ProviewR
OPEN SOURCE PROCESS CONTROL

Designer's Guide

1

2   Introduction

ProviewR is a modern, powerful and general process control system. It contains all functions normally required for successful sequential control, adjustment, data acquisition, communication, supervision, etc.

The configuration of a ProviewR system is done graphically, making the application adaptation simple, reliable, and flexible. ProviewR is a distributed system, which means that the system consists of several computers, connected via a network . Via the network, computers exchange data with each other. In this way, for instance, the measuring signals will be known on all the process- and operator stations of a ProviewR system.

A ProviewR system is defined with objects. Each object represents a physical or abstract entity in the system. The objects are structured in hierarchical tree structures, making it possible to describe the system in a structured way. The tree structure also imposes a hierarchical naming of all objects. By carefully choosing names when configuring a system, the full name of an object will identify its function, or role in the system.

The hierarchies are divided into two groups; one called the Plant Configuration, describing the logical view of a system; and one called the Node Configuration, describing the physical view of the system. The configuration of these two parts can be done independently. Eventually the two parts are connected to each other.

To configure a system you use the ProviewR Workbench. The workbench comprises a permanent database and a number of tools to help you configuring the objects needed to define the system. From the workbench you create a runnable system as well as documentation about the system.

The purpose of ProviewR is to help you with creating automated systems. Suppose you have a process that you wish to control, ProviewR helps you creating the control system for this process. When the system is created, you will find that you have also created the documentation for the system.

3 Overview

As this is a guide for designers, we start with a description of what the design of a control system implies. The description can also work as an introduction to the different concepts that are used in this guide.

A designer's starting point is, of course, the process that is to be controlled by the system, and the first task is to learn about the process and to figure out the best way to control it: what control loops are needed, what interlockings, how is the startup and shutdown carried out, how are operators and maintainers going to work with the system. This is summed up in a Design specification.

At the same time you have to consider what information about the process the control system needs to perform its task, i.e. which sensors should be placed in the plant. The control system also has to influence the process in various ways, for example with valves and engines. This ends in a Signal list, that is a list of all input and output signals of the system.

At this point the question of which control system is to be used is raised, and one alternative is of course ProviewR. You also have to decide which I/O-system to use and how to divide the function among the different process stations.

IO-systems

The task of the I/O-system is to bring signals from the process to the control system, and to put out signals to influence the process. The signals are usually digital or analog, but there are also other types as integers and counters. You can choose between a rack and card system connected to the process station, or distributed I/O, e.g. profibus.

Configure the system

When it's time to start to configure the system, you first create a new project in the Administrator. The Administrator is a tool for creating order among all the projects, as there can be plenty of them in time.

The configuration of a system is mainly done by creating objects in a database, the Workbench. There are a large amount of objects to configure everything from IO-channels to PLC programs. The ProviewR Object Reference Manual contains over 800 different types of objects. The objects are placed in a tree structure, and you use a tool called the Configurator to create objects and to navigate in the object tree.

The object tree is divided in two parts, the Plant hierarchy and the Node hierarchy. The Plant hierarchy reflects the different functions in the plant and in the process, while the Node hierarchy reflects the hardware of the control system, with computers, I/O-racks and I/O-cards.

When the control system is started later in runtime, a copy of the object tree is created in a realtime database, rtdb. The transfer from the Workbench to rtdb is done with so called loadfiles, files that are generated from the Workbench and contain all its objects.

Control program

ProviewR offers a graphical programming language in which logic, Grafcet sequences and control loops are programmed. It is named the PLC program. Also the PLC program is a part of the object tree. It is configured by placing specific program objects, PlcPgm, in the Plant hierarchy.
When opening a PlcPgm you enter the Plc Editor, in which the graphical programming is preformed. Function blocks are created, and connected in a signal flow of digital and analogous signals, where the input signals are fetched at the left side, transformed in different function blocks, and finally stored in output signals at the right side.

A complement to the PLC program are application programs, that are written in the c, c++ or java language. Applications are written and started as separate programs and connected to the realtime database through an API.

Simulation

The realtime database, the PLC program and possible applications kan easily be started in the development station. This makes it possible to test the programs in direct connection with the programming. You can even write special simulation programs which read the output signals, simulate the outputs influence of the process, calculate values of different sensors and put these values in the input signals.

The configuration and programming of the system is then a process where you switch between configuring/programming and testing. The result is carefully debugged programs and a fast and efficient commissioning of the plant. It also results in better programs, and more thoroughly worked through functions, because the feedback is greater in the creating process than the construction of a process control system implies.

At the simulation and commissioning it is of great importance to have access to tools that makes it possible to monitor and examine the system, and quickly localize possible malfunctions. In ProviewR this tool is called Xtt. Xtt contains a lot of functions to examine the content of the realtime database, to flow the signal flow, to log fast or slow sequences, etc.

Operator interface

There are a number of different groups of professionals that shall gain access to the system, operators running the plant on a daily basis, maintainers occasionally correcting some malfunctions, process engineers requesting various process data. All have their demands on the system interface. Furthermore the limits between the various groups of professionals might be fluid, operators that are both operators and maintainers, and perhaps even process engineers. This puts high demands on the functionality and flexibility of the operator interface. You can rapidly and easily trough so called methods that are activated from popup menues fetch all the information of various objects that are stored in the realtime database, or in different server systems.

Process graphics

Process graphics are built in a graphical editor. The graphics are vector based, which makes all graphs and components freely scalable. Components have a preprogrammed dynamic to change color and shape, depending of signals in the realtime database, or to respond to mouse clicks and set values in the database. In each component that is sensible to mouse click or input,

you can state access, and selectively allow or hinder users to influence the system.

Supervision

If any malfunction arise in the system, the operator has to be noticed. This is done with special supervisory objects that are configured in the plant hierarchy or in the PLC program, and that originate alarms and events. The alarms have four priority levels: A, B, C and D, and are displayed to the operator in the alarm list, the event list and the historical event list.

The alarmlist contains unacknowledged alarms and alarms in alarm state. An alarm normally has to be acknowledged before it disappears from the list. If the alarm state is active, the alarm remains in the list as long as it is active.

Alarms are also registered in the event list, that displays events in a chronological order.

The historical eventlist is a database, also registering events. Here you can search for events, stating criteria as priority and process part.

If a plant part is shut down, it is possible to block the alarms to avoid to distract the operator. Blocked plant parts are displayed in the Block list.

Data Storage

You often want to follow the change of a signal over time, in the shape of a curve. In ProviewR there are three kinds of functions for this, DsTrend, DsFast and SevHist.

DsTrend is a trend that is stored in the realtime database. The value of a signal is stored continuously with an interval of 1 second and upwards. For each curve there is space for about 500 samples, so if you choose to store a new sample every second, you have a trend of the signal of about 8 minutes.

SevHist stores signals in a similar way in a database on disk, which makes it possible to store values a longer period of time than DsTrend.

DsFast stores more rapid sequences, where the storage is started on a trigger condition, and continues a specified time. When the sequence is finished, it is displayed as a curve.

4   Database structure

As we have seen earlier, the main part of the configuration of a ProviewR system is taken place in a database, the Workbench. In the Workbench you create objects in a tree structure, and every object arise a certain function in the control system. ProviewR is so called object oriented, so let us look a little closer at what a ProviewR object really is.

4.1   Object

An object consists of a quantity of data, that in some way defines the state of the object or the properties of the object. The data quantity can be very simple, as for an And-gate, where it consists of a boolean value that is true or false. However, a PID controller has a more complex quantity of data. It contains gain, integration time, output, force etc. It consists of a mix of digital, analogous and integer values. Some values are configured in the development environment and some are calculated in runtime.

The quantity of data is called the body of the object. The body is divided in attributes, where every attribute has a name and a type. The body of an And-gate consists of the attribute Status which is of type Boolean, while the body of a PID controller consists of 47 attributes: ProcVal, SetVal, Bias, ForceVal etc.

All PID objects have their quantity of data structured in the same way, you say they are a member of the same class. The PID objects are members of the PID class, and the And-gates are members of the And class. A class is a kind of model of how objects that belong to the class appear, for example what the attributes are, and the name and type of the attributes.

Besides a body, an object also has a header. In the header, the class and identity of the object is found, and also its relation to other objects. The objects are ordered in a tree structure, and in the header there are links to the parent, and the closest siblings of the object.

4.2   Volumes

When configuring a system, and creating objects, you usually know which node the objects will belong to in runtime. You could group the objects after which node they will belong to, but a more flexible grouping is made, so instead you group the objects in volumes. A volume is a kind of container for objects. The volume has a name and an identity, and it contains a number of objects ordered in a tree structure.

There are a number of different types of volumes, and the first you get in contact with is a root volume. When configuring a node, you usually work in a root volume. Every node is connected to a root volume, i.e. when the node is starting up in runtime the root volume, and its objects, are loaded into the node. Below is a description of the different types of volumes.

RootVolume

A root volume contains the root of the object tree in a node. At startup, the node is loading

the root volume.

A node is connected to one and only one root volume. Furthermore, a root volume can be loaded into several nodes. When a process station is running in production, the same volume can concurrently be loaded into a development station for simulation, and a third node can run the volume in educational purposes. Though, you have to consider that the nodes have to run in different communication buses.

SubVolume

Some of the objects in a node can be placed in a subvolume. The reason to divide the objects of a node in a root volume, and in one or several subvolumes could be that a number of persons have to configure the node simultaneously, or that you plan to move parts of the control of some plant parts to another node later.

ClassVolume

The definition of different classes reside in a special type of volume, called ClassVolume. Here the description of a class is built with objects that define the name of the class and the attributes of the class.

There are two classvolumes that you always include in a ProviewR system, pwrs and pwrb. pwrs contains the system classes, mainly classes used in class definitions. pwrb contains base classes, i.e standard classes that are needed to build a process or operator station.

DynamicVolume

A dynamic volume contains dynamic objects, i.e. volatile objects created at runtime. If you have a material planning module in the system, an object is created for each material that is processed in the plant. When the processing is completed, the object is removed.

SystemVolume

The system volume is a dynamic volume that resides in every node, and that keeps various system objects.

DirectoryVolume

The Directory volume only exists in the development environment. Here the volumes and nodes of the system are configured.

Volume Identity

Each volume has a unique identity, that is written with four numbers, separated by periods, e.g. "_V0.3.4.23". The prefix _V states that it is a volume identity. To verify that the volume identities are unique, there is a global volume list that contains all volumes. Before creating a project, the volumes of the project should be registered in the volume list.

4.3   Attribute

The quantity of data for an object is divided into attributes. Each attribute has a name and a type. Here follows a description of the most common attribute types.

Boolean

Digital attributes are of type boolean. The value can be true (1) or false (0).

Float32

Analogue attributes are of type Float32, i.e. a 32-bit float.

Int32

Integer attributes are usually of type Int32, i.e. a 32-bit integer. There are also a number of other integer types, e.g. Int8, Int16, Int64, UInt8, UInt16, UInt32 and UInt64.

String

In a string attribute a character string is stored. There are different string types with various length, e.g. String8, String16, String40, String80, String256.

Time

Time contains an absolute time, e.g. 1-MAR-2005 12:35:00.00.

DeltaTime

DeltaTime contains an delta time, e.g. 1:12:22.13 (1 hour, 12 minutes, 22.13 seconds).

Enum

Enum is an enumeration type, used to choose one option of several alternatives.
It can be assigned one integer value, in a series of integer values, where each value is associated with a name. There are, for example, the enumeration ParityEnum which can have the values 0 (None), 1 (Odd) or 2 (Even).
Enum is a basic type and ParityEnum is a derived type.

Mask

Mask is used when choosing one, or several, of a number of alternatives. The alternatives are represented by the bits in a 32-bit integer.

An attribute can also consist of a more complex data structure. It can be an array with a specified number of elements, and it may also be another object, a so called attribute object.

4.4   Class

A Class is a description of how an object that is a member of the class shall look like. An object that belongs to the class is called an instance. The class defines how the data of the instances are structured in attributes of various types, or the graphic representation of objects in the PLCeditor or in the operator environment.

Each class has a template object, i.e. an instance of the class that contains default values for the attributes of the class.

ProviewR's base system contains about 1000 classes. See Object Reference Manual for detailed description. The designer can also create his own classes within a project.

4.5   Object Tree

The objects in a volume are ordered in a tree structure. In the volume there is one, or several
top objects, each top object can have one or several children, that can have children etc.
You usually talk about the relations between objects in the tree in terms as parent, sibling,
child, ancestor and descendants.

4.6   Object Name

Each object has a name that is unique within its sibling family. The object also has a path name
that is unique within the world. The path name includes, besides the object name, the volume
name and the name of all the ancestors, e.g

```
VolTrafficCross1:TrafficCross1-ControlSignals-Reset
```

If you want to be more specific and point out an attribute in an object, you add the attribute
name to the object name with a period between, e.g.

```
VolTrafficCross1:TrafficCross1-ControlSignals-Reset.ActualValue
```

Also an attribute can have several segments, since an attribute can consist of an object. The
attribute name segments are separated by periods, e.g

```
VolTrafficCross1:Hydr-Valve.OpenSw.ActualValue
```

4.7   Mounting

An operator station has to display values of signals and attributes that reside in the volumes
of process stations. This is achieved by a mechanism where an operator station mounts the
volumes of the process stations in its own object tree. A mounting means that you hang an
object tree of another volume in the own root volume. Where in the tree the volumes are hung,
is configured with a MountObject object. The MountObject states which object in the other
volumes that is mounted. The result is, that the MountObject is displayed as the mounted object,
with the object tree beneath it. It apparently looks as if the objects belong to the own root
volume, while they in reality reside in another node.

If you use sub volumes, they also have to be mounted in a root volume to make the objects
available.

When you choose mounting points and names of mounting points, it is suitable to do this in such a
way that the objects have the same pathname in both volumes.

4.8   Object Identity

An object has an identity that is unique. It consists of the volume identity, and an object index
that is unique within the volume. An object identity is written for example "_O0.3.4.23:34" where
0.3.4.23 is the volume identity, and 34 the object index. The prefix _O states that it is an
object identity.

5   A Case Study

In this chapter we will give you an idea of how a ProviewR system is created. The process to control is very simple in this case study - an intersection with four traffic lights - but it will give you an idea of the steps you have to go through when creating a ProviewR system.

The traffic lights should be able to operate in two different modes:

  - Normal: The traffic lights run normal cycle of red, yellow and green.
  - Flash: The traffic lights are flashing yellow.
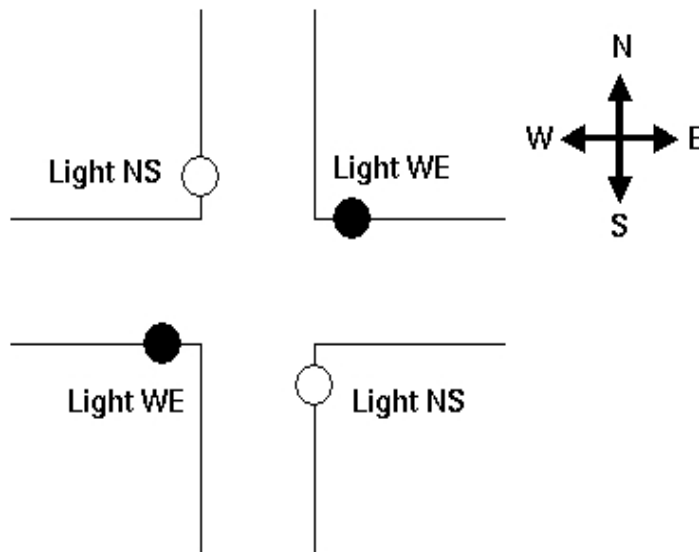
Fig Traffic lights in an Intersection

The operating mode of the traffic lights is decided by an operator via an operator station, or by a maintenance technician who influences a switch. The maintenance technician can change mode only if the operator has switched the traffic lights to service mode.

Figure 'Traffic Lights, Control Panels' shows the different switches and indicators needed by the operator and maintenance technician respectively to be able to monitor and control the system. These could be realized with plant graphics on the operator station or with hardware.
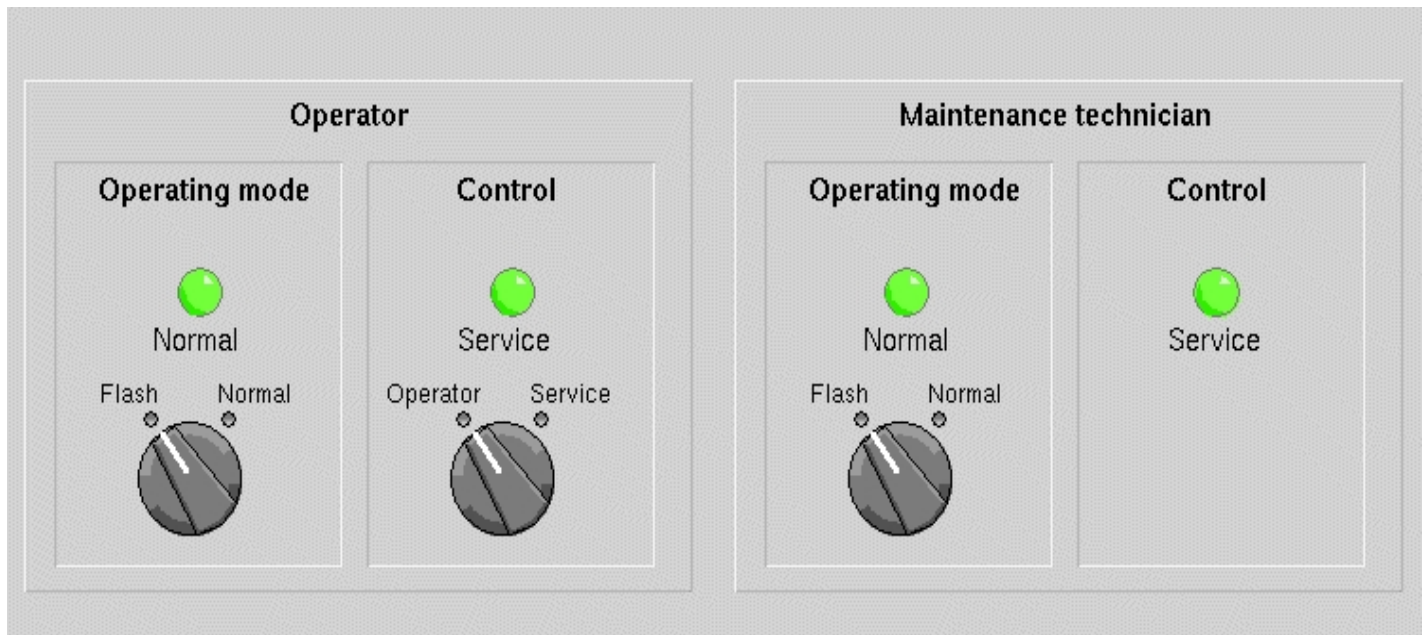
Fig Traffic Lights, Control Panels

5.1   Specification of I/O

We start with analysing the task to decide what hardware to use.

