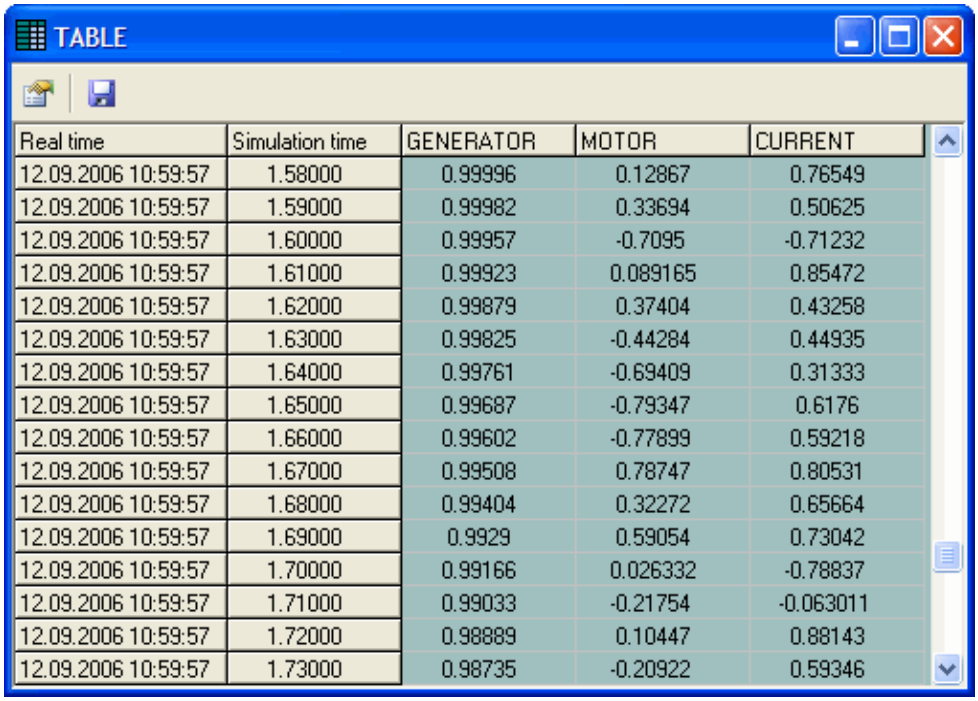
Table

Typename: TABLE

Function: This block creates a table output of up to 50 input signals. The output interval can be specified as well as the output format for the time column (real time and/or simulation time).

Display window:

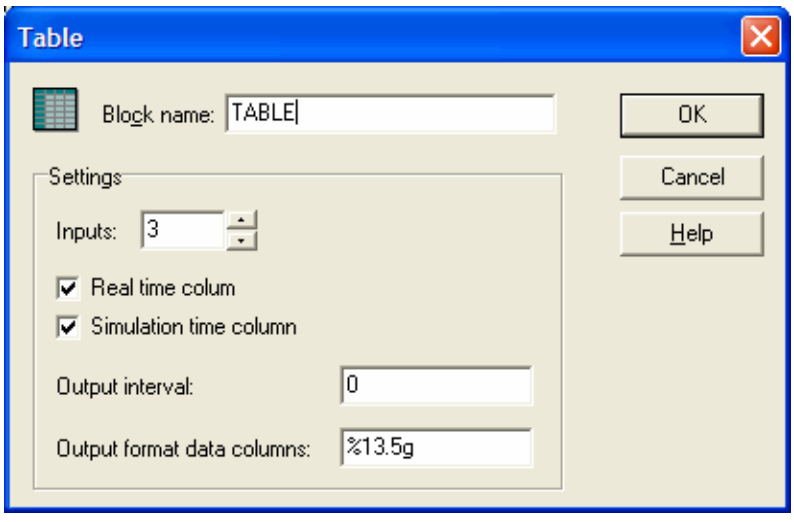




Real time	Simulation time	GENERATOR	MOTOR	CURRENT
12.09.2006 10:59:57	1.58000	0.99996	0.12867	0.76549
12.09.2006 10:59:57	1.59000	0.99982	0.33694	0.50625
12.09.2006 10:59:57	1.60000	0.99957	-0.7095	-0.71232
12.09.2006 10:59:57	1.61000	0.99923	0.089165	0.85472
12.09.2006 10:59:57	1.62000	0.99879	0.37404	0.43258
12.09.2006 10:59:57	1.63000	0.99825	-0.44284	0.44935
12.09.2006 10:59:57	1.64000	0.99761	-0.69409	0.31333
12.09.2006 10:59:57	1.65000	0.99687	-0.79347	0.6176
12.09.2006 10:59:57	1.66000	0.99602	-0.77899	0.59218
12.09.2006 10:59:57	1.67000	0.99508	0.78747	0.80531
12.09.2006 10:59:57	1.68000	0.99404	0.32272	0.65664
12.09.2006 10:59:57	1.69000	0.9929	0.59054	0.73042
12.09.2006 10:59:57	1.70000	0.99166	0.026332	-0.78837
12.09.2006 10:59:57	1.71000	0.99033	-0.21754	-0.063011
12.09.2006 10:59:57	1.72000	0.98889	0.10447	0.88143
12.09.2006 10:59:57	1.73000	0.98735	-0.20922	0.59346

The toolbar at the upper border of the window offers various functions which are self-explanatory.

Parameter dialog:



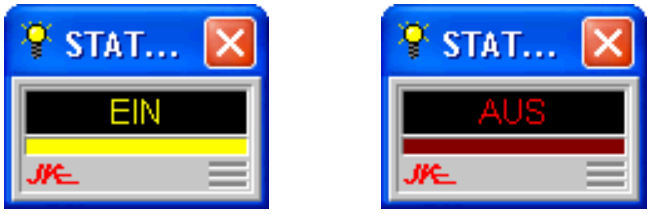
If the output interval is set to 0 (default), *each* simulation step is displayed.

Status display

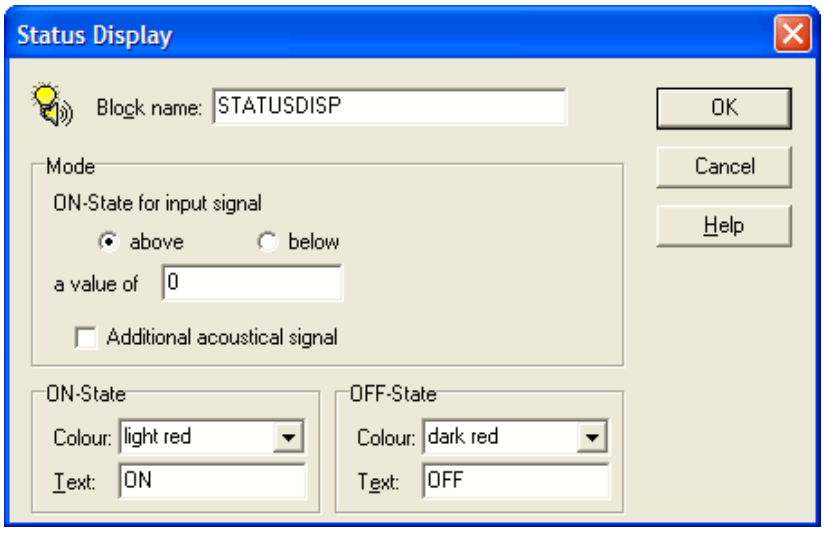
Typename: STATUSDISP

Function: This block produces a visual and (optionally) an additional acoustic signal if a threshold value is crossed in positive or negative direction. The text appearing in the display can separately be specified for the on/off-state.

Display window:



Parameter dialog:

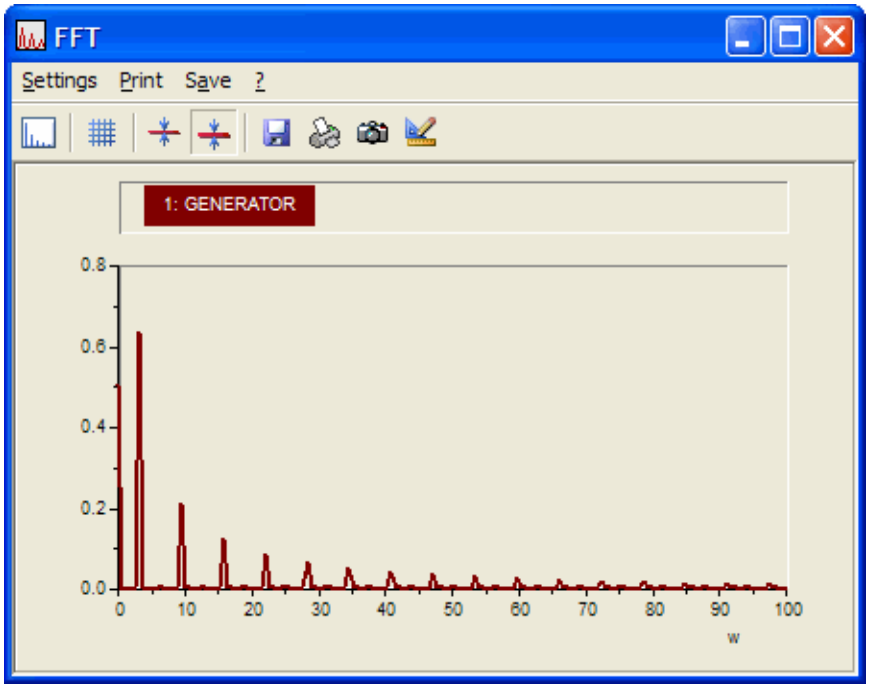


Fast Fourier Transformation (FFT)

Typename: FFT

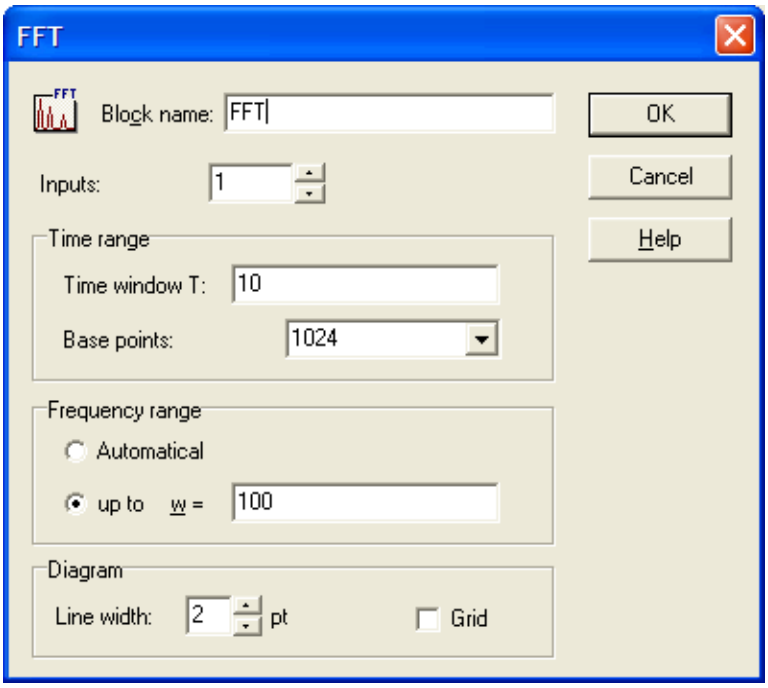
Function: This block allows the calculation of the amplitude spectrum of the input signal by the Fast-Fourier-Transformation (FFT). The time window and the number of base points for the transformation can be chosen by the user. For the representation of periodic signals the block can be used in a *discrete* mode; in this case only the maxima of the determined spectrum are displayed in line form.

Display window:



Above the diagram the name of the corresponding system block (in this case *Generator*) will be displayed. The toolbar below the window menu allows the direct access to the most important settings of the parameter dialog which can also be reached with the menu option SETTINGS.

Parameter dialog:



The different parameters have the following meaning:

- The *time window T* determines the length of the time window from which the base points for the transformation are taken. Basically you should choose the simulation time so that the transformation follows after the end of the simulation. The length of the time window may be not bigger than the simulation time, otherwise no transformation and therefore no display can be executed.
- The time window determines the smallest frequency $\omega_{\min}$ for which the spectrum will be calculated. It is

$$\omega_{\min} = \frac{2\pi}{T}.$$

- The number of *base points* for the transformation is always a power of two and determines the upper frequency of the calculated spectrum. If you have a number of *n* base points for transformation you will get an upper frequency of

$$\omega_{\max} = \left(\frac{n}{2} - 1\right)\omega_{\min}.$$

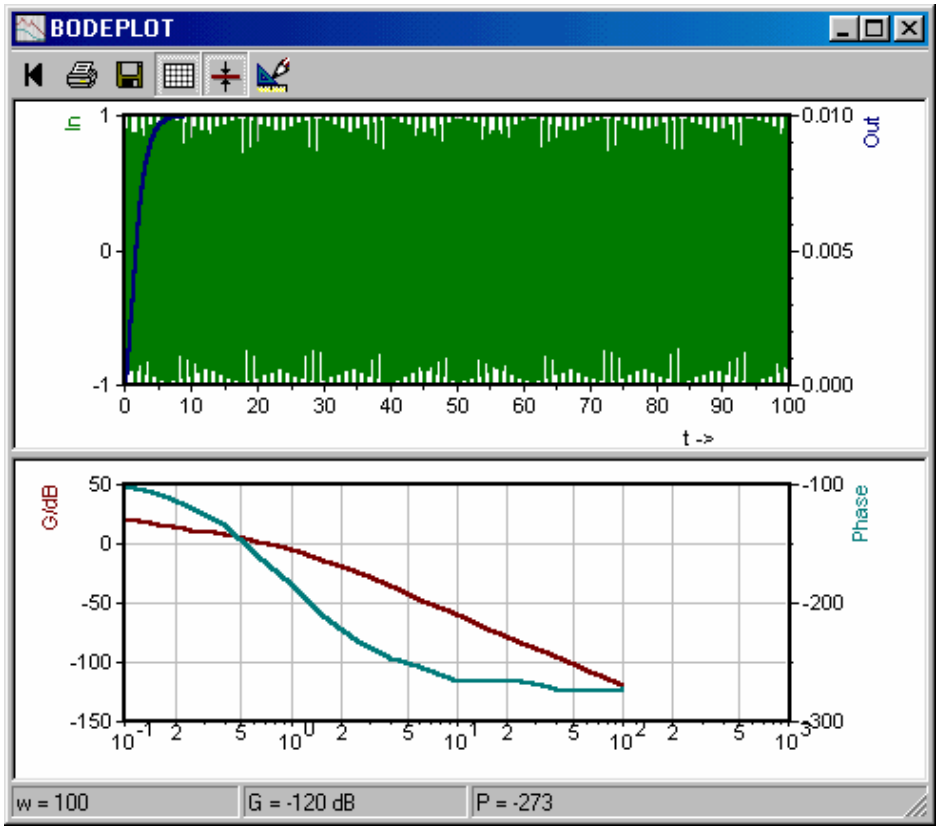
- The scaling of the frequency axis is influenced by the settings in the group field *Frequency range*. In the case of an automatical scaling the whole range up to $\omega_{\max}$ will be displayed.
- Within the group field *Diagram* the line width of the curve can be set and a grid can be activated additionally.

Frequency Response Plotter

Typename: BODEPLOT

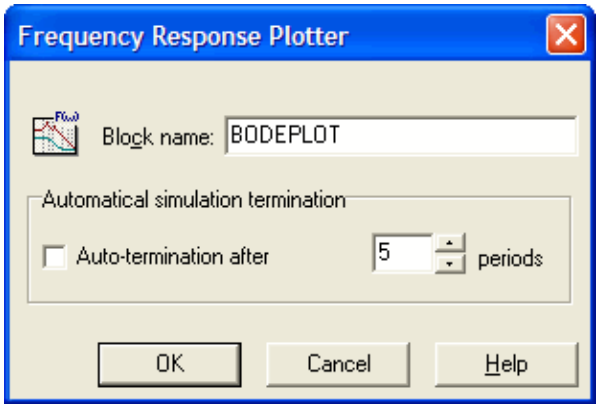
Function: This block allows the calculation of the gain (in dB) and phase (in degree) of two sinusoidal signals connected to the block inputs. Both signals must be of the same frequency. In combination with the batch mode this block can be used for the automatical calculation and representation of frequency responses (Bode resp. Nyquist plots). For details see chapter *Batch mode simulation*.

Display window:



In the upper part of the window the time responses of the input (block input 1) and output signal (block input 2) are displayed. The status bar contains the values determined for frequency, gain and phase. The lower part of the window contains in case of multiple simulations (e. g. in the batch mode) the resulting Bode plot with the gain (red) and phase (blue-green). By the reset-button of the toolbar both curves can be cleared (i. e. all recorded values of the Bode plot are deleted).

Parameter dialog:



If the option *Autom. Auto-termination after xxx periods* is activated, a running simulation is automatically terminated when the specified number of periods

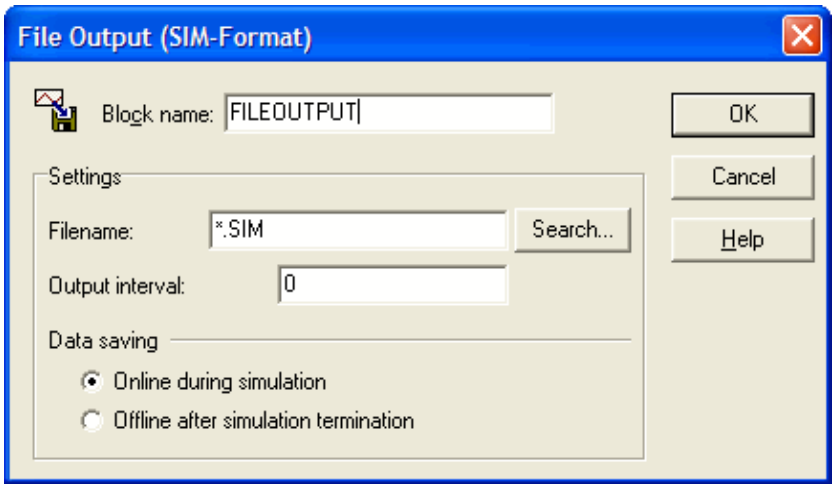
has been counted. This may result in a significant reduction of computation time especially when using the batch mode for calculating complete frequency responses. In some cases this option however might result in an unexact determination of phase values; so it should be used with care.

File output

Typename: FILEOUTPUT

Function: Allows the output of a signal $u(t)$ in form of pairs of values (t_i, u_i) to a file of type SIM.

Parameter dialog:



If you don´t specify an extension for the *Filename,* the extension SIM will be used. With the button *Search* a file input dialog can be called. The parameter *Output interval* determines the time distance of the storage of the pairs of values. If you choose an output interval of e. g. 0.5 and a simulation step size of 0.1, only each fifth value will be saved. If the output interval is set to 0 (default) each of the simulated values will be saved.

The storage of the data can take place *online* during the simulation or *offline* after the end of the simulation. The last setting is especially recommendable for real time simulations because in this case no access to the hard disc is necessary during the simulation. In the case of an offline storage the data will be temporarily hold in the memory of the computer.

If the block is to be used within the batch mode (see chapter *Batch mode simulation*), the substring {#b} can be used as a part of the filename; if not, the specified file is overwritten with each simulation of the batch job. If the sub-

string is used, it is replaced by the number of the current simulation of the batch job so that each simulation result is saved in a new file.

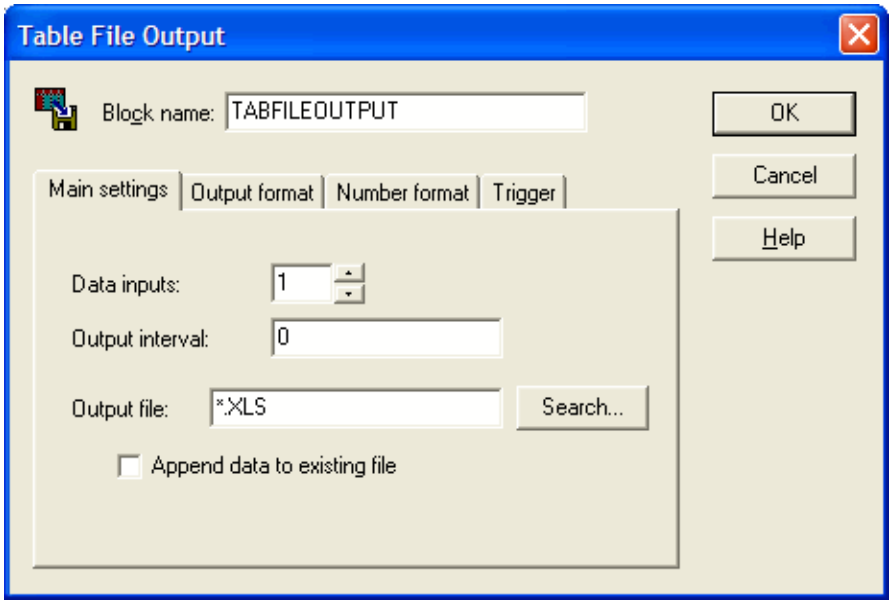
Example: The name for the output file may be TEST{#b}.SIM. If now a batch run with ten simulations is started, the output files will be named TEST1.SIM, TEST2.SIM, ..., TEST10.SIM. The comment (header) of each file will automatically contain information about the batch run and the parameter values used for the concrete simulation. In the non-batch mode the substring is simply ignored; in this case the file TEST.SIM would be created.

Table file output (Excel-format)

Typename: TABFILEOUTPUT

Function: This block allows the storage of up to 49 time responses in the format of a table. The saved data can be processed directly with spread sheet applications (e. g. Excel) in an easy way. The data then get the extension XLS. The storage of the data can be executed continuously or externally triggered. If the clock input C is open the data will be saved continuously. If it is connected, the input data will only be saved in the case of HIGH-level or a positive edge at the clock input.

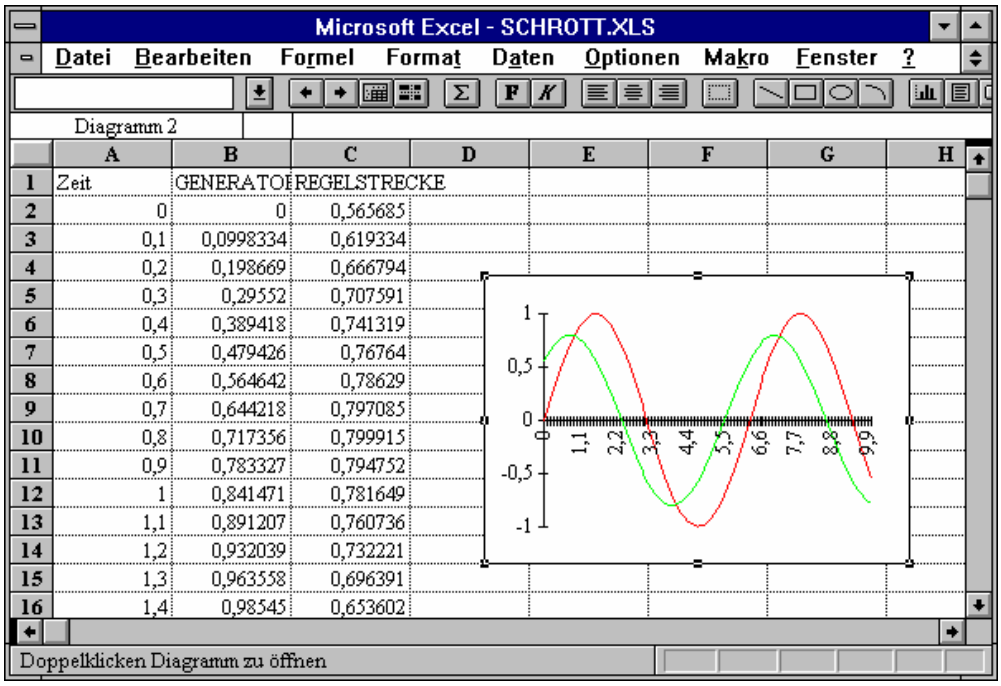
Parameter dialog:



The parameters have the following meaning:

- *Data inputs* specifies the number of block data inputs (1 to 49).

- *Output file* specifies the name of the file in which the data will be saved. With the menu option *Search* you can call a file open dialog. The time responses will be saved with a time intervall specified by *Output interval*. If this intervall is 0, the values will be saved at each simulation step.
- If the option *Block names as column caption* is activated, the names of the blocks which correspond to the input variables will be written to the first line of the file.
- The *Time format* option specifies the kind of time output within the first column of the file.
- The option *Column separator* specifies in which way the different columns of the table are separated. The setting of the option *Decimal separator* specifies which character is chosen for the presentation of the decimal point.



	A	B	C
1	Zeit	GENERATOR	REGELSTRECKE
2	0	0	0,565685
3	0,1	0,0998334	0,619334
4	0,2	0,198669	0,666794
5	0,3	0,29552	0,707591
6	0,4	0,389418	0,741319
7	0,5	0,479426	0,76764
8	0,6	0,564642	0,78629
9	0,7	0,644218	0,797085
10	0,8	0,717356	0,799915
11	0,9	0,783327	0,794752
12	1	0,841471	0,781649
13	1,1	0,891207	0,760736
14	1,2	0,932039	0,732221
15	1,3	0,963558	0,696391
16	1,4	0,98545	0,653602

Importing data into EXCEL

If the block is to be used within the batch mode (see chapter *Batch mode simulation*), the substring {#b} can be used as a part of the filename; if not, the specified file is overwritten with each simulation of the batch job. If the substring is used, it is replaced by the number of the current simulation of the batch job so that each simulation result is saved in a new file.

Example: The name for the output file may be TEST{#b}.XLS. If now a batch run with ten simulations is started, the output files will be named TEST1.XLS,

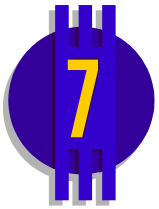
TEST2.XLS, ..., TEST10.XLS. The comment (header) of each file will automatically contain information about the batch run and the parameter values used for the concrete simulation. In the non-batch mode the substring is simply ignored; in this case the file TEST.XLS would be created.

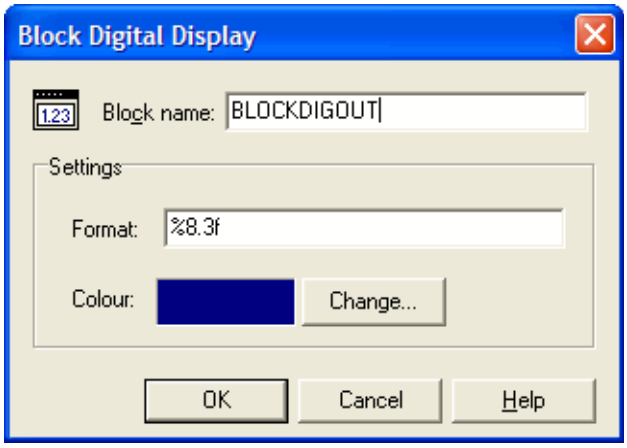
Block digital meter

Typename: BLOCKDIGOUT

Function: This block represents a digital meter with a user-defined output format. In contrast to the standard digital meter (DIGITALMETER block) this block does not realize the display within a separate window but within the block itself.

Parameter dialog:



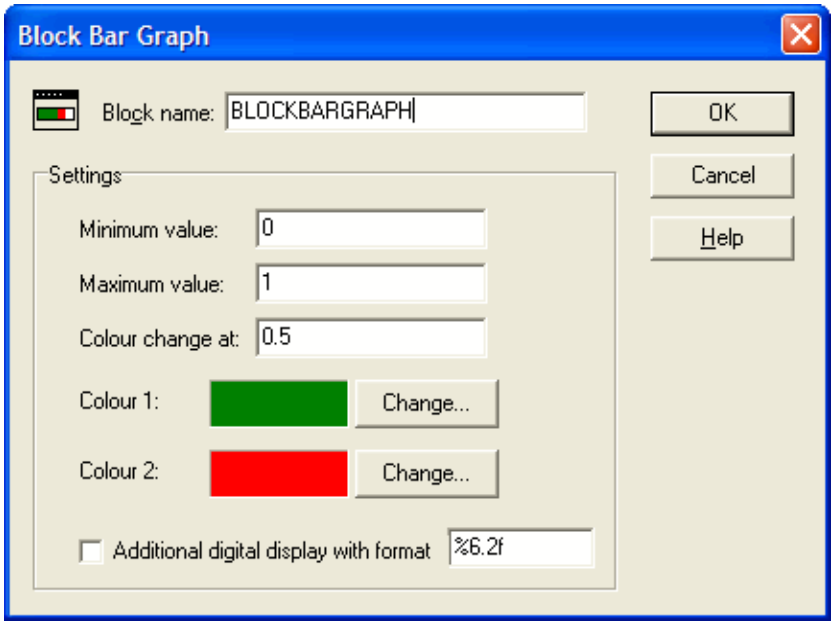


Block bar graph

Typename: BLOCKBARGRAPH

Function: This block represents a multi-colour bar graph with additional digital output. In contrast to the standard bar graph (BARGRAPH block) this block does not realize the display within a separate window but within the block itself.

Parameter dialog:

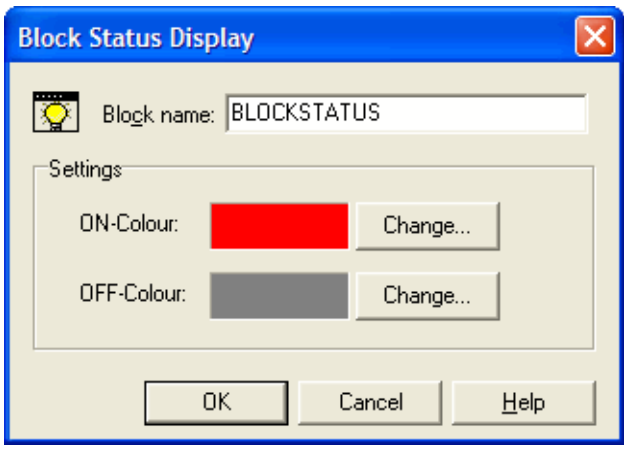


Block status display

Typename: BLOCKSTATUS

Function: This block represents a status display. In contrast to the standard status display (STATUSDISP block) this block does not realize the display within a separate window but within the block itself.

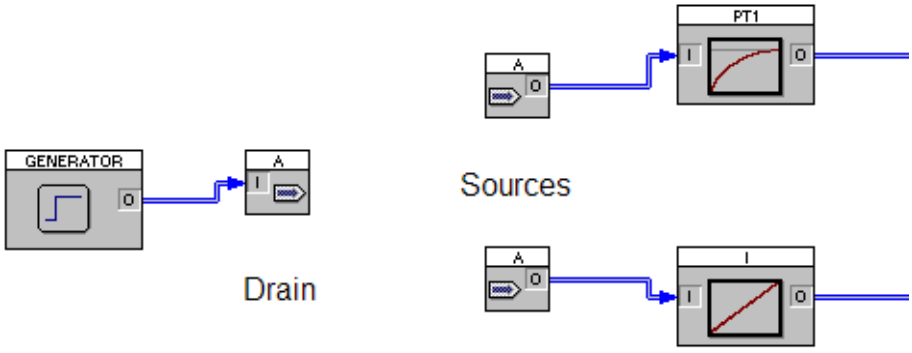
Parameter dialog:



Signal drain

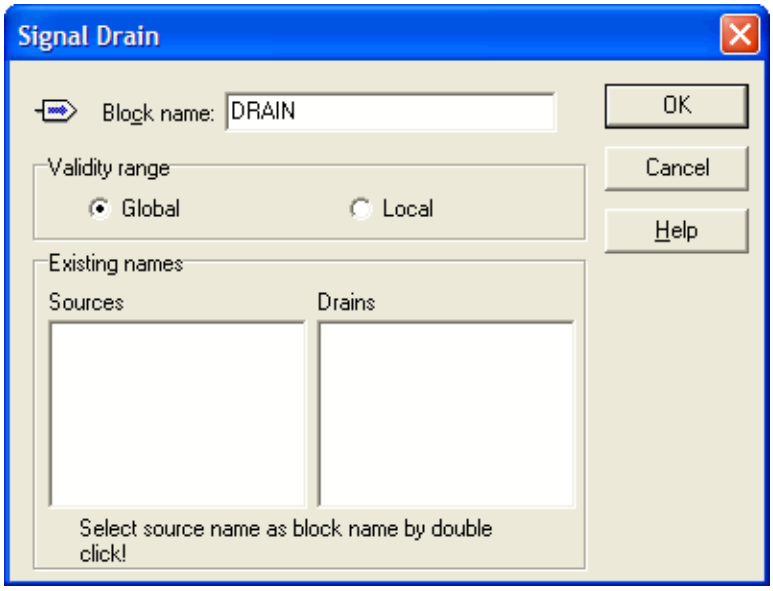
Typename: DRAIN

Function: With this block and the input block type *Signal source* you can realize "wireless" connections between blocks. The signal drain "sends" out its input signal under a special name (the name of the block). This signal can then be received from a signal source of the same name at any – even several – positions in the system structure. Block names are not case-sensitive. Signal drains can be of local or global validity. In the first case they are only known within the corresponding system or superblock file, in the second case they are valid out of the file limits as well.



The concept of sources and drains

Parameter dialog:



In the left list box the names of all existing sources are shown in alphabetical order, in the right list box the names of the drains. The name of a source can be accepted as the name of a drain by a double click.