Digital Outputs

We have four traffic lights, but the traffic lights in the same street can be connected in parallel, which means that we can treat them as two lights.

Three outputs per light: 2 * 3 = 6
Indication: Operating mode 1
Indication: Control 1

Total number of digital outputs: 8

Digital Inputs

The only digital input needed is for the maintenance technician's switch. The operator controls the process from the computer display, and this requires no physical signals.

Switch: Operating mode 1

Total number of digital inputs: 1

Analog I/O

No analog in- or outputs are needed for this task.

Specification of the Process Station

When we have decided upon the I/O needed, we can choose the hardware. We choose:

1 Linux PC with rack
1 card with 16 digital inputs
1 card with 16 digital outputs

Specification of the Operator Station

1 Linux PC

Specification of Plant Graphics

We need a display from which the operator can control and survey the traffic lights.

5.2   Administration

First we have to register a new volume, create a project, and, if necessary, create new users. For this we need:

- A name for the project. We call it trafficcross1.
- Two volumes, one for the process station and one for the operator station.
- We need three users: one developer, one operator and one maintenance technician.

Volumes, projects and users are registered and created in the administrator tool.

Register volume

For this project, we need two volumes, one for the process station, and one for the operator station. They are root volumes so we can choose an idle volume identity in the interval 0.1-254.1-254.1-254. We choose 0.1.1.1 to the operator station and 0.1.1.2 to the process station, and enter the volume mode in the Administrator to register these volumes.
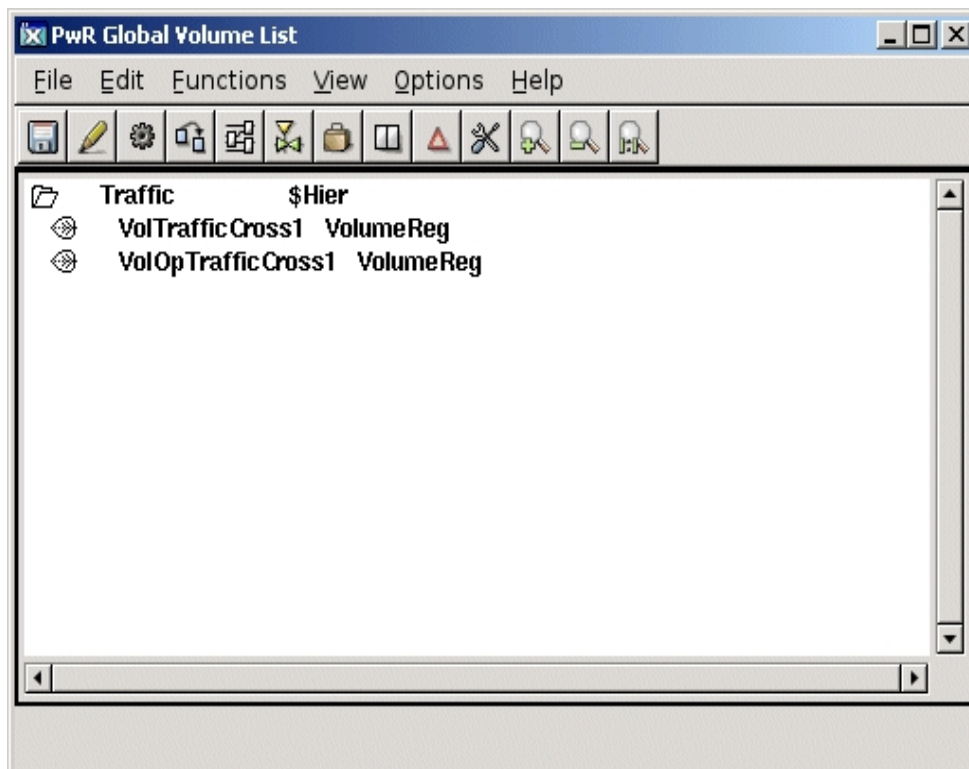

Fig Volume registration

Create users

Eric is a developer in the traffic department, an Carl is an operator. They are both involved in all the projects at the traffic department, so we create a common systemgroup for all the projects and let them share users. We grant Eric developer and system privileges, Carl operator privileges and Lisa maintenance privileges.
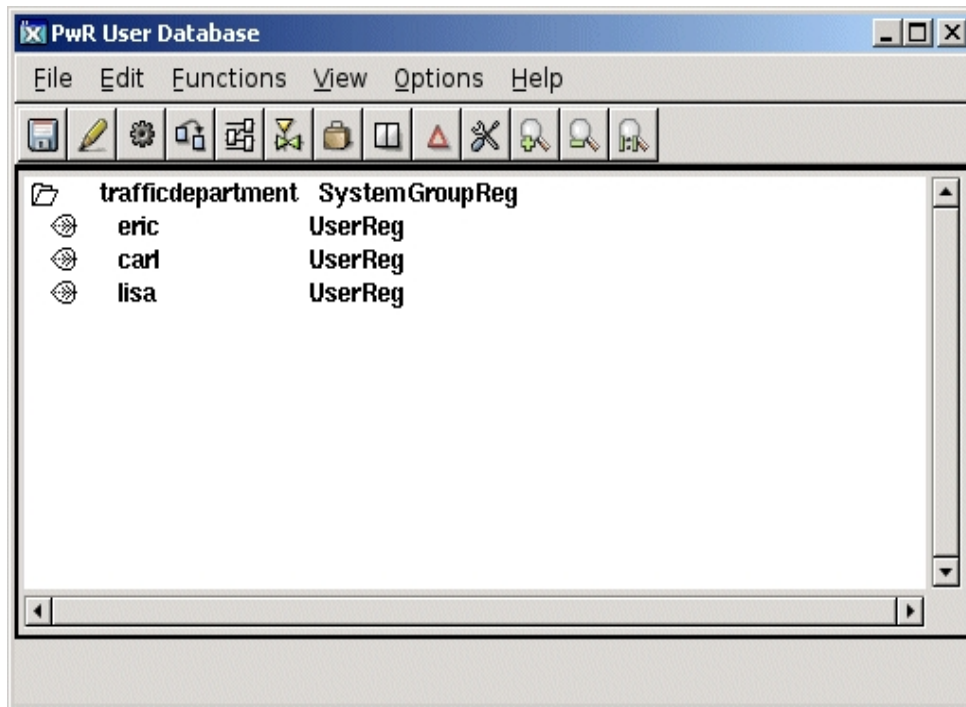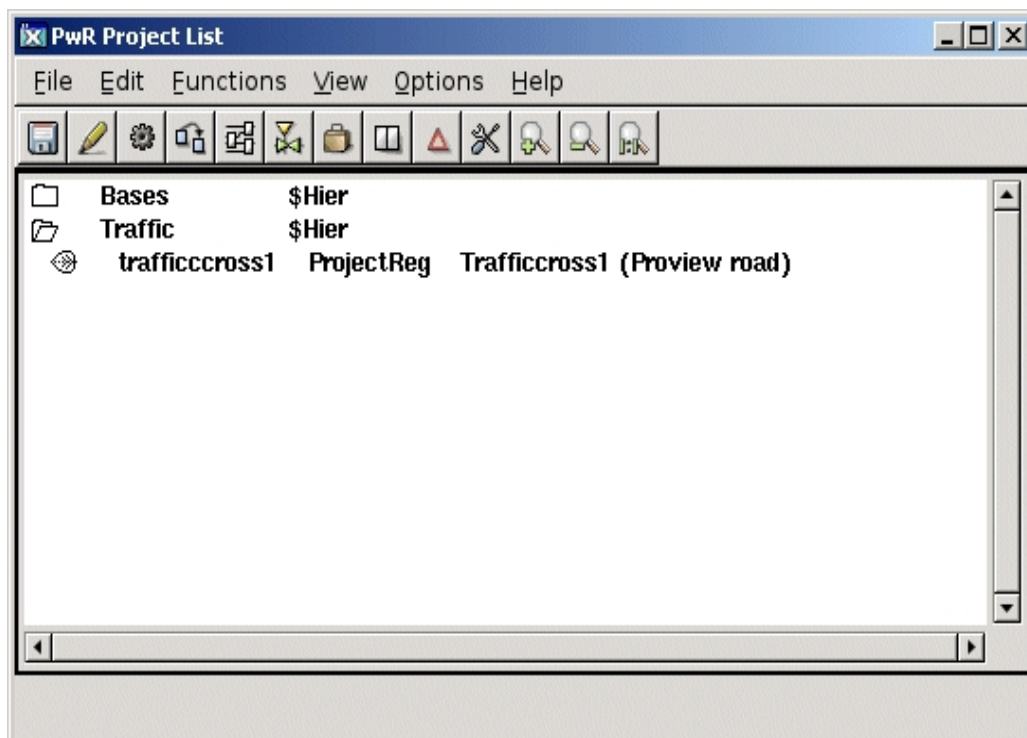

Fig Created users

Create Project

We create the project with the hierarchy name 'traffic-trafficcross1'.

Fig Created project
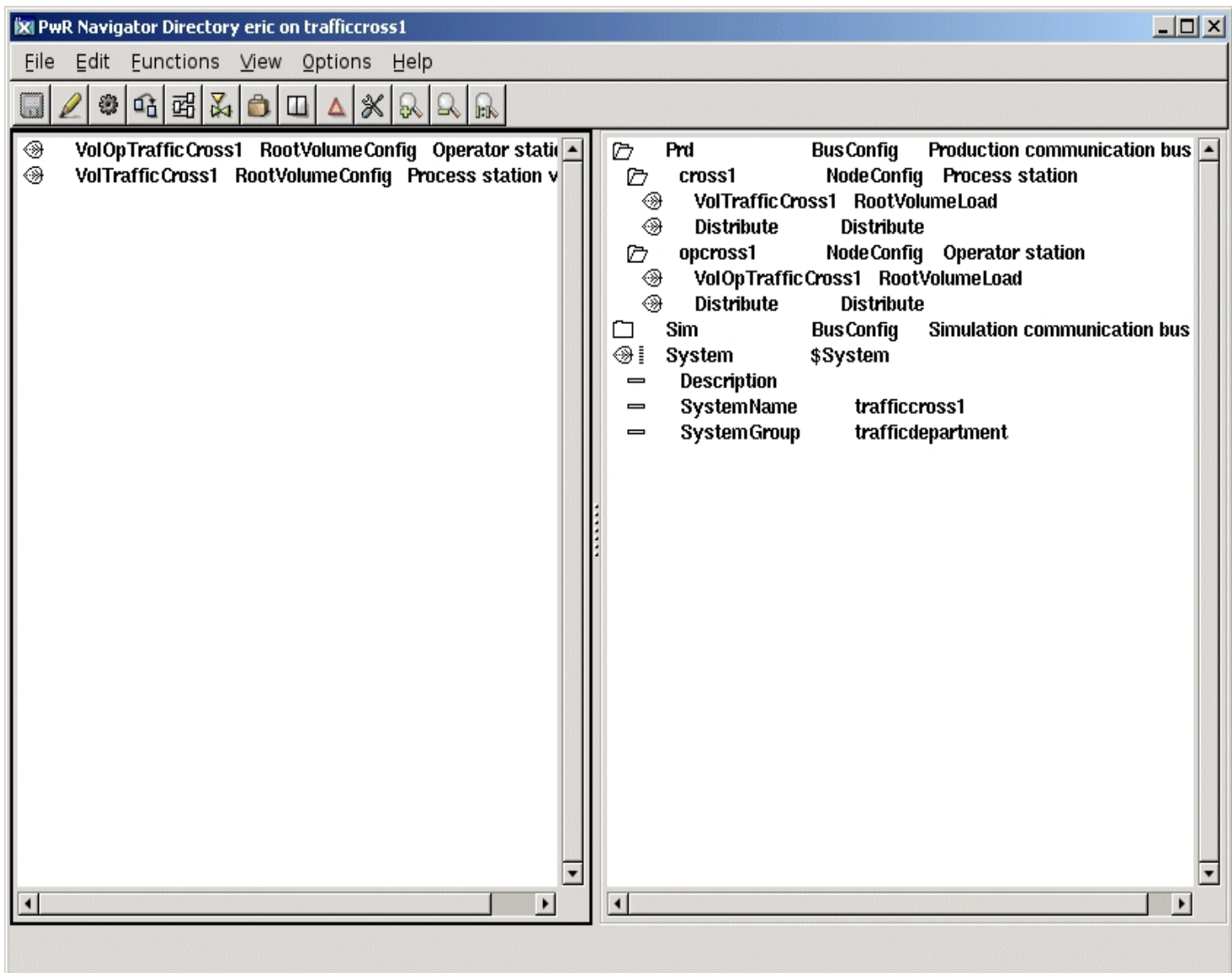
Configure the project

The project has a directory volume in which the nodes and volumes of the project are configured. In the left window the volumes are configured with RootVolumeConfig objects. In the right window the process and the operator station are configured with NodeConfig objects. The NodeConfig objects is put beneath a BusConfig object that states in which QCOM bus the nodes are communicating.

The NodeConfig objects contains

- Nodename
- ip address of the node

Below each NodeConfig object there is a RootVolumeLoad object that states the volume to load at runtime startup.

Note also the system object with the attribute SystemGroup that is assigned the value 'trafficdepartment'. This grants the users eric, carl and lisa access to the project.

5.3   Plant Configuration

Once the configuring of the project is done, next step is to configure the plant. The configuration is done in the Configuration Editor. The plant is a logical description of the reality, which is to be controlled and supervised.

The Process Station

The major part of the configuration is done in the volume of the process station, VolTrafficCross1. This because all physical hardware is configured here (the I/O), all the signals and the PLC-programs that work with the signals.

The plant is structured hierarchically. Examples of levels in the plant can be plant, process, process part, component, and signal. These are the logical signals represented by signal objects which will be connected to physical channels.

Sometimes it can be difficult to configure each signal in an initial stage, but it must at any rate be decided how possible signals shall be grouped.

The figure below illustrates how a plant has been configured. We see how signals have been configured on different levels, and also how the PLC programs are configured in the plant.
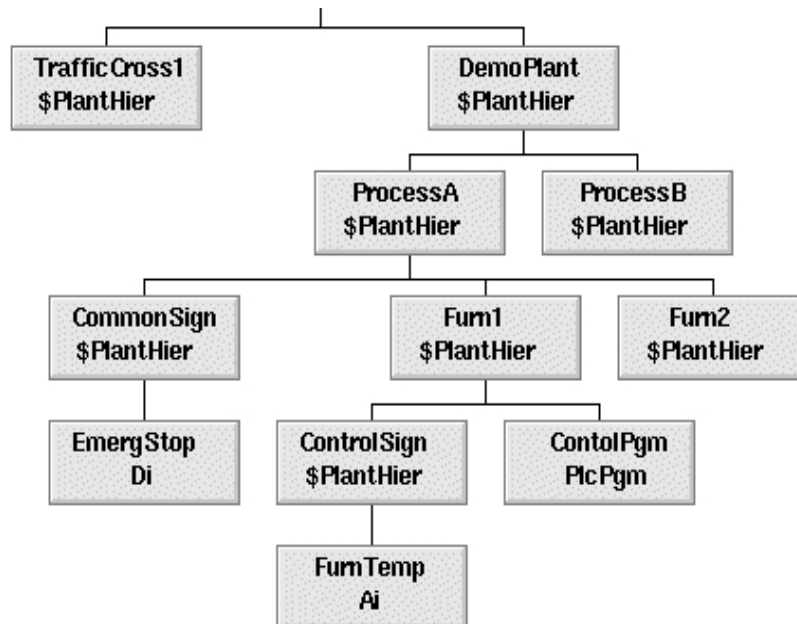
Fig An Example of a Plant Configuration

We choose to call our plant TrafficCross1 and we decide on the following structure:

- Two traffic lights, each one consisting of a green, a yellow, and a red lamp. Since the streets run north-south and west-east respectively, we call them TrafficLightNS and TrafficLightWE. Each lamp requires a signal. These are digital output signals and are called RedNS, RedWE, etc.

- A PLC program to control the traffic lights.

- A number of control signals to select operating mode and function. We choose to put them in

one folder, ControlSignals. The table below shows the signals required.

Figure shows the resulting Plant Configuration.