Miscellaneous system blocks

Parameter modifier

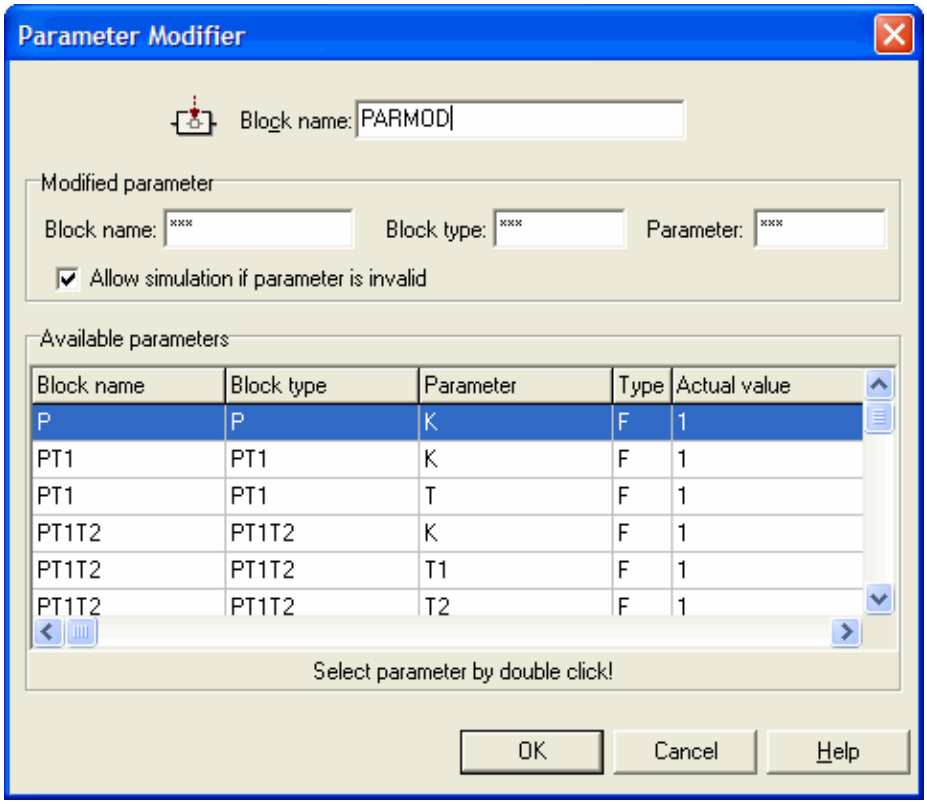
Typename: PARMOD

Function: This block type allows the modification of any (floating point) export parameter within the simulation. So well-aimed modifications of block parameters (e. g. gains or time constants) can be realized in a simple and comfortable way. If the enable input S has HIGH level (or if it is open), the specified export parameter value is set to the value of the data input D. If input S is connected but has LOW level, no modification is executed.

To guarantee a well-defined simulation order all PARMOD blocks are executed after all other blocks within one simulation step; so the new parameter value is active not until the beginning of the next simulation step. That means that in the first simulation step always the parameter value defined within the corresponding block parameter dialog is used.

After the simulation has terminated, all parameters modified by a PARMOD block are reset to their initial values.

Parameter dialog:



If the option *Allow simulation if parameter is invalid* is activated, the block allows simulation even if the specified export parameter does not exist (e. g. because the corresponding block had been renamed in the meanwhile); in this case the block is without any function. If the option is deactivated, a message box appears and the simulation can not be started.

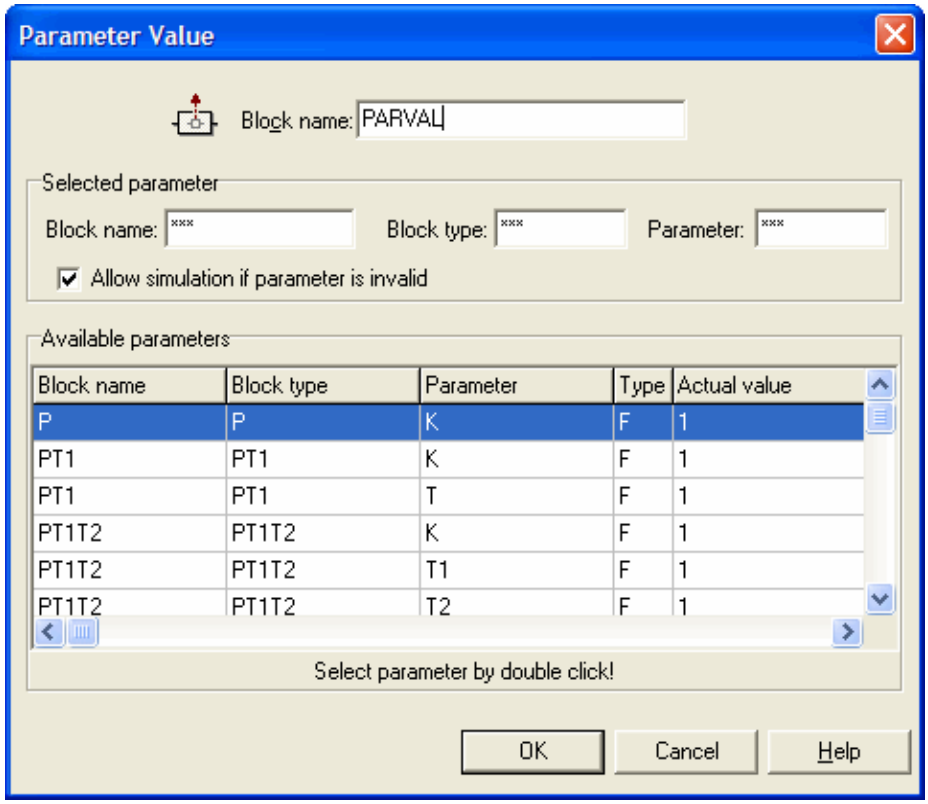
Remark: The block does *not* check the value to be transferred for validity; if such a check is desired, it must be realized within the simulation structure (e. g. by the usage of a limiter at the block data input).

Parameter value

Typename: PARVAL

Function: This block type delivers at its output the current value of any (floating point) export parameter.

Parameter dialog:



If the option *Allow simulation if parameter is invalid* is activated, the block allows simulation even if the specified export parameter does not exist (e. g. because the corresponding block had been renamed in the meanwhile); in this case the block is without any function. If the option is deactivated, a message box appears and the simulation can not be started.

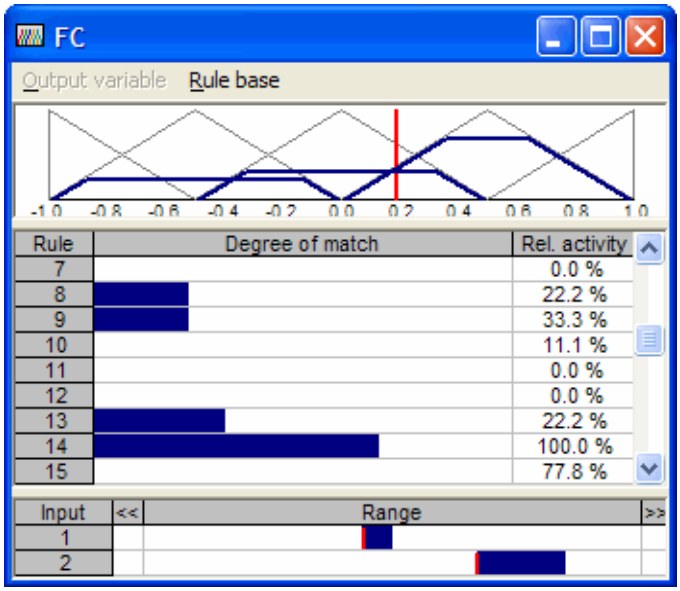
Fuzzy Controller

Typename: FC

Function: This system block realizes a fuzzy controller which is parameterized by a FUZ-file. This file can be produced with help of the WinFACT-Fuzzy-Shell FLOP.

The fuzzy controller block has a control window named *Fuzzy Debugger* which gives important hints to the inner function of the controller. The Fuzzy Debugger therefore supports the interactive design of the Fuzzy Controller. The screenshots below show the structure of the debugger and its parameter dialog. During the simulation the Fuzzy-Debugger shows the following variables online:

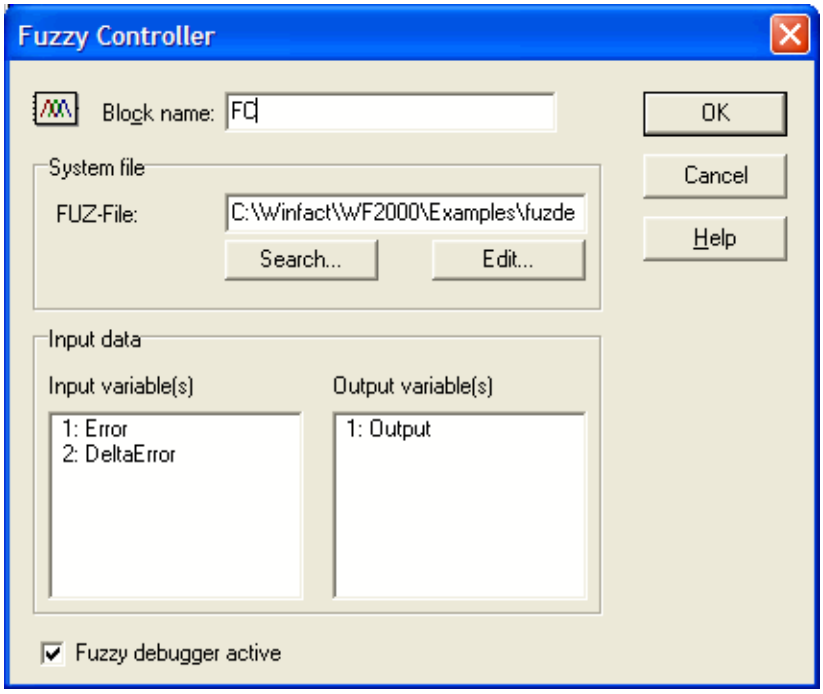
- In the upper third of the window the fuzzy sets of the chosen FC-output variable (grey), the current crisp output (red) and the resulting output fuzzy set (blue).
- In the middle third of the window the current match of degree of all rules (blue bars), and their relative activity. If a rule e. g. has a relative rule activity of 30%, this means that the rule in 30% of all simulation steps has a match of degree bigger than 0.
- In the lower third of the window the current crisp input values (red) and the so far used range of the input variable of the fuzzy controller (blue). Out of this range the displays left resp. right beside the range display take the colour yellow.



The Fuzzy debugger

Via the menu option OUTPUT VARIABLE the output variable of the fuzzy controller which is to be shown can be chosen. The menu option RULE BASE allows the textual output of the underlying rule base for control purposes.

Parameter dialog:



If you don´t specify any extension for *FUZ-File* the extension FUZ will be used. With the button *Search* a file open dialog can be called. The corresponding in- and output variables will be shown afterwards in the list boxes. With the button *Edit* the specified file can be opened directly into the fuzzy shell Flop. After leaving the dialog the number of block in- and outputs will be adjusted automatically if necessary. The fuzzy debugger can be deactivated to reach a higher simulation speed.

Online Fuzzy Controller

Typename: FCONLINE

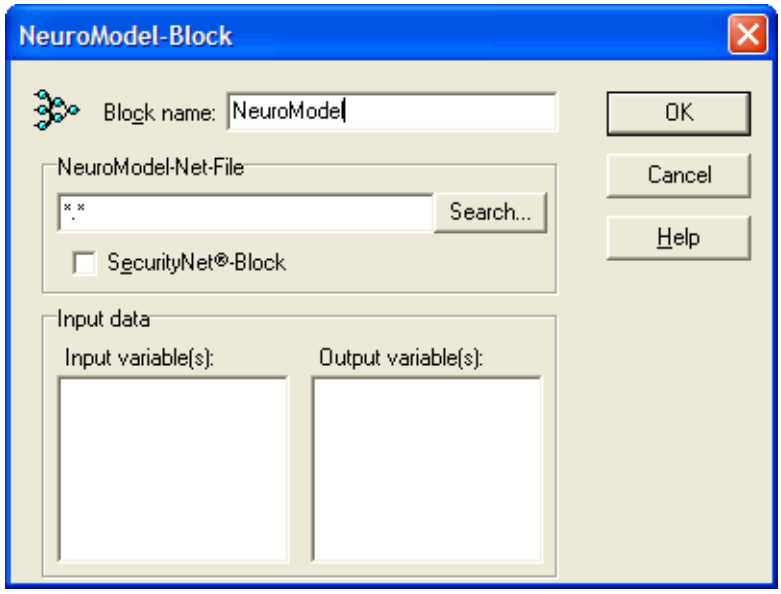
Function: This block type realizes a fuzzy controller designed with the WinFACT fuzzy shell FLOP which is connected to BORIS via Dynamic Data Exchange (DDE). For details please see the documentation resp. online help for FLOP.

NeuroModel block

Typename: NEUROMODEL

Function: This block type realizes a neural net generated with NeuroModel®. For further information please consult the documentation of NeuroModel.

Parameter dialog:

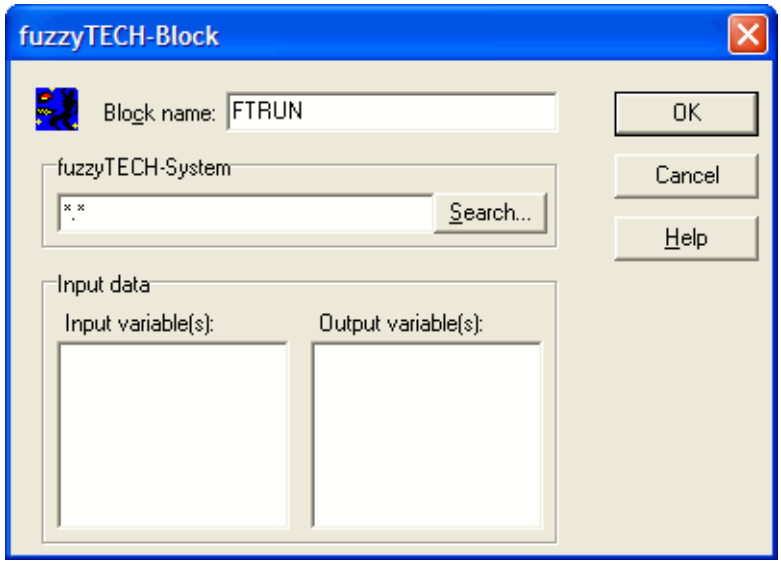


*fuzzy*TECH block

Typename: FTRUN

Function: This block type realizes a fuzzy controller generated with *fuzzy*TECH®. Further information you will get from the documentation of *fuzzy*TECH.

Parameter dialog:

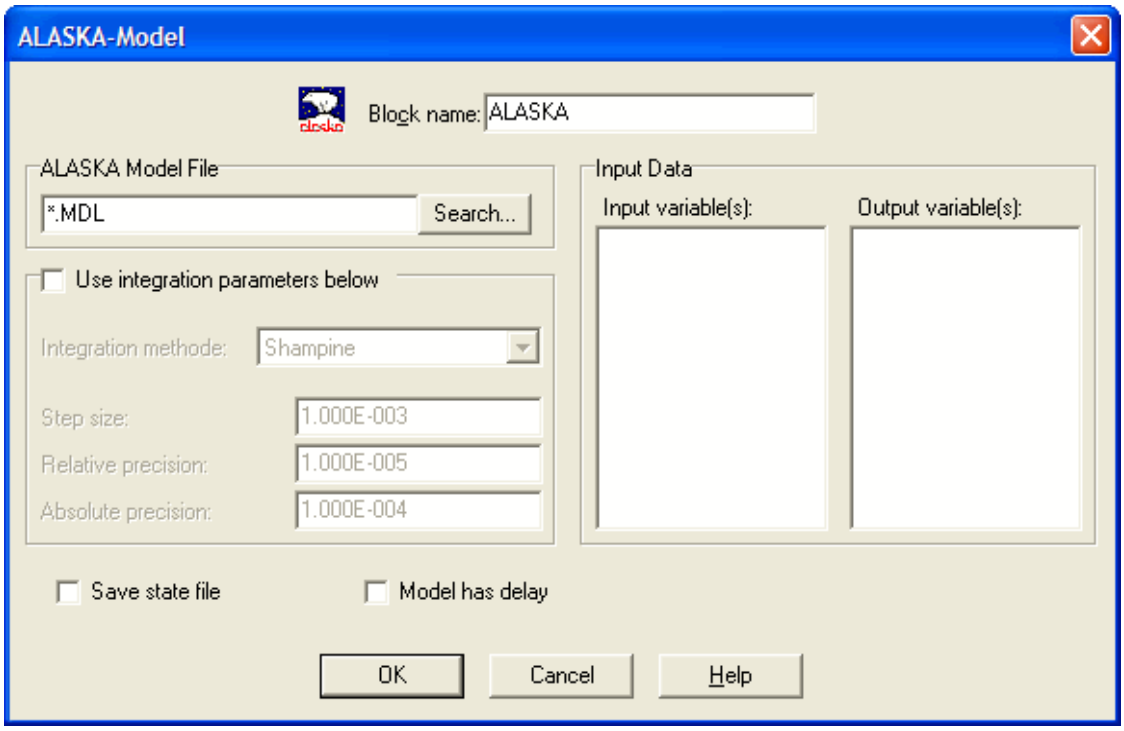


ALASKA 4 model

Typename: ALASKA

Function: This block type allows the integration of a simulation system designed with the simulation software *ALASKA 4*. Detailed information can be drawn from the corresponding software documentation.

Parameter dialog:



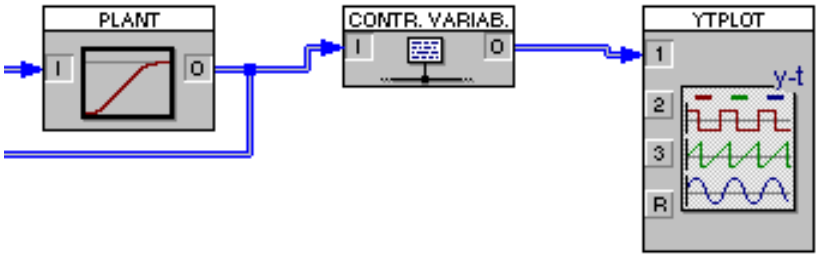
Label

Typename: LABEL

Function: Blocks of the *Label* type have no technical function but can only be used to give a name to signals with which they appear for example in time response blocks or - very important – in superblocks (see the chapter *Working with superblocks*).

Example: The output variable of a PT_1-element named *Plant* shall be connected to a time response block and appear there as *Contr. variable.* In order to

do so you have to put a label named *Contr. variable* between the PT_1-element and the time response (see screenshot below).



Use of label-blocks

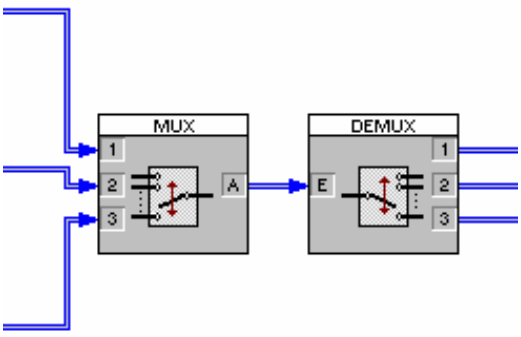
Parameter dialog:

Multiplexer

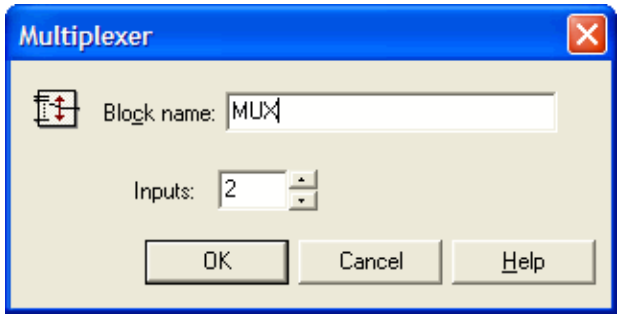
Typename: MUX

Function: This block allows the combination of several connections to a single one. Its use is therefore suitable if several connections should be drawn over a great area at the same time. The output connection of a multiplexer has to be connected to the input of a *Demultiplexer* block.

Example:



Parameter dialog:

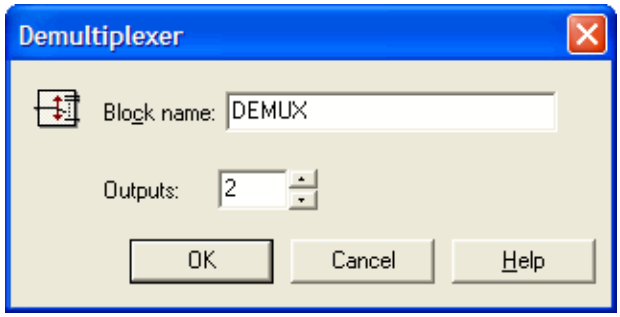


Demultiplexer

Typename: DEMUX

Function: This block splits up the multi-connection produced by a *Multiplexer* block. The input connection of a demultiplexer therefore has to start at the output of a multiplexer.

Parameter dialog:

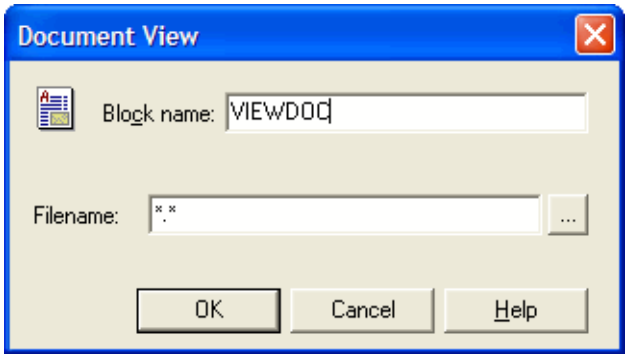


Document view

Typename: VIEWDOC

Function: This block can be used to display the content of a document file (e. g. a PDF file) by a mouse click on the button located within the system block. Ths document is displayed by the application specified for the corresponding file extension in the Windows registry.

Parameter dialog:

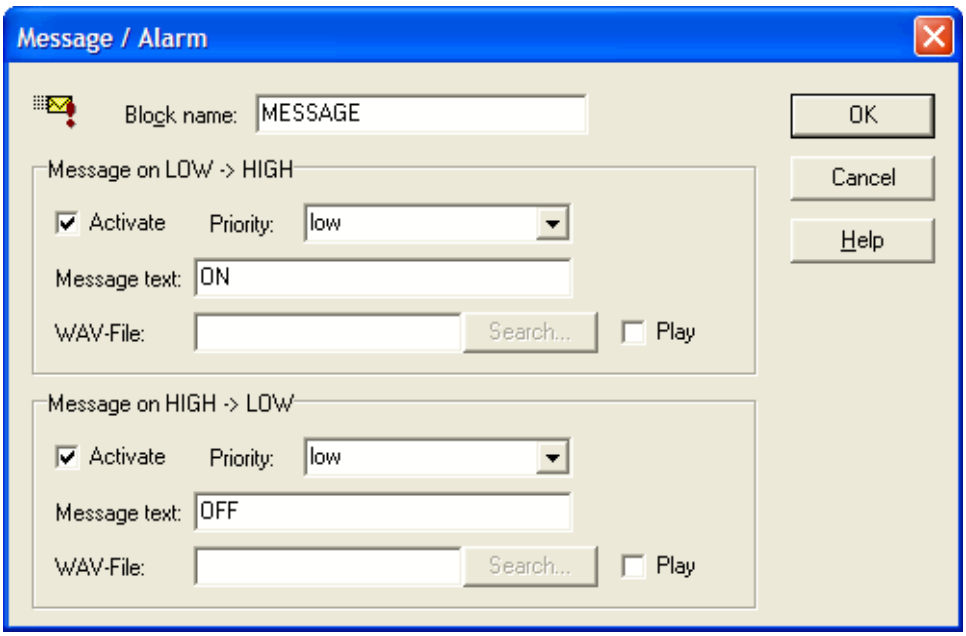


Message/Alarm

Typename: MESSAGE

Function: This block allows the generation of a message resp. a warning if a positive or negative edge appears at its input. Additionally to the graphical display each message can be confirmed by a WAV-file. For further information please see the chapter *Administration of messages/alarms.*

Parameter dialog:

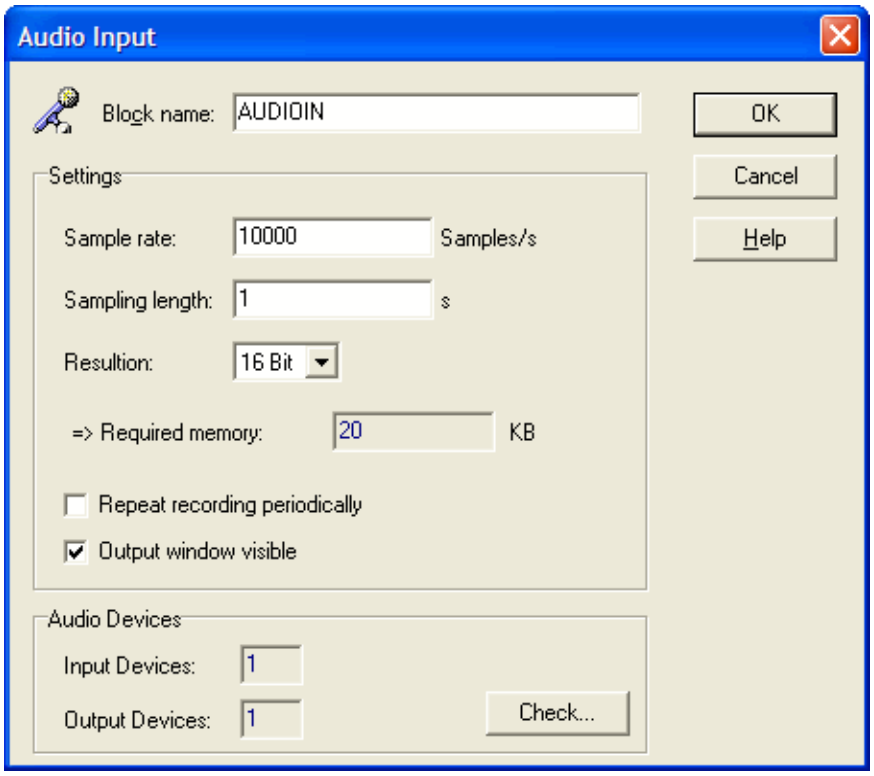


Audio input

Typename: AUDIOIN

Function: This block type allows input of audio signals via the microphone input of the sound card. The signal can be visualized within a separate window. The sample file *AudioIn.bsy* illustrates the usage of this block.

Parameter dialog:

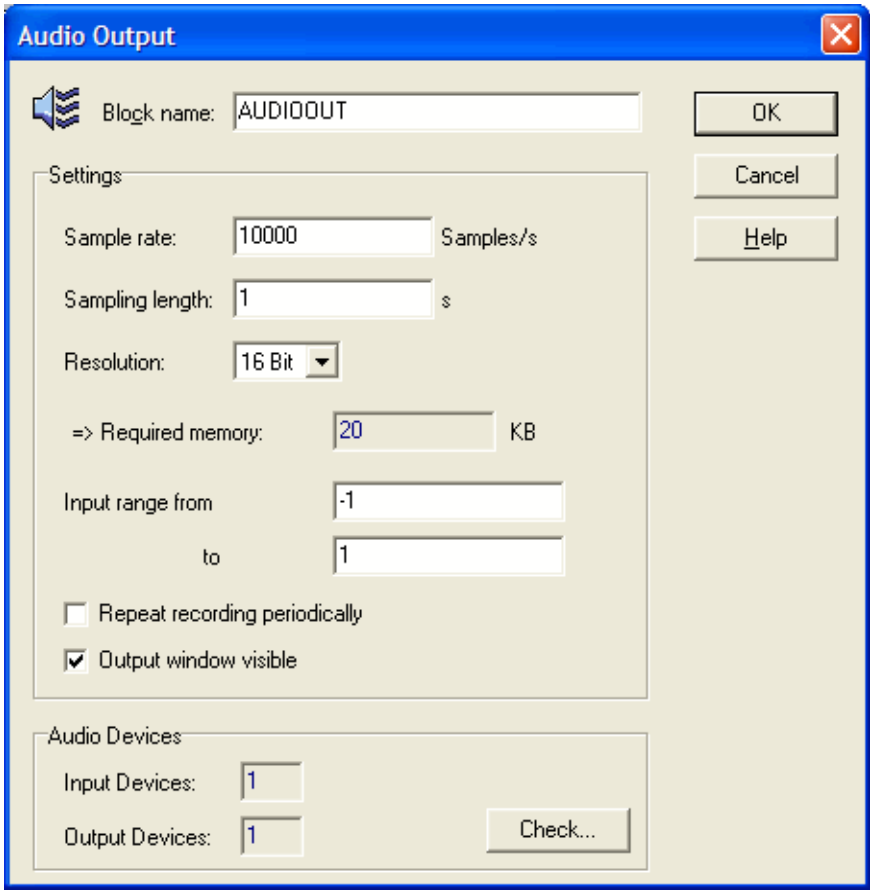


Audio output

Typename: AUDIOOUT

Function: This block type allows output of audio signals via the sound card. The signal can be visualized within a separate window. The sample file *AudioOut.bsy* illustrates the usage of this block.

Parameter dialog:



Hardware interface

Typename: HARDWARE

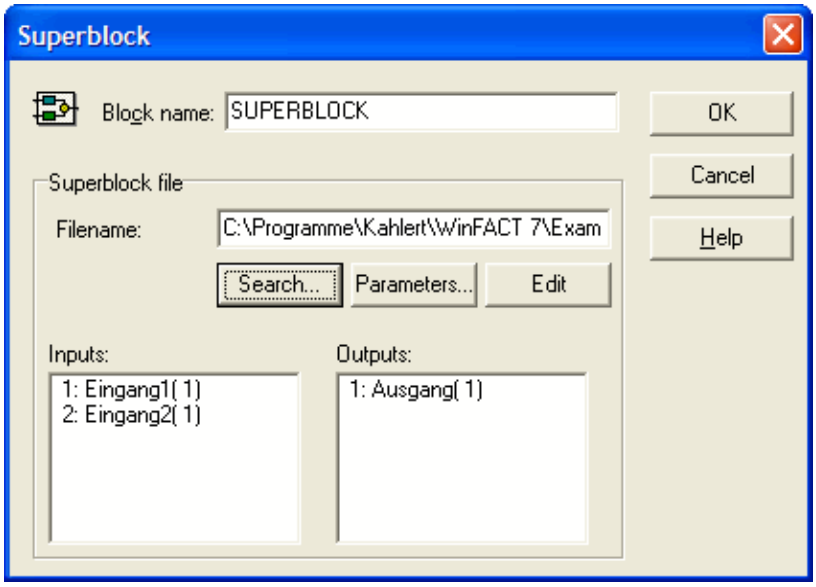
Function: This block - combined with a corresponding driver - is used as an interface to various hardware (e. g. PC-cards, external hardware at the RS-232-Port etc.) and can contain hardware inputs as well as hardware outputs. Further information you will get from the documentation delivered with the hardware driver.

Superblock

Typename: SUPERBLOCK

Function: A superblock contains any subsystem of blocks and connections. The work with superblocks is described in chapter *Working with superblocks*.

Parameter dialog:



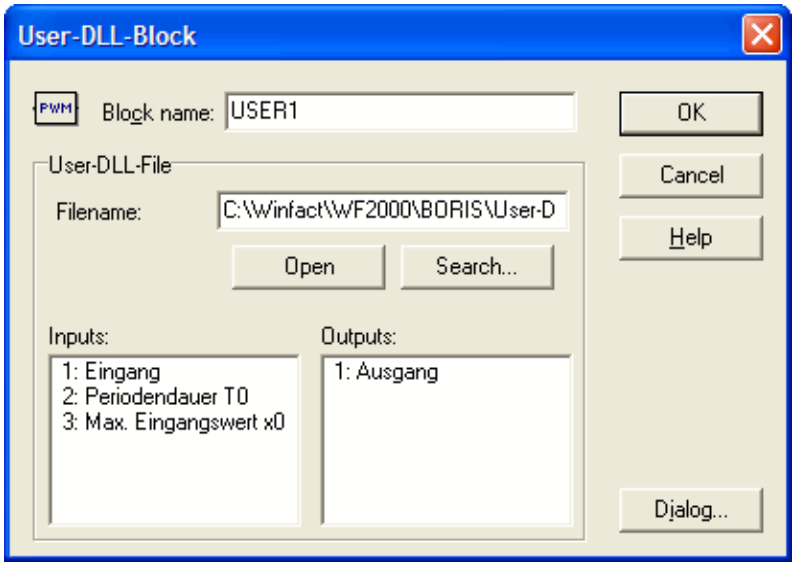
If you specify no extension for *Filename* the extension SBL will be used. With the button *Search* a file open dialog can be called. The corresponding in- and output variables will be shown afterwards in the list boxes for control purposes. The button *Parameters...* allows the work with superblock export parameters. With the button *Edit* the specified file can be opened directly into a new BORIS-instance. After leaving the dialog the number of the block in- and outputs will be adjusted automatically if necessary.

User-defined block (User-DLL)

Typename: USER1

Function: User-defined block realized as a Windows-DLL. Further information you will get in the chapter *User-defined system blocks*.

Parameter dialog:



If you specify no extension for *Filename* the extension DLL will be used. With the button *Search* a file open dialog can be called. The button *Open* reads the specified file. The corresponding in- and output variables will be shown in the list boxes afterwards. After leaving the dialog the number of block in- and outputs will be adjusted automatically if necessary. With the button *Dialog...* the block specific parameter dialog will be called.

Working with superblocks

What is a superblock?

A superblock is a special type of system block which results from the *grouping of several system blocks and their connections.* The superblock therefore is a subsystem which is combined to a new block, basically with in- and outputs. So the superblock has a kind of "Black-Box"-characteristic. Superblocks are therefore suitable especially for the clearly arranged structuring of complex systems and the grouping of frequently used subsystems.

Example: You want to test different controllers for a process you have simulated before that consists of the series of three system blocks. Then you transfer the series at first into a superblock you can access later on from any system.

How information about the structure of a superblock is managed

The superblocks of BORIS are *file referenced.* All the information about the blocks within the superblock and the connections are saved in a file with the extension SBL. These files are identical to the "normal" BORIS system files of type BSY with the exception of an additional file header in the superblock files. Thus superblock files can be saved as usual system files and vice versa, they can of course be simulated separately as well.