We choose to call our plant TrafficCross1 and decide the following structure:

Signal Name	Signal Type	Function
ServiceSwitch	Di	A switch which the maintenance technician can influence to change the operating mode.
OperatorSwitch	Di	A value which the operator can influence to change the operating mode.
ServiceMode	Di	A value which the operator can influence to change the function to service mode.
ServiceModeInd	Do	A signal which shows the maintenance technician that the program is in service mode.
Mode	Dv	Indicates whether the program is in normal or flashing mode.
ModeInd	Do	Indicates whether the program is in normal or flashing mode.
Reset	Dv	A value which is used to reset the program to initial mode.
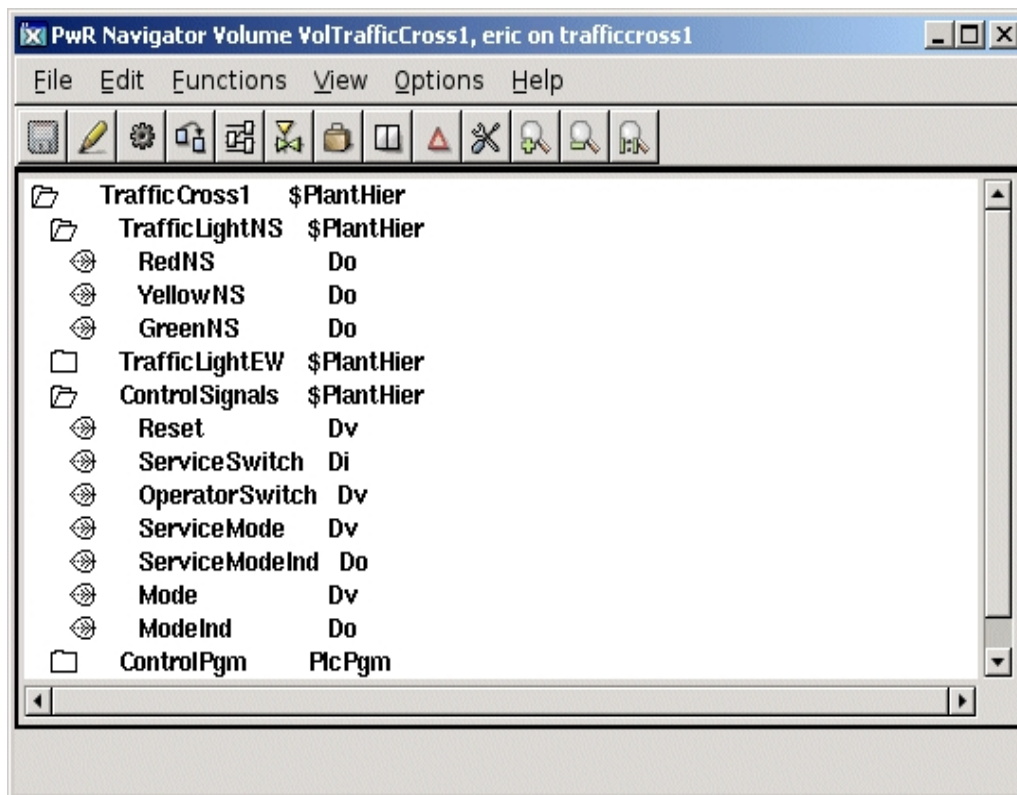
Fig The Plant Configuration of the Intersection

As you can see we have a plant object at the topmost level, TrafficCross1 of the class $PlantHier. We use other objects of class $PlantHier to group our objects. We also create an object, which defines a PLC program, the ControlPgm object of the class PlcPgm .

The Operator Station

The configuration of the operator station is performed in the volume VolOpTrafficCross1. In the Plant side there is only a mount object, that makes the plant hierarchy of the process node available in the operator station. We have mounted the topmost $PlantHier object, 'TrafficCross1' with a MountObject with the same name.
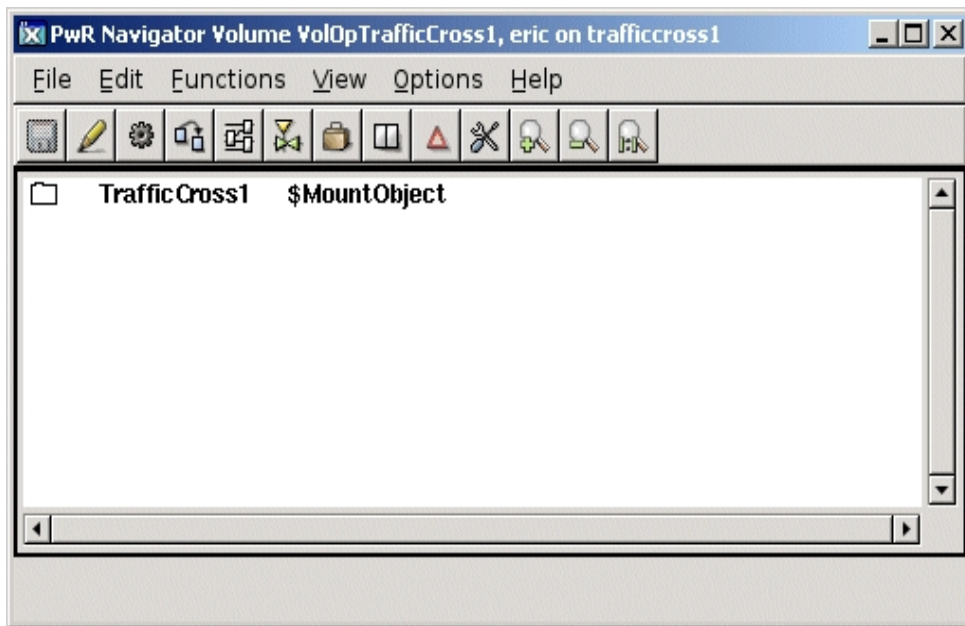
Fig The Plant Configuration in the operator volume.

5.4   Node Configuration

When you have configured the plant, continue to configure the nodes.

Processtation

In this example we choose to start configuring the process station. We name the process station "cross1". It is advisable to give the process stations descriptive names. In the node hierarchy we create a $NodeHier object 'Nodes' and below, a $Node object 'cross1' that configures the node.

In the analysis phase we decided that the process station should consist of the following hardware:

- 1 Linux PC
- 1 rack with 16 slots
- 1 card with 16 digital inputs (Di channels).
- 1 card with 16 digital outputs (Do channels).

The rack and the cards are configured in a manner much like what you would do physically. You have a node, place the rack in the node, the card in the rack, and the channels on each card.
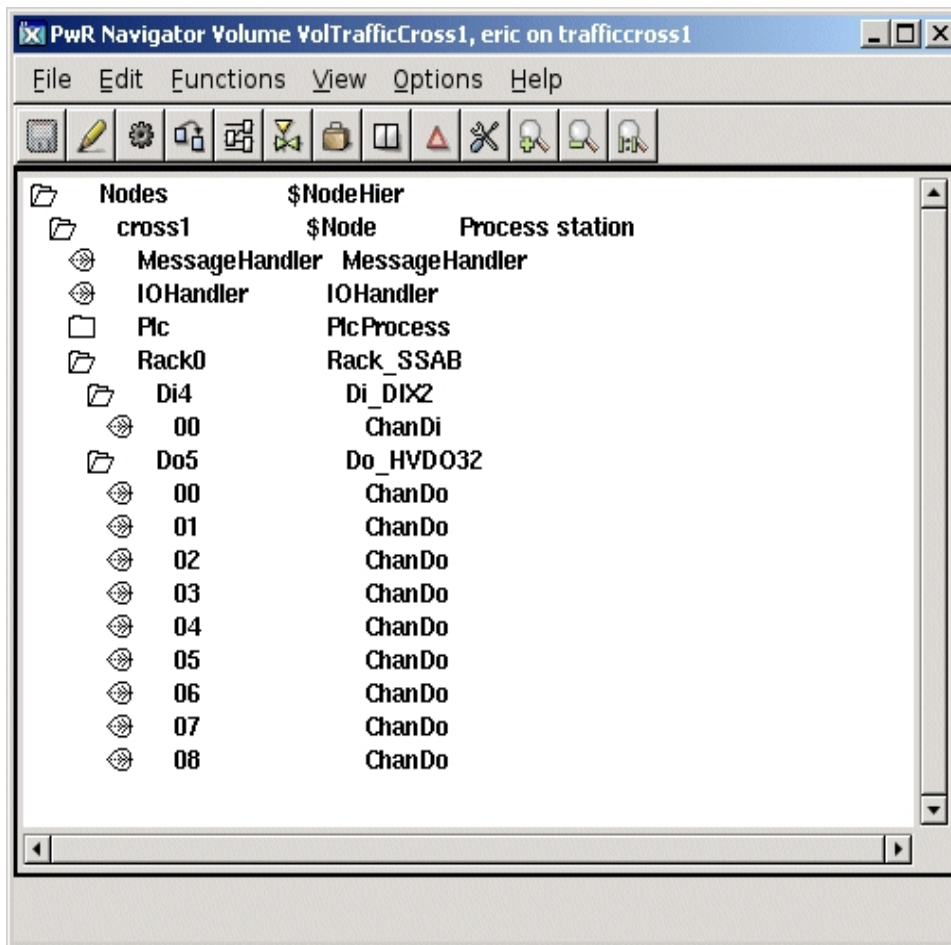
Fig Node Configuration of the Process Station

We also configure the PLC program with a PlcProcess object, and below this, a PlcThread object for each time base. We are content with one 100 ms timebase.
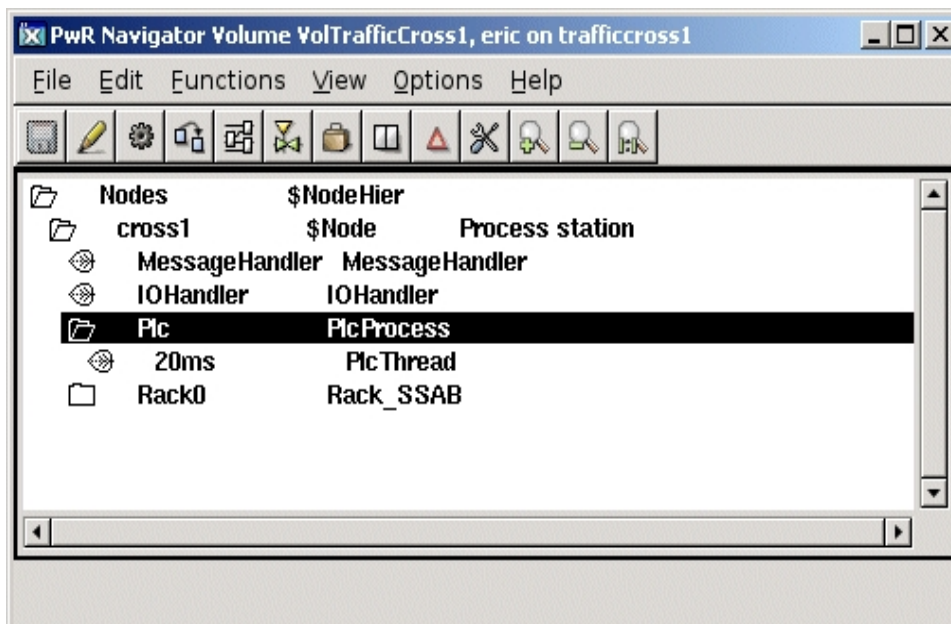
Fig Timebase configuration of the PLC program

Each object has a number of attributes that you may have to change. To give an understanding of

how to change attributes, some of the attributes in the PlcProcess object are edited below.
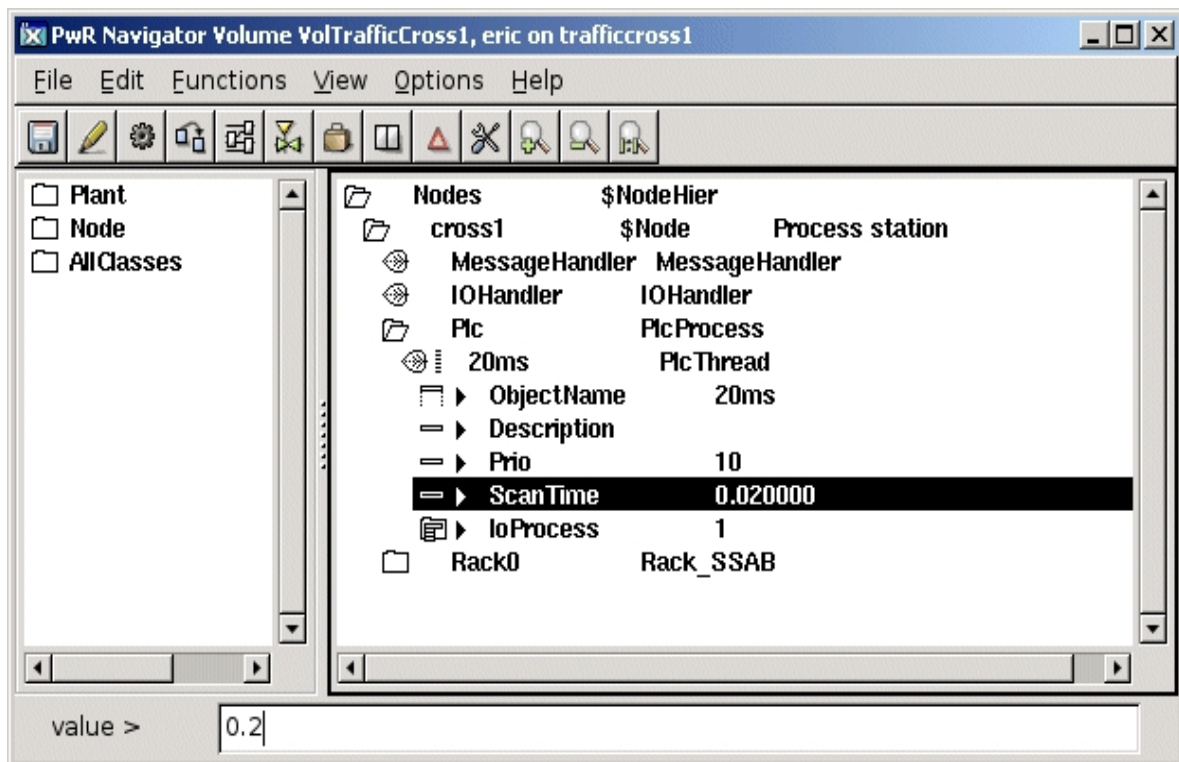
Fig Change of Attribute Value

Operator Station

The node hierarchy of the operator station is configured in the volume VolOpTrafficCross1.
Below the node object we find an OpPlace object that defines the operator place, and below this
a XttGraph object for the process graph of the operator station.
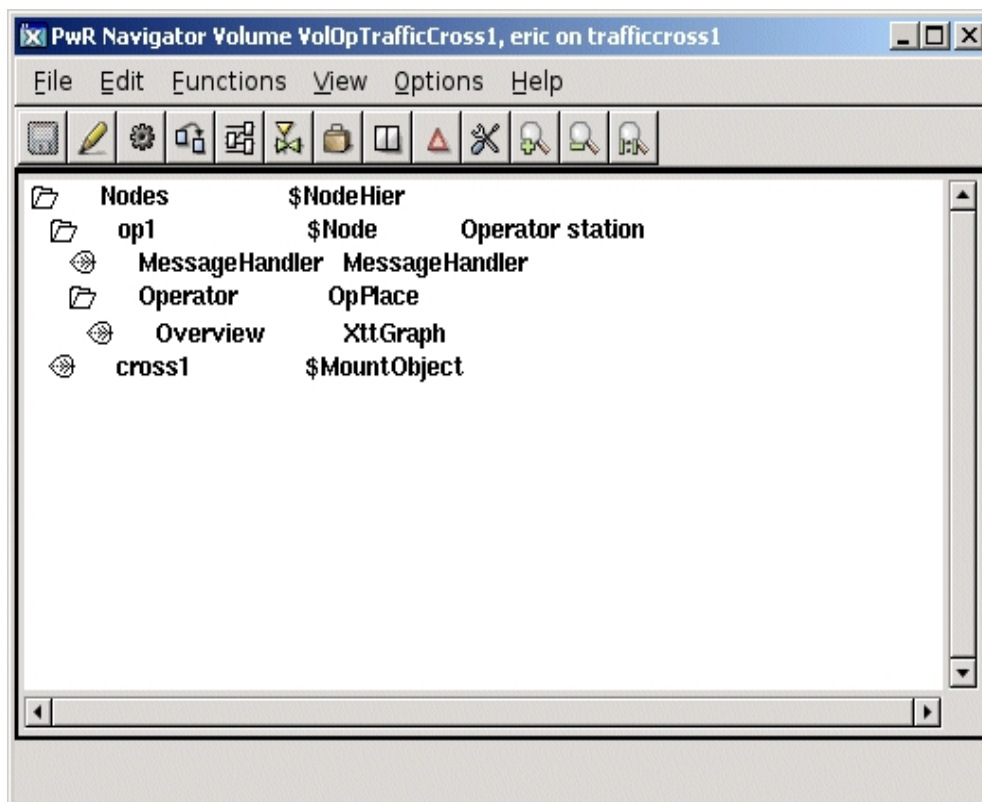
Fig The Node Configuration in the Operator Volume.

Connecting channels to signals

When you have configured the plant and the nodes, it is time to connect the logical signals to the physical channels. Each logical signal in the Plant Configuration must be connected to one channel in the Node Configuration; a Di to a ChanDi, a Do to a ChanDo, an Ai to a ChanAi and an Ao to a ChanAo , etc.