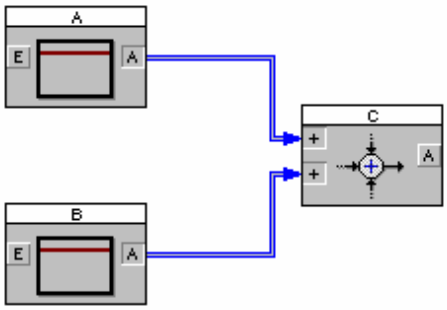
The advantage of the file referency is evident. If you change a superblock file, all system files which use this superblock will be updated automatically. The superblock itself is defined by its filename.

In- and outputs of superblocks

The in- and outputs of a superblock are determined by BORIS automatically. Please note the following:

- All open system block in- and outputs become in- and outputs of the superblock

  Example: The following structure has to be grouped to a superblock.



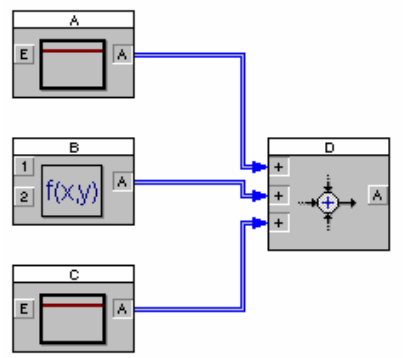
  The resulting superblock has two inputs (those of blocks A and B) and one output (the output of block C).

- If block outputs which already contain a connection (e. g. a feedback) shall become superblock outputs, it is suitable to use label blocks (see chapter *Superblocks and labels*). If vice versa open system inputs shall

not become superblock inputs, you have to connect e. g. to a constant block which is set to 0 (or another suitable value).

- The in- and outputs of the superblock will be numbered in the sequence in which the corresponding system blocks have been inserted.

Example:



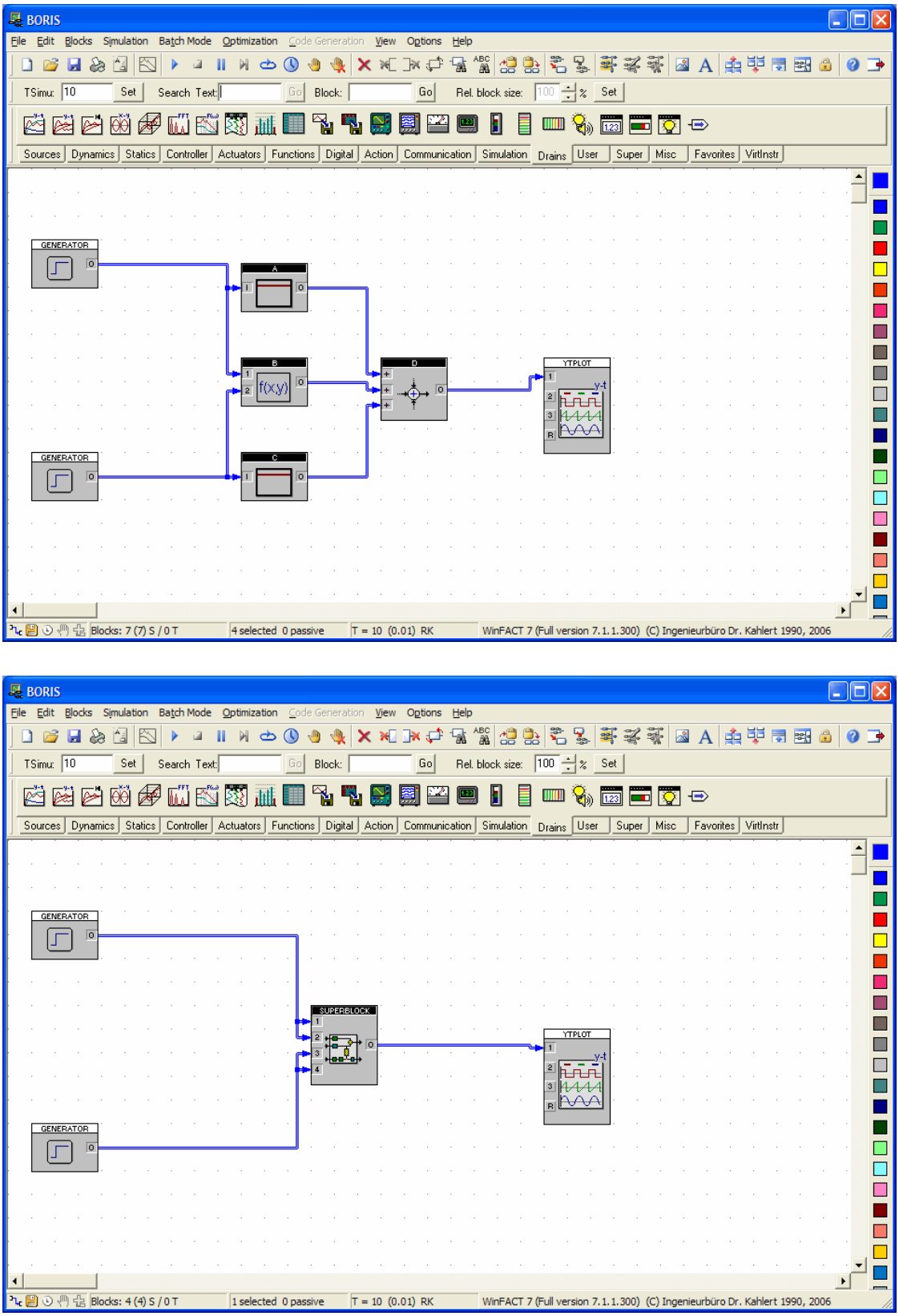
If the blocks have been inserted in the sequence A, B, C, D, the input of block A will become the first superblock input, the first input of block B becomes the second superblock input , the second input of block B becomes the third and the input of block C becomes the fourth superblock input. The same is valid for the existence of several outputs. If superblock in- and outputs shall be changed, you can either delete the corresponding system blocks and then insert them in a suitable order again or insert *Labels* (see chapter *Superblocks and labels*).

Creating a superblock

You basically can create a new superblock in two ways:

- ➢ You create the superblock as a separate file, save it with FILE | SAVE AS... and load it later on into the corresponding main system by specifying the name.
- ➢ If a subsystem of an already configurated system structure should be transferred to a superblock you have first to select the blocks which shall be grouped and then execute the grouping with EDIT | GROUP TO SUPERBLOCK... or the button of the system toolbar. Then BORIS will ask you for the name of the superblock file and will insert the superblock afterwards (see graphics below).

A superblock can be ungrouped by selection and the option EDIT | RESOLVE SUPERBLOCK or the button of the toolbar at every time. Pay attention that there is enough vacant room in the surrounding of the superblock.

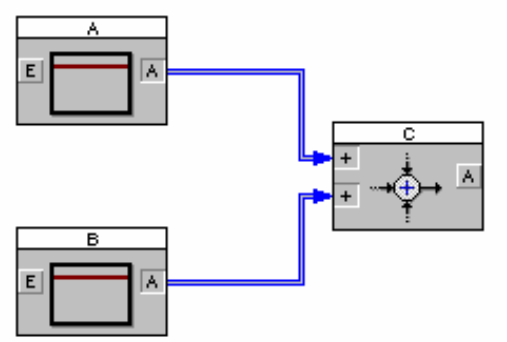


Grouping of blocks to a superblock: Selection of the blocks (top) and screen after inserting the superblock (bottom)

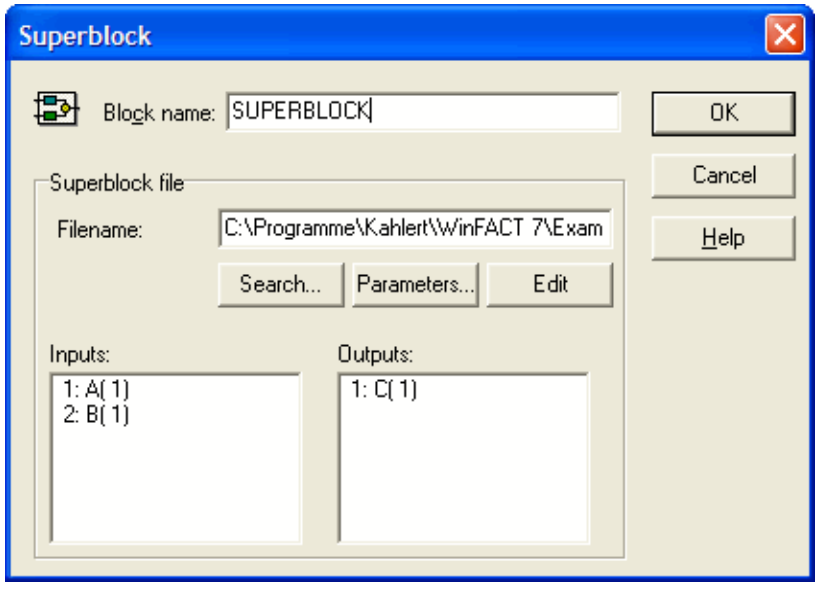
Superblocks and labels

The use of label blocks can be suitable in connection with superblocks in different ways:

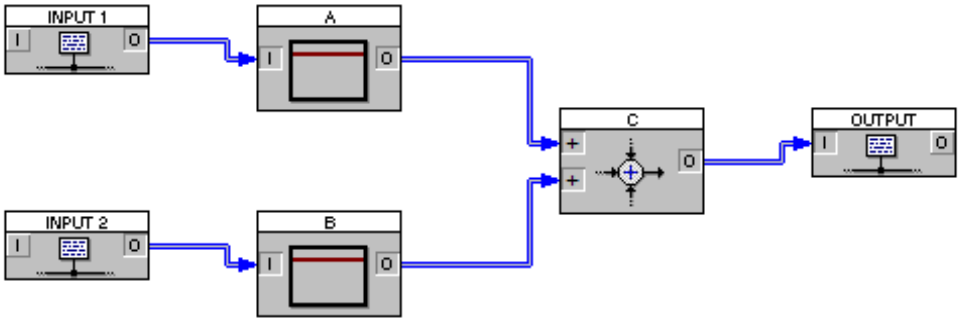
- Renaming of the superblock in- resp. outputs. All superblock in- and outputs get a name which is displayed in the parameter dialog of the superblock so that the user knows the function of every in- and output without having loaded the superblock file. By default the name of the block which is internally connected to the in- resp. output in connection with its in- resp. output number will be used. Please see the following example:



If this subsystem will be changed into a superblock, the inputs of the superblock will be named *A(1)* resp. *B(1),* the output will be named *C(1).* The parameter dialog of the superblock then looks as follows:



To give more expressive names to the in- and outputs you can use labels with the desired names. The modified superblock structure then will look as follows:



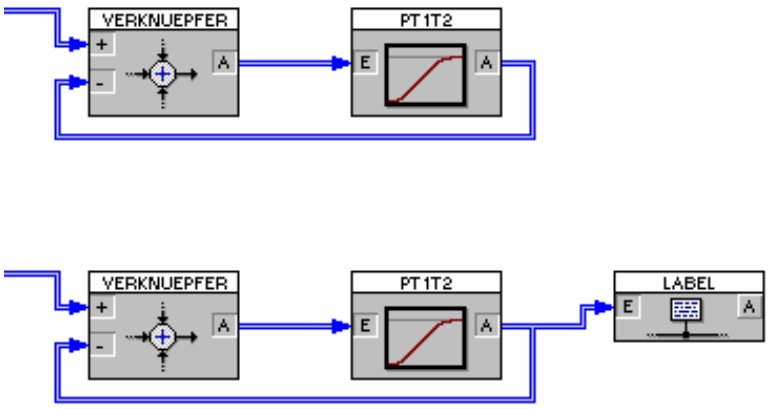
The inputs are named *input 1* resp. *input 2* , the output *Output* (practically you should use more expressive names!). If you then call the parameter dialog of the superblock, you will get the corresponding entries.

Rearrangment of in-/outputs

- For the rearrangement of the in- and outputs. If the superblock in- and outputs shall get a new sequence you only have to insert corresponding labels in the desired order.

Creating open outputs

- For creating open outputs. If an output of the superblock structure which already contains a connection shall become a superblock output, you have to put a label behind the output and by that produce an open output. The following graphic shows an example:



To change the output of a PT1T2-element (top) into a superblock output, a label is used (bottom).

Output blocks in superblocks

All output blocks (e. g. time response, oscilloscope, etc.) can be used within superblocks. Please note that the corresponding display window is only visible *during the simulation*! It appears at the start of the simulation and disappears immediately after the end of the simulation. A "post view" of the simulation results is therefore not possible. The same is valid for the display windows of other blocks (e. g. the fuzzy debugger of the fuzzy controller).

Sources and drains in superblocks

Signal sources and drains can also be used in superblocks. Basically you will operate with both block types in the *local* mode so that their validity is limited to the respective superblock. In principle *global* sources and drains can be used in a superblock as well. This means on the one hand that a signal which is sent by a superblock with a *global* signal drain can be received outside the superblock with the help of a corresponding global signal source. On the other hand superblocks can receive all signals which are generated by internal *global* signal sources outside the block.

Caution: If the same superblock is used several times in the same structure, you should avoid global signal sources inside the superblock - otherwise several drains with the same name will exist! Basically this will lead to unwelcome "effects" during the simulation!

Superblocks in superblocks in superblocks...

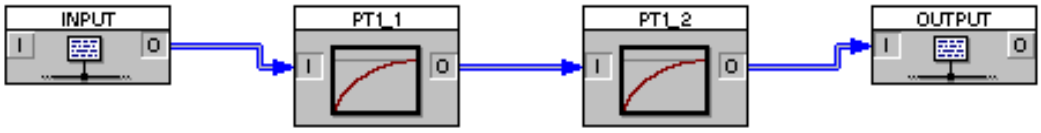
A system can contain several superblocks. A superblock itself can also contain one or several superblocks and so on – the depth is unlimited. Of course recursive references of superblocks are not allowed. If superblock A contains the superblock B, the latter one can not access to block A at the same time.

Export of parameters

One often wants to use the same structure several times, but at each time with different parameters. In this cases BORIS allows you to export parameters of

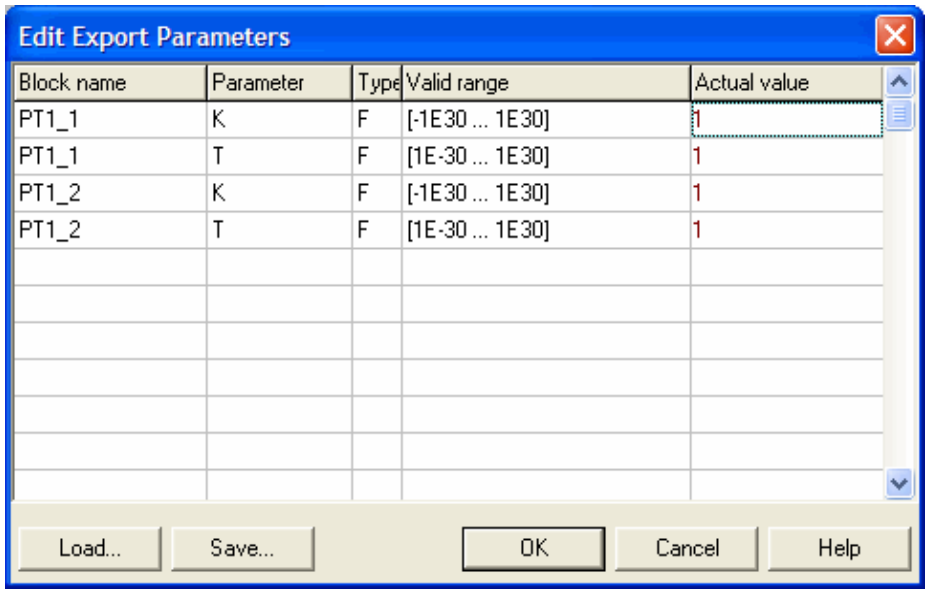
blocks to the "outside" so that they can be modified on the next level of the system structure. For that purpose you have to activate the corresponding export parameters within the superblock file so that they can be set with every use of the superblock independent of their value inside the superblock file.

Example: A simple superblock consists of the series circuit of two PT1-elements (see the following graphic). This superblock shall now be used by other structures. Gains and time constants should be modified inside the calling structure. For this purpose you only have to activate the export parameters in the parameter dialogs of the two PT1-elements (see the example file SUPEREXP.BSY in the examples directory).



Superblock with export parameters

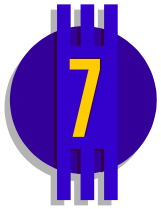
After the storage the superblock can be inserted into any system structure as usual. The exported parameters can be modified *for each superblock separately*. For that use the button *Parameters*...of the superblock parameter dialog. The parameters which are default at this option will not be saved inside the superblock file but inside the calling file. Because the assignment of the export parameters is made by the block name it is important to name all exporting blocks explicitly with different names. You can check this with the option FILE | CHECK FOR DUPLICATE BLOCK NAMES....



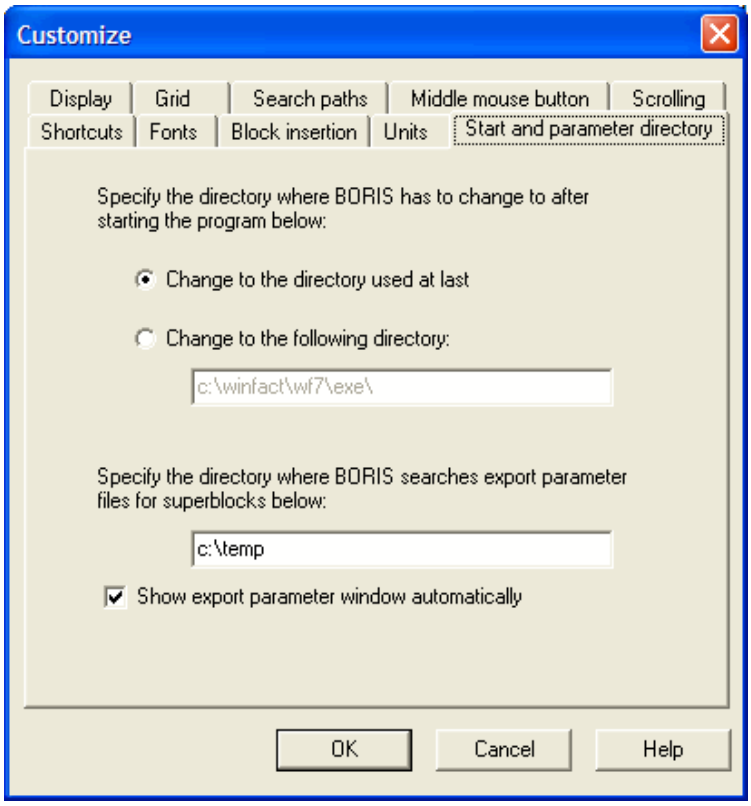
Dialog for the modification of superblock export parameters

Please note that the export of superblock parameters is only allowed to the *next* level, not further on. If e. g. a superblock A is used by a superblock B, the export parameters of the superblock A can be modified in B. If superblock B is inserted into a structure itself, you can not access to the export parameters of superblock A in there.

Reading export parameters from file

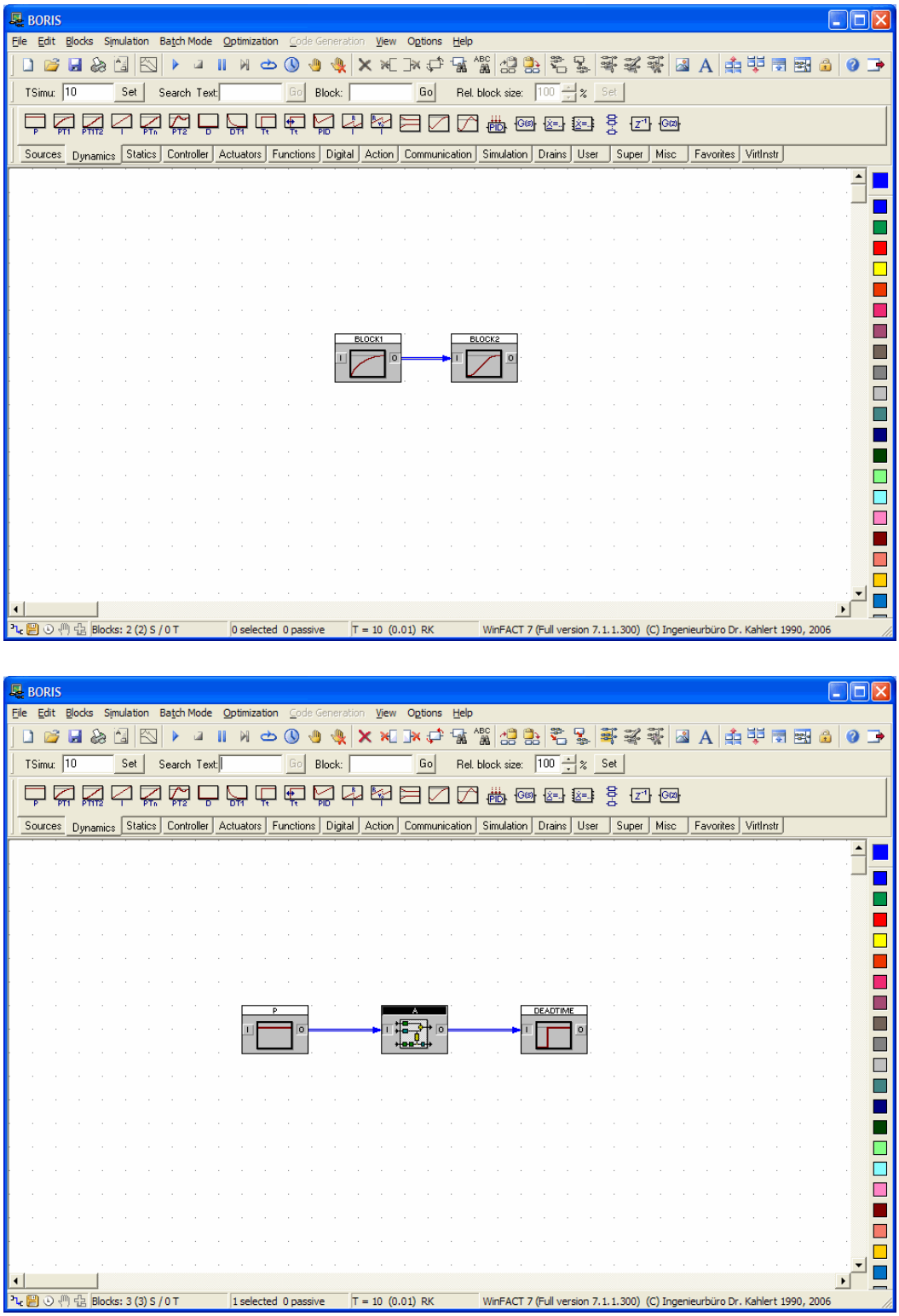


In more complex applications it may be useful to read the export parameters of a superblock from a text file. This text file must have the same name as the corresponding superblock and the file extension EXP. It must be located in the directory specified under OPTIONS | CUSTOMIZE... on the *Start and parameter directory* palette (see screenshot below).



*Selecting the directory for export parameter files (*c:\temp *here)*

The structure of the text file shall be explained with an example. The screenshots below show a superblock consisting of a serial connection of a PT1- and a PT1T2-element with activated export parameters (upper screenshot). The blocks are named *BLOCK1* resp. *BLOCK2*. The lower screenshot shows the system structure containing this superblock (middle system block) named *A*.



Internal structure of superblock (top) and system structure using superblock (bottom)

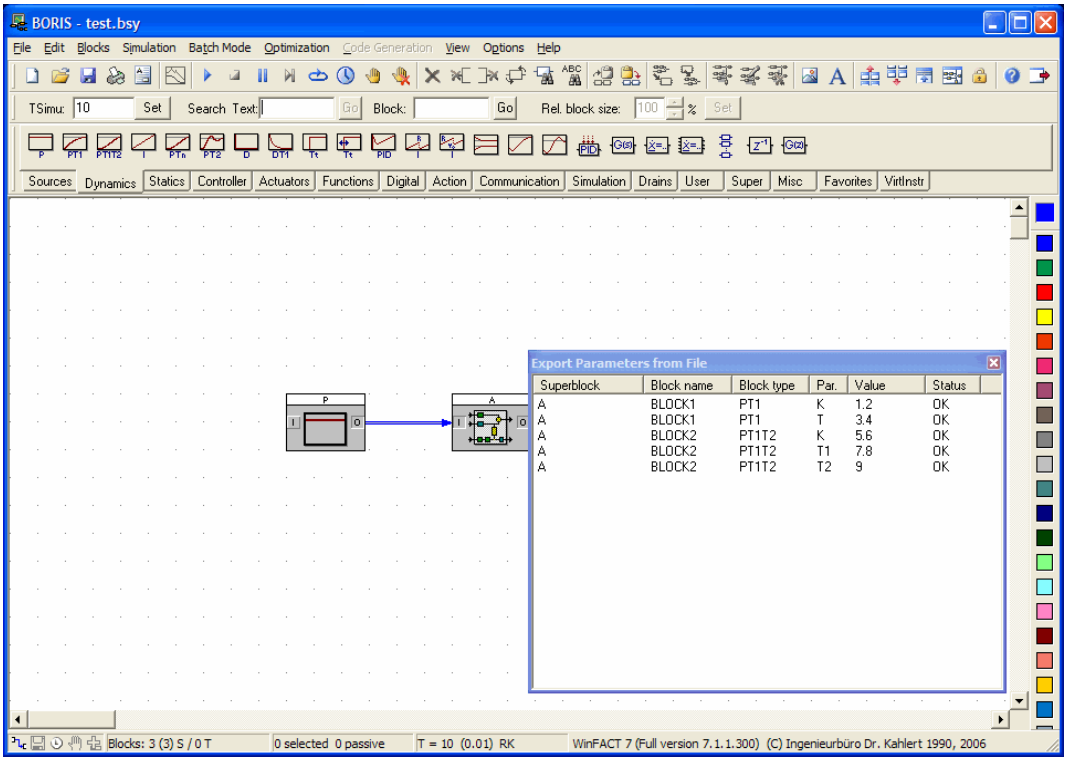
The listing below demonstrates the structure of the corresponding export parameter file, the filename of which has to be A.EXP. For each block contained

in the superblock this file has a specific *section* introduced by an identifier in brackets. This identifier contains the block name, a "|" character and the block type. The following lines contain all export parameters starting with the parameter name, a "=" char and the parameter value. Thus in the exmaple below the gain *K* of *PT1*-block *BLOCK1* is set to 1.2, its time constant *T* to 3.4; the gain *K* of *PT1T2*-block *BLOCK2* is set to 5.6, its first time constant *T1* to 7.8 and its second time constant *T2* to 9.

```
 [BLOCK1|PT1]
K=1.2
T=3.4

[BLOCK2|PT1T2]
K=5.6
T1=7.8
T2=9
```

When the simulation is started the export parameters read from file are listed in a separate window if this option was not deactivated under OPTIONS | CUSTOMIZE... before (see above). The screenshot below shows this window for the latest example. If a parameter could *not* be read from file, its status is set to *Failed* instead of *OK*. For those parameters the value specified manually is used for simulation.



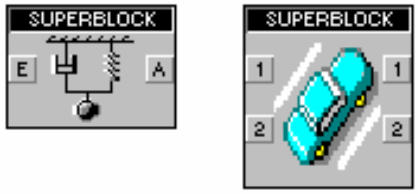
Window with export parameters read from file when starting the simulation

User-defined block bitmaps

Optionally the user is enabled to add a user*-defined block bitmap* to each of his superblocks. This block bitmap will be created as a BMP-file (e. g. with Paint-Brush), has to get the same name as the superblock file - but with the extension BMP – and has to be in the same directory as the superblock itself. Therefore the bitmap file with the name MOTOR.BMP belongs to the superblock with the name MOTOR.SBL. If you insert a superblock BORIS will check automatically if the corresponding BMP-file exists. In this case it will be used instead of the standard block bitmap. The bitmap should have a size of 48x42 pixels; bitmaps of other sizes will be stretched resp. compressed automatically so that they fit into the block graphic.

A user-defined bitmap (preferably a b/w-bitmap) can also be defined for the printer output. It has to be named with the identification code *_P*.

Examples: to MOTOR.SBL belongs the printer bitmap MOTOR_P.BMP
to GEAR.SBL belongs the printer bitmap GEAR_P.BMP



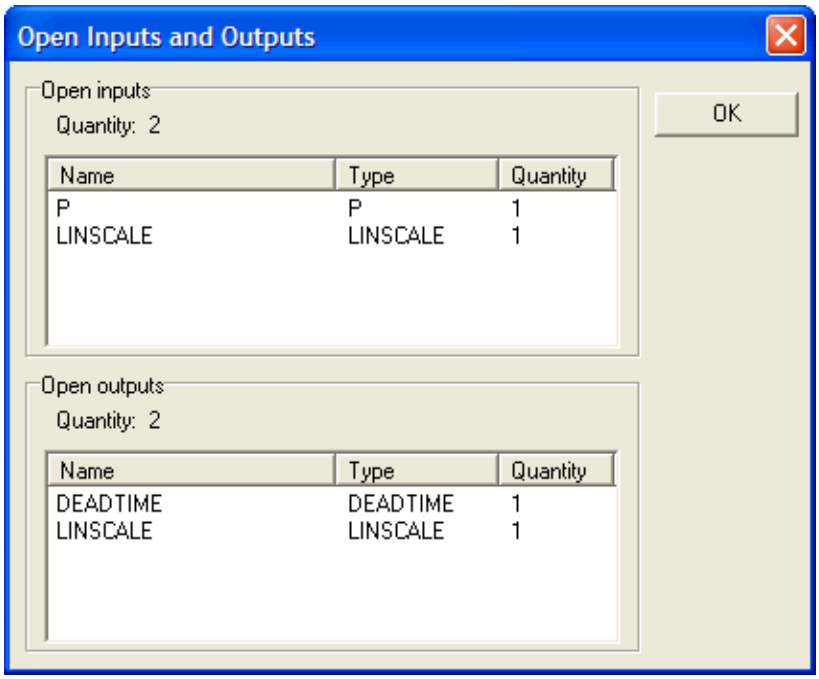
Superblock samples with used-defined bitmaps

Finally you can define a bitmap for the presentation of the superblock in the palette *Super* of the system block toolbar. It has the size of 18x18 pixels and the same name as the printer bitmap, but the identification code *_T*.

Examples: to MOTOR.SBL belongs the toolbar bitmap MOTOR_T.BMP
to GEAR.SBL belongs the toolbar bitmap GEAR_T.BMP

Further important information

To get a survey of the open in- and outputs of the current system you have to use the menu option FILE | OPEN INPUTS/OUTPUTS... . A dialog appears which lists the block name, the block type and the number of open in- resp. outputs.



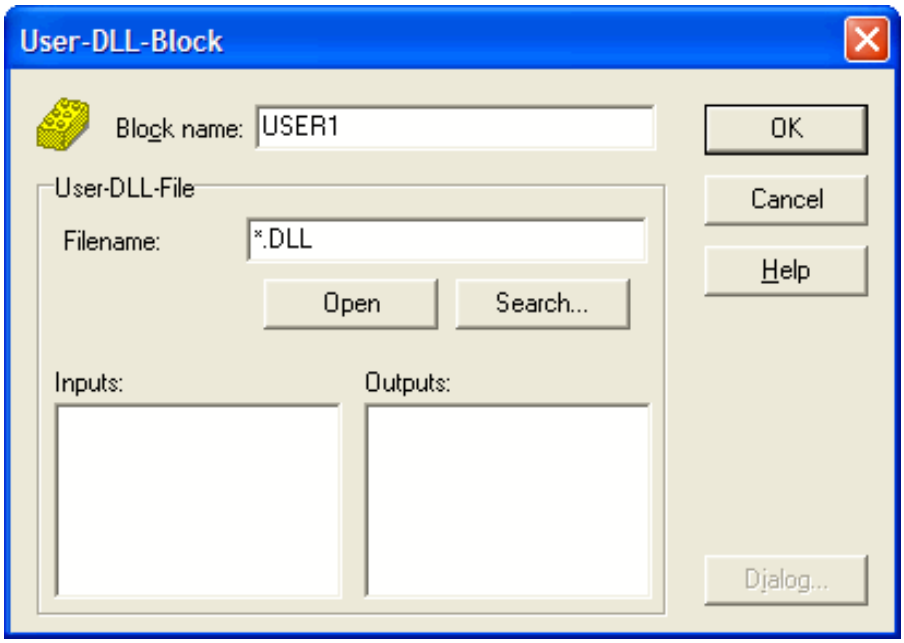
Dialog for listing open in- and outputs

It can happen that several inputs of a subsystem which should be grouped to a superblock are open but not needed (e. g. control inputs, etc.) and therefore should not be shown as inputs of the superblock. In this cases these inputs should be connected to a constant block which generates a suitable value (basically this will be 0).

User-defined system blocks (User-DLLs)

The concept of User-DLLs

BORIS provides the programming of user-defined block types, so-called *User-DLL-blocks*, based on a 32-bit Windows-DLL. These DLLs can be created with any 32-bit programming system like Delphi, Visual Basic, Borland C++ Builder or Visual C++.



Parameter dialog of a User-DLL-Block (no parameters defined here!)

In the following first the *data interface* and the *function interface* of the user-block will be explained prior to a deeper introduction to the programming of user-blocks based on the analysis of several examples of variable complexity.

Note: All programs resp. program segments listed below were implemented in DELPHI 3. A transfer to another programming environment or language however will succeed without any difficulty. Specific hints for programming User-DLLs under Microsoft Visual C++ can be found in chapter *Programming under Visual C++*.

Data interface of the user-block

All the data of a user-block which are relevant for the user are specified in a data structure called *TParameterStruct* resp. a pointer *PParameterStruct* to this structure. This structure first of all contains the *block parameters*. These data normally are block-specific constants (e. g. gains, time constants etc.) and can be modified by the user via the parameter dialog of the user-block. At the storage of a BORIS system structure including a user-block, these block parameters will be saved within the system file and thus are restored after reloading the file. Moreover these parameters can, of course, be "misused" for other purposes - e. g. as state variables, temporary variables, flags etc.

The following parameters are at the user's disposal:

- Up to 32 floating point parameters (10 byte, data type *extended* in PASCAL resp. *long double* in C)
- Up to 32 integer parameters (4 byte, data type *longint* in Pascal resp. *int* in C)
- Up to 32 boolean resp. byte parameters (1 byte, data type *byte* in Pascal resp. *char* in C)
- A string of length 256 (e. g. for filenames)

For each floating point- resp. integer-parameter a minimum and a maximum value can be specified which is checked automatically in the parameter dialog. Furthermore names have to be defined for all parameters which appear at the corresponding position in the parameter dialog. If desired, further parameters can be read from an external file. In addition to these block parameters the *TParameterStruct* structure contains some other variables for advanced users; these parameters are explained more detailed later on in connection with some examples. First of all we have a look at a short listing of the records of *TParameterStruct*.

```
type
  PParameterStruct = ^TParameterStruct;
  TParameterStruct = packed record
    NuE: Byte;
    NuI: Byte;
    NuB: Byte;
    E: Array[0..31] of Extended;
    I: Array[0..31] of LongInt;
    B: Array[0..31] of Byte;
    D: Array[0..255] of char;
    EMin: Array[0..31] of Extended;
    EMax: Array[0..31] of Extended;
    IMin: Array[0..31] of LongInt;
    IMax: Array[0..31] of LongInt;
    NaE: Array[0..31,0..40] of char;
    NaI: Array[0..31,0..40] of char;
    NaB: Array[0..31,0..40] of char;
    UserDataPtr: Pointer;
    ParentPtr: Pointer;
    ParentHWnd: Word;
    ParentName: PChar;
    UserHWindow: Word;
    DataFile: text;
  end;
```

Structure TParameterStruct *in Pascal*

Variable	Meaning
NuE, NuI, NuB	Number of float, integer resp. boolean parameters. Only those parameters have to be taken into account which shall appear in the parameter dialog ("real" block parameters); array items that contain variables (state variables, flags etc.) are excluded.
E, I, B	Parameter arrays with float, integer and boolean parameters
D	Can be used to save the name of an external file with additional data.
EMin, EMax	Arrays with the lower and upper bounds of the float parameters
IMin, IMax	Arrays with the lower and upper bounds of the integer parameters
NaE, NaI, NaB	Arrays with the names of the float, integer and boolean parameters. Each parameter name can have a maximum of 40 characters.
UserDataPtr	Pointer to optional block variables (e. g. state variables, temporary variables, flags etc.). These are normally variables which are no block parameters and therefore don´t have to be saved with the system file.
ParentPtr	obsolete
ParentHWnd	Handle of the main window of BORIS. Important for User-DLLs which create visualization windows or dialogs.
ParentName	Pointer to the name of the user-block. Only needed for blocks with a visualization window.
UserHWindow	This variable can be used to save the handle of a visualization window created by the user-block. If no such window exists, the variable is redundant.
DataFile	Text file for flexible usage. Can be used in combination with the variable *D* (see above).

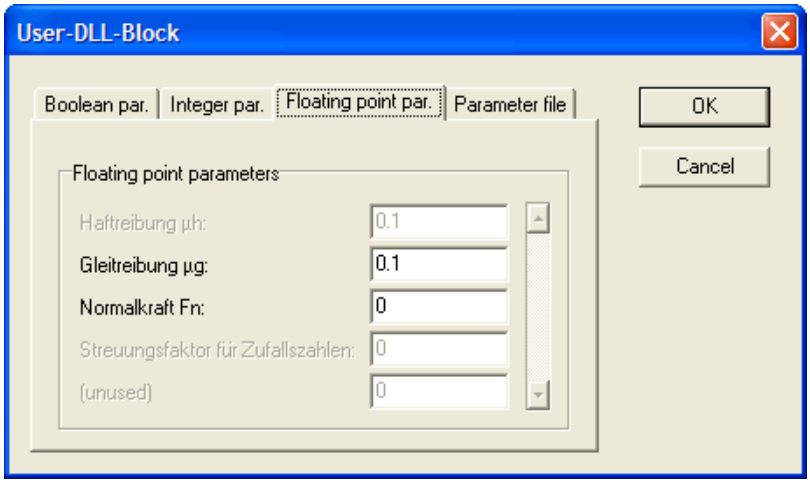
Besides the most important structure *TParameterStruct* some further data structures have to be declared within the User-DLL. First of all the structure *TDialogEnableStruct* resp. the corresponding pointer *PDialogEnableStruct* are discussed.

```
type
  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnablestruct=record
    AllowE: Longint;    { If a specific input field shall be  }
    AllowI: Longint;    { disabled/enabled, the corresponding bit}
    AllowB: Longint;    { of the Allow?-array must be 0/1}
    AllowD: Byte;
  end;
```

Structure of TDialogEnableStruct *(in Pascal)*

This structure is used for the control of the parameter dialog; combined with the *GetDialogEnableStruct* procedure (see later) the structure allows a conditional enabling and disabling of dialog elements (Example: A floating point parameter may only be modifiable if a certain boolean parameter is set). So *TDialogEnableStruct* contains one bit for each of the 32 input controls of the various parameter types that can be set (input field enabled) or reset (input field disabled). The *AllowD* parameter specifies whether the name of the optional parameter file (variable *D* of *TParameterStruct*) is enabled.

The screenshot below shows the *Floating point* register of the parameter dialog of a user-block for a friction simulation. The file REIBUNG.DLL is located in the *UserDLLs* directory, the source code file REIBUNG.DPR in the *Sources* subdirectory. This user-block contains four floating point parameters; at the moment the screenshot was captured only the second and the third parameter were enabled.



Sample parameter dialog of a user-block with four floating point parameters

The remaining data structures concern the specification of the inputs and/or outputs of the user-block. First of all the structure *TNumberOfInputsOutputs* has to be declared. This structure specifies the number of the inputs resp. outputs of the block as well as their names (max. 40 characters):

```
type
  PNumberOfInputsOutputs = ^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs = packed record
    Inputs  :Byte;                          {Number of inputs}
    Outputs :Byte;                          {Number of outputs}
    NameI : Array[0..49] of String[40];     {Input names}
    NameO : Array[0..49] of String[40];     {Output names}
  end;
```

Structure of TNumberOfInputsOutputs *(in Pascal)*

Finally the declaration of the data types for the transfer of the block inputs and outputs must be introduced. This is realized by the *TInputArray* resp. *TOutputArray* data type.

```
type
  PInputArray = ^TInputArray;
  TInputArray = packed Array[1..50] of extended;    {Input values}
  POutputArray = ^TOutputArray;
  TOutputArray = packed Array[1..50] of extended;  {Output values}
```

Data types TInputArray *resp.* TOutputArray *(in Pascal)*

Another data type that however is seldom used contains information which of the block inputs are connected to other blocks and which are not:

```
type
  PConnectedArray = ^TConnectedArray;
  TConnectedArray = packed Array[1..50] of byte;
```

Data type TConnectedArray *(in Pascal)*

In this array when calling the function *SetEnhancedInformation2* (see later) a 1 is set if the corresponding input is connected to another block and a 0 if it is not.

If specific block parameters should be activated as export parameters (e. g. to allow numerical optimization of these parameters or modification within the batch mode), another data type called *TExportParStruct* has to be declared:

```
type
  PExportParStruct = ^TExportParStruct;
  TExportParStruct = packed record
    ExportParName: array[0..20] of char;
    ExportParType: char;
    ExportParFValue: extended;
    ExportParIValue: integer;
    ExportParBValue: byte;
    ExportParFMin, ExportParFMax: extended;
    ExportParIMin, ExportParIMax: integer;
  end;
```

Data type TExportParStruct *(in Pascal)*

This data type contains the declaration of export parameters with floating point, integer or boolean type. The various record fields have the following meaning:

Variable	Meaning
ExportParName	Parameter name (max. 20 characters)
ExportParType	Specification of the parameter type: 'F' = Floating point parameter 'I' = Integer parameter 'B' = Boolean parameter
ExportParFValue	Parameter value in case parameter is of floating point type
ExportParIValue	Parameter value in case parameter is of integer type
ExportParBValue	Parameter value in case parameter is of boolean type
ExportParFMin, ExportParFMax	Valid parameter range in case parameter is of floating point type
ExportParIMin, ExportParIMax	Valid parameter range in case parameter is of integer type

Export parameters declared in User-DLLs are handled exactly as those of "normal" system blocks. The only difference is that they are always activated (except an activation/deactivation functionality was realized within the User-DLL), so that the BORIS menu options EDIT | EXPORT PARAMETERS | ACTIVATE ALL resp. EDIT | EXPORT PARAMETERS | DEACTIVATE ALL have no effect on these parameters.

Now all type declarations are explained. The declaration in the C language is absolutely analogous; the listing below summarizes all declarations for this programming language.

```
typedef struct{
    char NuE;
    char NuI;
    char NuB;
    long double E[32];
    int  I[32];
    char B[32];
    char D[256];
    long double EMin[32];
    long double EMax[32];
    int  IMin[32];
    int  IMax[32];
    char NaE[32][41];
```

```
    char NaI[32][41];
    char NaB[32][41];
    void FAR *UserDataPtr;
    void FAR *ParentPtr;
    unsigned int ParentHWnd;
    char *ParentName;
    unsigned int UserHWindow;
    FILE *DataFile
  } TParameterStruct, FAR *PParameterStruct;

typedef struct {
    long AllowE;
    long AllowI;
    long AllowB;
    char AllowD;
  } TDialogEnableStruct, FAR *PDialogEnableStruct;

typedef struct {
    char Inputs;
    char Outputs;
    char NameI[50][41];
    char NameO[50][41];
  } TNumberOfInputsOutputs, FAR *PNumberOfInputsOutputs;

typedef long double TInputArray[50];
typedef long double TOutputArray[50];

typedef char TConnectedArray[50];

typedef struct {
    char ExportParName[21];
    char ExportParType;
    long double ExportParFValue;
    int ExportParIValue;
    char ExportParBValue;
    long double ExportParFMin, ExportParFMax;
    int ExportParIMin, ExportParIMax;
  } TExportParStruct, FAR *PExportParStruct
```

Data interface of the user-block in C

Function interface of the user-block

The function interface of the User-DLL contains all functions resp. procedures which are necessary for the functionality of the block. There exist functions which are needed for the administration of the user-block resp. its parameter dialog (so-called *organizing functions*) on one hand, and functions specifying the mathematical functionality of the block (so-called *simulation functions*) on the other hand; the simulation functions are called by BORIS at each simulation step resp. directly before the simulation is started or after it was termi-

nated. The following listing gives a survey of all available functions and their parameters; a detailed description of each function follows thereafter. All functions must be declared as *stdcall*.

Organizing functions:

```
Procedure IsUserDLL32;

Procedure GetParameterStruct(D:PParameterStruct);

Procedure GetDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct)

Procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);
Procedure GetNumberOfInputsOutputs2(D:PNumberOfInputsOutputs;
                                    ParameterFileName: PChar;
                                    UserDataPtr: Pointer;
                                    var UserHWindow: THandle);

Procedure InitUserDLL(D:PParameterStruct);
Procedure DisposeUserDLL(D:PParameterStruct);

Procedure InitUserData(D: PParameterStruct);
Procedure DisposeUserData(D: PParameterStruct);

Procedure DialogOK(D:PParameterStruct);

Procedure ShowWindowDLL(D: PParameterStruct);
Procedure HideWindowDLL(D: PParameterStruct);

Function  SetInputChar: PChar;
Function  SetOutputChar: PChar;

Function  GetDLLName: PChar;

Procedure WriteToFile(AFileHandle: word; D: PParameterStruct);
Procedure ReadFromFile(AFileHandle: word; D: PParameterStruct);
Function  NumberOfLinesInSystemFile: word;

Function  WriteParToDocFileDLL(D: PParameterStruct): integer;

Procedure CallParameter dialogDLL(D1: PParameterStruct;
                                  D2: PNumberOfInputsOutputs);

Function  ExportParCountDLL(D1: PParameterStruct;
                            D2: PNumberOfInputsOutputs): integer;
Procedure GetExportParDLL(D1: PParameterStruct;
                          D2: PNumberOfInputsOutputs;
                          ParIndex: integer; D3: PExportParStruct);
Procedure SetExportParDLL(D1: PParameterStruct;
                          D2: PNumberOfInputsOutputs;
                          ParIndex: integer; D3: PExportParStruct);
```

Simulation functions:

```
Function  CanSimulateDLL(D:PParameterStruct):Integer;
Function  CanSimulateDLL2(T, DeltaT:Extended;
                          D:PParameterStruct):Integer;
```

```
Procedure InitSimulationDLL(D:PParameterStruct;
                            Inputs:PInputArray;
                            Outputs:POutputArray);
Procedure InitSimulationDLL2(D:PParameterStruct;
                             DeltaT:Extended;
                             Inputs:PInputArray;
                             Outputs:POutputArray);

Procedure SimulateDLL(T:Extended; D:PParameterStruct;
                      Inputs:PInputArray;
                      Outputs:POutputArray);
Procedure SimulateDLL2(T, DeltaT:Extended; D:PParameterStruct;
                       Inputs:PInputArray;
                       Outputs:POutputArray);
Function SimulateDLL3(T, DeltaT:Extended; D:PParameterStruct;
                      Inputs:PInputArray;
                      Outputs:POutputArray): integer;

Procedure EndSimulationDLL;
Procedure EndSimulationDLL2(D: PParameterStruct);

Procedure SetEnhancedInformation(DeltaT, TSimu: extended;
                                 D: PParameterStruct);
Procedure SetEnhancedInformation2(DeltaT, TSimu: extended;
                                  D1: PConnectedArray;
                                  D2: PParameterStruct);

Function HasDelayDLL(D: PParameterStruct): integer;
```

The following descriptions of the various functions contain the Pascal declarations (top) as well as the declarations in C (bottom).

IsUserDLL32

Signature

```
procedure IsUserDLL32; export stdcall;
```

```
void _export _stdcall IsUserDLL32
```

Function This dummy function contains no functionality, but it must be exported to declare the DLL as a BORIS-32-bit-User-DLL. Thus the function kernel can be empty.

GetParameterStruct

Signature

```
procedure GetParameterStruct(D:PParameterStruct);
                             export stdcall;
```

```
void _export _stdcall GetParameterStruct(PParameterStruct D)
```

Params *D* is a pointer to *TParameterStruct.*

Function In *GetParameterStruct* the number of the different parameter types of the user-block and the data used to initialize the parame-

ter dialog are specified. Also the names of the parameters, their initial values and their valid range of values are defined.

Example:

The example below defines a user-block with four floating point parameters named *Sticking friction*, *Slipping friction*, *Normal force*, *Variation coefficient* and two boolean parameters named *Real friction* and *Stick and Slip from file*. Moreover a parameter file with the extension *SIM* is admitted.

```
procedure GetParameterStruct(D:PParameterStruct); export stdcall;
var i : Integer;
begin
  D^.NuE:=4;  {Four float parameters}
  D^.NuI:=0;  {No integer parameter}
  D^.NuB:=2;  {Two boolean parameters}
  StrPCopy(D^.D,'*.sim');  {Accept files with extension SIM}
  {Initialization}
  for i:=0 to 1 do begin
     D^.E[i]:=0;    {Initialize float parameter 0 and 1 with 0}
     D^.EMin[i]:=0; {Lower bound of float par. 0 and 1 is 0}
     D^.EMax[i]:=1; {Upper bound of float par. 0 and 1 is 1}
     D^.B[i]:=0;    {Initialize boolean parameter with 0}
  end;
  D^.E[2]:=0;          {Initialize float parameter 2 with 0}
  D^.EMin[2]:=0;       {Lower bound of float par. 2 is 0}
  D^.EMax[2]:=100000;  {Upper bound of float par. 2 is 100000}
  D^.E[3]:=0;          {Initialize float parameter 3 with 0}
  D^.EMin[3]:=0;       {Lower bound of float par. 3 is 0}
  D^.EMax[3]:=1;       {Upper bound of float par. 3 is 1}
  {Names of inputs and outputs (max. 40 characters) !}
  strPCopy(D^.NaE[0],'Sticking friction:');     {Name of float par. 0}
  strPCopy(D^.NaE[1],'Slipping friction:');     {Name of float par. 1}
  strPCopy(D^.NaE[2],'Normal force:');          {Name of float par. 2}
  strPCopy(D^.NaE[3],'Variation coefficient:'); {Name of float par. 3}
  strPCopy(D^.NaB[0],'Real friction');    {Name of boolean par.. 0}
  strPCopy(D^.NaB[1],'"Stick and Slip" from file'); {N. of b. p. 1}
end;
```

Sample application of GetParameterStruct

GetDialogEnableStruct

Signature

```
procedure getDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct);
                                export stdcall;

void _export _stdcall GetDialogEnableStruct
                          (PDialogEnableStruct D,
                           PParameterStruct D2)
```

Params *D* is a pointer to *TDialogEnableStruct.*

D2 is a pointer to *TParameterStruct*

Function *GetDialogEnableStruct* controls the enabled and disabled

(grayed) elements of the parameter dialog, i. e. the access of the user to these elements. The function is called if the dialog is opened and *after each input* by the user within this dialog. Thus this function can react directly if necessary.

Example:

For a user-block with float and boolean parameters all parameters shall be enabled at any time. The filename of the external parameter file however shall only be enabled if both boolean parameters are set.

```
procedure GetDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct); export stdcall;
begin
  D^.AllowE:=$FFFFFFFF;  {all float parameters enabled}
  D^.AllowB:=$FFFFFFFF;   {all boolean parameters enabled}
  {Filename only enabled if both boolean parameters are set!}
  D^.AllowD:= Byte(D2^.B[0] and D2^.B[1]);
end;
```

Sample application of GetDialogEnableStruct

Naturally it is possible to transfer only one dialog element from the disabled to the enabled state (or vice versa) and let the others unaffected. If e. g. the third floating point parameter (i. e. float parameter 2) is to be enabled and all other floating point parameters shall hold their state, this can be realized by the following code sequence:

```
...
AllowE := AllowE or $00000004;   {Float parameter 2 enabled}
...
```

GetNumberOfInputsOutputs

Signature

```
procedure GetNumberOfInputsOutputs
          (D:PNumberOfInputsOutputs); export stdcall;

void _export _stdcall GetNumberOfInputsOutputs
                      (PNumberOfInputsOutputs D)
```

Params *D* is a pointer to *TNumberOfInputsOutputs.*

Function In *GetNumberOfInputsOutputs* number and names of block inputs and outputs are specified. The names are displayed in the *Input variable(s)* resp. *Output variable(s)* listboxes of the User-DLL dialog.

Example:

The listing below demonstrates the usage of the function for a user-block with two inputs and one output.

```
procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs); export
                                                              stdcall;
begin
  D^.Inputs:=2;   { Our block has two inputs..}
  D^.Outputs:=1;  {            and one output!}
  {Names of inputs and outputs}
  StrPCopy(D^.NameI[0],'Reference value'); {Name of first input}
  StrPCopy(D^.NameI[1],'Current value');   {Name of second input}
  StrPCopy(D^.NameO[0],'Control output');  {Name of ouput}
end;
```

Sample application of GetNumberOfInputsOutputs

GetNumberOfInputsOutputs2

Signature

```
Procedure GetNumberOfInputsOutputs2
                (D:PNumberOfInputsOutputs;
                 ParameterFileName: PChar;
                 UserDataPtr: Pointer;
                 var UserHWindow: THandle); export stdcall

void _export _stdcall GetNumberOfInputsOutputs2
                     (PNumberOfInputsOutputs D,
                      char* ParameterFileName,
                      void* UserDataPtr,
                      int* UserHWindow)
```

Params *D* is a pointer to *TNumberOfInputsOutputs.*

ParameterFileName is a pointer to the parameter *D* of *TParameterStruct.*

UserDataPtr is a pointer to the parameter *UserDataPtr* of *TParameterStruct.*

UserHWindow is a pointer to the parameter *UserHWindow* of *TParameterStruct.*

Function This function can be used instead of *GetNumberOfInputsOutputs* if the number of inputs and/or outputs can be variable.

Example:

The listing below shows a realization of the function for a user-block with the number of in- and outputs read from a file specified by *ParameterFileName*.

```
Procedure GetNumberOfInputsOutputs2(D:PNumberOfInputsOutputs;
           ParameterFileName: PChar; UserDataPtr: Pointer;
           var UserHWindow: THandle); export stdcall
var ParFile: TextFile;
    nIn, nOut, i: integer
begin
  AssignFile(ParFile, ParameterFileName);  //Assign filename
  ResetFile(ParFile);                      //Open file
  readln(ParFile, nIn, nOut);              //Read number of inputs/outputs
  D^.Inputs  := nIn;                       //...and assign them to block
```

```
  D^.Outputs := nOut;
  {Names of inputs and outputs}
  for i:=1 to nIn  do StrPCopy(D^.NameI[i-1], 'Input' + IntToStr(i));
  for i:=1 to nOut do StrPCopy(D^.NameO[i-1], 'Output' + IntToStr(i));
end;
```

Sample application of GetNumberOfInputsOutputs2

InitUserDLL
DisposeUserDLL

Signature

```
procedure InitUserDLL(D:PParameterStruct);
                                        export stdcall;
procedure DisposeUserDLL(D:PParameterStruct);
                                        export stdcall;

void _export _stdcall InitUserDLL(PParameterStruct D)
void _export _stdcall DisposeUserDLL(PParameterStruct D)
```

Params *D* is a pointer to *TParameterStruct.*

Function *InitUserDLL* is called by BORIS directly after initializing the user-block. Thus this function can e. g. be used to allocate dynamic memory for data which cannot be saved in the *TParameterStruct* structure of the block. These can be state variables, temporary data of any kind or more complex structures which can be read from an external file. The pointer to these dynamic data can be assigned to the *UserDataPtr* parameter of *TParameterStruct*.

Another sample application for this function are user-blocks with a visualization window. In such cases this function can be used to create the window (see *Liquid level control* example later).

The counterpart of *InitUserDLL* is the *DisposeUserDLL* function. This function is called directly before the block is destroyed. Thus the dynamic memory allocated in *InitUserDLL* has to be deallocated in *DisposeUserDLL*.

If the user data are not needed during the whole "life cycle" of the block but only during the simulation itself (which will be the common case), instead of *InitUserDLL* and *DisposeUserDLL* the functions *InitUserData* and *DisposeUserData* (see below) should be used.

The use of *InitUserDLL* and *DisposeUserDLL* is only necessary in complex user blocks.

Example:

A user block has to be developed which uses a 20x20-matrix with elements of the *extended* type and a vector of the dimension 50 with *integer* elements. The listing below shows the realization in Pascal.

```
...
...
type
  {Type declaration for user data}
  PUserData = ^TUserData;
  TUserData = record
    Matrix: array[1..20, 1..20] of extended;
    Vector: array[1..50] of integer;
  end;


procedure InitUserDLL(D: PParameterStruct); export stdcall;
begin
  ...
  ...
  {Allocate user data memory}
  GetMem(D^.UserDataPtr, SizeOf(TUserData));
  ...
  ...
end {of InitUserDLL};


procedure DisposeUserDLL(D: PParameterStruct); export stdcall;
begin
  ...
  ...
  {Free user data memory}
  FreeMem(D^.UserDataPtr, SizeOf(TUserData));
  ...
  ...
end {of DisposeUserDLL};
...
...
```

Sample application of InitUserDLL *and* DisposeUserDLL

To get access to an explicit matrix element e. g. the Pascal code below can be used:

```
...
...
{set 2nd row, 3rd column of matrix to 5}
PUserData(UserDataPtr)^.Matrix[2, 3] := 5;
...
...
```

Access to user data

InitUserData
DisposeUserData

Signature

```
procedure InitUserData(D:PParameterStruct);
                                        export stdcall;
procedure DisposeUserData(D:PParameterStruct);
                                        export stdcall;

void _export _stdcall InitUserData(PParameterStruct D)
void _export _stdcall DisposeUserData(PParameterStruct D)
```

Params *D* is a pointer to *TParameterStruct.*

Function *InitUserData* and *DisposeUserData* have the same range of application as the functions *InitUserDLL* and *DisposeUserDLL* described above. The only difference is that *InitUserData* is not called if the block is created but directly before the simulation starts. According to this the function *DisposeUserData* is called directly after the simulation has terminated. So these functions are appropriate for managing data which are only needed during the simulation (see example *3D-Characteristic map* later). For simple applications these functions are redundant.

DialogOK

Signature

```
procedure DialogOK(D:PParameterStruct);
                                          export stdcall;

void _export _stdcall DialogOK(PParameterStruct D)
```

Params *D* is a pointer to *TParameterStruct.*

Function Normally *DialogOK* is only needed in User-DLLs with a visualization window. The function is called by BORIS if the parameter dialog is left by clicking the *OK* button. It can be used e. g. to adjust the contents of the visualization window to the modified block parameters.

Example:

The listing below demonstrates the use of *DialogOK*.

```
procedure DialogOK(D:PParameterStruct); export stdcall;
begin
  {After the parameter dialog has been left by clicking the OK button,
   the visualization window of which handle is stored in
   UserHWindow shall be refreshed!}
  InvalidateRect(D^.UserHWindow, nil, true);
end;
```

Sample application of DialogOK

ShowWindowDLL
HideWindowDLL

Signature

```
Procedure ShowWindowDLL(D: PParameterStruct);
                                        export stdcall;
Procedure HideWindowDLL(D: PParameterStruct);
                                        export stdcall;

void _export _stdcall ShowWindowDLL(PParameterStruct D)
void _export _stdcall HideWindowDLL(PParameterStruct D)
```

Params *D* is a pointer to *TParameterStruct.*

Function These functions are only needed by User-DLLs with a visualization. They can be used to minimize resp. normalize the visualization window. The functions are called by BORIS if the menu option VIEW | SHOW ALL OUTPUT WINDOWS resp. VIEW | HIDE ALL OUTPUT WINDOWS is chosen.

Example:

The listing below demonstrates the use of both functions.

```
...
procedure ShowWindowDLL(D: PParameterStruct); export stdcall;
begin
  {Show output window in normal size}
  ShowWindow(D^.UserHWindow, sw_Normal);
end;

procedure HideWindowDLL(D: PParameterStruct); export stdcall;
begin
  {Minimize output window}
  ShowWindow(D^.UserHWindow, sw_Minimize);
end;
```

Sample application of ShowWindowDLL *resp.* HideWindowDLL

SetInputChar
SetOutputChar

Signature

```
Function SetInputChar: PChar; export stdcall;
Function SetOutputChar: PChar; export stdcall;

char* _export _stdcall SetInputChar(void)
char* _export _stdcall SetOutputChar(void)
```

Return value The return value is a pointer to a string containing one character for each block input resp. output.

Function *SetInputChar* resp. *SetOutputChar* can be used to change the default caption of the block inputs resp. outputs. Normally a block

input is labelled *I* (if only one input exists) and the output *O*; if several in-/outputs exist, they are numbered. By implementing *SetInputChar* resp. *SetOutputChar* each imput resp. output can be labelled individually.

Example:

For a User-DLL-block with two inputs and one output the inputs shall be labelled *w* resp. *x* and the output *y*. The listing below demonstrates the implementation of the two functions *SetInputChar* and *SetOutputChar*.

```
Function SetInputChar: PChar; export stdcall;
begin
  {Label input 1 "w", input 2 "x"}
  SetInputChar := 'wx';
end;

Function SetOutputChar: PChar; export stdcall;
begin
  {Label output "y"}
  SetOutputChar := 'y';
end;
```

Sample application of SetInputChar *and* SetOutputChar

GetDLLName

Signature

```
Function GetDLLName: PChar; export stdcall;
```

```
char _export _stdcall GetDLLName(void)
```

Return value Title of the User-DLL-block

Function This function provides the specification of a block title which appears in the BLOCKS | USER-DLL-BLOCKS... submenu resp. the hint window of the *User* register of the system block toolbar.

Example:

The listing below shows the realization of this function for a User-DLL called *Big Display*.

```
function GetDLLName: PChar; export stdcall;
begin
  GetDLLName := 'Big Display';
end;
```

Sample realization of GetDLLName

WriteParToDocFileDLL

Signature

```
Function WriteParToDocFileDLL(D: PParameterStruct;
                    Count: integer; s: PChar): integer;
                                        export stdcall;

int _export _stdcall WriteParToDocFileDLL
                (PParameterStruct D, Int Count, Char s)
```

Params *D* is a pointer to *TParameterStruct.*

Count is a counter for the parameters to be written.

s is the output text for the parameter.

Return value A return value 0 specifies that no more parameters have to be written. In case of any other return value the function is called again by BORIS with a *Count* parameter increased by 1.

Function This function can be used to write user-defined block parameters into a document file created by the BORIS RTF file generator. Therefor BORIS calls the function beginning with *Count = 0* until the function delivers a return value of 0. At each call the function has to transfer the corresponding parameter string in *s*.
Note: If only the standard parameters from *TParameterStruct* (i. e. the arrays *E*, *I* and *B*) are used, this function has not to be declared because these standard parameters are automatically written to the document file by BORIS!

Example:

The listing below shows the implementation of the function for the case that *UserDataPtr* contains three user-defined floating point parameters (*a*, *b* and *c*) which are to be written to the document file:

```
function WriteParToDocFileDLL(D: PParameterStruct; count: integer; s:
PChar): integer; export stdcall;
begin
  case Count of
    0: begin
         StrPCopy(s, 'a = ' + FloatToStr(D^.UserDataPtr.a) + #13#10);
         Result := 1;  // Further parameters follow!
       end;
    1: begin
         StrPCopy(s, 'b = ' + FloatToStr(D^.UserDataPtr.b) + #13#10);
         Result := 1;  // Further parameters follow!
       end;
    2: begin
         StrPCopy(s, 'c = ' + FloatToStr(D^.UserDataPtr.c) + #13#10);
         Result := 0;  // Parameter was the last one!
       end;
  end;
```

Sample implementation of WriteParToDocFileDLL

CallParameterDialogDLL

Signature

```
Procedure CallParameter dialogDLL(D1: PParameterStruct;
          D2: PNumberOfInputsOutputs); export stdcall;

void _export _stdcall CallParameter dialogDLL
     (PParameterStruct D1, PNumberOfInputsOutputs D2)
```

Params *D1* is a pointer to *TParameterStruct.*

D2 is a pointer to *TNumberOfInputsOutputs.*

Function This function provides the call of a user-defined parameter dialog instead of the standard dialog.

Example:

The listing of the sample User-DLL DLLDEMO2.DLL below demonstrates the call of a user-defined parameter dialog in DELPHI.

```
procedure CallParameterDialogDLL(D1:PParameterStruct;
                                 D2:PNumberOfInputsOutputs);export stdcall;
var s: string;
    Code: integer;
begin
  Form1 := TForm1.Create(Application);
  Str(D1^.E[0]:10, s);
  Form1.Edit1.Text := s;
  Form1.Checkbox1.Checked := D1^.B[0] > 0;
  if Form1.ShowModal = mrOk then begin
    Val(Form1.Edit1.Text, D1^.E[0], Code);
    D1^.B[0] := ord(Form1.Checkbox1.Checked);
  end;
  Form1.Free;
end;
```

Sample realization of CallParameterDialogDLL

ExportParCountDLL

Signature

```
Function ExportParCountDLL(D1: PParameterStruct;
                    D2: PNumberOfInputsOutputs): integer;
                                        export stdcall;

int _export _stdcall ExportParCountDLL
     (PParameterStruct D1, PNumberOfInputsOutputs D2)
```

Params *D1* is a pointer to *TParameterStruct*.

D2 is a pointer to *TNumberOfInputsOutputs.*

Return value The return value specifies the number of parameters exported by the User-DLL.

Function Combined with the functions *GetExportParDLL* and *SetExport-*

ParDLL described below this function is responsible for the realization of User-DLL export parameters. If no export parameters have to be declared, these functions are obsolete.

Example:

The following listing demonstrates the realizazion of this function in DELPHI for a block which exports two parameters:

```
function ExportParCountDLL(D1: PParameterStruct;
                           D2: PNumberOfInputsOutputs): integer;
                                                        export stdcall;
begin
  Result := 2; // Two export parameters!
end;
```

Sample realization of ExportParCountDLL

GetExportParDLL

Signature

```
Procedure GetExportParDLL(D1: PParameterStruct;
                                  D2: PNumberOfInputsOutputs;
                                  ParIndex: integer;
                                  D3: PExportParStruct);
                                                  export stdcall;

void _export _stdcall GetExportParDLL
       (PParameterStruct D1, PNumberOfInputsOutputs D2,
        int ParIndex, PExportParStruct D3)
```

Params *D1* is a pointer to *TParameterStruct.*

D2 is a pointer to *TNumberOfInputsOutputs.*

ParIndex is the index of the export parameter (starting with 1)

D3 is a pointer to *TExportParStruct.*

Function This function delivers the information needed by BORIS concerning the export parameter specified by *ParIndex*. The information are transferred via the pointer *D3*.

Example:

The listing below demonstrates the realization of this function in DELPHI for a block that exports two floating point parameters:

```
procedure GetExportParDLL(D1:PParameterStruct; D2:PNumberOfInputsOutputs;
                          ParIndex: integer; D3: PExportParStruct);
                                                        export stdcall;
begin
  // Example demonstrating the export of two floating point parameters.
  // Here exemplary the floating point parameters E[0] and E[3]
  // of the TParameterStruct (D1) structure are exported.
  case ParIndex of
```

```
    // First export parameter of the block
    1: with D3^ do begin
         ExportParName   := 'Gain K';        // Name of first
                                             // parameter
         ExportParType   := 'F';             // Parameter is of type
                                             // floating point
         ExportParFValue := D1^.E[0];        // It's the first floating
                                             // point parameter of the
                                             // TParameterStruct structure!
         ExportParFMin   := -1E30;            // Minimum valid value of the
                                              // parameter
         ExportParFMax   := +1E30;            // Maximum valid value of the
                                              // parameter
       end;
    // Second export parameter of the block
    2: with D3^ do begin
         ExportParName   := 'Damping D';     // Name of second
                                             // parameter
         ExportParType   := 'F';             // Parameter is of type
                                             // floating point
         ExportParFValue := D1^.E[3];        // It's the fourth floating
                                             // point parameter of the
                                             // TParameterStruct structure
         ExportParFMin   := -1E30;           // Minimum valid value of the
                                             // parameter
         ExportParFMax   := +1E30;           // Maximum valid value of the
                                             // parameter
       end;
  end;
end;
```

Sample implementation of GetExportParDLL

SetExportParDLL

Signature

```
Procedure SetExportParDLL(D1: PParameterStruct;
                          D2: PNumberOfInputsOutputs;
                          ParIndex: integer;
                          D3: PExportParStruct);
                                        export stdcall;

void _export _stdcall SetExportParDLL
       (PParameterStruct D1, PNumberOfInputsOutputs D2,
        int ParIndex, PExportParStruct D3)
```

Params

D1 is a pointer to *TParameterStruct.*

D2 is a pointer to *TNumberOfInputsOutputs.*

ParIndex is the index of the export parameter (starting with 1).