You can see the connection as a representation of the copper cable between the components in the plant, and the channel in the I/O rack. In the figure below there is a cable between the switch and channel 0 in the Di-card. As the Di-signal ServiceSwitch is representing the switch and Di-channel Di4-00 is representing the channel, we have to make a connection between these two objects.
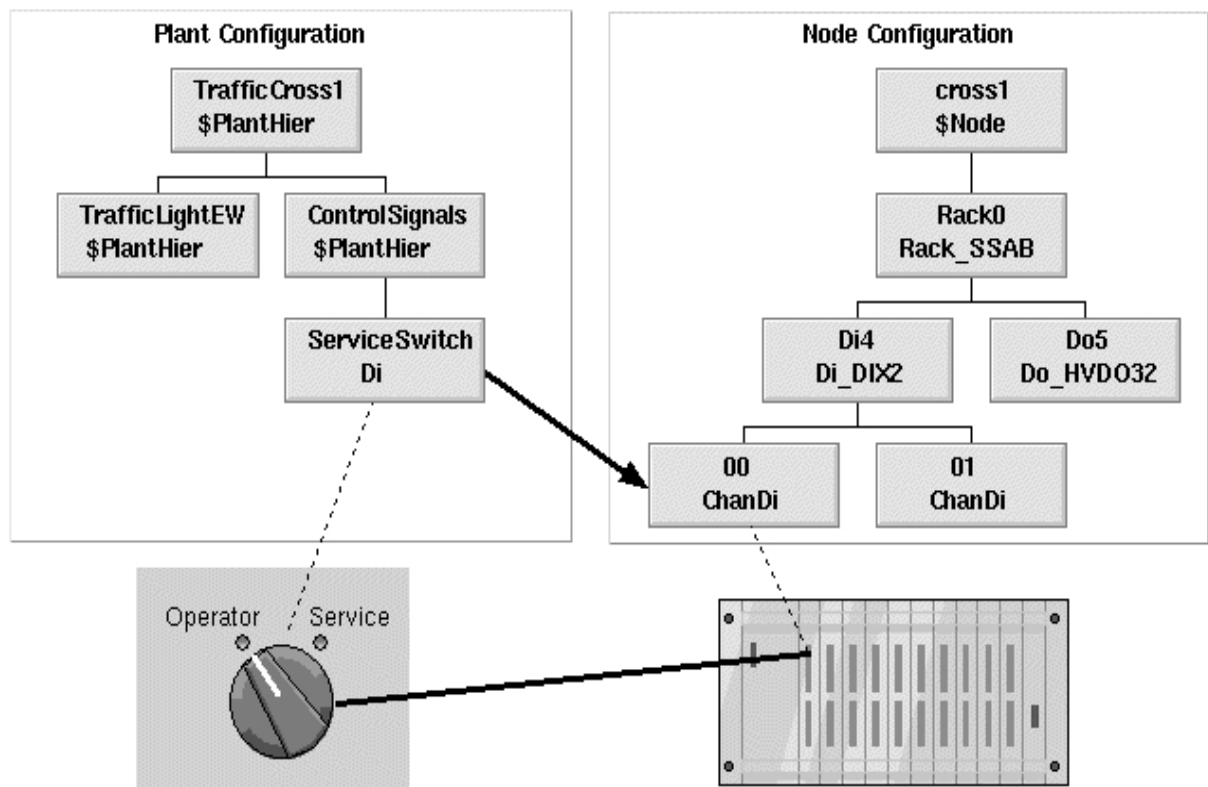
Fig Connection between a signal and a channel
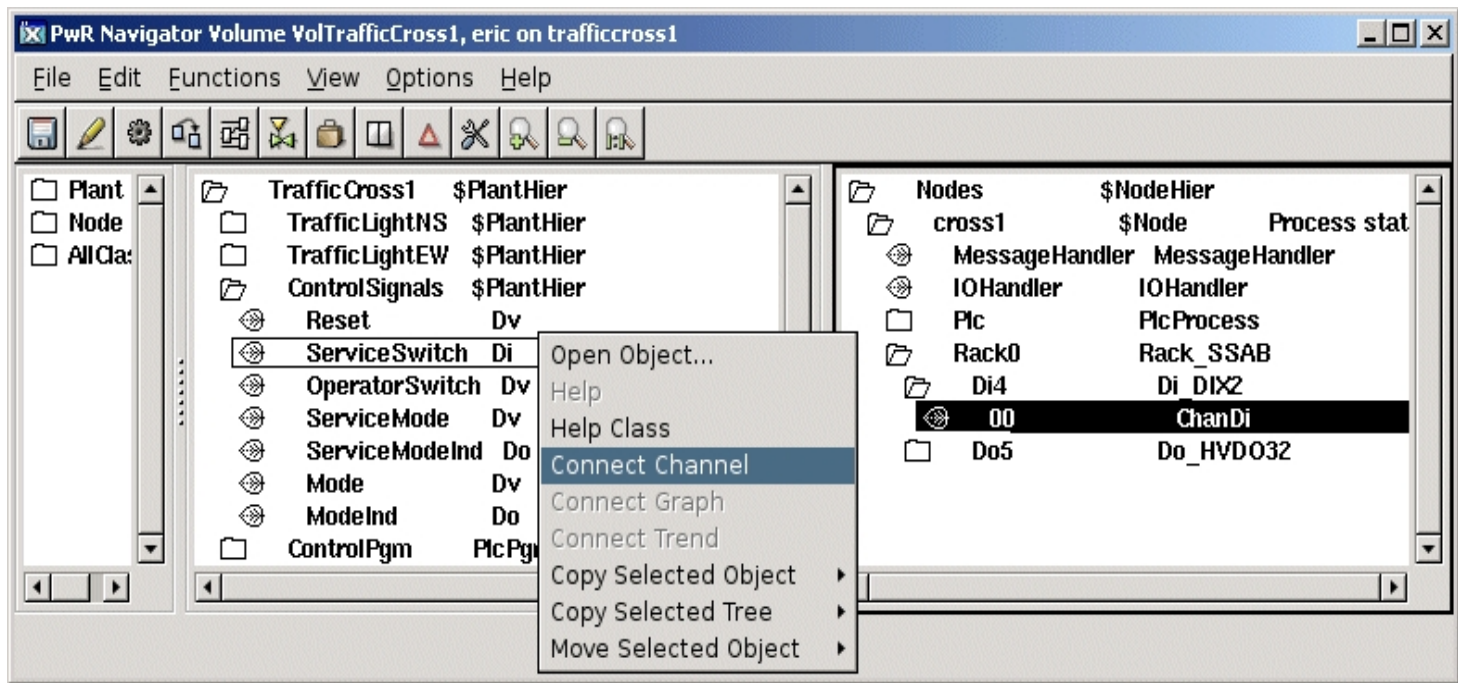
Fig Connect a signal to a channel

5.5 PLC program

We use the Graphical PLC Editor to create PLC programs.

However first we must connect the PlcPgm object to a PlcThread object in the node hierarchy.
This states which timebase the PLC program is executed on.
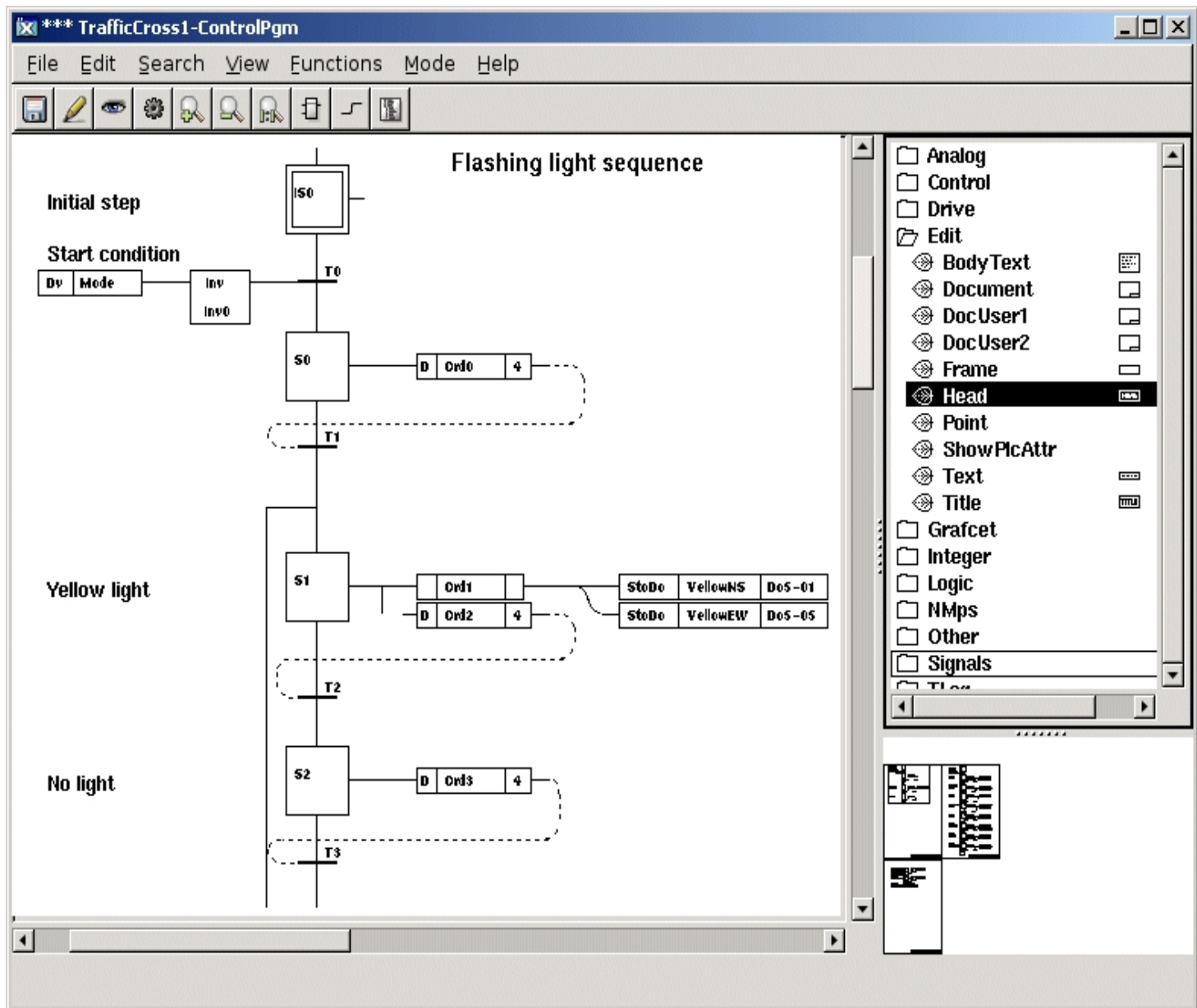
Fig The Graphical PLC Editor

We will use the PLC Editor to create a sequential control program for the traffic lights. There are two ways to solve the problem concerning the two operating modes for a traffic light, normal and flash:

1. Use one Grafcet sequence with conditional branches, i.e. one branch for the normal operating mode sequence and one for the flash operating mode sequence.
2. Use two separate Grafcet sequences with different start conditions.

Here we choose to use the second alternative. In chapter 4, Graphical PLC Programming a more detailed description of Grafcet and sequential control can be found.

Grafcet programs are based on activating and deactivating a number of steps in sequence. In linear sequences only one step at a time can be active. To each step you tie a number of orders that are to be executed when the step is active. This can be e.g. to set (with a StoDo object) a digital output signal, which turns on a lamp. The PLC programs thus control the logical signals.

Initial step

IS0

Start condition

| Dv | Mode |

Inv
Inv0

T0

S0 — | D | Ord0 | 4 |

T1

Yellow light

S1 — | | Ord1 | | — | StoDo | YellowNS | Do5-01 |
        | D | Ord2 | 4 | | StoDo | YellowEW | Do5-05 |

T2

No light

S2 — | D | Ord3 | 4 |

T3
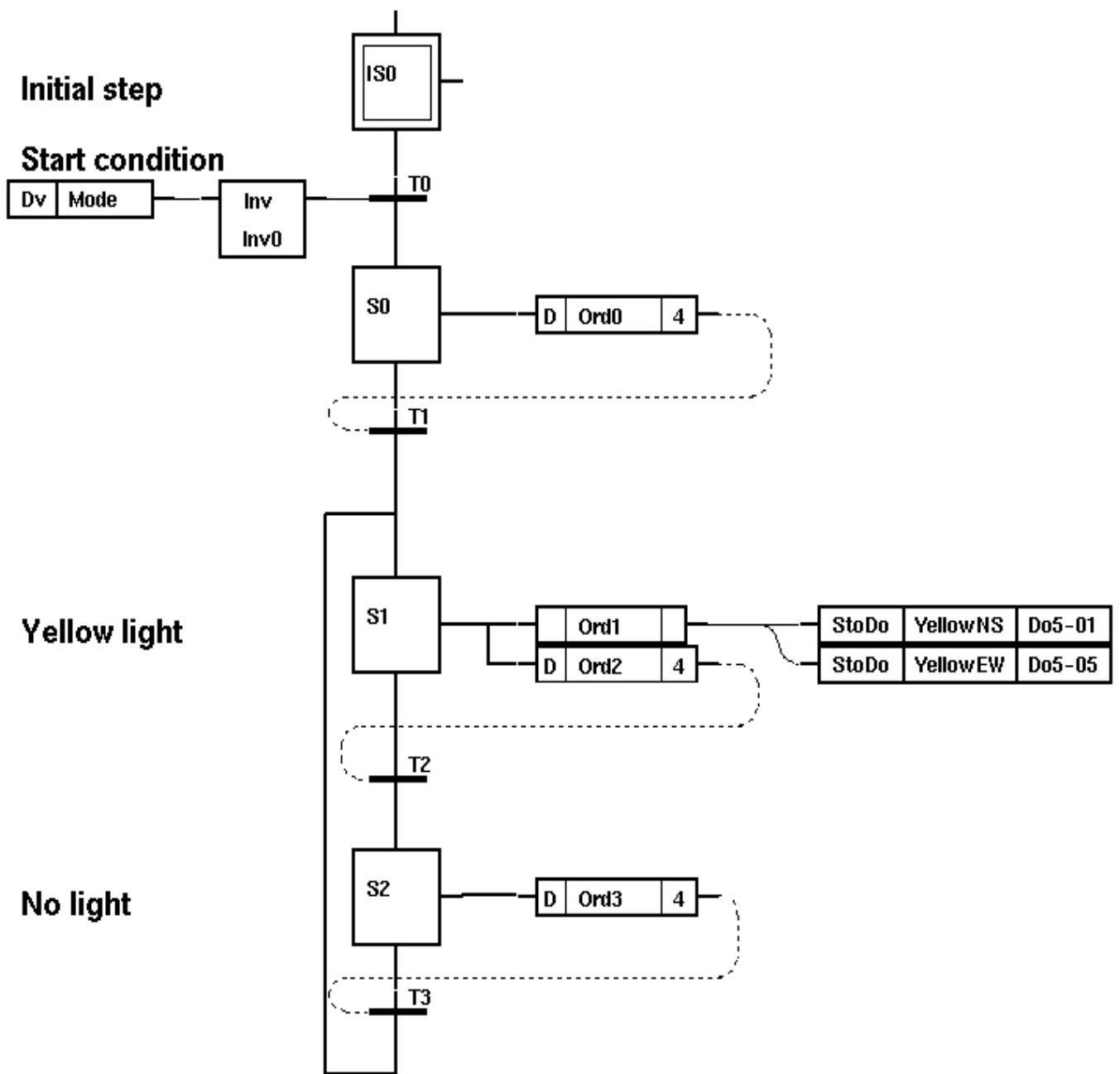
Fig The Flashing Light Sequence

This is the sequence that will be executed when you want the lights to flash yellow.

The start condition for this sequence is inverted in relation to the start condition for the normal operating mode sequence. This implies that the two sequences cannot execute at the same time.

Initial step

IS1

Start condition

Dv	Mode

T4

Both red

S3

	Ord4	
D	Ord5	120

StoDo	RedNS	DoS-00
StoDo	RedEW	DoS-03

T5

NS yellow, red
EW red

S4

	Ord6	
D	Ord7	10

StoDo	RedNS	DoS-00
StoDo	YellowNS	DoS-01
StoDo	RedEW	DoS-03

T6

NS green
EW red

S5

	Ord8	
D	Ord9	120

StoDo	GreenNS	DoS-02
StoDo	RedEW	DoS-03

T7

NS yellow, green
EW red

S6

	Ord10	
D	Ord11	10

StoDo	YellowNS	DoS-01
StoDo	GreenNS	DoS-02
StoDo	RedEW	DoS-03

T8

Both red

S7

	Ord12	
D	Ord13	120

StoDo	RedNS	DoS-00
StoDo	RedEW	DoS-03

T9

NS red
EW yellow, red

S8

	Ord14	
D	Ord15	10

StoDo	RedNS	DoS-00
StoDo	RedEW	DoS-03
StoDo	YellowEW	DoS-05

T10

NS red
EW green

S9

	Ord16	
D	Ord17	120

StoDo	RedNS	DoS-00
StoDo	GreenEW	DoS-06

T11

NS red
EW yellow, green

S10

	Ord18	
D	Ord19	10

| StoDo	RedNS	DoS-00
StoDo	YellowEW	DoS-05
StoDo	GreenEW	DoS-06

T12

Fig The Normal Sequence

The program for the normal operating mode is based on a traffic light following the sequence:

	North-South	West-East
1	Red	Red
2	Red, Yellow	Red
3	Green	Red
4	Yellow, Green	Red
5	Red	Red
6	Red	Red, Yellow
7	Red	Green
8	Red	Yellow, Green
9	Back to step 1	

The program starts in the initial step. If the start condition is fulfilled, step S1 will become active and the red lamps are turned on. After a certain time, step S1 will become inactive and step S2 will become active, and a yellow lamp will also be turned on, and so on. When step S8 has been active for a certain time, it will be deactivated, and the initial step is once again activated.
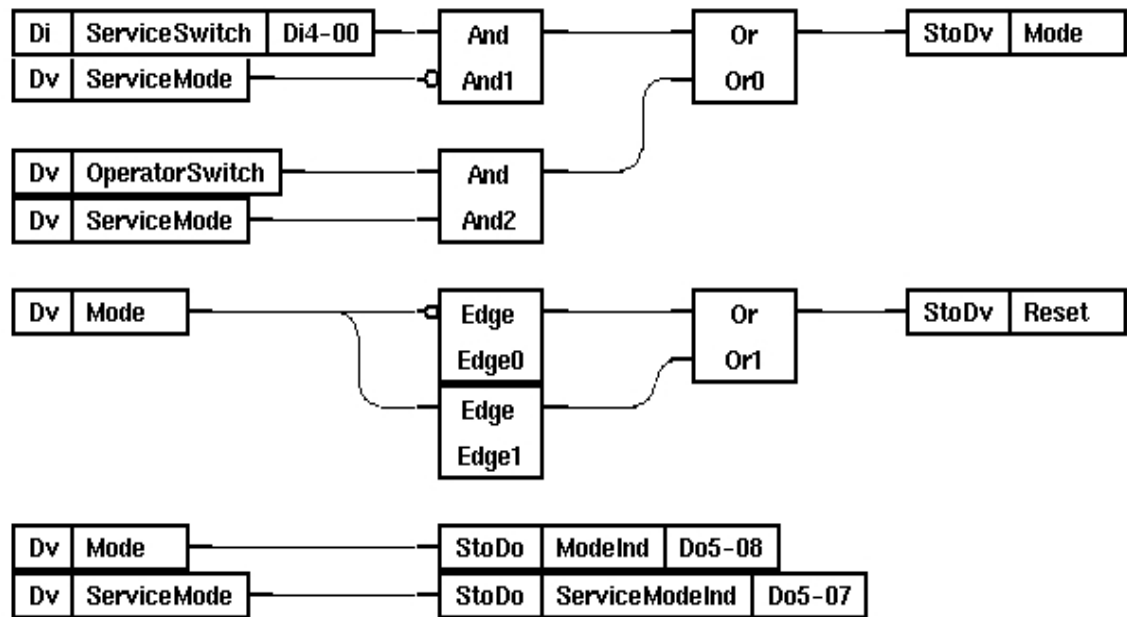
Fig Trigger Signals

The program above shows the logic that controls different operating modes.