D3 is a pointer to *TExportParStruct.*

Function

This function is the counterpart to *GetExportParDLL*. It is called by BORIS when the export parameter specified by *ParIndex* has to be modified. The new value is transferred via the pointer *D3*.

Example:

The listing below demonstrates the realization of this function in DELPHI for a block that exports two floating point parameters (compare listing at *GetExportParDLL*).

```
procedure SetExportParDLL(D1:PParameterStruct; D2:PNumberOfInputsOutputs;
                          ParIndex: integer; D3: PExportParStruct);
                                                              export stdcall;
begin
  // Example demonstrating the export of two floating point parameters.
  // Here exemplary the floating point parameters E[0] and E[3]
  // of the TParameterStruct (D1) structure are exported.
  case ParIndex of
    // First export parameter of the block
    1: with D3^ do begin
         D1^.E[0] := ExportParFValue;  // It's the first
                                       // floating point parameter of the
                                       // TParameterStruct structure!
       end;
    // Second export parameter of the block
    2: with D3^ do begin
         D1^.E[3] := ExportParFValue;  // It's the fourth
                                       // floating point parameter of the
                                       // TParameterStruct structure!
       end;
  end;
end;
```

Sample realization of SetExportParDLL

WriteToFile
ReadFromFile
NumberOfLinesInSystemFile

Signature

```
Procedure WriteToFile(AFileHandle: word;
            D: PParameterStruct); export stdcall
Procedure ReadFromFile(AFileHandle: word;
            D: PParameterStruct); export stdcall
Function  NumberOfLinesInSystemFile: word;
                                    export stdcall

void _export _stdcall WriteToFile
     (unsigned short AFileHandle, PParameterStruct D)
void _export _stdcall ReadFromFile
     (unsigned short AFileHandle, PParameterStruct D)

int _export _stdcall NumberOfLinesInSystemFile(void);
```

Params *AFileHandle* is the handle of the system file.

D is a pointer to *TParameterStruct.*

Function These functions provide the storage of block specific parameters which were defined outside the 32 standard float, integer and

boolean parameters. Thus these functions are only seldom needed. The sample User-DLL BIGDIS32.DLL located in the *UserDLLs* directory demonstrates the use of these functions.

CanSimulateDLL

Signature

```
function CanSimulateDLL(D:PParameterStruct):Integer;
                                          export stdcall;

int _export _stdcall CanSimulateDLL(PParameterStruct D)
```

Params *D* is a pointer to *TParameterStruct.*

Return value A return value of 1 indicates that the block can be simulated, a return value of 0 that the block cannot be simulated.

Function *CanSimulateDLL* is called by BORIS before the simulation starts (even before *InitSimulationDLL*) to check if the User-DLL block can be simulated. Thus this function can be used to prevent the simulation in special cases, e. g. if the block needs a parameter file and the specified file does not exist. To disable the simulation, the result value has to be set to 0, to enable it, the result value has to be set to 1.

Example:

The listings below demonstrate two different realizations of *CanSimulateDLL*.

```
Function CanSimulateDLL(D: PParameterStruct): integer; export stdcall;
begin
  CanSimulateDLL := 1; {Simulation always allowed!}
end;
```

```
Function CanSimulateDLL(D: PParameterStruct): integer; export stdcall;
begin
  // Simulation is only allowed if float parameter 0
  // and float parameter 1 have different signs
  if D^.E[0] * D^.E[1] < 0 then
    CanSimulateDLL := 1
  else
    CanSimulateDLL := 0;
end;
```

Sample applications of CanSimulateDLL

CanSimulateDLL2

Signature

```
function CanSimulateDLL2(T, DeltaT:Extended;
                         D:PParameterStruct):Integer;
                                          export stdcall;

int _export _stdcall CanSimulateDLL(long double T,
```

```
                                        long double DeltaT,
                                        PParameterStruct D)
```

Params *T* is the simulation length specified in BORIS
DeltaT is the simulation step size.
D is a pointer to *TParameterStruct.*

Return value A return value of 0 means, that the User-DLL does not allow a simulation; a return value of 1 means, that the block allows the simulation.

Function May be used alternatively to *CanSimulateDLL* if the simulation length and/or step size is to be used within the function.

InitSimulationDLL

Signature

```
procedure InitSimulationDLL (D:PParameterStruct;
                             Inputs:PInputArray;
                             Outputs:POutputArray);
                             export stdcall;

void _export _stdcall InitSimulationDLL (PParameterStruct D,
                              TInputArray FAR Inputs,
                              TOutputArray FAR Outputs)
```

Params *D* is a pointer to *TParameterStruct.*
Inputs is a pointer to *TInputArray.*
Outputs is a pointer to *TOutputArray.*

Function *InitSimulationDLL* initializes the block at the time $t = 0$ resp. the first simulation step. Thus it can be used for setting default values needed before the simulation starts (e. g. initial values for state variables or outputs etc.). If required this function can also be used for reading data from an external parameter file (see example *3D-characteristic map* below). The current inputs of the block are passed through the *Inputs* pointer, the outputs through the *Outputs* pointer.

Example:

The listing below demonstrates the realization of *InitSimulationDLL* for a block with two inputs and two outputs. At $t = 0$ the first output is set by the first input and the second output is set to zero.

```
procedure InitSimulationDLL (D:PParameterStruct; Inputs:PInputArray;
                             Outputs:POutputArray); export stdcall;
begin
  Outputs^[1] := Inputs^[1];  {Output 1 = Input 1}
  Outputs^[2] := 0.0;         {Initialize output 2 by 0}
end;
```

Sample application of InitSimulationDLL

InitSimulationDLL2

Signature

```
procedure InitSimulationDLL2(DeltaT:Extended;
                             D:PParameterStruct;
                             Inputs:PInputArray;
                             Outputs:POutputArray);
                             export stdcall;

void _export _stdcall InitSimulationDLL2
                             (long double DeltaT,
                              PParameterStruct D,
                              TInputArray FAR Inputs,
                              TOutputArray FAR Outputs)
```

Params *DeltaT* is the simulation step size.

D is a pointer to *TParameterStruct*.

Inputs is a pointer to *TInputArray*.

Outputs is a pointer to *TOutputArray*.

Function *InitSimulationDLL2* may be used alternatively to *InitSimulationDLL* if the simulation step size *DeltaT* is to be used within the function.

SimulateDLL

Signature

```
Procedure SimulateDLL(T:Extended; D:PParameterStruct;
                      Inputs:PInputArray;
                      Outputs:POutputArray);
                      export stdcall;

void _export _stdcall SimulateDLL(long double T,
                            PParameterStruct D,
                            TInputArray FAR Inputs,
                            TOutputArray FAR Outputs)
```

Params *T* is the time of the current simulation step.

D is a pointer to *TParameterStruct.*

Inputs is a pointer to *TInputArray.*

Outputs is a pointer to *TOutputArray*.

Function *SimulateDLL* specifies the numerical kernel of the block, i. e. the algorithm executed at each simulation step. Thus this function is called at each simulation step. The parameter *T* contains the current simulation time, the parameter *D* a pointer to the block data. The current inputs of the block are passed through the pointer *Inputs*, the outputs determined by the function through the pointer *Outputs*.

If the simulation step size ΔT is needed for execution, the function *SimulateDLL2* (see below) should be used instead of *SimulateDLL*.

Example:

The listing below demonstrates the use of *SimulateDLL* for a block with two inputs and two outputs. The first output is determined as the arithmetical mean of both inputs multiplicated with float parameter 0, the second output as the geometrical mean multiplicated with float parameter 1.

```
Procedure SimulateDLL(T:Extended; D:PParameterStruct;
                      Inputs:PInputArray;
                      Outputs:POutputArray); export stdcall;
begin
  Outputs^[1] := D^.E[0] * (Inputs^[1] + Inputs^[2]) | 2;
  Outputs^[2] := D^.E[1] * sqrt((Inputs^[1] * Inputs^[2]));
end;
```

Sample realization of SimulateDLL

SimulateDLL2

Signature

```
Procedure SimulateDLL2(T, DeltaT:Extended;
                       D:PParameterStruct;
                       Inputs:PInputArray;
                       Outputs:POutputArray);
                       export stdcall;

void _export _stdcall SimulateDLL2(long double T,
                                   long double DeltaT,
                                   PParameterStruct D,
                                   TInputArray FAR Inputs,
                                   TOutputArray FAR Outputs)
```

Params *T* is the time value of the current simulation step.

DeltaT is the simulation step size.

D is a pointer to *TParameterStruct.*

Inputs is a pointer to *TInputArray.*

Outputs is a pointer to *TOutputArray.*

Function *SimulateDLL2* may be used instead of *SimulateDLL* if the simulation step size is to be used within the function.

SimulateDLL3

Signature

```
Function SimulateDLL3(T, DeltaT:Extended;
```

```
                                D:PParameterStruct;
                                Inputs:PInputArray;
                                Outputs:POutputArray): integer;
                                 export stdcall;

int _export _stdcall SimulateDLL3(long double T,
                                  long double DeltaT,
                                  PParameterStruct D,
                                  TInputArray FAR Inputs,
                                  TOutputArray FAR Outputs)
```

Params *T* is the time value of the current simulation step.

DeltaT is the simulation step size.

D is a pointer to *TParameterStruct*.

Inputs is a pointer to *TInputArray*.

Outputs is a pointer to *TOutputArray*.

Return value A return value of 0 effects that the simulation continues; any other return value results in a simulation termination after the current simulation step.

Function *SimulateDLL3* may be used instead of *SimulateDLL* or *SimulateDLL2* if the simulation is to be terminated by the User-DLL block under specific conditions.

EndSimulationDLL

Signature

```
procedure EndSimulationDLL; export stdcall;

void _export _stdcall EndSimulationDLL(void)
```

Function *EndSimulationDLL* is called after the simulation has been terminated. Thus this function can be used e. g. to close open files. In most applications this function is not needed.

The function *EndSimulationDLL2* (see below) can be used instead of *EndSimulationDLL* if the pointer to the *TParameterStruct* structure is needed.

Example:

The listing below demonstrates the application of *EndSimulationDLL*.

```
procedure EndSimulationDLL; export stdcall;
begin
  Close(DataFile);   {Close parameter file}
end;
```

Sample application of EndSimulationDLL

EndSimulationDLL2

Signature

```
procedure EndSimulationDLL2(D: PParameterStruct);
                                        export stdcall;

void _export _stdcall EndSimulationDLL2
                                    (PParameterStruct D)
```

Params *D* is a pointer to *TParameterStruct.*

Function *EndSimulationDLL2* can be used instead of *EndSimulationDLL* if the parameter structure *TParameterStruct* is required for execution.

SetEnhancedInformation

Signature

```
procedure SetEnhancedInformation
                                (DeltaT, TSimu: extended;
                                 D: PParameterStruct);
                                 export stdcall;

void _export _stdcall SetEnhancedInformation
                                  (long double DeltaT,
                                   long double TSimu,
                                   PParameterStruct D)
```

Params *DeltaT* is the simulation step size.

TSimu is the simulation length.

D is a pointer to *TParameterStruct.*

Function *SetEnhancedInformation* is used seldom. The function delivers the current simulation step size and time. Both parameters can e. g. be saved in unused float parameters. This function is called by BORIS at the beginning of a simulation run (i. e. before *InitSimulationDLL* is called).

Example:

In the example below *DeltaT* and *TSimu* are saved in the float parameters 3 and 4.

```
procedure SetEnhancedInformation(DeltaT, TSimu: extended;
                                 D: PParameterStruct); export stdcall;
begin
  D^.E[3] := DeltaT;
  D^.E[4] := TSimu;
end;
```

Sample application of SetEnhancedInformation

SetEnhancedInformation2

Signature

```
procedure SetEnhancedInformation2
                                  (DeltaT, TSimu: extended;
                                   D1: PConnectedArray;
                                   D2: PParameterStruct);
                                   export stdcall;

void _export _stdcall SetEnhancedInformation2
                                    (long double DeltaT,
                                     long double TSimu,
                                     PConnectedArray D1,
                                     PParameterStruct D2)
```

Params *DeltaT* is the simulation step size.

TSimu is the simulation length.

D1 is a pointer to *TConnectedArray.*

D2 is a pointer to *TParameterStruct.*

Function This function may be used instead of *SetEnhancedInformation* to get information which block input is connected to another system block and which is not. The function is called by BORIS at the beginning of a simulation (i. e. before *InitSimulationDLL*).

HasDelayDLL

Signature

```
function HasDelayDLL(D: PParameterStruct): integer;
                                           export stdcall;

int _export _stdcall HasDelayDLL(PParameterStruct D)
```

Params *D* is a pointer to *TParameterStruct.*

Return value A return value <> 0 characterizes that the block has a delay.

Function This function can be used to characterize that the User-DLL block has a delay and thus e. g. can be used to resolve algebraic loops. The consequence is that when calling the *SimulateDLL* (resp. the *SimulateDLL2* or *SimulateDLL3*) function BORIS does *not* store the current input values in the *Inputs* vector but those of the previous simulation step. Furthermore when the *InitSimulationDLL* resp. *InitSimulationDLL2* function is called the *Inputs* vector contains *no valid data* at all. Therefore in these initialization functions the block has to set the block outputs to fixed, i. e. input independent values.

If the function is not used, the User-DLL block is automatically interpreted as a non-delaying block.

Examples

Example 1: A simple User-DLL

We want to start with a simple example of a User-DLL with two inputs and one output. Depending on the value of the first input (control input) x_1 the value of the second input (data input) x_2 has to be multiplicated with different gains k_1, k_2, k_3 and passed to the output y. A boolean parameter b_1 effects the inverting of the output:

- If x_1 is negative, x_2 has to be multiplicated with k_1.
- If x_1 is zero, x_2 has to be multiplicated with k_2.
- If x_1 is positive, x_2 has to be multiplicated with k_3.
- If b_1 is set, the output additionally has to be inverted.

The listing below demonstrates the implementation of the DLL in Pascal. The compiled file DLLDEMO1.DLL is located in the *UserDLLs* directory, the corresponding source code DLLDEMO1.DPR in the *Sources* subdirectory. All important comments regarding the implementation are included.

```
Library DLLDemo1;
(************************************************)
(*                                              *)
(*         Simple demo DLL for BORIS            *)
(*         (Example 1 of handbook)              *)
(*                                              *)
(************************************************)

uses WinProcs,
  SysUtils,
  WinTypes;

type
  PParameterStruct=^TParameterStruct;
  TParameterStruct=packed record
    NuE : Byte;
    NuI : Byte;
    NuB : Byte;
    E:Array[0..31] of Extended;
    I:Array[0..31] of Integer;
```

```
    B:Array[0..31] of Byte;
    D:Array[0..255] of char;}
    EMin:Array[0..31] of Extended;
    EMax:Array[0..31] of Extended;
    IMin:Array[0..31] of Integer;
    IMax:Array[0..31] of Integer;
    NaE : Array[0..31,0..40] of char;}
    NaI : Array[0..31,0..40] of char;
    NaB : Array[0..31,0..40] of char;
    {the rest is not needed in this block!}
  end;

  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnablestruct=packed record
    AllowE: Longint;
    AllowI: Longint;
    AllowB: Longint;
    AllowD: Byte;
  end;

  PNumberOfInputsOutputs=^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs=packed record
    Inputs :Byte;                    {Inputs}
    Outputs:Byte;                    {Outputs}
    NameI : Array[0..49,0..40] of char;
    NameO : Array[0..49,0..40] of char;
  end;

  PInputArray = ^TInputArray;
  TInputarray = packed array[1..50] of extended;
  POutputArray = ^TOutputArray;
  TOutputarray = packed array[1..50] of extended;


procedure GetParameterStruct(D:PParameterStruct);export stdcall;
begin
  D^.NuE:=3; {3 float parameters}
  D^.NuI:=0; {0 integer parameters}
  D^.NuB:=1; {1 boolean parameter}
  {Names of parameters}
  StrCopy(D^.NaE[0], 'Gain k1 for control input < 0');
  StrCopy(D^.NaE[1], 'Gain k2 for control input = 0');
  StrCopy(D^.NaE[2], 'Gain k3 for control input > 0');
  StrCopy(D^.NaB[0], 'Output inverted');
  {Initialize parameters}
  D^.E[0] := 1;
  D^.E[1] := 5;
  D^.E[2] := 10;
  D^.B[0] := 0;
  {Set range for parameters}
  D^.EMin[0] := 0.0; D^.EMax[0] := 100000;
  D^.EMin[1] := 0.0; D^.EMax[1] := 100000;
  D^.EMin[2] := 0.0; D^.EMax[2] := 100000;
end;

procedure GetDialo-
gEnableStruct(D:PDialogEnableStruct;D2:PParameterStruct);export stdcall;
begin
  {All dialog controls are enabled}
  D^.AllowE := $FFFFFFFF;
  D^.AllowI := $FFFFFFFF;
  D^.AllowB := $FFFFFFFF;
```

```
end;

procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);
            export stdcall;
begin
  D^.Inputs:=2;            { Our block has 2 inputs}
  D^.Outputs:=1;           {         and 1 output !}
  StrPCopy(D^.NameI[0],'data input');    {Name of 1st input}
  StrPCopy(D^.NameI[1],'control input');   {Name of 2nd input}
  StrPCopy(D^.NameO[0],'output');         {Name of 1st output}
end;

function CanSimulateDLL(D:PParameterStruct):Integer; export stdcall;
begin
  {Simulation always allowed!}
  CanSimulateDLL := 1;
end;

procedure SimulateDLL(T:Extended;D:PParameterStruct;Inputs:PInputArray;
                      Outputs:POutputArray);export stdcall;
begin
  if Inputs^[1] < 0 then {Input 1 negative}
    Outputs^[1] := D^.E[0] * Inputs^[2]
  else if Inputs^[1] = 0 then {Input 1 zero}
    Outputs^[1] := D^.E[1] * Inputs^[2]
  else {Input 1 positive}
    Outputs^[1] := D^.E[2] * Inputs^[2];
  if D^.B[0] = 1 then {Boolean parameter 0 = 1 ==> Invert output}
    Outputs^[1] := -Outputs^[1];
end;

procedure InitSimulationDLL(D:PParameterStruct;Inputs:PInputarray;
                            Outputs:POutputArray);export stdcall;
begin
  {The initialization step is the same as a normal step. So
   we just call the standard procedure SimulateDLL!}
  SimulateDLL(0, D, Inputs, Outputs);
end;

procedure EndSimulationDLL; export stdcall;
begin
  {not needed}
end;

function SetInputChar: PChar; export stdcall;
begin
  {Label control input 'S', data input 'D' }
  SetInputChar := 'SD';
end;

procedure IsUserDLL32; export stdcall;
begin
end;

{Function export }
exports
  GetParameterStruct,
  GetDialogEnableStruct,
  GetNumberOfInputsOutputs,
  CanSimulateDLL,
  InitSimulationDLL,
  SimulateDLL,
```

```
  EndSimulationDLL,
  SetInputChar,
  IsUserDLL32;
begin
  {Initialization of DLL (not necessary)}
end.
```

Pascal listing for example 1

Example 2: User-defined 3D-Characteristic map

As an enhancement of the user-defined characteristic curve - which is a standard block of the BORIS block library - a User-DLL is to be implemented which realizes a user-defined *3D-characteristic map* $z = f(x, y)$. The User-DLL has to fulfill the following specifications:

- The 3D-characteristic is to be entered in matrix form as a FWM-file (see paragraph *WinFACT file types* in chapter 2 *Basics*). The name of the file is specified in the parameter dialog. The underlying abscissa range $[\, x_{\min,} x_{\max} \,]$ resp. $[\, y_{\min,} y_{\max} \,]$ as well is set in the parameter dialog. The maximum dimension of the matrix is 20×20.
- If desired, a linear interpolation can be carried out between the base points. If a pair of input values (x, y) is outside the range covered by the base point matrix, the output value is calculated by extrapolation.
- In addition to the *z*-output a second output shall indicate whether a pair of input values (x, y) is out of range covered by the base point matrix, e. g. extrapolation is activated. In this case the second output takes High-level, otherwise Low-level. The values for High- and Low-level have to be set in the parameter dialog.

The following listing illustrates the implementation in Pascal. The compiled file DRDFELD.DLL is located in the *UserDLLs* directory, the source code file DRDFELD.DPR in the *Sources* subdirectory. All relevant comments regarding the implementation are included.

```
Library DRDFeld;

(************************************************)
(*                                              *)
(*              3D-Characteristic-DLL           *)
(*             (Example 2 of handbook)          *)
(*                                              *)
(************************************************)

uses Windows, SysUtils;

type
  PParameterStruct=^TParameterStruct;
```

```
  TParameterStruct= packed record
    NuE : Byte;
    NuI : Byte;
    NuB : Byte;
    E:Array[0..31] of Extended;
    I:Array[0..31] of Integer;
    B:Array[0..31] of Byte;
    D:Array[0..255] of char;
    EMin:Array[0..31] of Extended;
    EMax:Array[0..31] of Extended;
    IMin:Array[0..31] of Integer;
    IMax:Array[0..31] of Integer;
    NaE : Array[0..31,0..40] of char;
    NaI : Array[0..31,0..40] of char;
    NaB : Array[0..31,0..40] of char;
    UserDataPtr: Pointer;
    {The rest of TParameterStruct is not needed here!}
  end;

  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnablestruct=packed record
    AllowE: Longint;
    AllowI: Longint;
    AllowB: Longint;
    AllowD: Byte;
  end;

  PNumberOfInputsOutputs=^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs=packed record
    Inputs :Byte;                       {Inputs}
    Outputs:Byte;                       {Outputs}
    NameI : Array[0..49,0..40] of char;
    NameO : Array[0..49,0..40] of char;
  end;

  PInputArray = ^TInputArray;
  TInputArray = packed array[1..50] of extended;
  POutputArray = ^TOutputArray;
  TOutputarray = packed array[1..50] of extended;


  {TUserData contains the characteristic data}
  TSingleMatrix = Array[1..20, 1..20] of Single;
  PUserData = ^TUserData;
  TUserData = record
                z: TSingleMatrix; {Function value matrix}
                nx,ny: Word;      {Columns resp. rows of matrix}
                dx,dy: Extended;  {x- resp. y-distance of two base points}
              end;

Const
  {Two constants for later...}
  SingleMin=-3.4E38;
  SingleMax= 3.4E38;


procedure GetParameterStruct(D:PParameterStruct);export stdcall;
begin
  D^.NuE:=6;              {Six float}
  D^.NuI:=0;              {No integer }
  D^.NuB:=1;              {One boolean}
  StrPCopy(D^.D,'*.fwm');  {Files have extension FWM}
```

```
  {Data initialization}
  D^.B[0]:=0;
  D^.E[0]:=-1; D^.EMin[0]:=SingleMin; D^.EMax[0]:=SingleMax;
  D^.E[1]:=1;  D^.EMin[1]:=SingleMin; D^.EMax[1]:=SingleMax;
  D^.E[2]:=-1; D^.EMin[2]:=SingleMin; D^.EMax[2]:=SingleMax;
  D^.E[3]:=1;  D^.EMin[3]:=SingleMin; D^.EMax[3]:=SingleMax;
  D^.E[4]:=0;  D^.EMin[4]:=SingleMin; D^.EMax[4]:=SingleMax;
  D^.E[5]:=5;  D^.EMin[5]:=SingleMin; D^.EMax[5]:=SingleMax;
  {Naming of parameters (max. 40 characters) !}
  StrPCopy(D^.NaE[0],'x - axis minimum');
  StrPCopy(D^.NaE[1],'x - axis maximum');
  StrPCopy(D^.NaE[2],'y - axis minimum');
  StrPCopy(D^.NaE[3],'y - axis maximum');
  StrPCopy(D^.NaE[4],'Low-level for output 2');
  StrPCopy(D^.NaE[5],'High-level for output 2');
  StrPCopy(D^.NaB[0],'linear Interpolation');
end;

procedure
GetDialogEnableStruct(D:PDialogEnableStruct;D2:PParameterStruct);export
stdcall;
begin
  {All dialog controls are enabled!}
  D^.AllowE:=$FFFFFFFF;
  D^.AllowB:=$FFFFFFFF;
  D^.AllowI:=$FFFFFFFF;
  D^.AllowD:= 1;
end;

procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);export
stdcall;
begin
  D^.Inputs:=2;            { The block has 2 inputs...}
  D^.Outputs:=2;           {           and 2 outputs !}
  StrPCopy(D^.NameI[0],'Abscissa value (x)');    {Name of 1st input}
  StrPCopy(D^.NameI[1],'Abscissa value (y)');    {Name of 2nd input}
  StrPCopy(D^.NameO[0],'Ordinate (z) ');         {Name of 1st output}
  StrPCopy(D^.NameO[1],'Extrapolation ON/OFF');  {Name of 2nd output}
end;

function CanSimulateDLL(D:PParameterStruct):Integer; export stdcall;
var Datei: Text;
    Dateiname : Array[0..255] of char;
begin
  {Simulation only allowed if specified FWM file exists!}
  StrCopy(Dateiname, D^.D);
  if Pos('*',StrPas(Dateiname))=0 then begin
    Assign(Datei,Dateiname);
    {$I-} Reset(Datei); {$I+}
    if IOResult = 0 then begin
      CanSimulateDLL := 1;
      Close(Datei);
    end else CanSimulateDLL := 0;
  end else CanSimulateDLL:=0;
end;

procedure SimulateDLL(T:Extended;D:PParameterStruct;
                    Inputs:PInputArray;Outputs:POutputArray);export
stdcall;
var ix, iy: integer;
    x0, y0, xp, yp: Extended;
    Extrapolation, Interpolation: boolean;
```

```
begin
  with PUserData(D^.UserDataPtr)^ do begin
    Extrapolation := (Inputs^[1]<D^.E[0]) or (Inputs^[1]>D^.E[1]) or
                     (Inputs^[2]<D^.E[2]) or (Inputs^[2]>D^.E[3]);
    if Extrapolation then
      Outputs^[2]:=D^.E[5]   {Output 2 to High-level}
    else
      Outputs^[2]:=D^.E[4];  {Output 2 to Low-level}
    Interpolation := D^.B[0] = 1;
    {Calculate indices ix, iy of next base point}
    ix := trunc((Inputs^[1] - D^.E[0])/dx) + 1;
    iy := trunc((Inputs^[2] - D^.E[2])/dy) + 1;
    if ix < 1 then ix := 1;
    if iy < 1 then iy := 1;
    if Interpolation then begin
      if ix > nx-1 then ix := nx-1;
      if iy > ny-1 then iy := ny-1;
      x0 := D^.E[0] + (ix-1)*dx; {Reference point, x-Coordinate}
      y0 := D^.E[2] + (iy-1)*dy; {Reference point, y-Coordinate}
      xp := (z[ix+1, iy] - z[ix, iy]) / dx; {Gradient in x-direction}
      yp := (z[ix, iy+1] - z[ix, iy]) / dy; {Gradient in y-direction}
      Outputs^[1] := z[ix, iy] + (Inputs^[1] - x0)*xp + (Inputs^[2]-y0)*yp;
    end else begin
      if ix > nx then ix := nx;
      if iy > ny then iy := ny;
      Outputs^[1] := z[ix, iy];
    end;
  end;
end;

procedure InitSimulationDLL(D:PParameterStruct;Inputs:PInputArray;
                            Outputs:POutputArray);export stdcall;
var i, j: Integer;
    Datei: text;
begin
  {First read parameter file...}
  assign(Datei, D^.D);
  reset(Datei);
  with PUserData(D^.UserDataPtr)^ do begin
    readln(Datei,ny,nx);
    for i:=1 to ny do
      for j:=1 to nx do readln(Datei,z[j, i]);
    dx:=(D^.E[1]-D^.E[0])/(nx-1); {Segment size in x-direction}
    dy:=(D^.E[3]-D^.E[2])/(ny-1); {Segment size in y-direction}
  end;
  close(Datei);
  {...and then execute a normal simulation step}
  SimulateDLL(0, D, Inputs, Outputs);
end;

procedure EndSimulationDLL;export stdcall;
begin
  {not needed}
end;

procedure InitUserData(D: PParameterStruct); export stdcall;
begin
  {Allocate user-data memory}
  GetMem(D^.UserDataPtr, sizeof(TUserData));
end;

procedure DisposeUserData(D: PParameterStruct); export stdcall;
```

```
begin
  {Free user-data memory}
  FreeMem(D^.UserDataPtr, sizeof(TUserData));
end;

function SetInputChar: PChar; export stdcall;
begin
  {Label first input 'x', second 'y'}
  SetInputChar := 'xy';
end;

function SetOutputChar: PChar; export stdcall;
begin
  {Label first output 'z', second 'E'}
  SetOutputChar := 'zE';
end;

procedure IsUserDLL32; export stdcall;
begin
end;

{Function export }
exports
  GetParameterStruct, GetDialogEnableStruct, GetNumberOfInputsOutputs,
  CanSimulateDLL, InitSimulationDLL, SimulateDLL,
  InitUserData, DisposeUserData, EndSimulationDLL,
  SetInputChar, SetOutputChar, IsUserDLL32;

begin
  {Initialzation of the DLL (not necessary).}
end.
```

Pascal listing for example 2

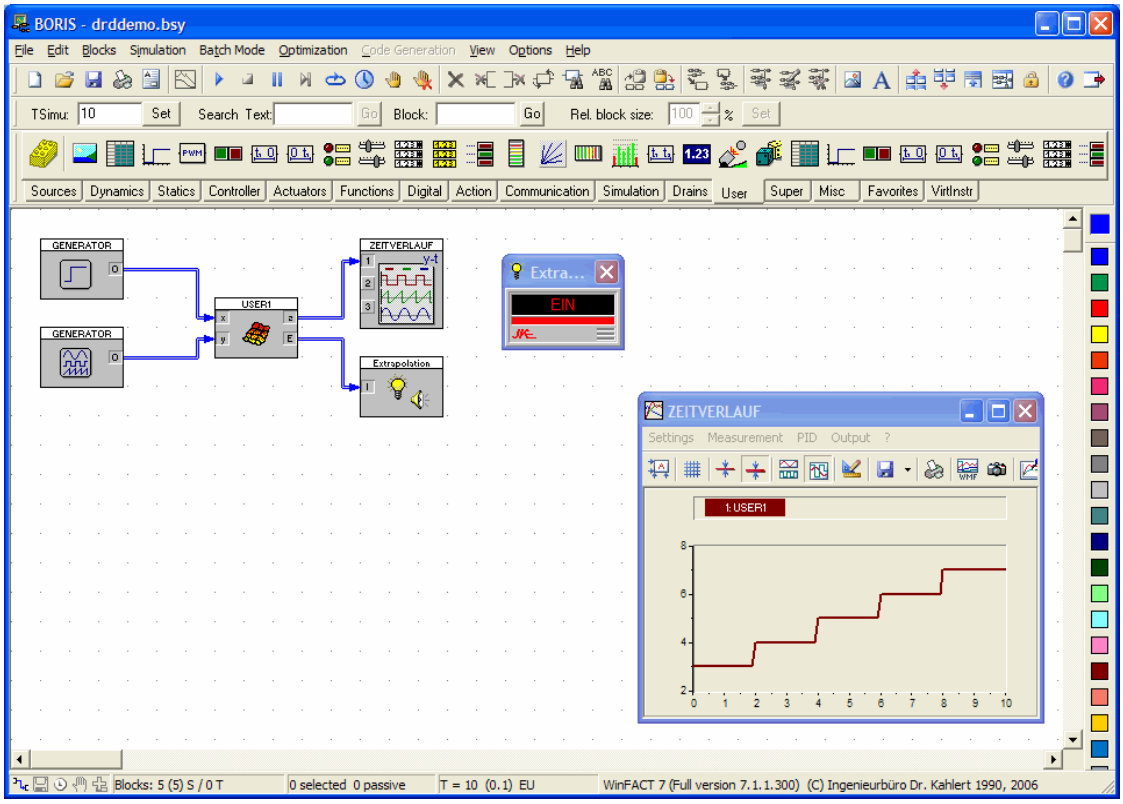
Application sample for 3D-characteristic-map-DLL

A 3D-characteristic map is to be defined for the range $0 \le x \le 4,\ 0 \le y \le 4$ in which the output (z-axis) increases linearly from 1 to 9. The characteristic may be defined by 5 base points for each direction. So the corresponding characteristic matrix has the following form:

$$\underline{M} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}.$$

The FWM-file DRDDEMO.FWM located in the *UserDLLs* directory contains the described matrix. The response of the output has to be simulated for the case of a constant input x of 2 and an input y which increases linearly from 0 to 5. The interpolation shall be inactivated. The screenshot below shows the de-

termined simulation result. The corresponding system file DRDDEMO.BSY is located in the *UserDLLs* directory.

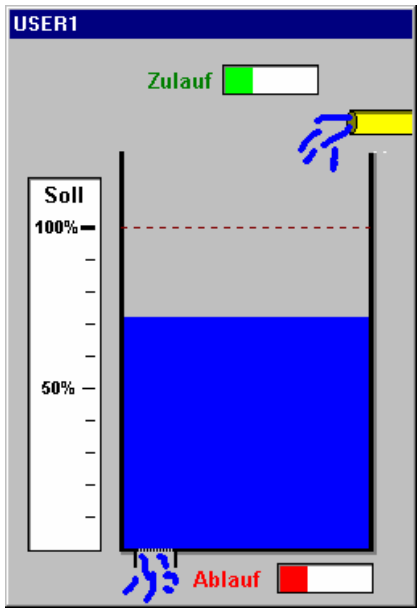


Simulation result

Example 3: Visualization of a liquid level control

We want to design a User-DLL for the visualization of a liquid level control. The output window shall appear on the first simulation start and visualize the system inputs (inflow, outflow, level) as shown in the screenshot below. The output window has to be destroyed if the block itself is deleted within BORIS. It may be supposed that all inputs are between 0 and 1.

The following listing shows the realization of the DLL based only on the Windows API. The compiled file FUELLSTD.DLL is located in the *UserDLLs* directory, the source FUELLSTD.DPR in the *Sources* subdirectory. A more comfortable programming of such block types is supported by visual programming systems like DELPHI, VISUAL BASIC etc; some examples realized in DELPHI 3 (most of them including sources) can be found in the *UserDLLs* resp. *Sources* subdirectory.



Visualization window of User-DLL

```
Library FuellStd;

(************************************************)
(*                                              *)
(*          DLL for visualization of a          *)
(*             liquid level control             *)
(*            (Example 3 of handbook)           *)
(*                                              *)
(************************************************)

uses WinProcs, WinTypes, Windows, SysUtils, Messages;

{$R FUELLBMP}  {RES file with background bitmap}


const
  cxBitmap = 250;  (* Bitmap width *)
  cyBitmap = 350;  (* Bitmap height *)

 {To enable the simulation of several blocks of this type simultaneously,
  we have to save the window handle and the current inputs for all existing
blocks.
  To make the programming task not too complicated, we save these values
  in a static array of dimension 100.}
var
  nWindows: integer;  {current number of windows}
  LevelWindows: array[1..100] of record
    HW: HWnd;                       {window handle}
    VPos, VNeg, Height: extended; {current block inputs: Inflow, outflow,
level}
  end;

type
  PParameterStruct=^TParameterStruct;
  TParameterStruct=packed record
    NuE : Byte;
    NuI : Byte;
```

```
    NuB : Byte;
    E:Array[0..31] of Extended;
    I:Array[0..31] of Integer;
    B:Array[0..31] of Byte;
    D:Array[0..255] of char;
    EMin:Array[0..31] of Extended;
    EMax:Array[0..31] of Extended;
    IMin:Array[0..31] of Integer;
    IMax:Array[0..31] of Integer;
    NaE : Array[0..31,0..40] of char;
    NaI : Array[0..31,0..40] of char;
    NaB : Array[0..31,0..40] of char;
    UserDataPtr: Pointer;
    ParentPtr: Pointer;
    ParentHWnd: HWnd;
    ParentName: PChar;
    UserHWindow: HWnd;
    DataFile: text;
  end;

  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnablestruct=packed record
    AllowE: Longint;
    AllowI: Longint;
    AllowB: Longint;
    AllowD: Byte;
  end;

  PNumberOfInputsOutputs=^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs=packed record
    Inputs :Byte;                          {Inputs}
    Outputs:Byte;                          {Outputs}
    NameI : Array[0..49,0..40] of char;
    NameO : Array[0..49,0..40] of char;
  end;

  PInputArray = ^TInputArray;
  TInputarray = packed array[1..50] of extended;
  POutputArray = ^TOutputArray;
  TOutputarray = packed array[1..50] of extended;


function GetVPos(H: HWnd): extended;
{This function delivers input inflow for the window with handle H}
var i: integer;
begin
  {Scan window list for correct window}
  for i:=1 to nWindows do if LevelWindows[i].HW = H then
    GetVPos := LevelWindows[i].VPos;
end;

function GetVNeg(H: HWnd): extended;
{This function delivers input outflow for the window with handle H}
var i: integer;
begin
  {Scan window list for correct window}
  for i:=1 to nWindows do if LevelWindows[i].HW = H then
    GetVNeg := LevelWindows[i].VNeg;
end;

function GetHeight(H: HWnd): extended;
{This function delivers input level for the window with handle H}
```

```
var i: integer;
begin
  {Scan window list for correct window}
  for i:=1 to nWindows do if LevelWindows[i].HW = H then
    GetHeight := LevelWindows[i].Height;
end;
procedure SetInputs(H: HWnd; VP, VN, Hgt: extended);
{Sets the inputs of the window with handle H}
var i: integer;
begin
  for i:=1 to nWindows do if LevelWindows[i].HW = H then begin
    LevelWindows[i].VPos := VP;     {Inflow}
    LevelWindows[i].VNeg := VN;     {Outflow}
    LevelWindows[i].Height := Hgt;  {Level}
  end;
end;

procedure Paint(DC: HDC; VP, VN, Hgt: extended);
(* Main paint procedure *)
var MemDC: HDC;
    RedBrush, BlueBrush, GreenBrush: HBrush;
    HBM: HBitmap;
    BlockBitmap: TBitmap;
    Rect: TRect;
begin
  MemDC := CreateCompatibleDC(DC);
  HBM := LoadBitmap(HInstance, 'BEHAELTER_BITMAP');
  SelectObject(MemDC, HBM);
  GetObject(HBM, SizeOf(BlockBitmap), @BlockBitmap);
  RedBrush := CreateSolidBrush(RGB(255, 0, 0));
  BlueBrush := CreateSolidBrush(RGB(0, 0, 255));
  GreenBrush := CreateSolidBrush(RGB(0, 255, 0));
  {Inflow}
  with Rect do begin
    Left := 134;
    Right := Left + round(VP*57);
    Top := 17;
    Bottom := 33;
    FillRect(MemDC, Rect, GreenBrush);
  end;
  {Outflow}
  with Rect do begin
    Left := 168;
    Right := Left + round(VN*57);
    Top := 327;
    Bottom := 343;
    FillRect(MemDC, Rect, RedBrush);
  end;
  {Level}
  with Rect do begin
    Left := 71;
    Right := 223;
    Bottom := 316;
    Top := Bottom - round(Hgt*200);
    FillRect(MemDC, Rect, BlueBrush);
  end;
  BitBlt(DC, 0, 0, BlockBitmap.bmwidth, BlockBitmap.bmheight, MemDC, 0, 0,
         SrcCopy);
  DeleteObject(RedBrush);
  DeleteObject(GreenBrush);
  DeleteObject(BlueBrush);
  DeleteDC(MemDC);
```

```
  DeleteObject(HBM);
end;

function WndFunc(Wnd: HWnd; Msg, wParam: word; lParam: longint): longint;
                                                                    stdcall;
{Window function of the output window}
var PaintStruct: TPaintStruct;
    DC: HDC;
    i, Index: integer;
begin
  case Msg of
    wm_Paint: begin
                DC := BeginPaint(Wnd, PaintStruct);
                Paint(DC, GetVPos(Wnd), GetVNeg(Wnd), GetHeight(Wnd));
                EndPaint(Wnd, PaintStruct);
              end;
    wm_Destroy: begin
                  {The window has to be destroyed. So we have to
                   delete it from the window list and move all
                   other entries one index to the top}
                  Index := 0;
                  repeat inc(Index) until LevelWindows[Index].HW = Wnd;
                  for i:=Index+1 to nWindows do
                    LevelWindows[i-1] := LevelWindows[i];
                  dec(nWindows);
                end;
  end;
  WndFunc := DefWindowProc(Wnd, Msg, wParam, lParam);
end;

const
  WindowClass: TWndClass = (
    style: 0;
    lpfnWndProc: @WndFunc;
    cbClsExtra: 0;
    cbWndExtra: 0;
    hInstance: 0;
    hIcon: 0;
    hCursor: 0;
    hbrBackground: 1;
    lpszMenuName: nil;
    lpszClassName: 'LevelWindow');

procedure GetParameterStruct(D:PParameterStruct); export stdcall;
begin
  (* no parameters needed ! *)
  D^.NuE := 0;
  D^.NuI := 0;
  D^.NuB := 0;
end;

procedure GetDialogEnableStruct(D:PDialogEnableStruct;D2:PParameterStruct);
export stdcall;
begin
  (* not necessary *)
end;

procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs); export
stdcall;
begin
  D^.Inputs:=3;    {Our block has three inputs...}
  D^.Outputs:=0;   {             and no outputs  }
```

```
  {Names of inputs}
  StrCopy(D^.NameI[0], 'Inflow');
  StrCopy(D^.NameI[1], 'Outflow');
  StrCopy(D^.NameI[2], 'Level');
end;

procedure InitUserDLL(D:PParameterStruct); export stdcall;
{This procedure is called when the User-DLL is initialized. We need a flag
.that indicates whether the output windows already exists. For this purpose
 we take integer-parameter D^.I[0]. Naturally we could create the window
 already here, but we want it to be shown first when the simulation starts.
 So the programming is a little bit more complicated!}
begin
  D^.I[0] := 0;
end;


procedure DisposeUserDLL(D:PParameterStruct); export stdcall;
{This procedure is called if the block is deleted or replaced by another
DLL.
 In these cases we have to destroy the output window}
begin
  DestroyWindow(D^.UserHWindow);
end;

function CanSimulateDLL(D:PParameterStruct):Integer; export stdcall;
begin
  CanSimulateDLL:=1;  {Simulation always allowed}
end;

procedure InitSimulationDLL(D:PParameterStruct;Inputs:PInputArray;
                            Outputs:POutputArray); export stdcall;
var Rect: TRect;
    x, y: integer;
begin
  with Rect do begin
    Top := 0;
    Bottom := cyBitmap;
    Left := 0;
    Right := cxBitmap;
  end;
  (* Adjust window size to bitmap size! *)
  AdjustWindowRect(Rect, ws_Popup or ws_Caption, false);
  {If no output window exists, it is created here...}
  if D^.I[0] = 0 then begin
    D^.I[0] := 1; {Flag, that window now exists}
    {To avoid overlapping windows, we move each new window
     50 pixels to the upper left}
    x := 350 + nWindows*50;
    y := 70 - nWindows*50;
    {The window is created...}
    D^.UserHWindow := CreateWindowEx(ws_ex_TopMost, 'LevelWindow', '',
                              ws_Popup or ws_Caption or ws_MinimizeBox,
                              x, y, Rect.Right-Rect.Left, Rect.Bottom-
                              Rect.Top, D^.ParentHWnd, 0, HInstance, nil);
    {...displayed...}
    ShowWindow(D^.UserHWindow, sw_Show);
    {...and add to window list!}
    inc(nWindows);
    LevelWindows[nWindows].HW := D^.UserHWindow;
  end;
  {We want to give the window the block's name}
```

```
  SetWindowText(D^.UserHWindow, D^.ParentName);
end;

procedure SimulateDLL(T:Extended;D:PParameterStruct;Inputs:PInputArray;
                      Outputs:POutputArray);export stdcall;
var DC: HDC;
begin
  {Put inputs to window list...}
  SetInputs(D^.UserHWindow, Inputs^[1], Inputs^[2], Inputs^[3]);
  {...and repaint window}
  DC := GetDC(D^.UserHWindow);
  Paint(DC, GetVPos(D^.UserHWindow), GetVNeg(D^.UserHWindow),
        GetHeight(D^.UserHWindow));
  ReleaseDC(D^.UserHWindow, DC);
end;

procedure EndSimulationDLL; export stdcall;
begin
  (* not necessary *)
end;

function SetInputChar: PChar; export stdcall;
begin
  {Label first input with '+', second with '-' and third with 'H'}
  SetInputChar := '+-H';
end;

procedure ShowWindowDLL(D: PParameterStruct); export stdcall;
begin
  {Normalize window}
  ShowWindow(D^.UserHWindow, sw_Normal);
end;

procedure HideWindowDLL(D: PParameterStruct); export stdcall;
begin
  {Minimize window}
  ShowWindow(D^.UserHWindow, sw_Minimize);
end;

procedure IsUserDLL32; export stdcall;
begin
end;

exports
  InitUserDLL, DisposeUserDLL, GetParameterStruct,
  GetDialogEnableStruct, GetNumberOfInputsOutputs,
  CanSimulateDLL, InitSimulationDLL, SimulateDLL,
  EndSimulationDLL, SetInputChar, ShowWindowDLL,
  HideWindowDLL, IsUserDLL32;

begin
  (* Register window class *)
  WindowClass.hInstance := HInstance;
  WindowClass.hIcon := LoadIcon(0, idi_Application);
  WindowClass.hCursor := LoadCursor(0, idc_Arrow);
  RegisterClass(WindowClass);
  nWindows := 0;
end.
```

Pascal listing for example 3

Further examples

The complete installation of WinFACT resp. BORIS includes a lot of sample User-DLLs. Most of them are written in Pascal (Delphi 3). The compiled DLLs are located in the *UserDLLs* subdirectory, the corresponding sources - if delivered - in the *UserDLLs\Sources* subdirectory. Some of these DLLs can be inserted directly from the register *User* of the system block toolbar.

Register User *of the system block toolbar*

Programming in Visual C++

When programming User-DLLs under *Microsoft Visual C++* (at least under the current version 6) a problem occurs concerning the 10 byte floating point data type (*extended* in Pascal resp. *long double* in C) used within the User-DLL interface. Microsoft namely has given the *long double* data type an internal length of 8 byte instead of 10 byte normally used. So if this data type is used directly when programming a User-DLL problems may later occur when BORIS calls specific functions of the User-DLL.

Nevertheless of course Visual C++ can be used for writing BORIS User-DLLs. Therefor only a little work-around is necessary which defines an own "Dummy" data type of 10 byte length; within the User-DLL interface then all *long double* declarations are substituted by this data type. Two small conversion functions are then used to convert the 10 byte data type to an 8 byte floating point of *double* type resp. vice versa.

Within the *UserDLLs\Sources\C++\VisualC++* subdirectory you find a Visual C++ sample project named CPPDEMO.CPP which implements *Example 1: A simple User-DLL* listed earlier using this work-around. The source code for this example is written in such a way that it can be compiled with Visual C++ as well as with the Borland C++ Builder. Therefor the data type *NUMType* was declared that represents the 10 byte floating point type mentioned above. The following listing shows the source code of the corresponding header resp. C file.

```
#ifndef WFNumType
#define WFNumType WFNumType
```

```
// This header file offers a work around to VC++.
//
// Background:
// -----------
// Because the ANSI-C type  'long double' is mapped to 'double'
in VC++ (this is quite correct due to ANSI-C),
// the precision is reduced from 80 to 64 bits. Internally the
floating point unit does perform every
// caclulation with the complete 80 bits and BORIS also works
with the highest prescision.

// -->  So there is no way in VC++ to define a floatingpoint
data type sized 10 bytes !!

// Solution:
// ---------
// We define a new data type 'NUMType' which is has
sizeof(NUMType)==10 and write
// conversion for the type 'double'.
//
// Rules of usage:
// ---------------
//   1.) Replace every location of 'long double' by the new
//       data type 'NUMType' !
//   2.) Convert 'NUMType' to 'double' at entry point of every
//       dll function and vice versa
//       at the exit point.

// Example:
//
//  long double f(long double a)
//  {
//      a+=1;
//      return a;
//  }
//
// has to be changed to:
//
//  NUMType f(NUMType a)
//  {
//      double d=NUMType2double(a);
//      d+=1;
//      return double2NUMType(d);
//  }
//
//  Of cause, this could be written a little bit shorter, but
here clearness is the main goal!
//


// This header file checks if you use MC-VC++ and resets
everything back (take a look at the
// defines below), so that it won't hurt using it with another
compiler.
```

```
#ifndef  _MSC_VER

#define NUMType long double
#define NUMType2double(a) a
#define double2NUMType(a) a

#else
#define MS_LONG_DOUBLE_PROBLEM


typedef struct{
    char d[10];
}NUMType;



inline double NUMType2double(const NUMType mem)
{
    double d;
    _asm fld TBYTE PTR mem
    _asm fstp QWORD PTR d
    return d;
}

inline NUMType double2NUMType(const double d)
{
    NUMType mem;
    _asm fld QWORD PTR d
    _asm fstp TBYTE PTR mem
    return mem;
}


#endif

#endif
```

Listing of WFNumType.h

```
#ifndef cppdemo
#define cppdemo cppdemo

#pragma pack(1)

/***********************************************
 *                                             *
 *       simple demo DLL for BORIS             *
 *       (Example 1 of the manual)             *
 *                                             *
 ***********************************************/

#ifdef _MSC_VER
```

```
#define WFCallingConvention  __stdcall
#elif   __BORLANDC__
#define WFCallingConvention  _stdcall _export
#endif


typedef struct{
    char    NuE;
    char    NuI;
    char    NuB;
    NUMType E[32];        // replacement for long double !!!
    int     I[32];
    char    B[32];
    char    D[256];
    NUMType EMin[32];     // replacement for long double !!!
    NUMType EMax[32];     // replacement for long double !!!
    int     IMin[32];
    int IMax[32];
    char NaE[32][41];
    char NaI[32][41];
    char NaB[32][41];
   /* the rest of the type TParameterStruct is not accessed,
                   so we can simply stop here */
  } TParameterStruct, *PParameterStruct;

typedef struct {
    long AllowE;
    long AllowI;
    long AllowB;
    char AllowD;
  } TDialogEnableStruct, *PDialogEnableStruct;


typedef struct {
    char Inputs;                //number of inputs
    char Outputs;               //number of outputs
    char NameI[50][41];
    char NameO[50][41];
  } TNumberOfInputsOutputs, *PNumberOfInputsOutputs;

typedef NUMType TInputArray[10]; //replacement for long double
!!!
typedef NUMType TOutputArray[10];//replacement for long double
!!!

#pragma pack()

extern "C"
{
 void WFCallingConvention
   GetParameterStruct(PParameterStruct);
 void WFCallingConvention
   GetDialogEnableStruct(PDialogEnableStruct,PParameterStruct);
 void  WFCallingConvention
```

```
   GetNumberOfInputsOutputs(PNumberOfInputsOutputs D);
 int   WFCallingConvention
   CanSimulateDLL(PParameterStruct D);
 void  WFCallingConvention
   InitSimulationDLL(PParameterStruct D, TInputArray Inputs,
                     TOutputArray Outputs);
 void  WFCallingConvention
   SimulateDLL(NUMType T,PParameterStruct D,
               TInputArray Inputs, TOutputArray Outputs);
 char* WFCallingConvention SetInputChar(void);
 void  WFCallingConvention EndSimulationDLL(void);
 void  WFCallingConvention IsUserDLL32(void);
}

#endif
```

Listing of CPPDemo.h

```
#include <string.h>
#include "WFNUMType.h"
#include "cppdemo.h"

void WFCallingConvention GetParameterStruct(PParameterStruct D)
{
  D->NuE=3;
  D->NuI=0;
  D->NuB=1;
  strncpy(D->NaE[0],"Gain k1 if control input < 0",39);
  strncpy(D->NaE[1],"Gain k2 if control input = 0",39);
  strncpy(D->NaE[2],"Gain k3 if control input > 0",39);
  strncpy(D->NaB[0],"inverse output signal",39);
  D->E[0]=double2NUMType(1);
  D->E[1]=double2NUMType(5);
  D->E[2]=double2NUMType(10);
  D->B[0]=0;
  D->EMin[0]=double2NUMType(0.0);
  D->EMax[0]=double2NUMType(100000);
  D->EMin[1]=double2NUMType(0.0);
  D->EMax[1]=double2NUMType(100000);
  D->EMin[2]=double2NUMType(0.0);
  D->EMax[2]=double2NUMType(100000);
}


void WFCallingConvention
  GetDialogEnableStruct(PDialogEnableStruct D,
                        PParameterStruct D2)
{
//All dialog elements available
  D->AllowE=0xFFFFFFFF;
  D->AllowI=0xFFFFFFFF;
  D->AllowB=0xFFFFFFFF;
}
```

```
void  WFCallingConvention
  GetNumberOfInputsOutputs(PNumberOfInputsOutputs D)
{ D->Inputs=2;
  D->Outputs=1;
  strncpy(D->NameI[0],"control input",39);
  strncpy(D->NameI[1],"data input",39);
  strncpy(D->NameO[0],"output",39);
}

int  WFCallingConvention CanSimulateDLL(PParameterStruct D)
{ //Simulation always allowed!
  return 1;
}

void WFCallingConvention
  SimulateDLL(NUMType T,PParameterStruct D,
              TInputArray Inputs,TOutputArray Outputs)
{
  double I0=NUMType2double(Inputs[0]);
  if (I0<0) //Input 1 negative
    Outputs[0] = double2NUMType(NUMType2double(D->E[0]) *
                                NUMType2double(Inputs[1]));
  else if (I0 == 0) //Input 1 zero
    Outputs[0] = double2NUMType(NUMType2double(D->E[1]) *
                                NUMType2double(Inputs[1]));
  else //Input 1 positive
    Outputs[0] = double2NUMType(NUMType2double(D->E[2]) *
                                NUMType2double(Inputs[1]));
  if (D->B[0] == 1) //output should be inverted
    Outputs[0] = double2NUMType(-NUMType2double(Outputs[0]));
}

void WFCallingConvention
  InitSimulationDLL(PParameterStruct D,
                    TInputArray Inputs,TOutputArray Outputs)
{/*Cause the initial step can be evaluated exactly like all
other
     steps, we can call SimulateDLL(...) here.*/
 SimulateDLL(double2NUMType(0), D, Inputs, Outputs);
}

void WFCallingConvention EndSimulationDLL(void)
{//is not needed
}

char* WFCallingConvention SetInputChar(void)
{
  return "CD";
}

void WFCallingConvention IsUserDLL32(void)
{
}
```

Listing of CPPDemo.cpp

User-defined block bitmaps

Instead of using the standard block bitmap for User-DLLs BORIS supports user-defined bitmaps (see also chapter *Working with superblocks*). These block bitmaps are created as BMP files (e. g. with PaintBrush) and must have the same name as the DLL, however with the file extension replaced by BMP. The BMP file must be located in the same directory as the User-DLL. A user bitmap for a DLL called MOTOR.DLL therefore must be called MOTOR.BMP. The recommended bitmap size is 48x42 pixels; bitmaps of other size are stretched automatically.

In addition to the user-defined bitmap for screen output another (recommended b/w) bitmap can be defined for the printer output. This printer bitmap must have a file name extended by *_P*. If the User-DLL is to appear on the register *User* of the system block toolbar, an additional bitmap (18x18 pixels) with extension *_T* has to be created.

Example:	with MOTOR.DLL	corresponds printer bitmap	MOTOR_P.BMP
		and toolbar bitmap	MOTOR_T.BMP

Designing PID-controllers

In addition to the pure simulation of (control) systems BORIS provides the comfortable design of PID-controllers*. Two different ways are at the user's disposal:

- Design of PID-controllers based on design rules called "*rules of thumb*". This method is described in the following chapter.
- Design of PID-controllers by *numerical parameter optimization* using the parameter optimization module of BORIS (see chapter *Numerical optimization of systems*).

* For a more flexible and comfortable design of PID-controllers we recommend our *PID Design Center* which can be purchased separately.

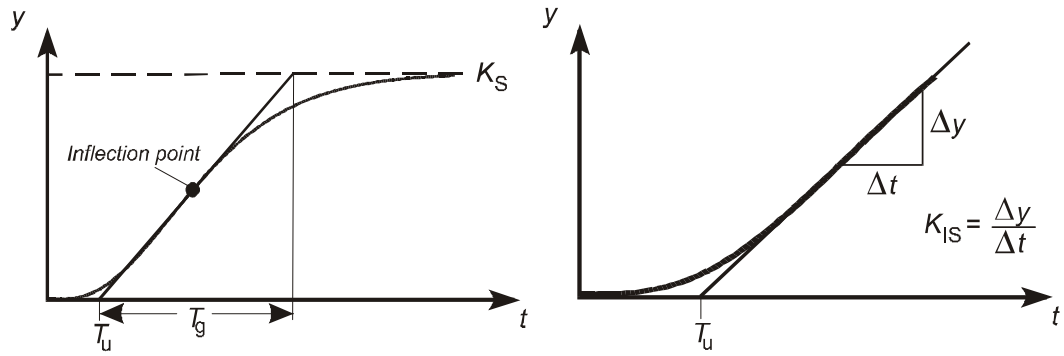
PID-Design by "rules of thumb"

Characteristic values of the plant

The basis of the PID-design by rules of thumb realized in BORIS is the *step response* of the plant, i. e. the response of the plant to a step with amplitude 1 at its input. This step response delivers the *delay time* T_u and the *compensation time* T_g of the plant (in case of a plant with compensation) resp. the delay time alone in case of a plant without compensation. Combined with the *gain* of the plant K_S resp. K_{IS} these parameters represent the base of the controller design (see figure below, see also [7]). Normally these design rules may only be applied to non-oscillating plants.

Note: Design rules for "non-standard plants" are just under construction and will follow in a later version. Further design rules are offered via the *PID Design Center* add-on for BORIS.

The evaluation of the characteristic values of the plant can be executed manually or automatically. As a first step the step response of the plant - received by measurement, by simulation or read from an external file - has to be passed into a time response block.



Characteristic values of a plant with compensation (left) resp. without compensation (right)

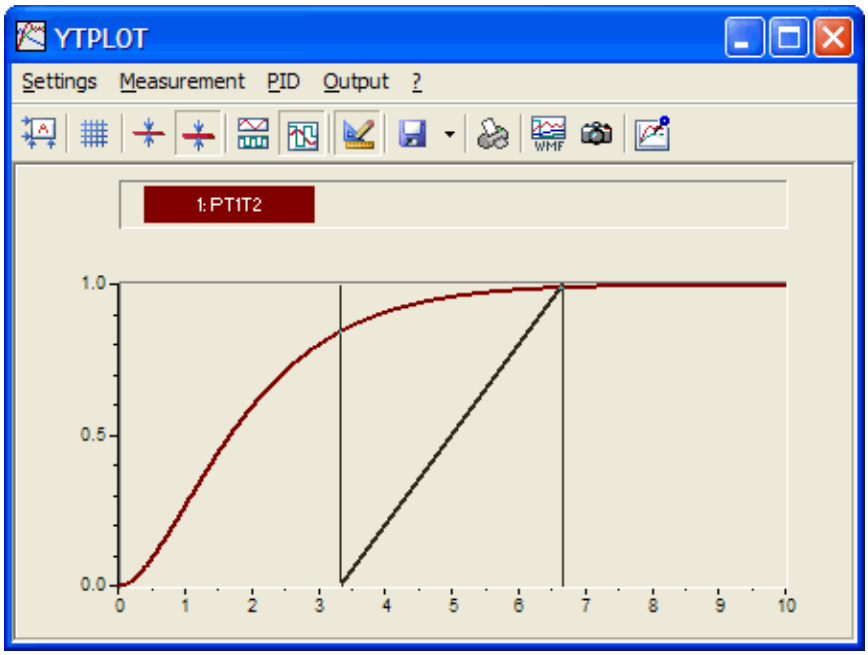
If the evaluation shall be executed manually, the measurement function of the y-t-plotter window has to be activated. Moreover the tangent function has to be activated by the MEASUREMENT | SHOW TANGENT menu option. This effects that in addition to the measurement cursors a tangent appears which can also be moved with the mouse:

- First the tangent is "clipped" to the cursors and moved simultaneously with them.
- By holding the <Shift> or <Ctrl> key the tangent can be "untied" and moved independently from the cursors.

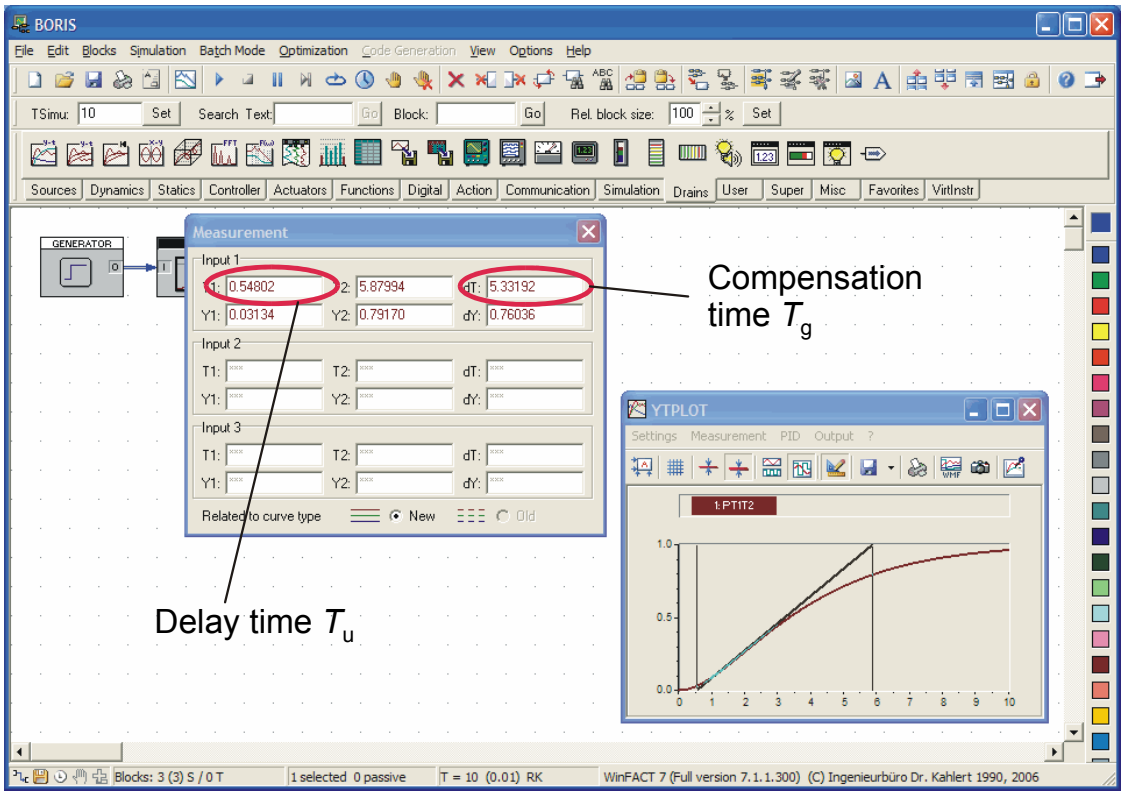
In this way the tangent can be moved to the turning-point of the step response. After that the measurement dialog delivers the values for delay and compensation time (see figure below).

To evaluate the characteristic values automatically, choose the PID | PID-DESIGN.. menu option. This option leads to a combined analysis-/design-dialog. The upper group box titled *Plant parameters* is substantial for the characteristic values evaluation.

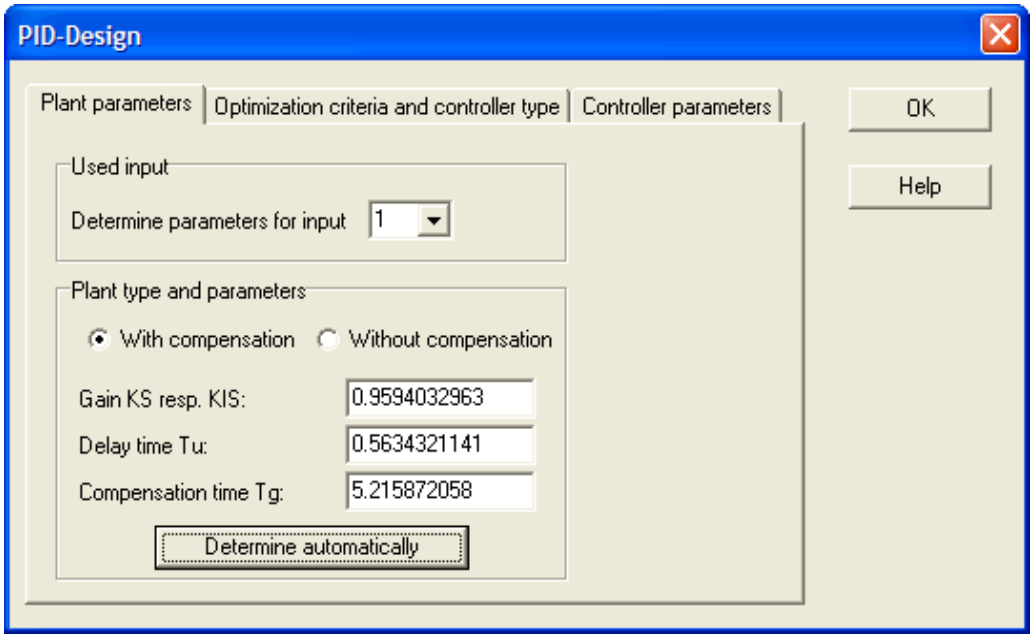
The combobox *Determine parameters for input* specifies the y-t-plotter's input the design is to be executed for. The evaluation is started by clicking the *Determine automatically* button. After that BORIS first determines the plant type (with/without compensation) and then the characteristic values themselves. If desired the values can be modified manually.



Output window of the y-t-plotter after activating the measurement and tangent function



Manual evaluation of delay and compensation time



Analysis- and design-dialog (after executed evaluation of the characteristic values)

Evaluating the controller parameters

After having finished the evaluation of the characteristic values of the plant the controller design can be started. If not already done, the Analysis/Design-dialog has to be called by the PID | PID-DESIGN... option. The design rules are those of Chien, Hrones and Reswick (see tables below). Before the parameter evaluation is executed, it can be specified whether an overshoot of the closed loop system is acceptable and whether the priority should be set on good disturbance or good reference input behavior. These settings are selected within the *Reference/Disturbance* resp. *Controller type* group box.

The parameter evaluation is started by clicking the *Calculate parameters* button. The resulting parameters are displayed in the corresponding edit fields within the *Controller parameters* group box. They can be modified any time if desired. If a PID block already exists, the parameters can be transferred to this block by clicking the *Transfer parameters* button.