At the very top to the right you set the Dv signal "Mode". If this is set to a logical 1, the sequence for the light's normal operating mode will be run, otherwise the sequence for flashing lights will be run.

The Dv signal "Reset" will be set to a logical 1 during one execution cycle when the signal Mode changes value. This implies that the two Grafcet sequences will return to the initial step. The chosen sequence will be executed again when Reset is set to a logical 0.

The PLC programs you have created must be compiled before they can be executed on a process station.

5.6   Plant Graphics

Plant graphics are often used as an interface between the operator and the process. Plant graphics are created with the Plant Graphics Editor.
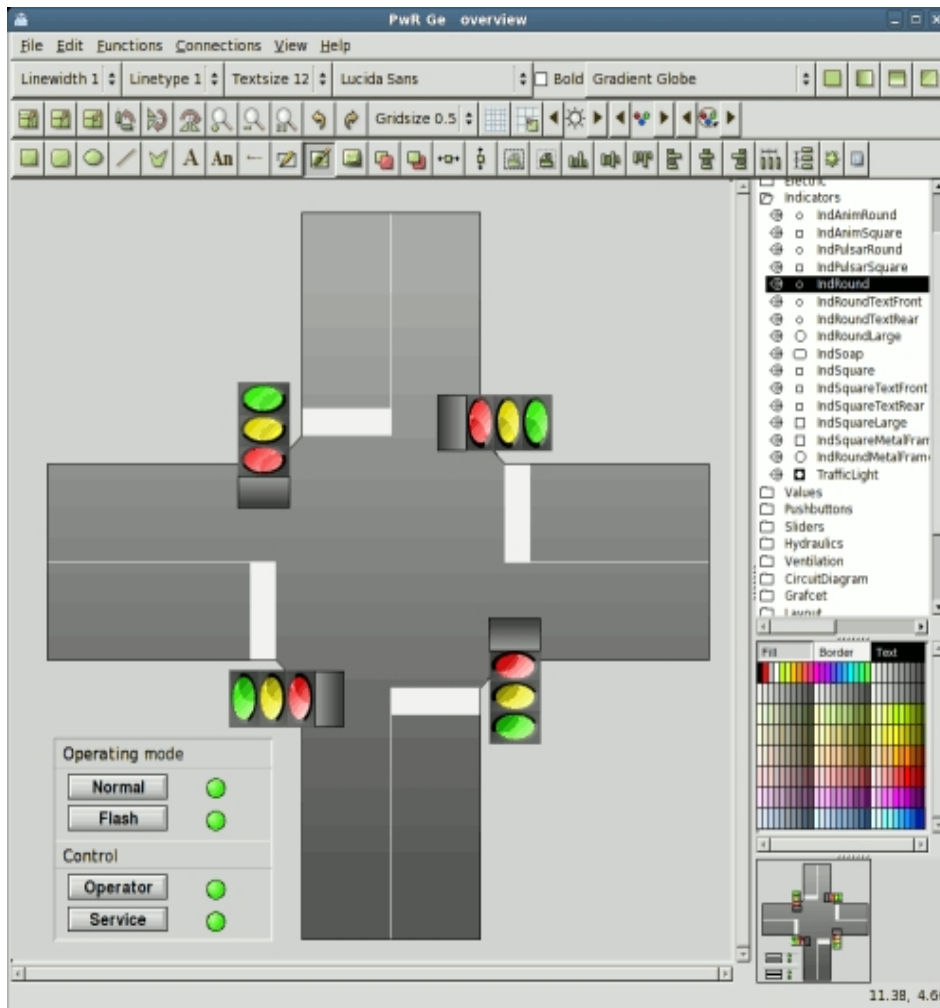
Fig The Plant Graphics Editor

Plant graphics can contain dynamics, which are connected to the logical signals, e.g.:

- Text that becomes visible when a signal reaches a certain value
- Graphical objects that change color when a signal reaches a certain value
- Graphical objects that become invisible when a signal reaches a certain value
- Graphical objects that move depending on the value of a signal

You can also place push buttons in the plant graphics, which the operator can use for changing values of digital signals. To change analog signals you use an input entry field.

In our example we choose to make a plant graphics, showing a road crossing, where the traffic lights (red, yellow, and green) are dynamic as shown in figure. How to create the plant graphics is described in chapter 5 Creating Plant Graphics .
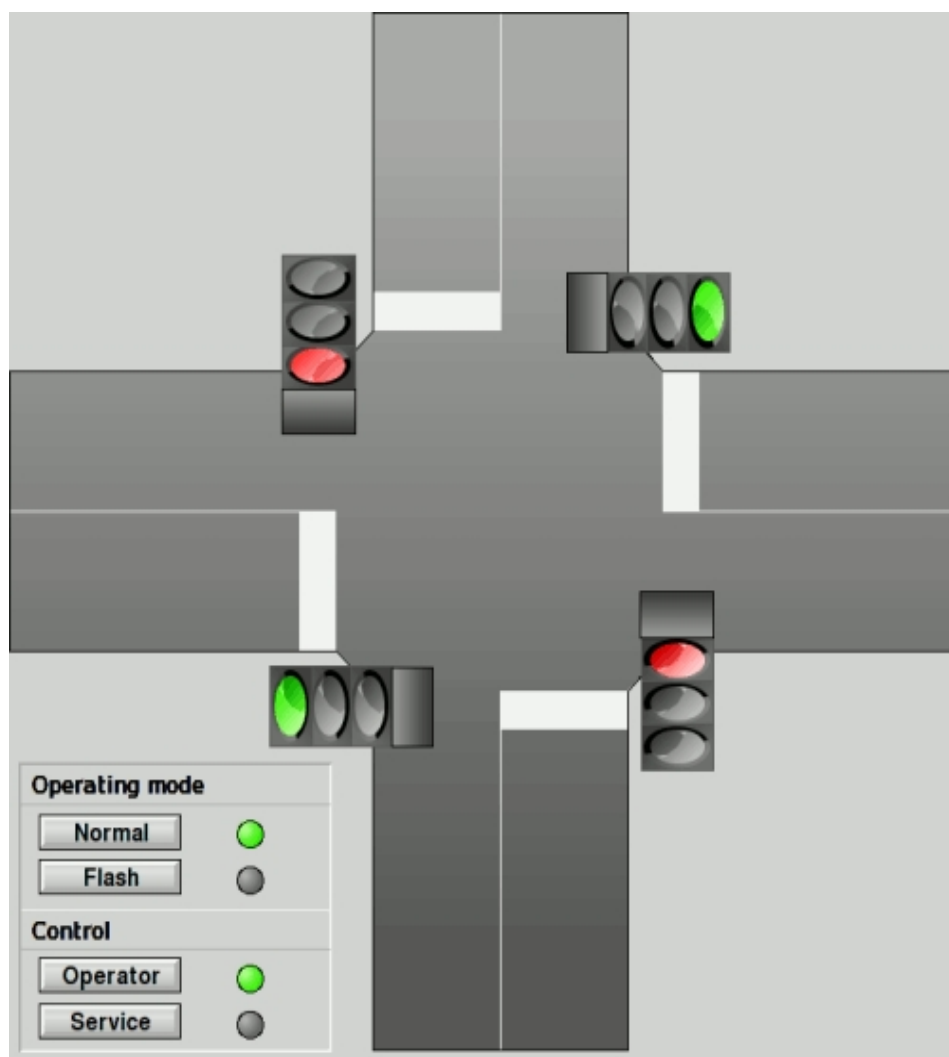
Fig The Plant Graphics for the Intersection

6 Create a project

Installation of development environment

Before you can begin to work with ProviewR you have to install the ProviewR development environment. There are a number of packages for different Linux distributions available, and if there is no package for the desired distribution you can also download the ProviewR source and build from sources.

The installation package is named with the version number in the name, e.g. pwr47. This makes it possible to install several versions side by side, which is an advantage when you have plenty of projects in production running at different versions.

You will find more information about the installation on the Download page on www.proview.se.

setup script

At the installation the Linux user 'pwrp' with password 'pwrp' is created. By logging in as pwrp you can start ProviewR by clicking on the ProviewR icon on the screen.

If you want to run ProviewR as another user, you have to initiate ProviewR at login. Insert the following line into the file .bashrc in the home directory

```
source /src/pwrp/adm/db/pwr_setup.sh
```

If you have several users with common ProviewR projects, you have to make sure that they have write access to files in the projects. One way to achieve this is to set umask to 002, and let all users have the same group, e.g. pwrp.

Users

In the Case Study above, we learned how to create users belonging to different system groups. This is described in the Administration chapter. At installation of the development package, a user database is included containing the systemgroup 'common' with five users, pwrp, op1, op2, op3 and op4. pwrp is a development user and has system privileges in runtime. The op users are operators and have operator privileges in runtime. These users work for many applications, and we will settle with them for the moment.
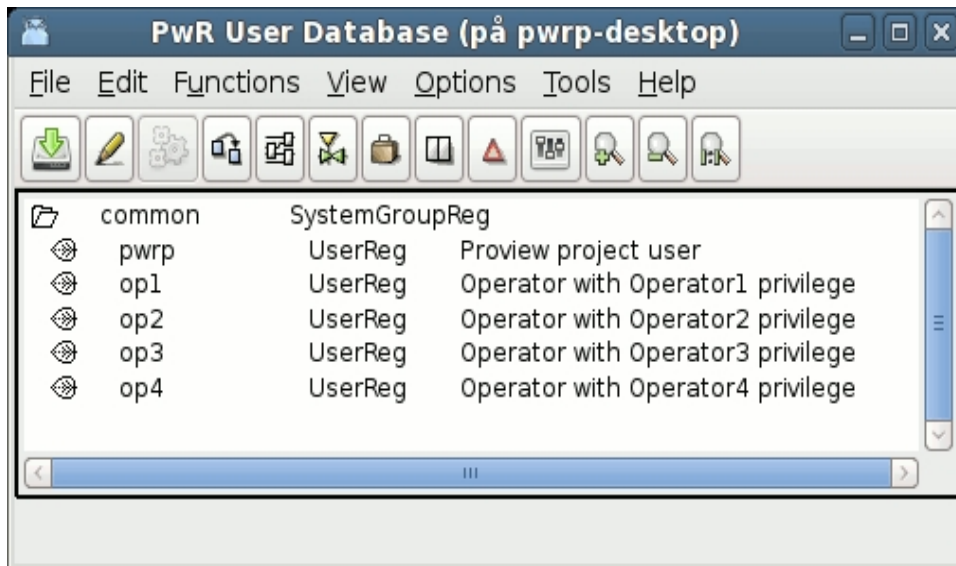
Users included in the installation

Note that when building ProviewR from sources, you have to create users before creating a project.

Register volumes

As we also saw in the Case Study you can register volumes in the GlobalVolumeList. This can be done later, by letting the configuration guide for the project register the volumes and fetch the next free volume identity. We choose this way for the moment, but if you have several independent development stations you should handle the registration manually to avoid collisions of volume identities. If you have a larger plant with plenty of volumes, we also recommend that you assign the volume identities manually and group them in a suitable way.

Create a project

To create a project you enter the project list of the administrator, by clicking on the ProviewR icon, or by the command

```
> pwra
```

pwra is defined as 'wb -p pwrp pwrp', i.e. it is logging in as user pwrp. If you have defined other users for development, you have to redefine 'pwra' or use the 'wb -p' command directly. wb takes user and password as arguments.

In the project list you enter edit mode and create a ProjectReg object on the top level or under a hierarchy object. In the ProjectReg object you specify project name, ProviewR version and path for the project. When you save, the project directories are created.

How to create a project is described in the Getting Started Guide. We recommend that you go trough the sections for creating the project and configuring the directory volume before continuing.

7   Directory Volume Configuration

Open a project

When the project is created, it is found in the administrator project tree. You open a project by activating 'Open Project' in the popupmenu for a ProjectReg object.

You can also use the 'sdf' command to attach to a project. sdf takes the project name as argument

```
> sdf trafficcross1
```

The directory volume is now opened by the command

```
> pwrs
```

pwrs is defined as 'wb pwrp pwrp', i.e. it is logging in as user pwrp. If you have defined other users for development, you have to redefine 'pwrs' or use the 'wb' command directly. wb takes user and password as arguments (and also volume as third argument).

If the volume is empty, a guide to help with the configuration is started. To create a simple project with a node and a volume you just have to press the 'Next' button.

The guide looks for volumes registered for the project. If there are no volumes registred it suggests volumes with suitable volume names and free volume identities, and registers the volumes if the suggestions are approved. The guide also creates all the configuration objects in the directory volume and inserts suitable data into them.

If you later will expand the systems with more nodes and volumes, it is good to have some knowledge of how the configuration is done, thus we describe how to configure the volume manually.

The Configuration Editor

The configuration editor displays two windows, and for the DirectoryVolume, the left shows the volume configuration, and the right the node configuration.

Configure Volumes

First we configure all the root volumes, sub volumes and class volumes in the project. This is done in the volume window in the directory volume. We start by creating a RootVolumeConfig object the configures a root volume.

- Enter the edit mode from the menu 'Edit/Edit mode'. Now the palette is visible to the right in the window, and maps can be opened with a click on the map symbol or a double click on the text.
- Open the Volume map and select the 'RootVolumeConfig' class.
- Click with MB2 in the volume configuration window, and the object is created.
- Select the object and open the object editor from the menu 'Functions/Open Object'.
- Select ObjectName and activate 'Functions/Change value' in the object editor menu.
- Enter the name of the object. The name should be the same as the name of the volume.

- Close the object editor.

Create the RootVolumeConfig objects for the other rootvolumes of the project. For the following objects you can control the position of the object. If you click with MB2 on the object name of an object, the new object will be a sibling to this object. If you click on the leaf or map symbol, the object will be a child.

Also subvolumes and classvolumes are configured in a similar way with SubVolumeConfig and ClassVolumeConfig objects.

It is also possible to display the attributes of an object directly in the configuration editor:

- Press Shift and click MB1 on the object to open the object
- Select an attribute and activate Functions/Change value to modify a value.

Configure the nodes

In the right window, the nodes in the project are configured. You group the nodes by which QCOM bus they communicate on. We create two BusConfig objects, one for the production nodes and one for simulation. In the BusNumber attribute the busnumber is defined.

As children to the BusConfig object, the NodeConfig objects are created, one for each process and operator station. When the NodeConfig objects are created, some additional objects are created

- a RootVolumeLoad object that states the rootvolume to load when the runtime environment is
  started on this node. The name of the object should be equal to the name of the root volume.
- a Distribute object that configures which files are copied from the development environment
  to the process or operator station.

Open the NodeConfig object an enter nodename, operating system and ip address.

Below the BusConfig object for the simulation bus it is suitable to place a NodeConfig object for the development station, and below this, a RootVolumeLoad that states the volume of the process station you are going to work with first. In this way you can start the volume in runtime and test it on the development environment. State the name, operating system and ip address of the development station in the NodeConfig object.

System object
Create also a $System object in the node configuration window. The system object has the attributes SystemName and SystemGroup.

- The system name in this state is often equal to the project name.
- The system group attribute makes the system a member of a system group in the user database,
  which defines the users for the system. Once the system object is created you have to state
  a valid username and password when entering the workbench.

Save
Save the session from the menu 'File/Save'. If the configuration passes the syntax check, you will receive a question if you want to create the configured volumes. Answer Ok to these questions and create the volumes.

If the volume selection window is opened now, 'File/Open' in the menu, all the configured volumes are displayed. The next step is to configure a RootVolume.

8   Configure a Root Volume

A root volume is opened from the volume selection window. Select the volume and click on the Ok button. This will start the configuration editor for the root volume. As for the DirectoryVolume it is split in two windows, but this time, the left window shows the plant configuration and the right the node configuration.

Plant Configuration

The Plant Configuration describes the different plants that you can find in the ProviewR system. A plant is a logical description of e.g. a production process, functions, equipment, that is to be controlled, supervised, etc.
See an example of a plant configuration

$PlantHier Object

The top object in the plant hierarchy is the $PlantHier object. This object identifies the plant or parts of it.

The $PlantHier object is used to group objects and to structure the plant. This object can, for instance, be used to group signal objects.

Signal Objects

The signal objects define logical signals, or points, representing a quantity or value somewhere in the process; as contrast to the channel objects which define physical signals. The signal objects are generic, i.e. they can be used with any I/O-system.

There are some classes of signals that cannot be connected to hardware signal, i.e. the Dv, Iv, Av and Sv objects (DigitalValue, IntegerValue, AnalogValue and StringValue). These objects are used to store logical values, integer value, real numbers and strings respectively.

The actual value of the signal is defined by the attribute ActualValue.

At present the following signal objects are available:

Ai   Analog input.
Ao   Analog output.
Av   Analog value.
Ii   Integer input.
Io   Integer output.
Iv   Integer value.
Di   Digital input.
Do   Digital output.
Po   Pulsed digital output.
Dv   Digital value.
Co   Counter input.
Sv   String value.

Note! The PLC program can read signals placed on remote nodes, but cannot write to them.

PlcPgm Object

The PlcPgm object defines a PLC program. It is possible to have several PLC programs in a plant. The following attribute must be given a value:

- ThreadObject indicates the plc thread where the program is executed. It references a PlcThread object in the node configuration.
- If the program contains a Grafcet sequence, the ResetObject must be given. This is a Dv, Di or Do that resets the sequence to its initial state.

Backup Object

The Backup object is used to point out the object or attribute, for which the backup will be made. It is also indicated whether storing will take place with fast or slow cycle time.

MountObject

The MountObject mounts an object in another volume. The attribute Object specifies the mounted object.

Node Configuration

The Node Configuration defines the nodes of your ProviewR system. The nodes are named and their contents specified.
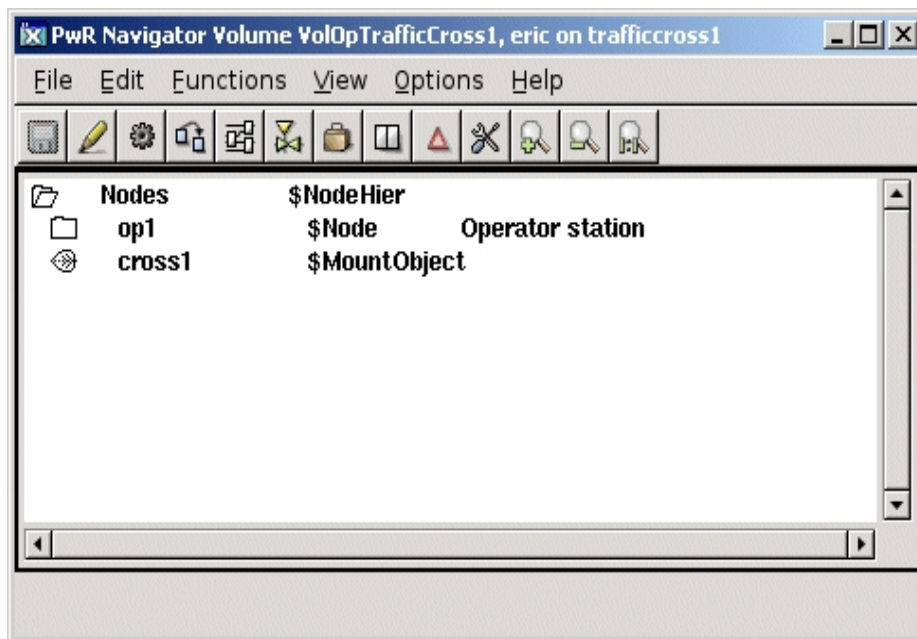
Fig Node Configuration

$NodeHier Object

The node hierarchy object is used to group objects in the Node Configuration. This object is of the $NodeHier class. The object can be used to group for instance $Node objects or XttGraph objects.
See $NodeHier in Object Reference Manual

$Node

To define the nodes of the system, you use node objects. The node object is of the $Node class. When the node object is created, a number of server and operator objects are created.

See $Node in the Object Reference Manual

I/O Objects

The configuration of the I/O system is dependent of which type of I/O system you use. ProviewR has a modular I/O that can handle different types of I/O systems: rack and card systems, distributed bus systems, or systems connected with some network.

The modular I/O is divided in four levels: agent, rack, card and channel.

Rack and Card System

We will take the PSS9000 as an example of a rack and card I/O. The system consists of analog and digital input and output cards that are mounted in racks. The rack is connected via a bus cable to a busconverter card in the computer that converts the PSS9000 bus to the computers PCI bus.

In this case, the agent level is not used, so the $Node object works as an agent. The rack level is configured with SSAB_Rack objects that are placed below the $Node object, one for each rack in the system. The cards are configured with objects below the rack object, that are specific for different kind of IO cards. For PSS9000 there are card objects like Ai_Ai32uP, Ao_Ao4uP, Di_DIX2 and Do_DVDO32. Below a card object, channel objects are placed, one for each channel on the card.

Common for the different I/O systems are the channel objects, that define the input or output channels of a card or module. There are some different types of channels.

ChanDi	Digital input
ChanDo	Digital output
ChanAi	Analog input
ChanAit	Analog input with conversion of the signalvalue from a table
ChanAo	Analog output
ChanIi	Integer input
ChanIo	Integer output
ChanCo	Counter input
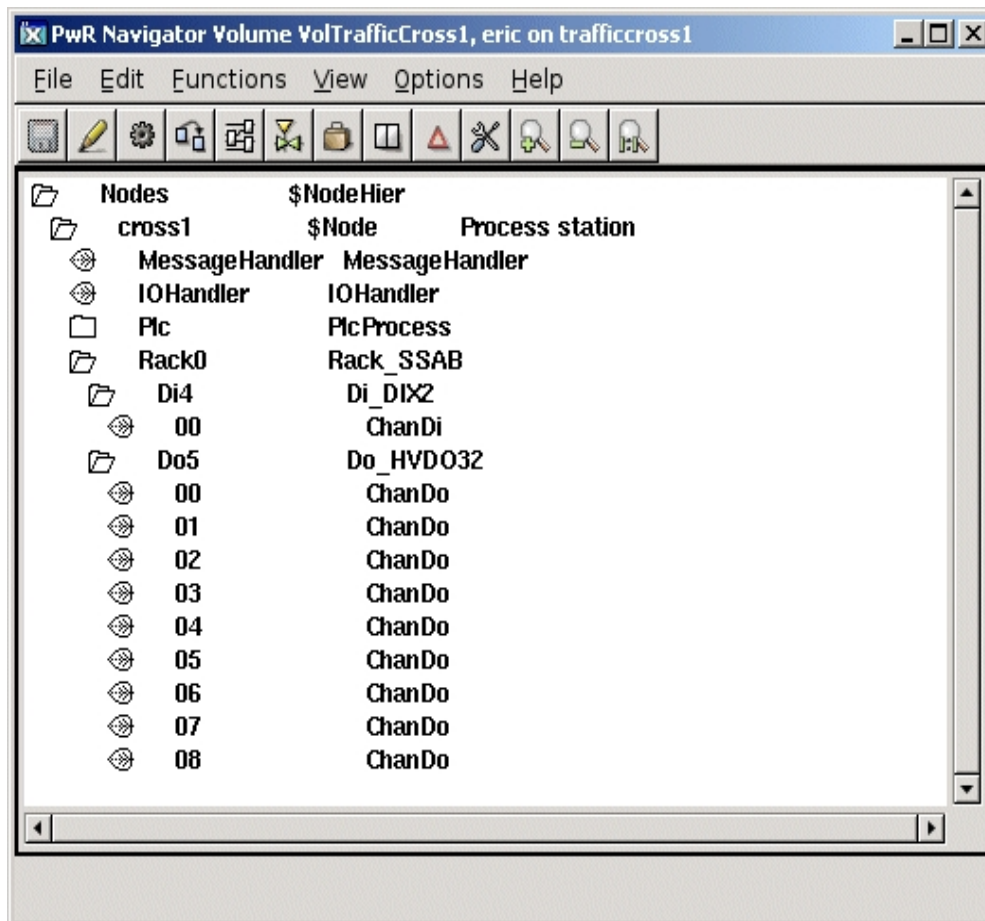
Fig I/O configuration

Distributed I/O
As an example of distributed I/O we choose profibus. In this case, all the four levels are used.
In the PCI bus of the computer, there is a mastercard that communicates with a number of slaves
on the profibus circuit. The mastercard is configured with a Pb_Profiboard card on the agent
level. Below this, we find the different slaves configured with Pb_DP_Slave objects. They
represent the rack level. Below the slave objects there are module objects of type Pb_Ai,
Pb_Ao, Pb_Di, Pb_Do etc, that are placed on the card level. Below the module objects finally,
the channels are configured with the channel objects ChanDi, ChanDo etc.

Process and thread for I/O objects
I/O objects of the card level, often contains the attributes Process and ThreadObject. Which
process that shall handle the card is defined in Process.

The card can be handled by the PLC program, that is, reading and writing is made synchronized
with the execution of the PLC. You can also specify a thread in the PLC that should handle the
card, i.e. which timebase is used to read or write the card (the PlcThread attribute).

The card can also be handled by the rt_io process, that usually has a lower priority than the
PLC, and that is not syncronized with the PLC. Certain types of analog inputcards that take
some time to read are with advantage handled by this process.

You can also write an application that handles reading and writing of cards. There is an
API to initiate, read and write the cards. This is useful if the reading and writing of a
card has to be syncronized with the application.

MessageHandler Object

The MessageHandler object configures the server process rt_emon, that handles the supervision objects (DSup, ASup, CycleSup). When an event is detected by the server, a message is sent to the outunits that have interests in this specific event.

In the object is indicated for example the number of events that are stored in the node. The object is automatically created below a $Node object.
See MessageHandler in the Object Reference Manual

IOHandler object

IOHandler configures properties for the I/O handling.

- ReadWriteFlag specifies whether to address physical hardware or not.
- IOSimulFlag indicates whether to use the hardware or not.
- The timebase for the rt_io process, i.e. the process that handles slower types of I/O cards
  that are not suitable to be handled by the PLC.

In the production system for a process station, ReadWriteFlag is set to 1 and IOSimulFlag is set to 0. If you want to simulate the process station, for example on the development station, ReadWriteFlag is set to 0 and IOSimulFlag is set to 1.

The IOHandler object is created automatically, when creating a $Node object.
See IOHandler in the Object Reference Manual

Backup_Conf Object - Configuration Object for Backup

Sometimes it may be desirable to have a backup of a number of objects in your system. In that case you place a backup configuration object, Backup_Conf, under the node object. The backup is carried out with two different cycles, one fast cycle and one slow.

In order to indicate which objects/attributes that should be backed up you use backup objects.
See description of the Backup object
See Backup_Conf in Object Reference Manual

Operator Place Object

To define an operator place you place an object of the OpPlace class under the $Node object.

The following attributes must be given values:

- UserName is a name of a ProviewR user, defined in the UserDatabase. The privileges of the
  user determines the access to the system.
- MaxNoOfEvents indicates the number of events that the operator's event list can hold at
  the same time.
- EventSelectList indicates the object hierarchies in the Plant Configuration, from which the
  operator will be receiving events.

If we look at the figure below, that illustrates the plant A, and assume that we want to receive events only from the encircled objects, we state 'A-C' as an alternative in the select list. This choice means that we will be receiving events from the supervised object C, and from all supervised objects that have C as their parent.
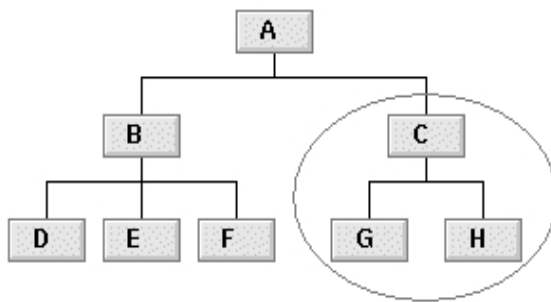
Fig EventSelectList example 1

Another example:

We look at the figure below, which illustrates the plant TrafficCross1. If you want to receive all events from the plant TrafficCross1, you state TrafficCross1 as an alternative. TrafficC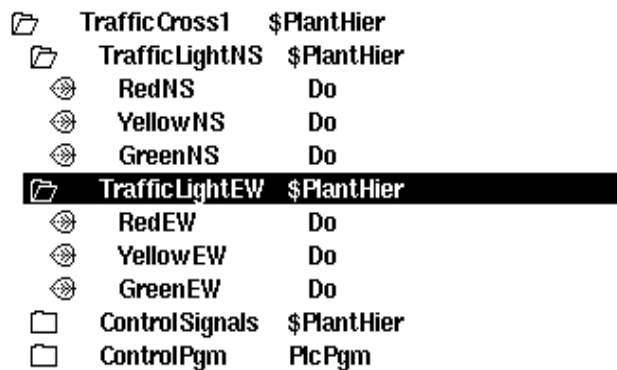ross1 handles two traffic lights, TrafficLightNS and TrafficLightWE. Let us say that we want events only from TrafficLightNS. In that case we state 'TrafficCross1-TrafficLightNS' instead of TrafficCross1.

Fig EventSelectlist example 2

If you want to receive messages from the CycleSup object that supervises the plc threads, you must also state the hierarchy name of the $Node object in EventSelectList.

In FastAvail you specify the complete hierarchy name of the XttGraph object, which will be possible to start from the graphics buttons of the Operator Window. You can have 0 - 25 push buttons. Buttons that are not used become invisible.

See OpPlace in Object Reference Manual
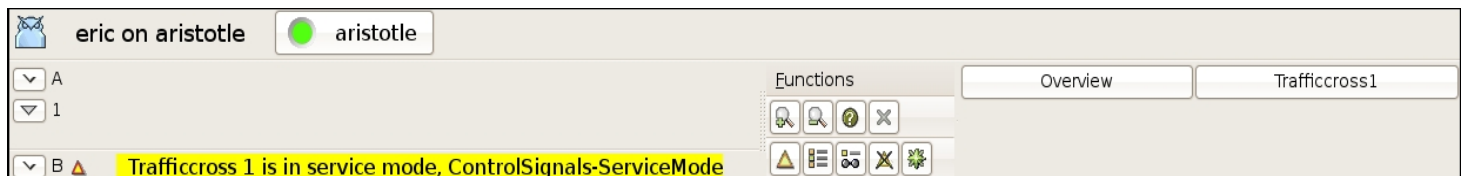
Fig Operator Window

The Plant Graphics Object - the XttGraph Object

In order to be able to show plant graphics that are unique to the project in the operator station, you must configure XttGraph objects. These objects define, for instance, what the files with the plant graphics are called. The objects are referred to in the FastAvail attribute of the OpPlace object, and in the DefGraph attribute that is found in $PlantHier and signal objects and makes it possible to open the graph from the popup menu for the object.

When the object is referred to in a FastAvail you can use the possibility to execute a Xtt command from the XttGraph object. In this way, you can set a signal from a pushbutton in the operator window.

- Action. States a Ge graph to open, or an Xtt command to execute.

See XttGraph in Object Reference Manual

Multiview object

Multiview is an operator window organized as table where each cell can contain a graph, trend, sev history curve, alarm list, event list or another multview window. The example below displays a multiview window with one column and two rows. The first cell contains another multiview with two columns and tree rows, and the second cell contain an alarm window. The different alarm windows show alarms from different parts of the plant, specified by AlarmView objects. An multiview window is configured by a XttMultiView object. The Action array contains the specifications for each cell.

It is also possible to exchange a graph or curve in a cell with the 'set subwindow' command.
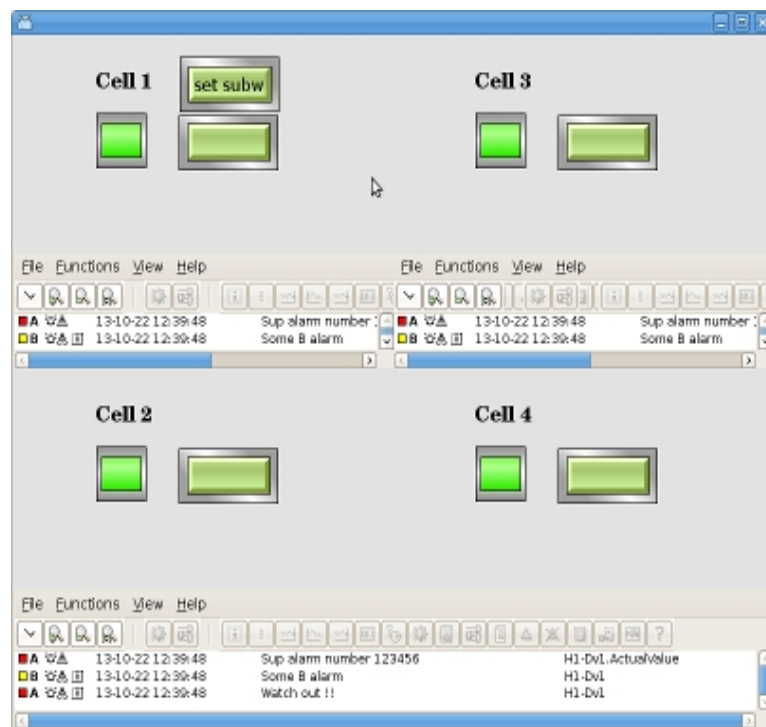
Fig Multiview window

9   Navigating the project

This chapter describes the projects structure and explains the purpose of the different directories. The chapter also describes some important files that you can use to configure and control different things in your project.

The directory structure is divided into two branches, one which contains all source files that your project consists of and the other is the build tree where all produced files are placed.

9.1   Introduction

The figure below shows a project structure of the directory tree.
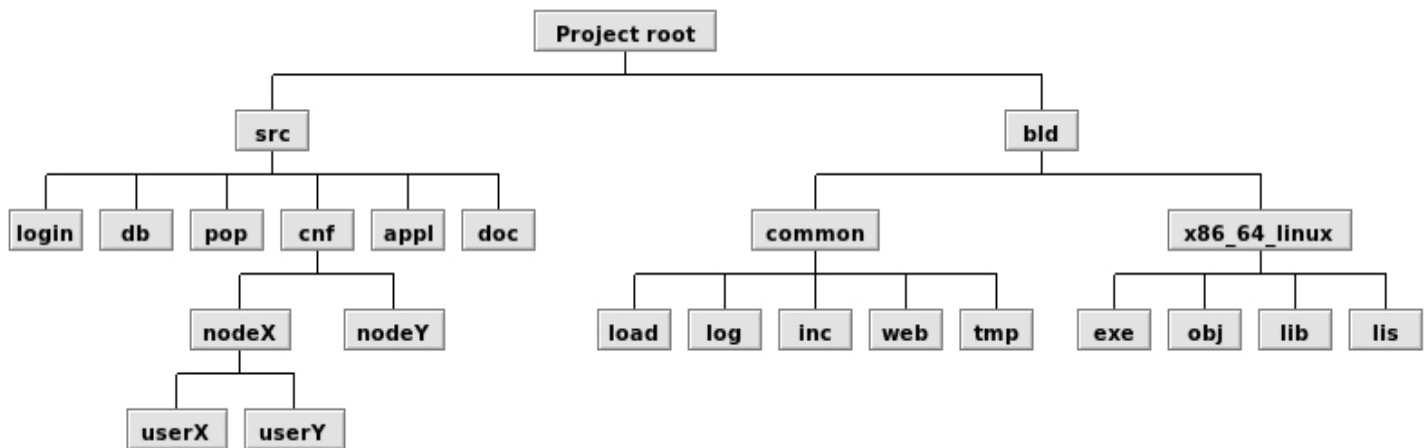
This directory structure was modified in ProviewR V4.7. The reason for this was to make it more clear what the sources and configuration files of the project are, and what is generated content. Earlier versions did not make this clear and the idea is that everything that resides in the build tree can be regenerated from the source tree.

All directories have an environment variable defined, so it will be easy to reach all directories in the project. These variables are always defined as:

```
$pwrp_<directory>
```

For example $pwrp_exe for the directory <project_root>/bld/x86_64_linux/exe.

9.2   The source tree

The source tree contains all the files that are the sources and configuration files of the projects. The top level $pwrp_src only contains subdirectories and no source or configuration files. In the following the contents and purpose of the subdirectories are described.