Controller	With overshoot		No overshoot	
	Disturbance	**Reference**	**Disturbance**	**Reference**
P	$K_R = 0.71\frac{1}{K_S}\frac{T_g}{T_u}$	$K_R = 0.71\frac{1}{K_S}\frac{T_g}{T_u}$	$K_R = 0.3\frac{1}{K_S}\frac{T_g}{T_u}$	$K_R = 0.3\frac{1}{K_S}\frac{T_g}{T_u}$
PI	$K_R = 0.71\frac{1}{K_S}\frac{T_g}{T_u}$ $T_N = 2.3T_u$	$K_R = 0.59\frac{1}{K_S}\frac{T_g}{T_u}$ $T_N = T_g$	$K_R = 0.59\frac{1}{K_S}\frac{T_g}{T_u}$ $T_N = 4T_u$	$K_R = 0.34\frac{1}{K_S}\frac{T_g}{T_u}$ $T_N = 1.2T_g$
PID	$K_R = 1.2\frac{1}{K_S}\frac{T_g}{T_u}$ $T_N = 2T_u$ $T_V = 0.42T_u$	$K_R = 0.95\frac{1}{K_S}\frac{T_g}{T_u}$ $T_N = 1.35T_g$ $T_V = 0.47T_u$	$K_R = 0.95\frac{1}{K_S}\frac{T_g}{T_u}$ $T_N = 2.4T_u$ $T_V = 0.42T_u$	$K_R = 0.59\frac{1}{K_S}\frac{T_g}{T_u}$ $T_N = 1.35T_g$ $T_V = 0.5T_u$

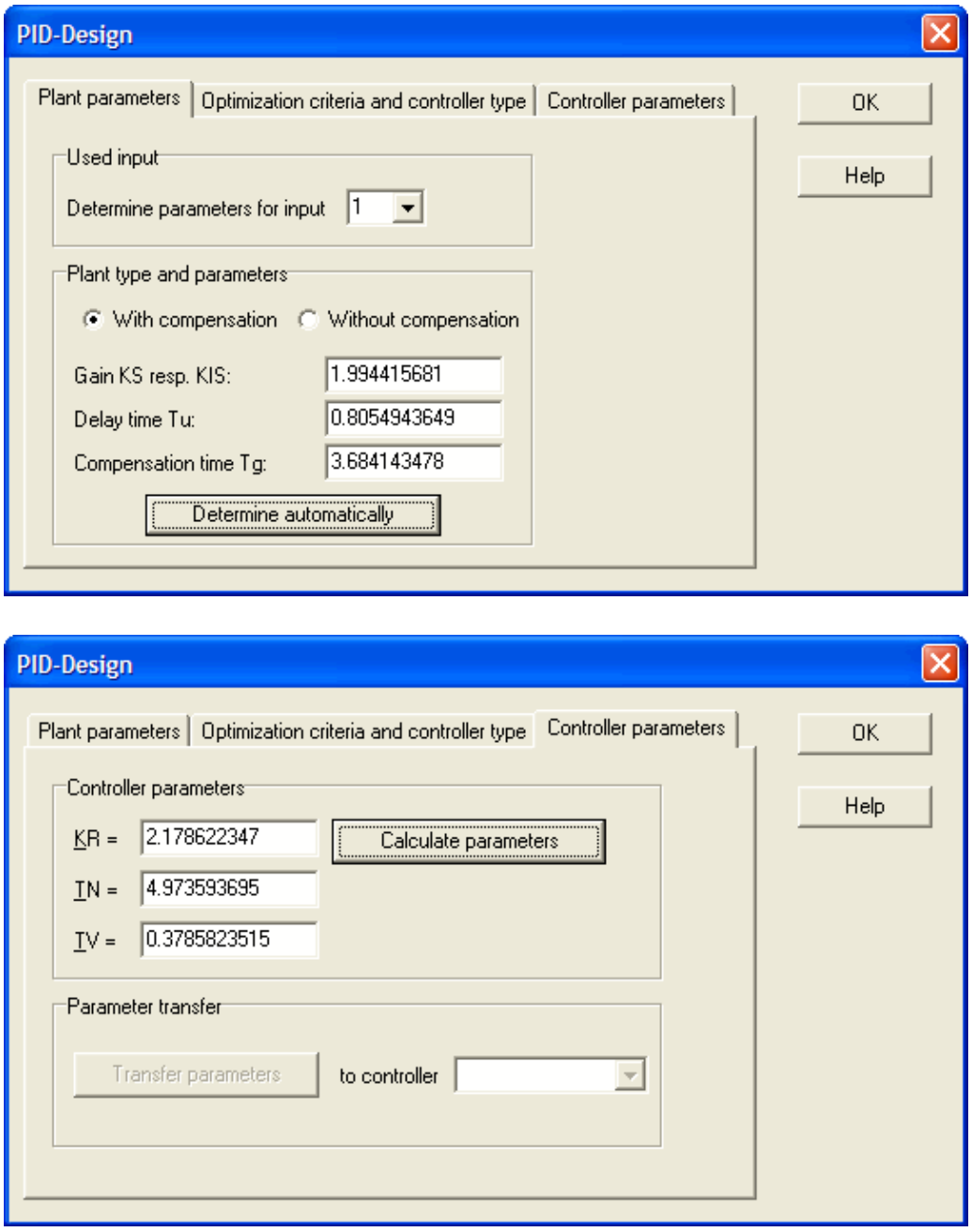
Con-troller	With overshoot		No overshoot	
	Disturbance	**Reference**	**Disturbance**	**Reference**
P	$K_R = 0.71\frac{1}{K_{IS}}\frac{1}{T_u}$	$K_R = 0.71\frac{1}{K_{IS}}\frac{1}{T_u}$	$K_R = 0.3\frac{1}{K_{IS}}\frac{1}{T_u}$	$K_R = 0.3\frac{1}{K_{IS}}\frac{1}{T_u}$
PI	$K_R = 0.71\frac{1}{K_{IS}}\frac{1}{T_u}$ $T_N = 2.3T_u$	$K_R = 0.59\frac{1}{K_{IS}}\frac{1}{T_u}$	$K_R = 0.59\frac{1}{K_{IS}}\frac{1}{T_u}$ $T_N = 4T_u$	$K_R = 0.34\frac{1}{K_{IS}}\frac{1}{T_u}$
PID	$K_R = 1.2\frac{1}{K_{IS}}\frac{1}{T_u}$ $T_N = 2T_u$ $T_V = 0.42T_u$	$K_R = 0.95\frac{1}{K_{IS}}\frac{1}{T_u}$ $T_V = 0.47T_u$	$K_R = 0.95\frac{1}{K_{IS}}\frac{1}{T_u}$ $T_N = 2.4T_u$ $T_V = 0.42T_u$	$K_R = 0.59\frac{1}{K_{IS}}\frac{1}{T_u}$ $T_V = 0.5T_u$

Design rules of Chien/Hrones und Reswick for plants with compensation (top) resp. without compensation (bottom)

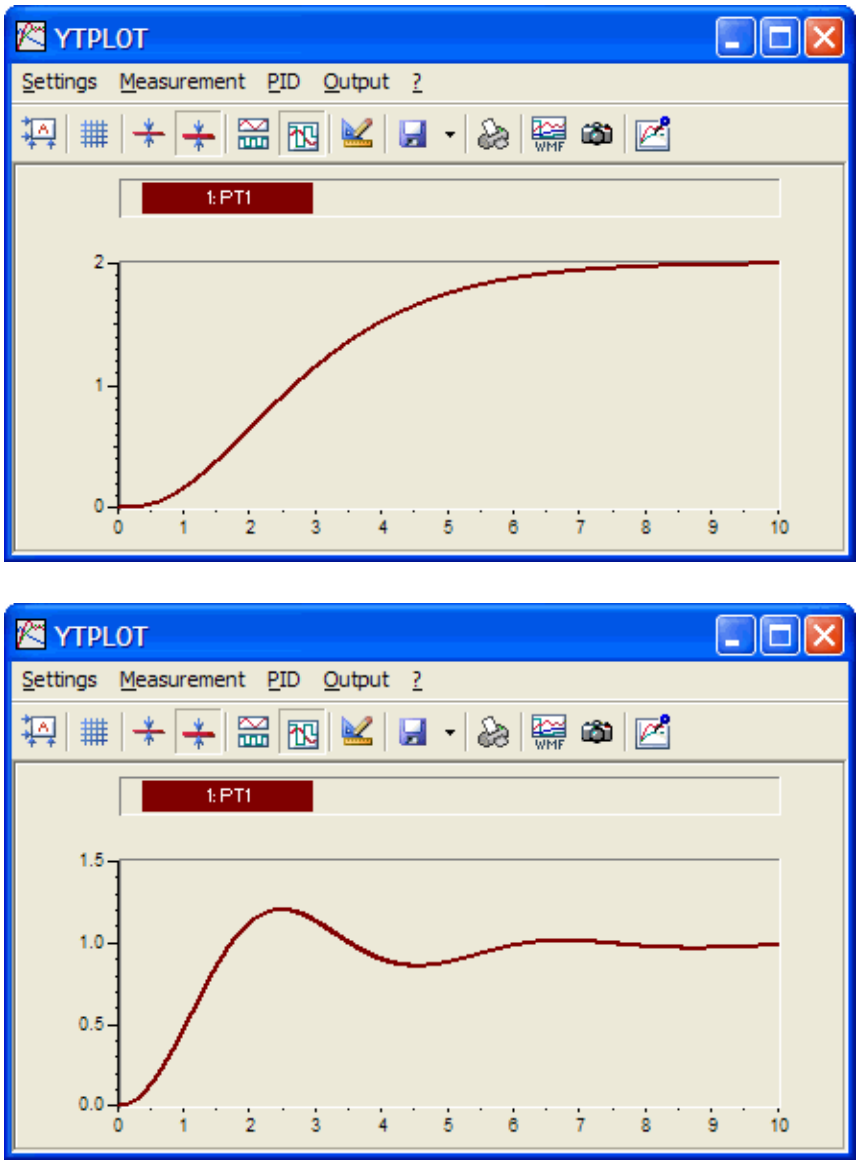
Example

A PID controller has to be designed for a PT_3-plant with a triple time constant of 1 and a gain of 2. An overshoot of the closed loop system can be accepted.

The following screenshots show the PID design dialog after analyzing the plant and designing the controller as well as the step responses of the plant and the closed loop system.



Plant analysis (top) and controller design (bottom) for PT_3 sample system



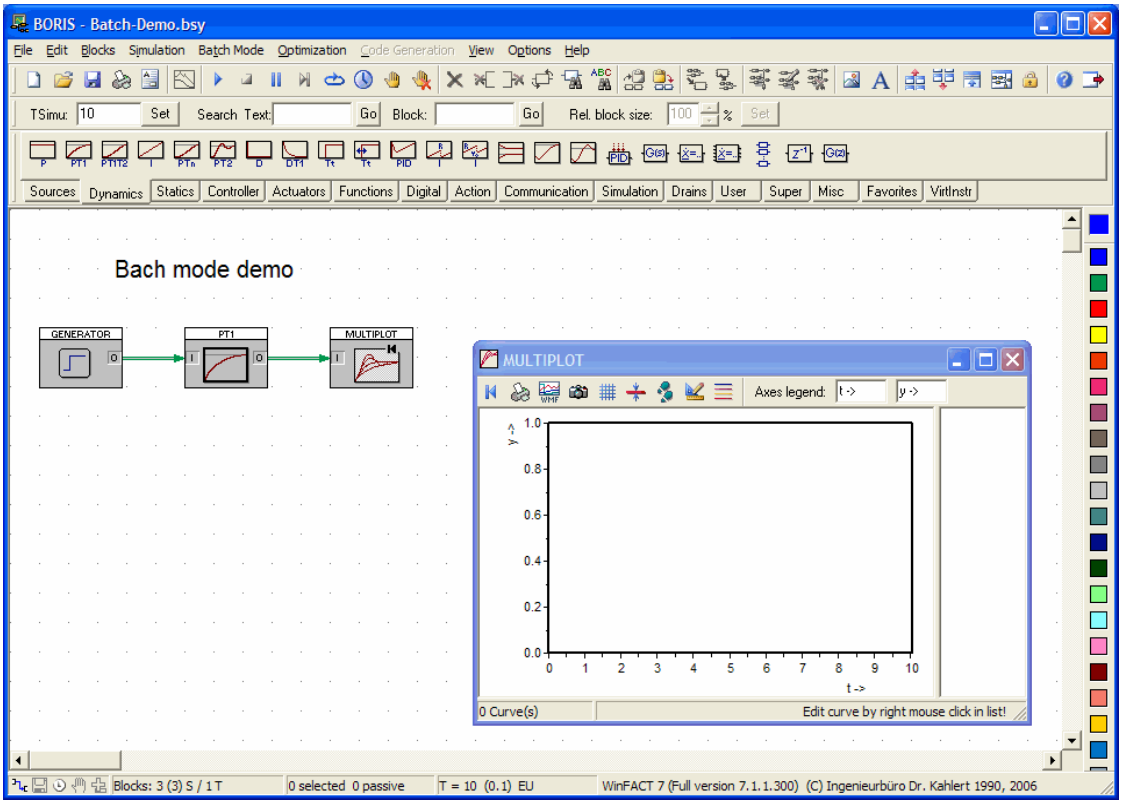
Step responses of plant (top) resp. closed loop system (bottom)

Note: The example is located in the examples directory under PID_DESI.BSY.

Batch mode simulation

Besides single simulations BORIS allows the automatical execution of complete series of simulations, e. g. for the examination of the influence of specific parameters on the system response. This so-called batch mode is based on the principle of export parameters, i. e. all parameters activated for export can be modified within a batch run.

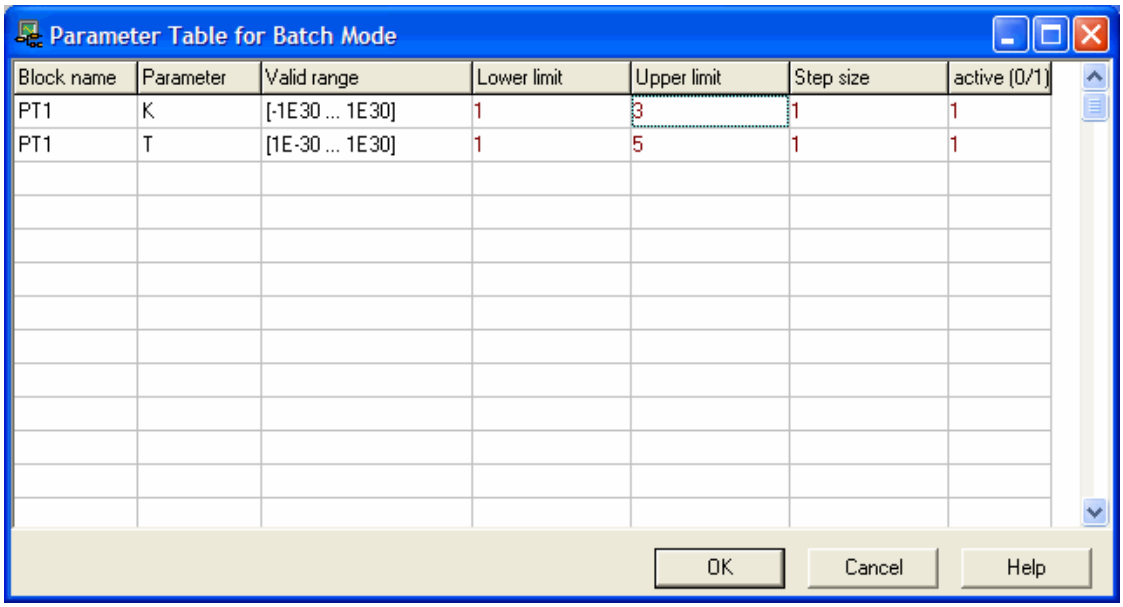
In the following the principle of the batch mode is illustrated by a simple example. Therefor the step response of a PT_1-element is to be examined dependent on its gain *K* and its time constant *T*.



Demo structure for batch mode simulation (file BATCH-DEMO.BSY) before starting the batch run

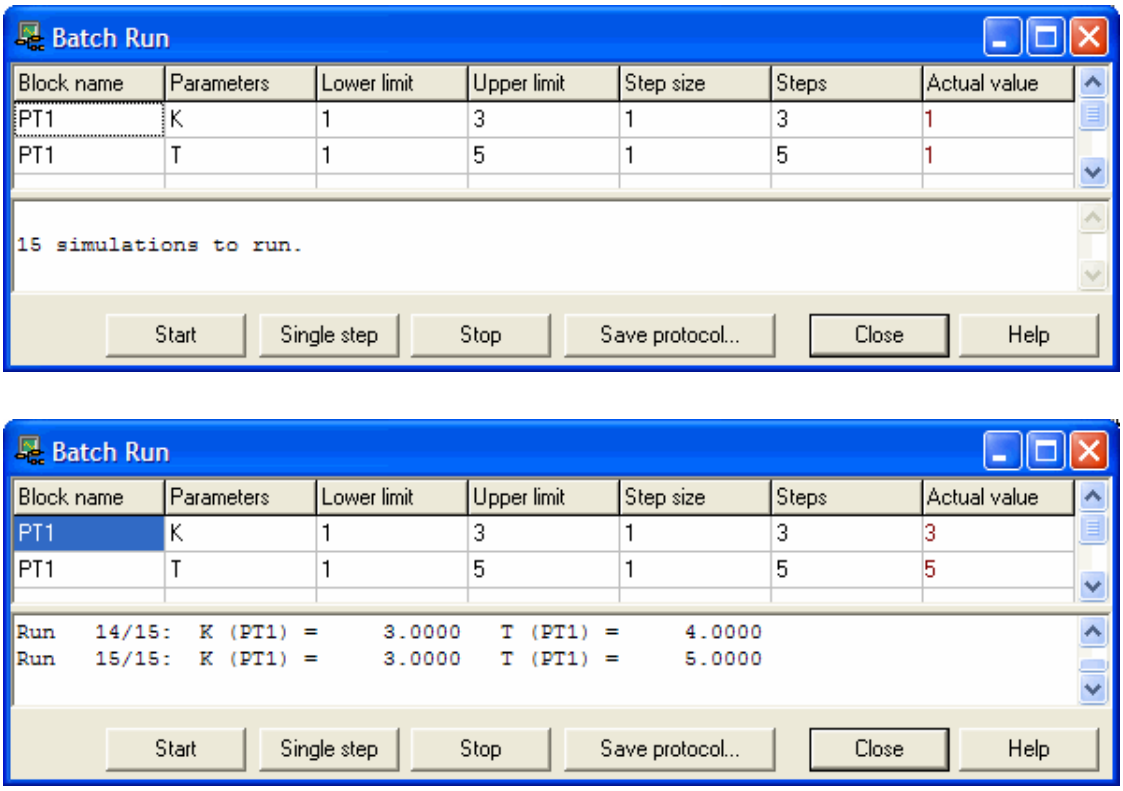
To configure the batch mode proceed as follows (you can find the already configured system in your WinFACT examples directory under BATCH-DEMO.BSY):

- Create the system structure consisting of a generator, a PT_1-element and a multi-plotter as shown above.
- Activate the parameters of the PT_1-element as export parameters.
- Now select the BATCH MODE / PARAMETER TABLE... menu option to get into the batch mode parameter configuration dialog. For the gain *K* of the PT_1-element we want to simulate the values 1, 2 and 3 and for the time constant *T* the values 1, 2, 3, 4 and 5s. Specify the parameters as shown in the screenshot below. Don't forget to input a 1 into the last column of both rows of the table to activate the modification of the parameters.



Configuring the batch mode parameters

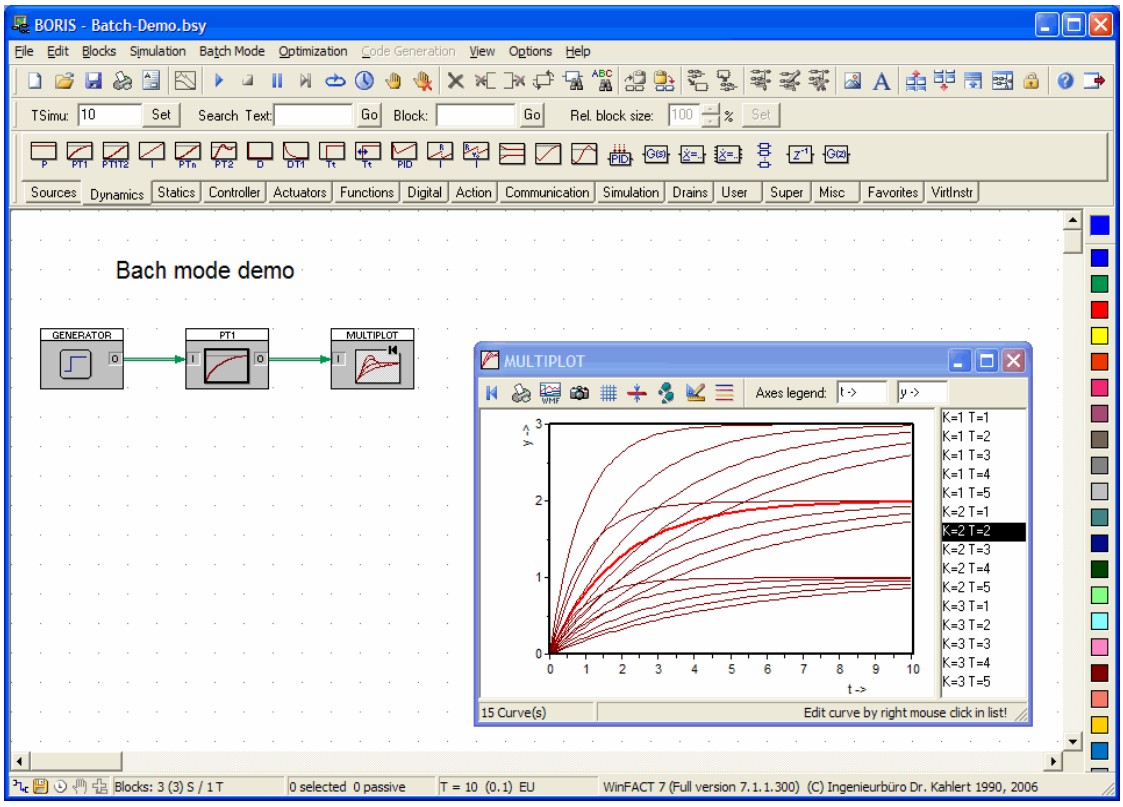
- Now close the dialog and choose the BATCH MODE / START BATCH RUN... menu option. The control dialog for the batch run itself appears. Start the batch run by pressing the *Start* button. Within the dialog each simulation can be supervised; if desired the batch run can be terminated via the *Stop* button. The *Single step* button allows a stepwise execution of the batch run. The *Save protocol* button stores the complete batch protocol to a file.



Batch dialog before starting the batch run (top) and after batch run is complete (bottom)

➢ Now close the batch dialog and have a look at the batch results presented in the multi-plotter. It contains one step response for each simulation (i. e. for each combination of block parameters); all curves are listed at the right border of the plotter. By clicking at a list item the corresponding curve is automatically highlighted.

Thus the multi-plotter is especially recommended for the presentation of batch run results. If the results of the single simulations of a batch run additionally or alternatively should be saved in files, the FILEOUTPUT resp. TABFILEOUTPUT block types can be used. These can be configured in such a way that for each simulation a new file is created that contains information about the current parameter values used for that simulation in its file header. For details please refer to the description of the corresponding block types in chapter *The BORIS system block library*.

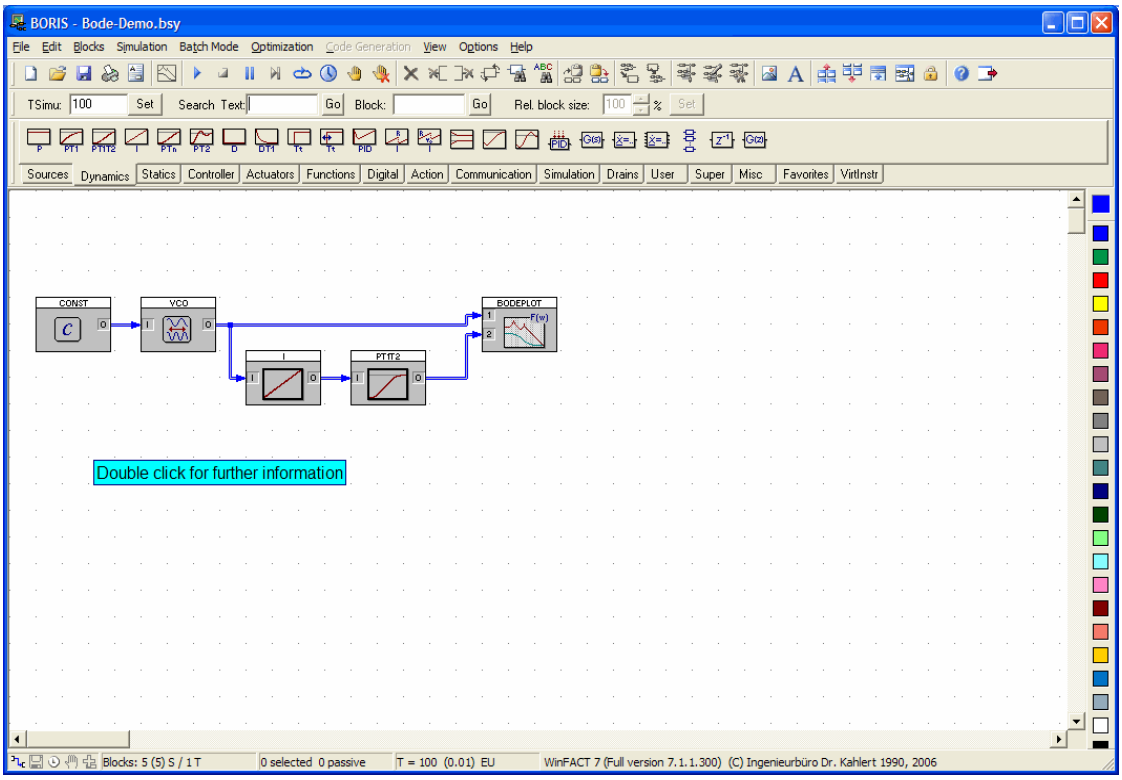


Batch run results presented in the multi-plotter (here step response for $K = T = 2$ selected)

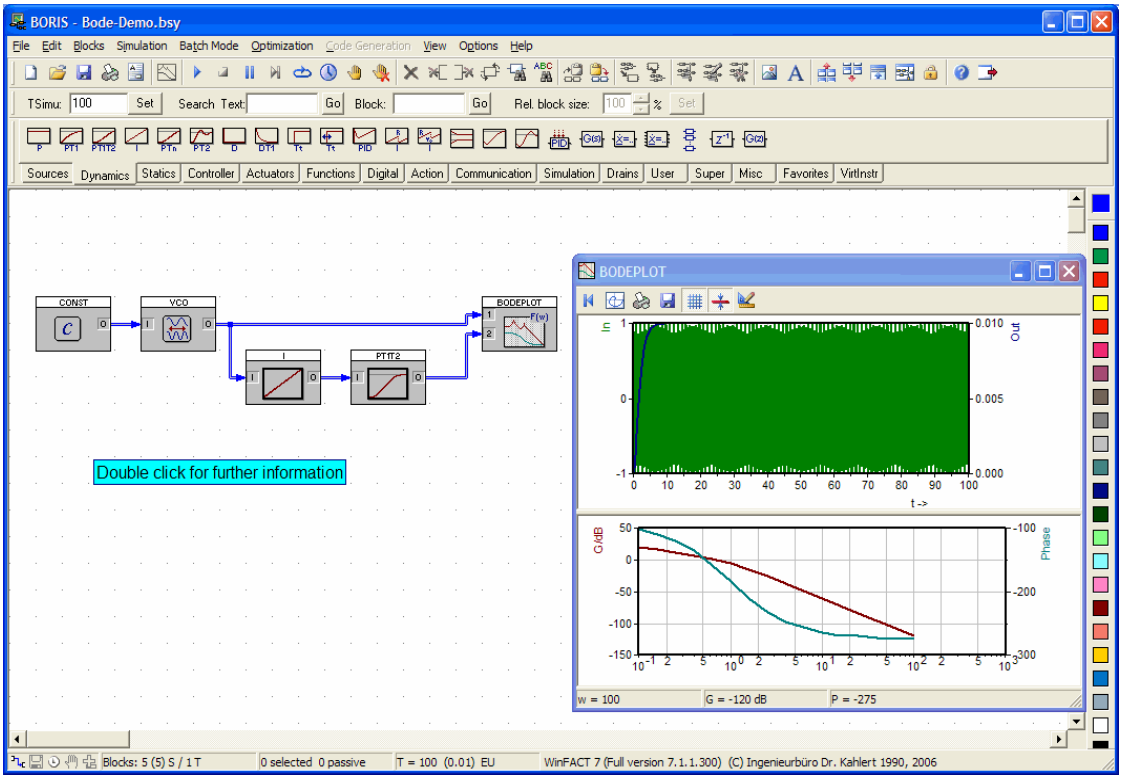
Frequency response determination

Combined with the frequency response plotter (block type BODEPLOT) the batch mode may be used to determine the frequency response of a system in form of the Bode or Nyquist plot. Therefor first under usage of CONST block and a VCO (VCO block type) a *wobble generator* is created and then the system to be examined is stimulated with an increasing frequency within the batch mode. The following screenshot illustrates this principle based on the demo file BODE-DEMO.BSY located in the examples directory.

The input signal for the system to be examined is connected to input 1 of the BODEPLOT block, the output signal to input 2. The VCO should be configured to the *exponential* operating mode. The constant of the CONST block (i. e. its output value) then specifies the current frequency and is modified via the batch mode. If e. g. the frequency response is to be determined for a frequency range from $\omega = 0.1$ ($= 10^{-1}$) to $\omega = 100$ ($= 10^{2}$), the constant has to be modified between -1 and 2 (e. g. with a step size of 0.2). The screenshot below shows the result determined with the demo file BODE-DEMO.BSY.



System structure for determining the frequency response of a system (IT_2-system here)

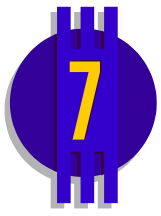


Results from BODE-DEMO.BSY

While the determination of the amplitude response (gain) is relative uncritically, for an exact determination of the phase response the settings for the simulation time and step size have to be chosen carefully. The simulation time (duration) should be long enough so that even for the smallest frequency at least two periods are included. On the other hand the simulation step size should be selected small enough so that the highest frequency is yet simulated with an adequate resolution. Of course also the characteristic values of the examined system (time constants and/or eigenfrequencies) are of importance. In case of an unfavourable selection of simulation time and/or step size the phase response might be determined unexactly.

In case of very time-consuming frequency response calculations it might be useful to adapt the simulation step size automatically with help of the VCO; for this purpose the VCO block offers a corresponding option. If this option is activated the simulation step size of BORIS is changed dependent on the current VCO frequency so that lower frequencies are simulated with greater step size to save computation time. Furthermore the parameter dialog of the frequency response plotter offers an option to terminate a simulation automatically after the output signal consists of a sufficient number of periods so that especially in case of high frequencies computation time can be saved. Both options however have to be used carefully!

Getting frequency responses



Besides the representation of simulation results in the time domain BORIS allows to determine the *frequency response* of a linear system. This frequency response can be represented in following ways:

- Bode plot (amplitude and phase response)
- Nyquist plot (real and imaginary part)
- *s*-Transfer function (numerator and denominator polynom)

To get the frequency of a linear substructure first the corresponding system blocks have to be selected. Afterwards the frequency response can be determined via the EDIT | EVALUATE FREQUENCY RESPONSE menu option or the  button. It has to be noted that always a *serial connection* of the selected blocks

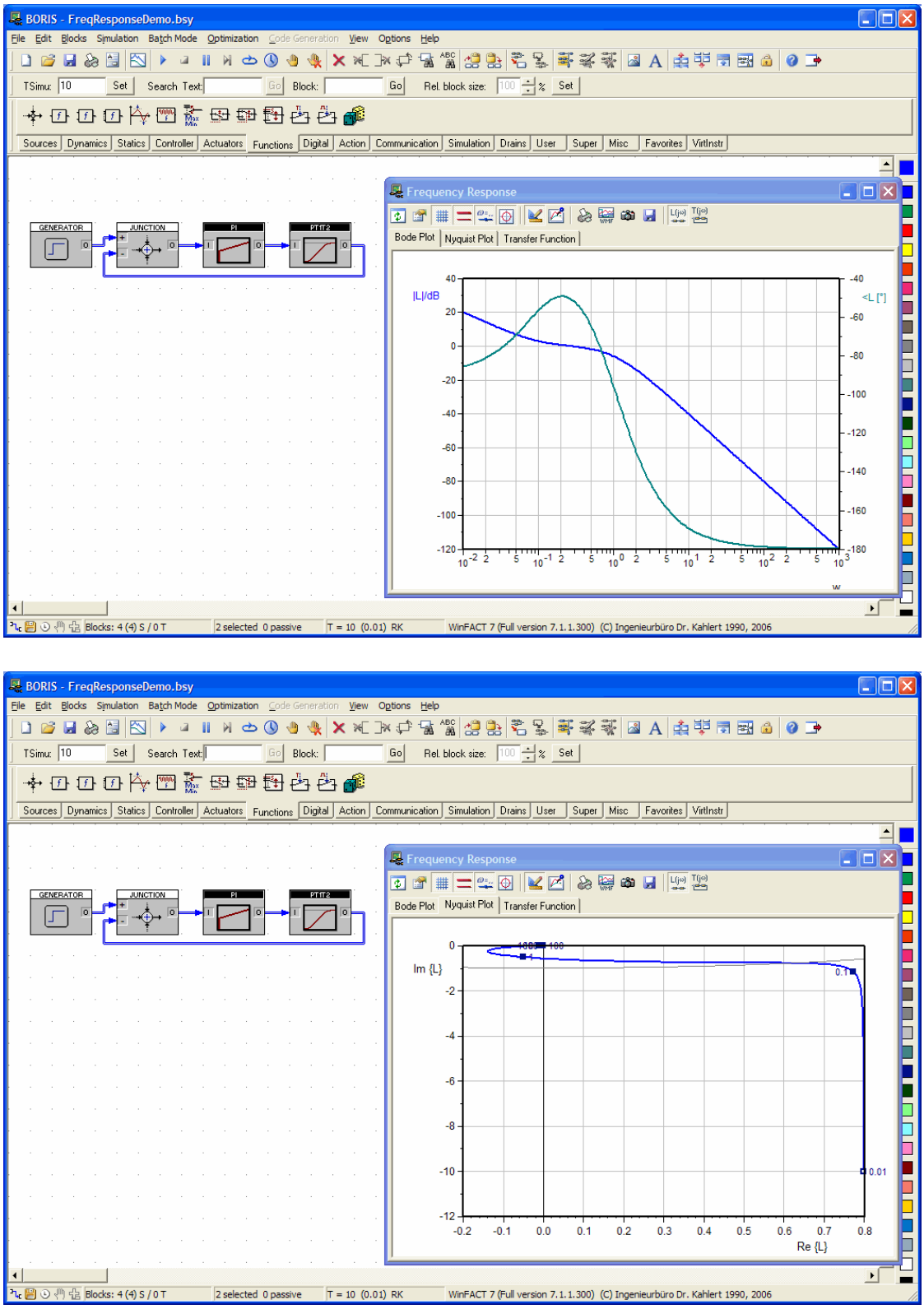
is supposed, independent from the actual combination. That means e. g. that also in case of two *parallel* blocks the frequency response of their serial connection is calculated! If any of the selected blocks is a non-linear one the frequency response calculation is disabled.