9.2.1 $pwrp_login

This is the directory you will start in when you move to a project with the 'sdf'-command.
Two files here are of interest.

```
login.sh
sysinfo.txt
```

login.sh is a script that is run when you go to the project. It can be used to set up project-specific environment variables and alikes.

sysinfo.txt is a text-file that will be printed in the terminal-window when you come to the project. Information of what is going on or what is done in the project can be put here.

9.2.2 $pwrp_db

This is the directory where the databases for all your local volumes will reside (including the directory volume where the project is configured). Each database resides in it's own sub-directory. This is valid if you choose to create your databases as BerkleyDB-databases. If you instead choose to have mysql-databases the databases will be created on your mysql-server.

In this directory the files for user-defined classes also reside, the UserClassVolumes. They are text-files with file-end wb_load. The user-classvolume usually has a similar name as the RootVolume in the project that uses the classes. If your RootVolume is named VolMyProject then the ClassVolume will be named CVolMyProject and the name of the file thus will be:

```
cvolmyproject.wb_load
```

9.2.3 $pwrp_pop

This is the directory where the pictures developed with the ge-editor will be stored (fileend *.pwg). Finished files should be copied to the exe-directory in the build-tree ($pwrp_exe). Also xtt-helpfiles should be developed here and copied to the $pwrp_exe-directory.
Pictures that have a corresponding XttGraph-object in the node-hierarchy of a RootVolume will automatically be copied to the $pwrp_exe-directory when that volume is built.

9.2.4 $pwrp_appl

This is the directory where you should keep source codes for your own applications belonging to the project and also for codes that you want to link with the plc-program.

One file of special interest that should be kept here is

```
ra_plc_user.h
```

This file will by default be included when you compile the plc-code. It is however included from a directory in the build-tree (<project_root>/bld/common/inc).

Note that the ra_plc_user.h is automatically generated on $pwrp_inc when the first PlcPgm is compiled. If you need to modify ra_plc_user.h, you should copy it to $pwrp_applcode and keep the original there.

All header-files located $pwrp_appl or in subdirectories and which should be included with the plc-program must be copied to the $pwrp_inc-directory (<project_root>/bld/common/inc).

If you have some small functions that you only link with the plc-program and nothing else then you typically place this code in a file called:

```
ra_plc_user.c
```

9.2.5 $pwrp_doc

This is the place where you put documentation related to the project. This can be your own produced documentation or for example DataSheet's on components existing in your plant that you might want to distribute to the operator stations.

9.2.6 $pwrp_cnf

This directory contains all configuration files for your project. Some configuration files are common for the whole project and they are placed here. Some configuration files are specific for each node in your project. Create a subdirectory here for each node that has specific configuration files. Sometimes a configuration is unique to a specific user. If this is the case, then create a subdirecty in the node-directory for that user.

Files that you should keep here is:

Configuration of Global function keys.
```
Rt_xtt
```

Configuration of menus and quick commands in rt_xtt.
```
xtt_setup.rtt_com
```

Startup file for ProviewR.
```
ld_appl_<nodename>_<bus_no>.txt
```

File to decide which libraries to link with the plc-program.
```
plc_<nodename>_<bus_no>_<plc_name>.opt
```

File to control setting of initial values when starting ProviewR.
```
pwrp_alias.dat
```

File with help texts
```
xtt_help.dat
```

All the above files are further described below.

9.3   The build tree

The build tree contains all the files needed when building the plc-program. It contains also all the files produced when you build. Files needed when building the plc-program should be in this tree, but the master for all files should be in the source tree. The idea is that the build tree should be possible to remove completely and regenerated from the source tree.

9.3.1 $pwrp_exe

This is the directory where the exe-files for plc-programs are created. The plc-program is named:

```
plc_<nodename>_<bus_no>_<version_no>
```

If you have your own applications in the project these exe-files should also be generated here.

9.3.2 $pwrp_obj

This is the directory where all object files produced during compilation should be placed.

Object-files for the plc-program are automatically placed here. The object file for a PlcPgm-object placed in the PlantHier is named after the object identity of this object.

Object files for all other code should also be placed here.

9.3.3 $pwrp_lis

This is the directory where you place list-files produced during compilation.

9.3.4 $pwrp_lib

This is the directory where libraries will be created containing object files belonging to a certain volume. You should also place your own libraries here.

9.3.5 $pwrp_load

This is the place where load-files for the projects volumes will be created. The load-files are named:

```
<volumename>.dbs
```

9.3.6 $pwrp_log

This directory contains log-files that are produced during simulation of your project. ProviewR's main log-file is named

```
pwr_<nodename>.log
```

If you restart simulation, logging will be appended to this file. Remove it if you want a fresh one.

9.3.7 $pwrp_tmp

This directory contains temporary files. These files will be created at certain operations. For example if you compile a plc-program with debug-mode the source files created for this program will be created here.

9.3.8 $pwrp_inc

This directory contains include-files that will be included when you build the plc-program. A file called
```
ra_plc_user.h
```
will always be searched in this directory. If you have other header-files to include, then include them in this one.
The master for all these include files should be kept in the source tree and copied here.

Header-files for userclasses will be created here when you build a classvolume. The files are named:

```
pwr_<classvoumename>classes.h
pwr_<classvoumename>classes.hpp
```

9.3.9 $pwrp_web

This directory contains all files for the web-interface, for example pwg files for Ge graph, flw files for plc trace and xtt helpfiles converted to html.
here.

9.4 Special files

All special files that can be used for different kinds of configuration, or are of other interest and are located somewhere in the project directory tree, are described here. They are all mentioned above in this chapter.

9.4.1 Rt_xtt

This file is read by rt_xtt when started and the file is searched from the directory where you start rt_xtt. The file configures hot-keys to perform different kinds of commands.

Valid commands are:

```
Command      // This will perform a xtt command
SetDig       // This will set a digital signal to TRUE
ToggleDig    // This will toggle the state of a digital signal
ResetDig     // This will reset a digital signal to FALSE
```

To bind a hot-key to a command you first define the key and then state the command.
For example to bind the keystroke <ctrl>F5 to a command that acknowledges a type A alarm:

```
Control <Key>F5: Command(event ack /prio=A)
```

A typical Rt_xtt-file could look something like this:

```
#
# Function key definition file
#
Control <Key>F5: Command(event ack /prio=A)
Control <Key>F6: Command(event ack /prio=NOA) # ack non A-alarms
Control <Key>F7: Command(show alarm)          # open alarm list
Control <Key>F8: Command(show event)          # open event list
# Below opens a graph defined by a XttGraph-object in the node hierarchy.
# The $Node-expression will be replaced by the node-object on this node.
# This makes the Rt_xtt-file work on different nodes.
Alt <Key>F12: Command(open graph /object=$Node-Pictures-rkt_overview)
# Below opens a graph defined by a XttGraph-object in the node hierarchy.
# The /focus-syntax sets focus on a object in the graph named NewPlate
Control <Key>F9: Command(open graph /object=*-Pictures-rkt_platepic/focus="NewPlate")
Control <Key>F10: Command(open graph /object=*-Bilder-rkt_cells/focus="Check_no")
# Below closes all open graphs except rkt_overview.
Control <Key>F11: Command(close all/except=rkt_overview)
Shift <Key>F1: SetDig(VWX-RKT-RB-DS-OnOffMan_1_2.ActualValue)
Shift <Key>F6: ResetDig(VWX-RKT-RI-RP-CalcPrePos.ActualValue)
Shift Control <Key>v: ToggleDig(VWX-RKT-COM-VWXSVR-BlockOrder_RKT.ActualValue)
```

9.4.2 xtt_setup.rtt_com

This file is read by rt_xtt when started and the file is searched from the directory where you start rt_xtt. The file configures the appearance of rt_xtt. You can build your own menus and make entries that perform certain commands.

All commands in the file follow standard xtt command syntax. There is a built in help in rt_xtt which explain most of the commands. To view this help type <ctrl>-b when in xtt to open

the command line and write help. Navigate through the commands to understand them.

When you start rt_xtt there is by default some menues where the first one is named Database.
If you, for example, would like to create a menu on top (before) this, use this command:

```
create item/text="Maintenance"/menu/dest=DataBase/before
```

To create a sub menu (first child) to this, use this command:

```
create item/text="VVS"/menu/dest=Maintenance/firstchild
```

To create an entry that performs a command below the VVS menu, use this command:

```
create item/text="Graph P1"/command="open graph/object=$Node-pics-h4_proc1"\
        /pixmap=graph/dest=Maintenance-VVS/lastchild
```

The pixmap qualifier defines the appearance (icon) of this entry. Without this qualifier the
icon will be a leaf. The command opens a graph defined by an XttGraph-object in the node hierarchy.

The result will look like this:
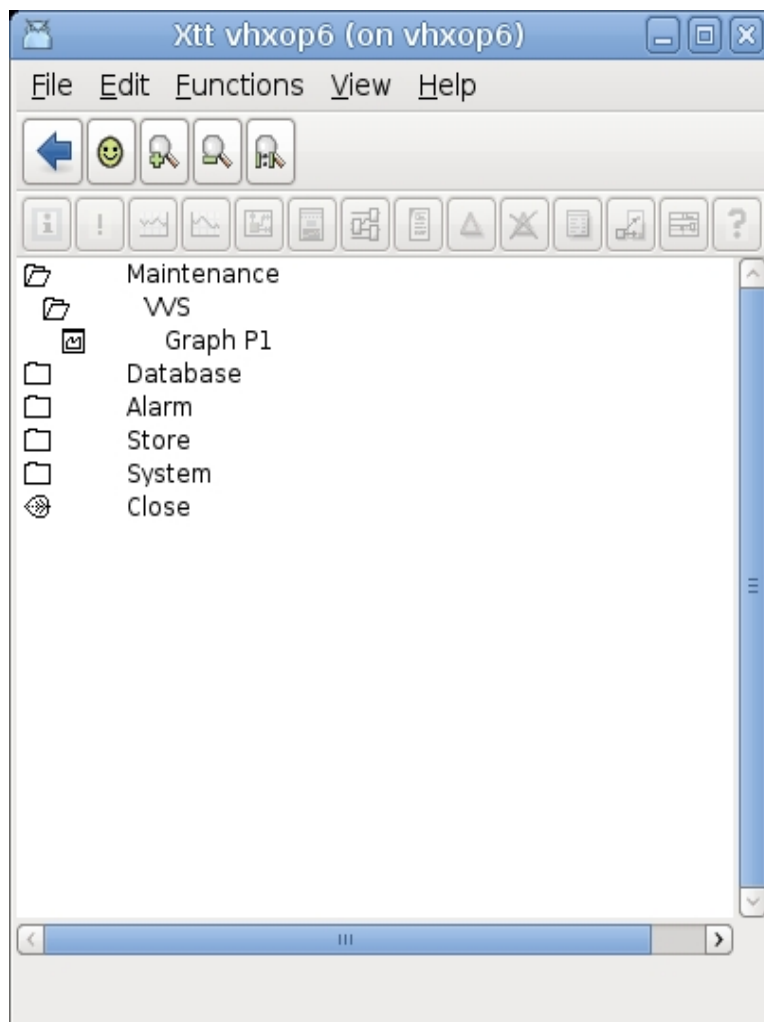
In addition to building a menu you can also define symbols (shortcuts) that can be used as commands.
The symbols can be entered on the command line, and the command defined by the symbol will
be executed.
The following defines a symbol 'h4' that will open a graph:

```
            define h4 "open graph /object=*-pics-h4_process1"
```

In the xtt_setup-file you create one command per line. Comment lines starts with an exclamation mark.

9.4.3   ld_appl_<node>_<bus_no>.txt

This file controls which applications should start when you start ProviewR runtime. You can add your own applications as well as turn off one or more of the ProviewR kernel applications.

A typical ld_appl-file can look like this:

```
# Startup file for ProviewR
#
# id,    name,    load/noload run/norun,  file,   prio,   debug/nodebug,  "arg"
#pwr_neth,        , noload, norun, , 5, debug, ""
#pwr_plc,         , noload, norun, ,  , debug, ""
#pwr_alim,        , noload, norun, , 5, debug, ""
#pwr_emon,        , noload, norun, , 5, nodebug, ""
#pwr_tmon,        , noload, norun, , 5, debug, ""
#pwr_qmon,        , noload, norun, , 19, debug, ""
#pwr_nacp,        , noload, norun, , 5, debug, ""
#pwr_bck,         , noload, norun, , 5, debug, ""
#pwr_io,          , noload, norun, , 5, debug, ""
#pwr_linksup,     , noload, norun, , 5, debug, ""
#pwr_trend,       , noload, norun, , 5, debug, ""
#pwr_fast,        , noload, norun, , 5, debug, ""
#pwr_remh,        , noload, norun, , 5, debug, ""
pwr_remlog,       , noload, norun, , 5, debug, ""
#pwr_sysmon,      , noload, norun, , 5, debug, ""
#pwr_elog,        , noload, norun, , 5, debug, ""
pwr_websocketserver,      , noload, norun, , 5, debug, ""
#pwr_opc_server,       , noload, norun, , 5, debug, ""
#pwr_sevhistmon,       , noload, norun, , 5, debug, ""
#pwr_sev_server,       , noload, norun, , 5, debug, ""
#rs_nmps_bck, rs_nmps_bck, noload, run, rs_nmps_bck, 12, nodebug, ""
ra_utl_track, ra_utl_track, noload, run, ra_utl_track, 12, nodebug, ""
```

The sharp sign means that those lines are commented away. Almost all of the ProviewR kernel applications are commented away since we want those to start. If I take away the hash sign then this kernel application will not be started. For example in this case I have no web-interface so I don't want the web server to start (pwr_websockerserver).

If I use Nmps-cells and want the content of the cells to be backed up I take away the sharp sign on rs_nmps_bck.

On the last line I have added an application produced by myself. I've chosen priority 12. I don't want this application to interfere with the kernel applications and they run between 17 and 19.

9.4.4   plc_<node>_<bus_no>_<plc_name>.opt

Note! This file is deprecated since V4.8.2. A BuildOptions object should be used instead.

This file (if it exists) will be used as the link options when I build the plc-program.
ProviewR by default links against some libraries and object-files. If you have your own
opt-file you need to include these. A default opt-file would look like:

```
$pwr_obj/rt_io_user.o -lpwr_rt -lpwr_usbio_dummy -lpwr_usb_dummy -lpwr_pnak_dummy
 -lpwr_cifx_dummy -lpwr_nodave_dummy -lpwr_epl_dummy
```

Add your own libraries at will. The syntax to be used is the syntax for ld (The GNU linker). ProviewR
will create a template option file .opt_template that can be renamed to .opt and used as a template.

9.4.5   pwrp_alias.dat

File to control setting of initial values when starting ProviewR.

There are some different ways of setting values through the pwrp_alias-file.
The same file is used for all nodes in the project. Each row in the file should start
with the following expression:

```
<nodename>_setval
```

The different ways of setting things is described below:

1. Setting an attribute value

```
<nodename>_setval <attribute_name> = <value>
```

example:

```
bslds1_setval bsl-ds1-par-maxtemp.actualvalue = 70.0
```

Using the above described syntax will set the value before the backup is loaded and before
the plc-program is started. This means that if a value is backed up then the backuped value
will always be valid.

If you instead really want the setting in this file to have effect then use this syntax:

```
<nodename>_setvalp <attribute_name> = <value>
```

In this case the setting will take effect after the backup-file is loaded and the plc-program
is started.

2. Setting simulation mode

Setting the simulation mode means that no physical i/o will be handled. You can write simulation
programs to set correct values on the input i/o. Add this line to the file:

```
<nodename>_setval plcsim = yes
```

3. Set all plc-programs to scan off at startup

To set all plc-programs to scan-off use this line:

```
<nodename>_setval plcscan = off
```

Turn a plc-program on by finding the corresponding WindowPlc-object (child to PlcPgm-object) and
set the attribute ScanOff to 0. Observe that there might be subwindows in this program
that also need to be turned on.

9.4.6   /etc/proview.cnf

A configuration file for definition of various parameters both in developmen, runtime and

storage environment. In the development environment the definitions are valid for all projects
on the station.

Parameters in the development environment

mysqlSocket	mysql socket for volumes with mysql database. Default value "/var/run/mysqld/mysqld.sock".
mysqlServer	mysql server node.
defaultProjectRoot	The default path to projects when creating new projects. Default value "/usr/local/pwrp".
defaultProductionQBus	The default production bus when creating new projects. Default value 1.
defaultSimulationQBus	The default simulation bus when creating new projects. Default value 999.
defaultSystemGroup	The default system group when creating new projects. Default value "Common".
defaultNodeHierRoot	Default name of the top level object in the node hierarchy when creating new projects. Default value "Nodes".
defaultSecurity	Default name of the Security object. Default value "Security".
defaultXttPriv	Default value of XttPriv in the Security object. Default value "Security".
defaultOpPlaces	Default name of map for OpPlace objects in the node hierarchy. Default value "OpPlaces".
defaultServers	Default name of map for server objects in the node hierarchy. Default value "Servers".
defaultIO	Default name of map for IO objects in the node hierarchy. Default value "IO".
defaultApplications	Default name of map for application objects in the node hierarchy. Default value "Applications".
defaultOpOp	Default name of OpPlace object för operators in the node hierarch Default value "Op".
defaultOpMaintenance	Default name of OpPlace object for maintenance in the node hiera Default value "Maintenance".
defaultOpDefault	Default name of default OpPlace object in the node hierarchy. Default value "OpDefault".
defaultWebBrowser	Default name of WebBrowser object in the node hierarchy. Default value "WebBrowser".
qcomBusId	QCom bus identity for simulation.

Parameters in the runtime environment

qcomBusId	QCom bus identity.
curveExportFile	Default file name for the history curve export function.
webDirectory	Directory where the web files are placed and accessed by the web

Parameters in the storage environment

sevDatabaseType	Database type, mysql or sqlite.
sevXttDefaultPriv	Default privileges for access of the sev database from sev_xtt.
sevMysqlEngine	Mysql engine for created tables, innodb or myisam.

10 Graphical PLC Programming

This chapter describes how you create PLC programs.

The Editor

You enter the plc editor from a PlcPgm object in the plant configuration. Select the object and activate 'Functions/Open Program' in the menu. The first time the program is opened, you will find an empty document object. The program consists of functions blocks and Grafcet sequences.