The frequency response is displayed in a separate window which allows the selection between the various representation modes via different palettes. The screenshot below shows Bode and Nyquist plot of an open-loop system containing a PI-controller and a PT1T2-plant.

Besides the frequency response of an open-loop (i. e. the serial connection of the selected blocks) also the frequency response of the corresponding closed-loop system can be determined; the buttons  and  of the frequency response window switch between both modes. If a new substructure was selected or parameters of selected blocks were changed, the frequency response must be updated via the refresh button  . The  button calls a dialog which e. g. allows the specification of the frequency range. Most other buttons of the frequency response window are self-explanatory.

If instead of the graphical representation of the frequency response only the transfer function is to be calculated, this can be done via the EDIT | EVALUATE TRANSFER FUNCTION... menu option.

Remark: If the open-loop system contains a dead time, this dead time is *not* considered in the frequency response resp. transfer function of the closed-loop system.



Bode plot (top) and Nyquist plot sample (bottom) for an open-loop system (sample file FreqResponseDemo.bsy*)*

Numerical optimization of system parameters

Besides the pure simulation of systems BORIS offers a powerful module for the numerical optimization of system parameters using intelligent optimization techniques, so-called evolutionary strategies [14]. These strategies are able to find the global optimum of a user-defined quality function with a high reliability even if the function is very complex.

As an introduction we first want to consider a mathematical optimization problem. The task is to find the optimization parameters $x_1 \ldots x_5$ so that the quality function

$$Q(x_1,...,x_5) = (x_1 - 1)^2 + (x_2 - 2)^2 + (x_3 - 3)^2 + (x_4 - 4)^2 + (x_5 - 5)^2$$

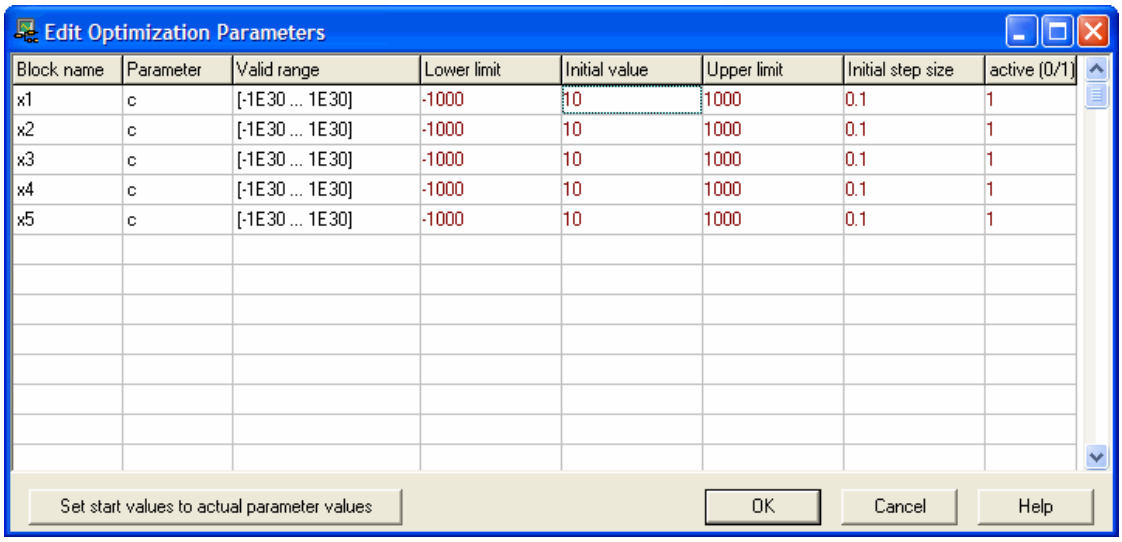
delivers its minimal value. The parameter range for the optimization parameters is given as $-1000 \le x_i \le 1000$. This example allows the checking of the solution found by BORIS because the exact solution can be analytically calculated as

$$x_1 = 1,\ x_2 = 2,\ x_3 = 3,\ x_4 = 4,\ x_5 = 5\ .$$

The corresponding value of the quality function is zero.

Choosing the optimization parameters

BORIS can optimize all active export parameters of the current system (top level). In the example above a quality function of five parameters has to be minimized. So we choose five *constant* blocks and denominate $x_1 \ldots x_5$. The parameter *c* of each block is activated as an export parameter. To edit the optimization parameters we use the OPTIMIZATION | OPTIMIZATION PARAMETERS... menu option.

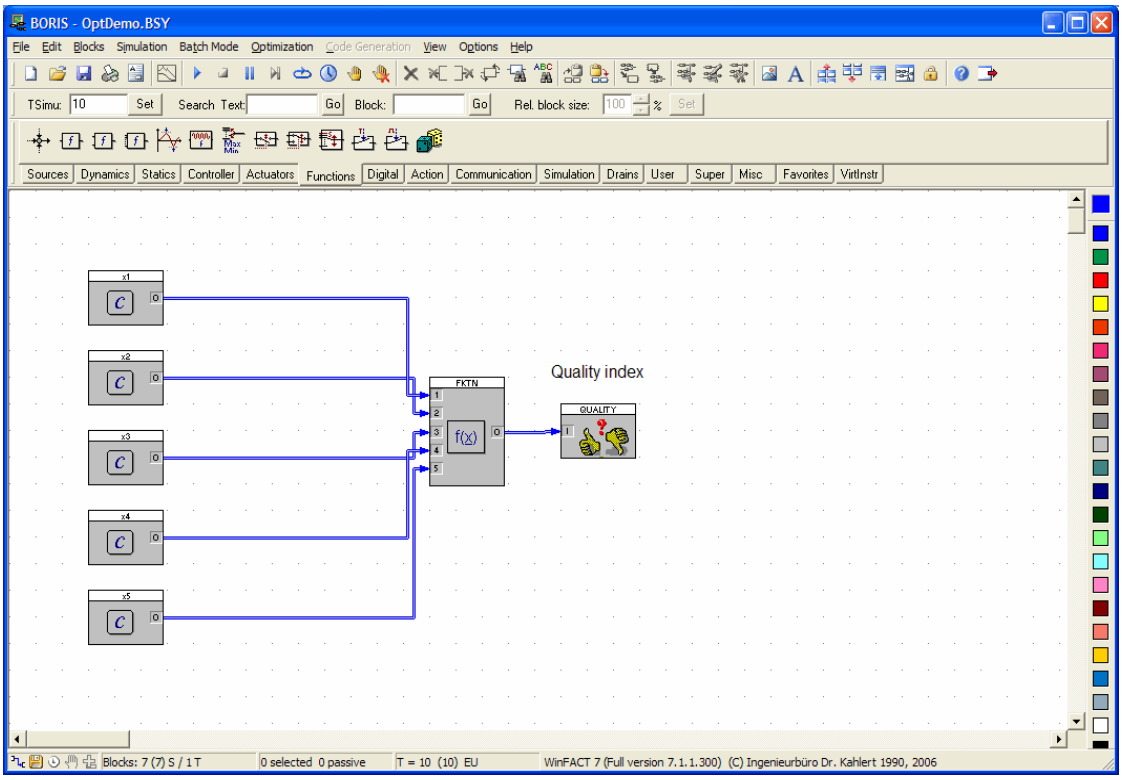


Optimization parameter dialog

Within this dialog the lower and upper limit of each optimization parameter can be specified (-1000 resp. 1000 above) as well as the starting value of the optimization parameter (10 above) and its step size (0.1 above). In addition to that specific parameters can be excluded from optimization by setting the value in the *active* column to zero. This might be sensible e. g. for a system block with three export parameters if only two of these parameters should be optimized. Clicking the *Set initial values to current parameter values* button after finishing an optimization effects that a following optimization starts with start values and step sizes that correspond to the results of the previous run.

Specifying the quality function

Optimization in BORIS always means the *minimization* of a quality or target function that is defined within the regarding system structure in a graphical way. For that purpose BORIS introduces a so-called *quality index block* which is inserted in the same way as a normal system block by the OPTIMIZATION | INSERT QUALITY INDEX BLOCK menu option. The signal connected with the input of this quality block defines the function to be minimized. The screenshot below shows the optimization structure for the sample problem realized in the OPTDEMO.BSY file (installed in the examples directory). The function block serves for the calculation of the quality function $Q(x_1,...,x_5)$.

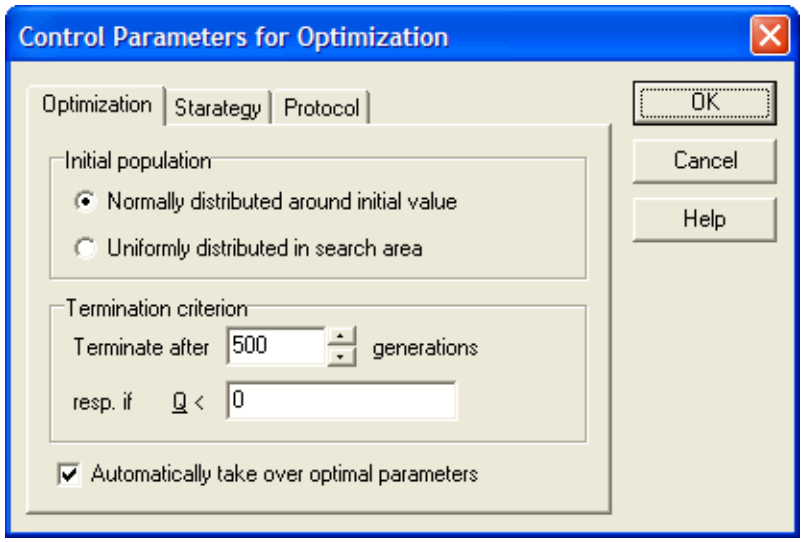


Optimization structure for sample problem

Optimization control parameters

The integrated optimization module of BORIS is based on evolutionary strategies (genetic algorithms). The optimization can be controlled by a set of control parameters (strategy parameters). These can be selected by the menu option OPTIMIZATION | CONTROL PARAMETERS... The control parameter dialog consists of three registers.

Register *Optimization*



This register contains the following options:

Initial population

This setting specifies the type of the initial population. If the option *Normally distributed around initial value* is activated, the initial individuals are located normally distributed around the initial value set under *Optimization parameters*. The variance of the normal distribution corresponds to the selected initial step size. If the option *Uniformly distributed in search area* is selected, the initial population is located uniformly distributed within the specified parameter bounds.

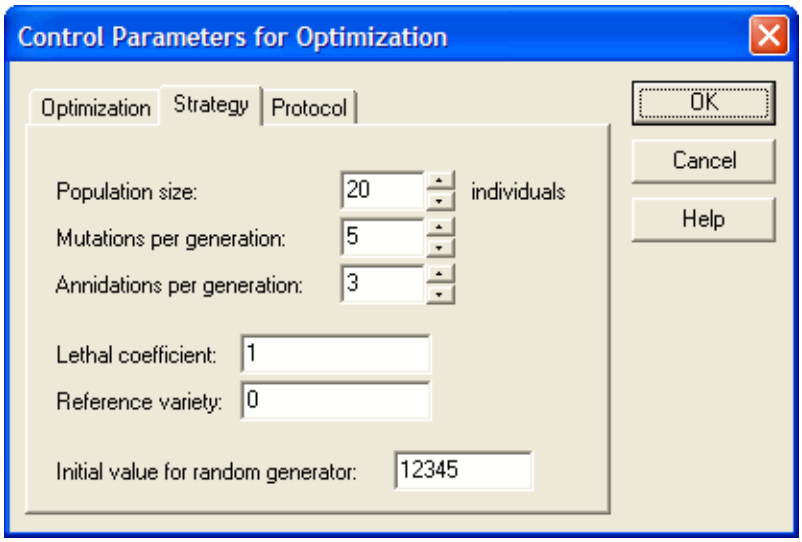
Termination criterion

Specifies the termination condition for the optimization. The optimization is stopped either if the specified number of generations is exceeded or the value of the quality function is less than a specified limit value.

Automatically take over optimal parameters

If this option is activated, the optimal parameters found by the optimization strategy are automatically copied to the corresponding block parameters after the optimization has terminated.

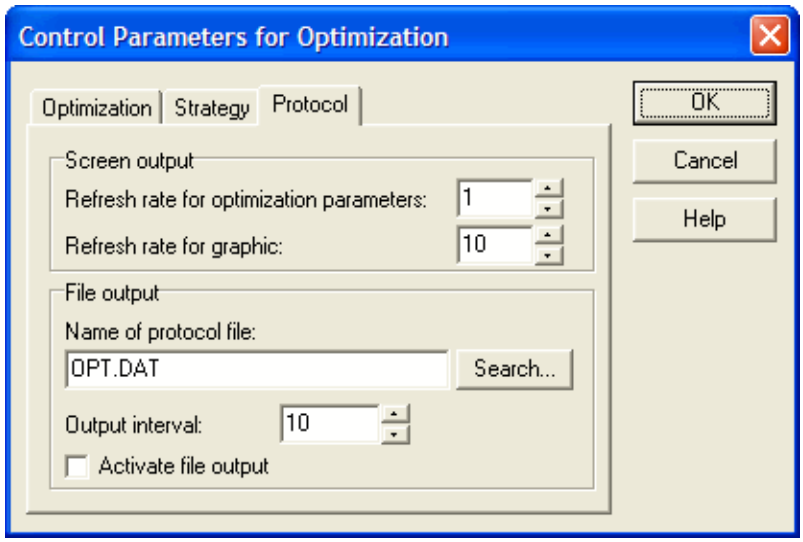
Register *Strategy*



This register contains the following options:

Population size	Number of individuals within the population. This value should be adjusted to the number of optimization parameters (at least four or five times greater).
Mutations per generation	Specifies the number of mutated individuals within one generation. A greater value results in a more global search for the optimum, however it can result in a lower convergence speed (see [14]).
Annidations per generation	Specifies the number of annidations within one generation. A greater value results in a more global search for the optimum, however it can result in a lower convergence speed (see [14]).
Lethal coefficient	Specifies the lethal coefficient for a single mutation. A smaller value results in a more global search for the optimum, however it can result in a lower convergence speed (see [14]).
Reference variety	Specifies the nominal variety of individuals within the population. A greater value results in a more global search for the optimum, however it can result in a lower convergence speed (see [14]).
Initial value for random generator	Specifies the initial value for the random number generation.

Register *Protocol*

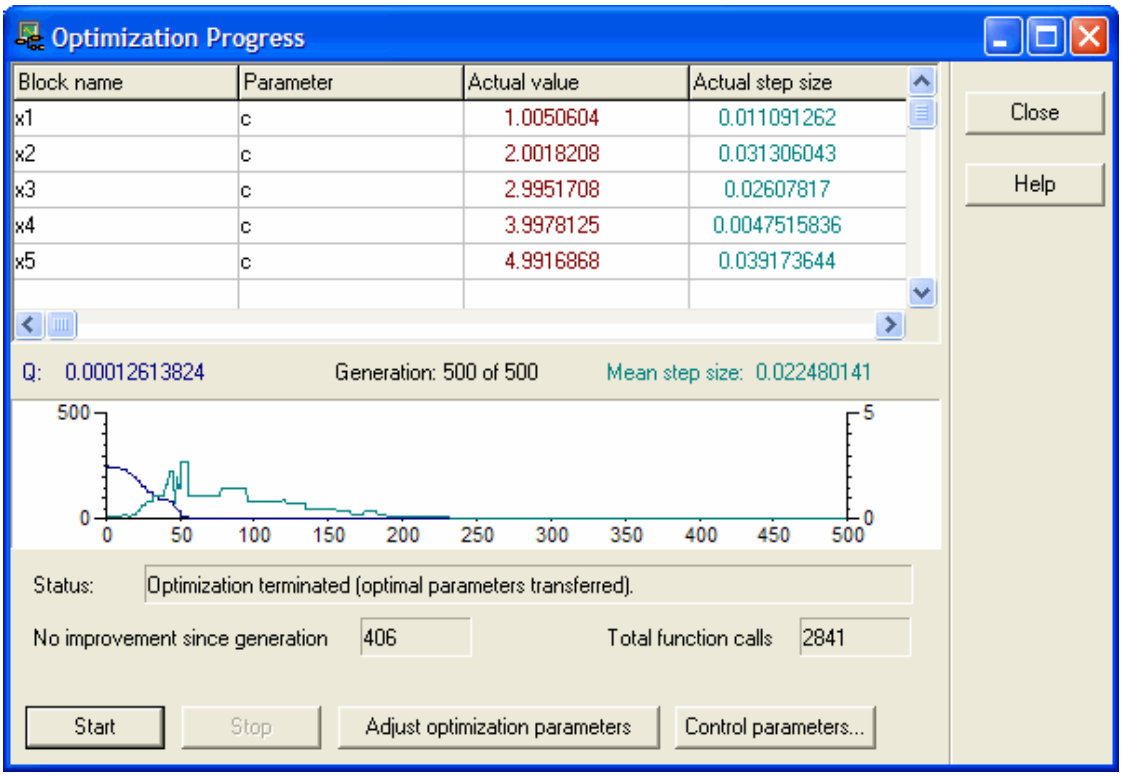


This register contains the following options:

Screen output	Specifies the refresh rate for the screen output (see chapter *Controlling the optimization* process below).
File output	Specifies the file output during the optimization process. The value of the *Output interval* field specifies the number of generations between two file outputs in case of an activated protocol file.

Controlling the optimization process

Having done all preparations for optimization, the optimization control dialog can be called by the OPTIMIZATION | START OPTIMIZATION... menu option. The screenshot below shows the dialog after running a successful optimization for the sample problem described earlier.



Optimization control dialog (here after having finished an optimization run)

The upper part of the dialog shows all optimization parameters with their current values (red) and step sizes (blue-green). The chart in the middle part of the dialog records the progress of the quality index Q and the mean value of the step size. The display of the last successful generation and the total number of function calls (simulations) complete the information.

A new optimization is started by clicking the *Start* button. A running optimization can be stopped any time by clicking the *Stop* button. After finishing the optimization procedure the parameters found as optimal can be set to the start values for the next optimization by the *Adjust optimization parameters* button. In this case the current step sizes of all parameters are analogously set to the start step sizes for the next optimization.

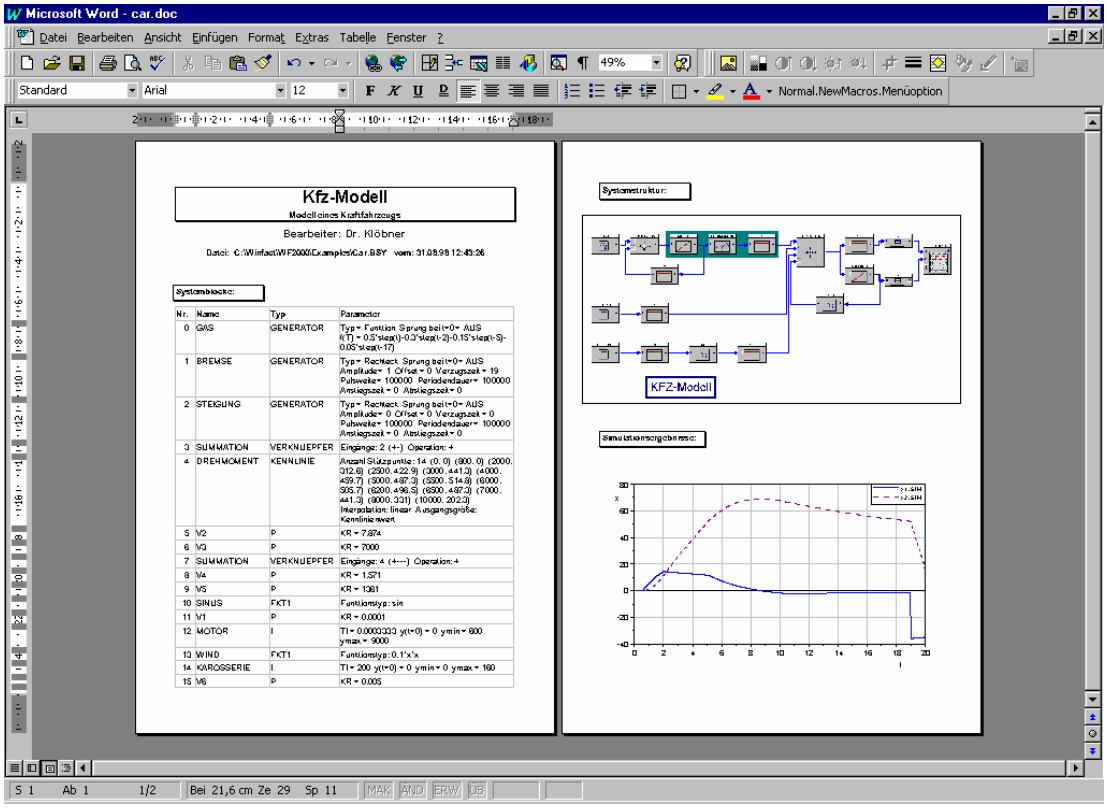
Application for controller design

A main application of the numerical parameter optimization is the automatical design of controllers, e. g. based on integral criteria as the ITAE-criterion or other performance indices. It's no problem to include the maximum control output of the controller or other restrictions into the design. The simultaneous design of several controllers can be realized as well. Two examples of the

controller design by numerical optimization are delivered in the files ITAEOPT.BSY resp. ITAEOPT2.BSY.

Documenting systems

The integrated document generator of BORIS provides a comfortable way to document your system files in graphical as well as in textual form. The document output can directly be sent to the printer or exported in a different manner. The export options represent a flexible way to make the documents disposable for other types of Windows applications (word processing software, graphic suites ...).

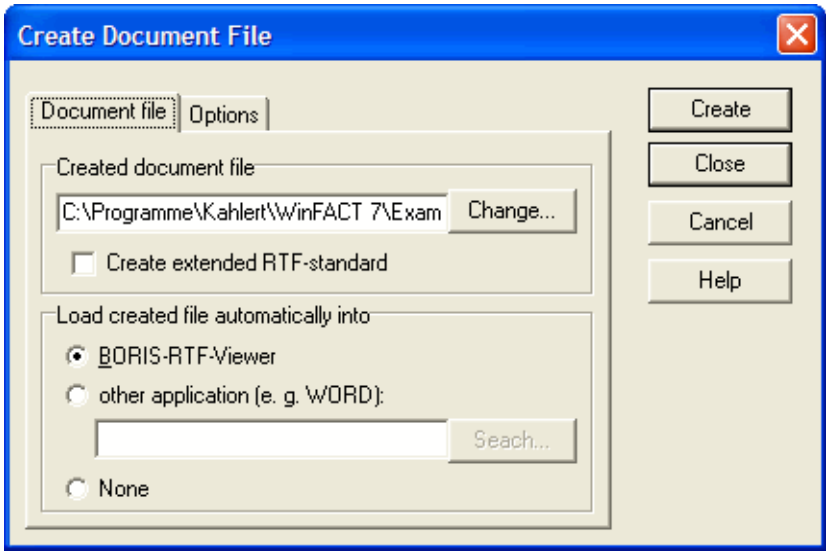


Documenting a system with the integrated document generator

Creating a document file

After saving a BORIS system or superblock file the first time, a document file can be created which includes all information regarding the system structure. The document file is of RTF type (*Rich Text Format*) and can later on be imported into nearly each word processing software. In case that no word processing software is at disposal, the integrated RTF viewer can be used. In comparison to standard word processing software this viewer however provides only the main features used for text editing.

To create the document file choose the FILE | CREATE DOCUMENT FILE... menu option resp. the button of the control toolbar. A dialog consisting of several registers appears which controls the document generation.



Dialog for document file control (Register Document file*)*

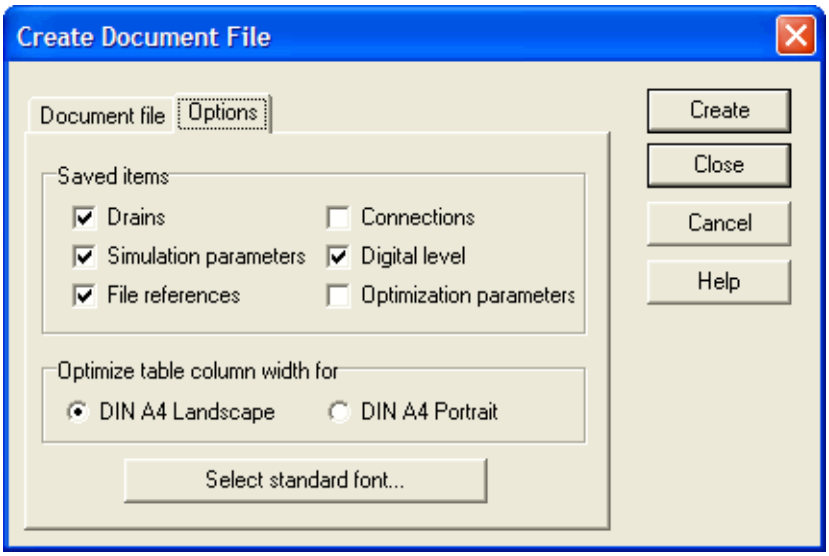
The edit field *Created document file* specifies the name of the created RTF file. The default name is the name of the current system file whereas the extension is replaced by RTF. If the checkbox *Create extended RTF-standard* is checked, the file is created in table form. This form allows a much better representation of the data and is therefore recommended. The integrated RTF viewer however does not support tables; so in this case the option should be deactivated.

If requested the created file can directly be opened by the specified RTF editor resp. the word processing program. This option is activated by the according entry within the *Load created file automatically into* group box.

The *Options* register specifies the items saved within the document file (group box *Saved items*). The *Optimize column width for* group box provides the se-

lection of the paper format used for later printing of the file. The file font can be chosen by the *Choose standard font*... button.

The document file generation itself is started by clicking the *Start* button. After generation is complete, the file will automatically be opened into the specified text editor (if this option was not deactivated).



Register Options

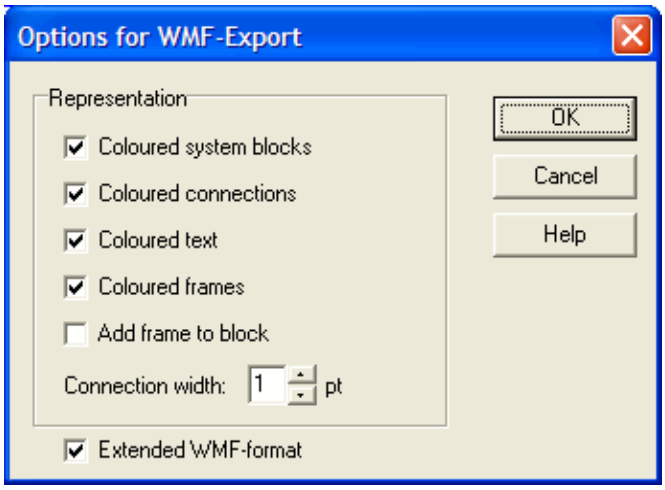
Exporting the system structure

The current system structure can be exported in two different formats:

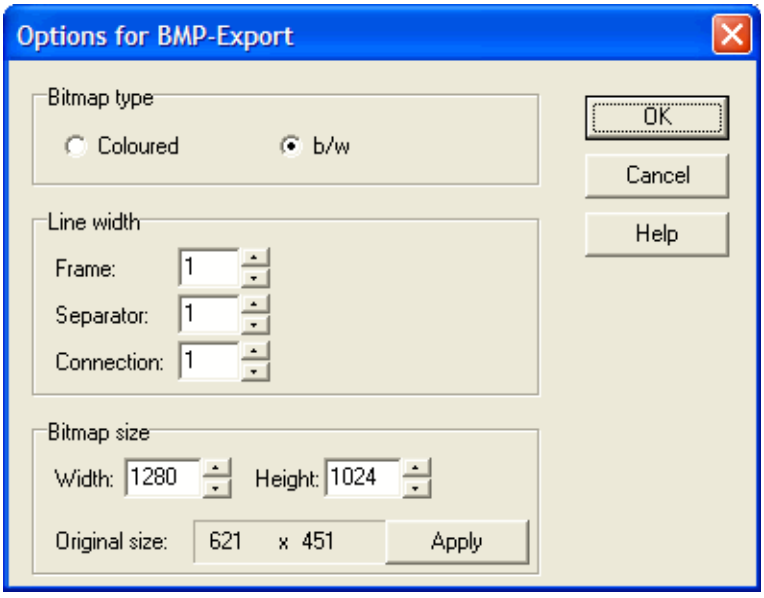
- As a bitmap (BMP format)
- As a vector graphic (WMF format)

Both formats can be exported in color as well as in b/w mode. Nearly each Windows application provides the import of these formats.

To start the structure export use the FILE | EXPORT... menu option. After choosing the output format and file via a standard file open dialog, another dialog allows the selection of some format specific options.



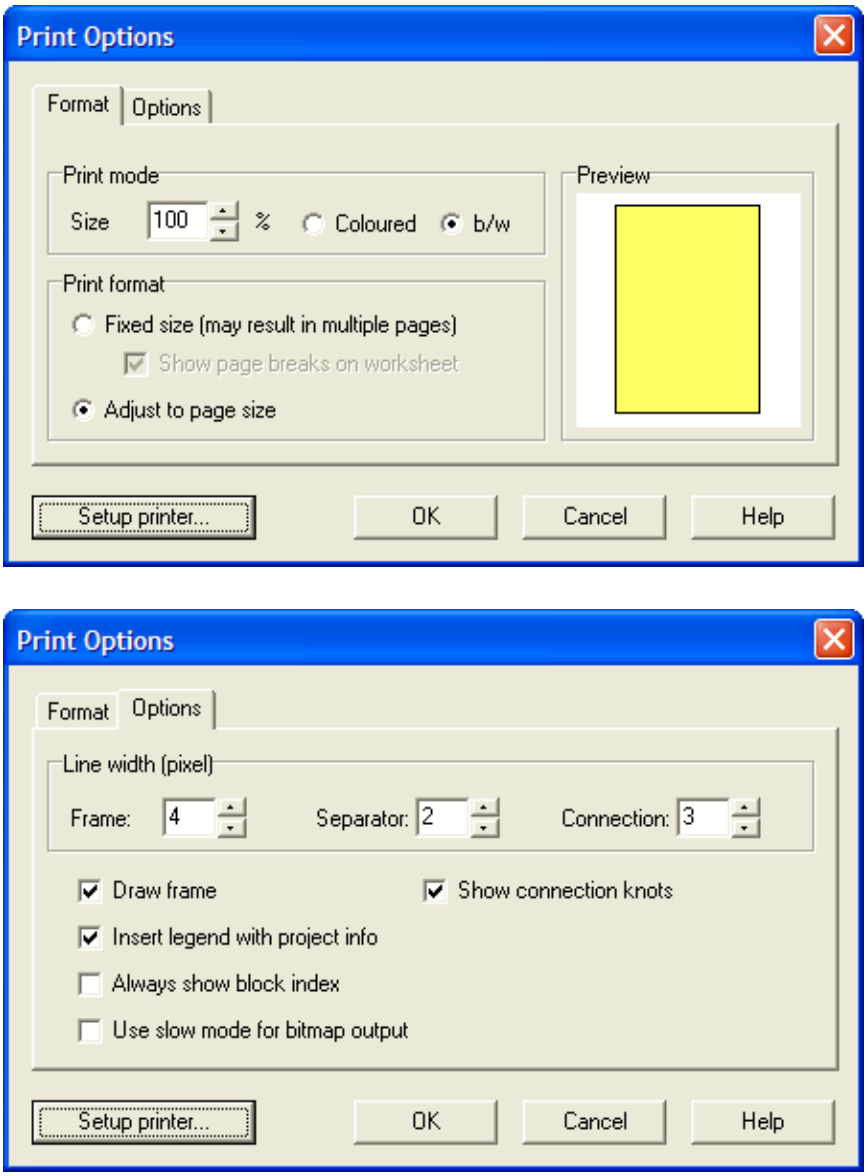
Export options for WMF format



Export options for BMP format

Printing the system structure

In addition to the capability of exporting the system structure BORIS offers different ways of printing. These can be activated by the menu option FILE | PRINT... resp. the button of the control toolbar. The printing can be modified by several options which are accessed by the print options dialog.



Print options dialog containing a Format *(top) and an* Options *register (bottom)*

The dialog options have the following functions:

Option	**Meaning**
Print mode	Specifies the print size and the color resolution
Print format	Specifies the print format. BORIS is able to print with a fixed size or to fit the printer output automatically to the page size of the printer. If the fixed size mode is selected, complex systems result in a multiple page print. In this case the page breaks are displayed previously on the screen if the *Show page*

	breaks on worksheet checkbox is activated. The number of printed pages can also be controlled via the *Preview* display.
Line width	Specifies the line width for the different components of the system structure
Draw frame	Specifies whether a frame has to be printed around the system structure
Insert legend with project info	If this option is activated, BORIS automatically includes the project info of the current file to the printer output.
Always show block index	If this option is activated, the block index is always printed within the block caption.