Programming with function block is made in a horizontal net of nodes and connections from left to right in the document. Signals or attributes are fetched on the left side of the net, and the values are transferred via connections from output pins to input pins of functions blocks. The function blocks operate on the values, and on the left side of the net, the values are stored in signals or attributes.

Grafcet sequences consist of a vertical net of nodes and connections. A state is transferred between the steps in the sequence via the connections. Grafcet and function block nets can interact with each other and be combined to one net.

Edit function objects

The plc editor consists of
- a working area
- a palette with grafcet objects and function blocks, and a palette with connections
- a navigation window, from which the work area can be scrolled and zoomed
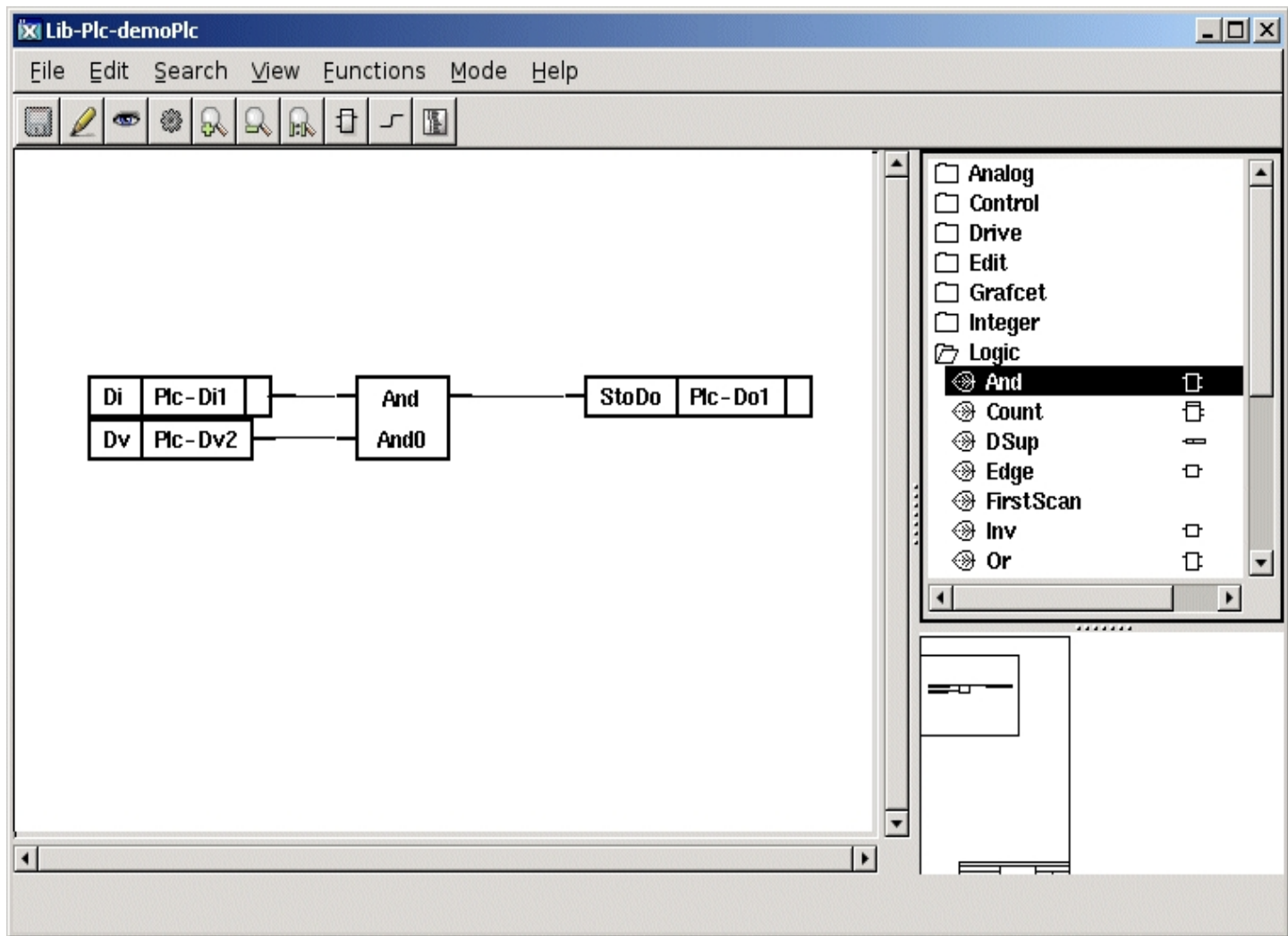
Fig The Plc editor

A function object is created by selecting a class in the palette, and pressing MB2 in the working area.

Modify the object

The object is modified from the object editor. This is opened by selecting the object and activating 'Functions/Open Objects' in the menu. Values of the object attributes are modified with 'Functions/Change value' in the object editor menu, or by pressing the Arrow Right key. If an input or output is not used, it can be removed with a check box. There is also a check box which states that the value of a digital input should be inverted.

Connect function objects

A output pin and a input pin is connected by

- Place the cursor on the pin, or in an area in the function object close to the pin, and press MB2.
- Drag the cursor to the other pin, or to an area in the function object close to the pin, and release MB2.

A connection is now created between the function objects.

Fetch a signal value

The value of a Di signal is fetched with a GetDi object. The GetDi object has to point at a Di signal and this is done by selecting the signal in the plant configuration, and then press Ctrl and double click on the GetDi object. The name of the signal is now displayed in the drawing. Dv signals, Do signals and attributes are fetched in the same way, with GetDv, GetDo and GetDp objects.

The easiest way to create a Get object is to draw a connection from input point where the Get object should be connected, and release it in an empty space in the work area. A generic Get object is now created that will be transformed to a Get object of the correct type when the signal is specified. The signal is specified as before by selecting the signal in the plant hierarchy and Ctrl/double click on the Get object.

Store a value to a signal

The value of an output from a function object is stored in Do signal with a StoDo objects. The StoDo object is connected to a Do signal in the same way as the Get objects. Dv signals and attributes are stored with StoDv and StoDp objects.

Grafcet Basics

This section gives a brief introduction to Grafcet. For a more detailed description, please read a reference manual on Grafcet. Grafcet is an international norm or method to use at sequential control.

Grafcet consists of a number of steps, and to each step one or more orders are connected, which will be executed when the step is active. In order to move from one step to another, a transition is used. For each transition you have transition conditions, and the move can only take place when the transition conditions have been fulfilled.

Single Straight Sequence
We look at the single sequence below and assume that the step is active, which means that the order connected with the initial step will be carried out. This order will be carried out until the initial step becomes inactive. Step 1 becomes active when the transition condition for transition 1 has been fulfilled. Then the initial step becomes inactive.

A Grafcet program is always a closed sequence.
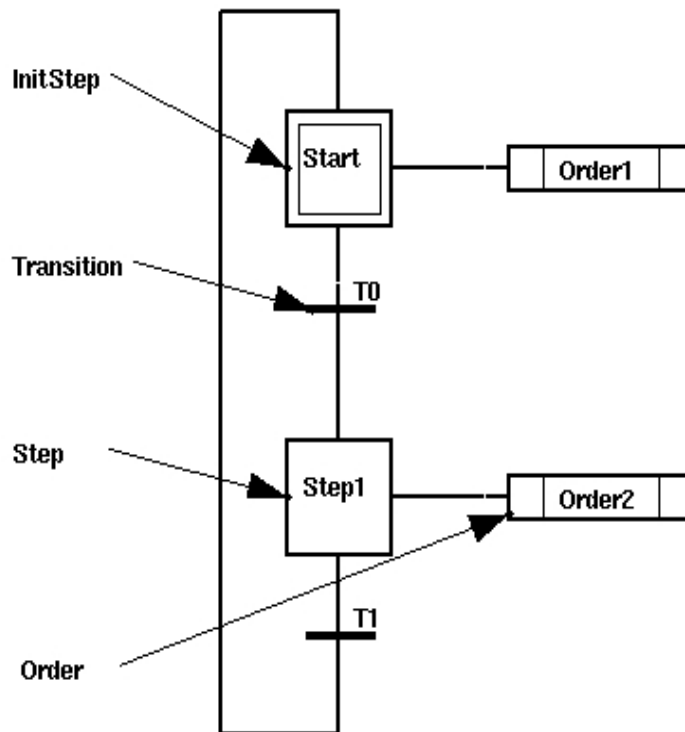
Fig A Simple Straight Grafcet Sequence

Diverged sequence

A straight sequence is the most simple variant of sequences. Sometimes you may require alternative branches in your program, for instance when you have a machine which can manufacture three different products. At the points where the production differs, you introduce alternative branches.
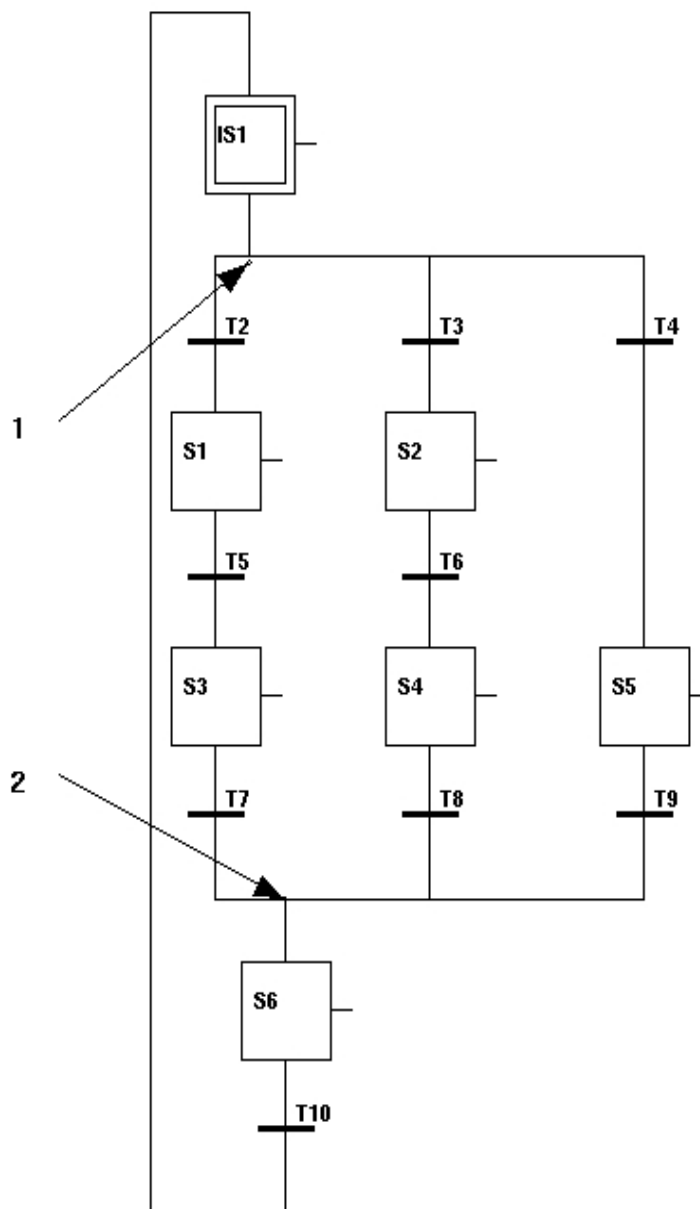
Fig Sequence Selection

The example in the figure above shows the sequence for a machine which can manufacture the three products, Red, Green, and Blue. At the point of divergence, point 1 in the figure, you choose the desired branch depending on the product to produce. The alternative branches diverge from a step, that is followed by one transition condition in each branch. It is the constructors task to see that only one of the transition conditions is fulfilled. If several are fulfilled, it is undefined which one that is selected. At point 2 in the figure, the branches are converging to a common step.

Parallel Sequences

Sometimes it may be necessary to start several parallel working procedures at the same time. It must be possible for these working procedures to operate independent of each other. In order to do this, parallel sequences are used.
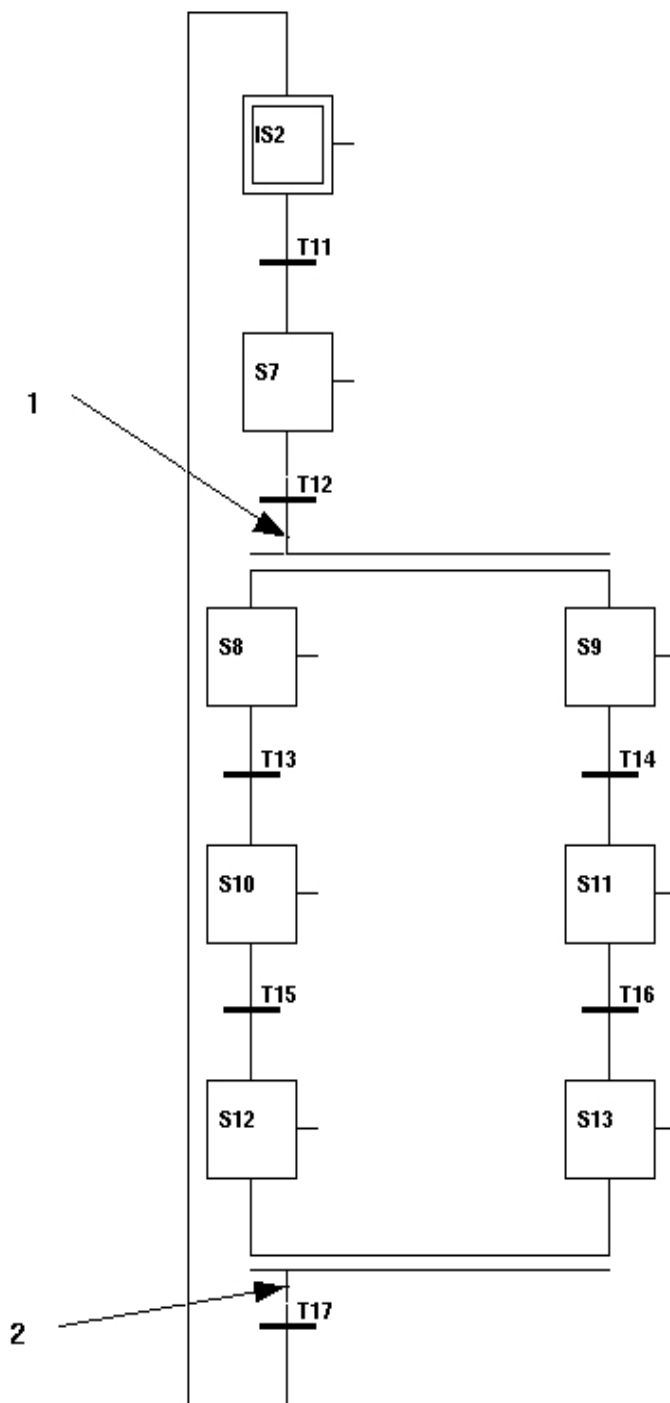
Fig Parallel Sequences

The example in the figure above illustrates the sequence for two machines that are drilling two holes at the same time and independent of each other. When the transition condition before the parallel divergence (point 1 in the figure) is fulfilled, the activity is moved to both branches, and the machines start drilling. The drilling is performed independent of each other.

The branches are converging to a transition condition (point 2 in the figure), and when the drilling is completed in both machines, i.e. both S12 and S13 are active, and the transition condition T17 is fulfilled, the activity is moved to the init step IS2.

Step
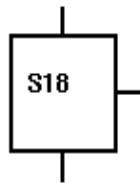
A Step is used to describe a state in the process. The following applies to a step:

- A step can be active or inactive.
- An attribute, Order, indicates whether the step is active or not.
- You can connect one or more orders to a step.
- The step can be allocated a name at your own discretion.

InitStep
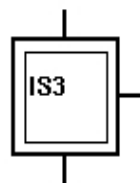
In each sequence you must have an initial step (InitStep) which differs from a usual step in the following way:

- You should only have one initial step in a sequence.
- When the program starts its execution, the initial step is active.
- You can always make the initial step active by setting the reset signal.

Transition - Trans

As mentioned above, the transition (Trans) is used to start a transition between an active and an inactive step. A logical condition, for instance a digital signal, is connected to a transition, and determines when the transition is taken place.
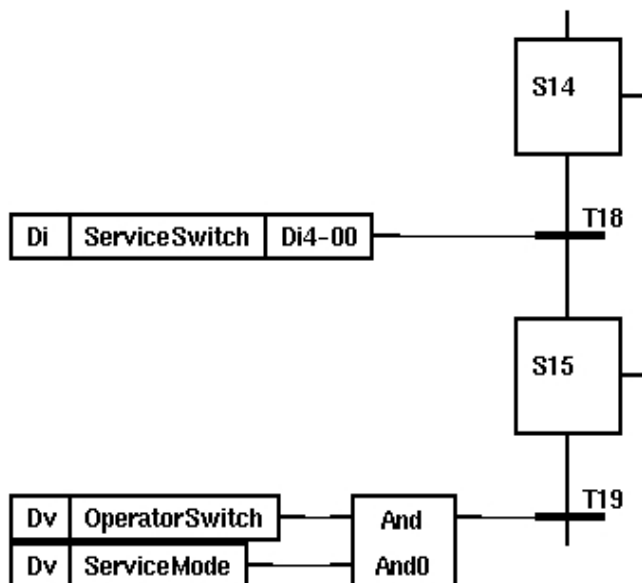
Fig A Transition Example

Order

It is possible to connect one or more orders to each step.
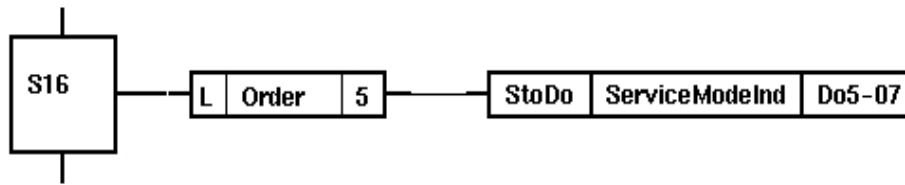
Fig An Order Example

Normally the output is active when the input is active, but for each order you have a number of attributes, with which you can influence the function of the output:

- D Delay
- L Time limit
- P Pulse
- C Conditional
- S Stored

These functions are described in detail in ProviewR Objects Reference Manual. The principles are that you indicate the name of the attribute (capital letters) and possible time by means of the Object Editor. The figure below illustrates how to delay an order from being active for 10 seconds.
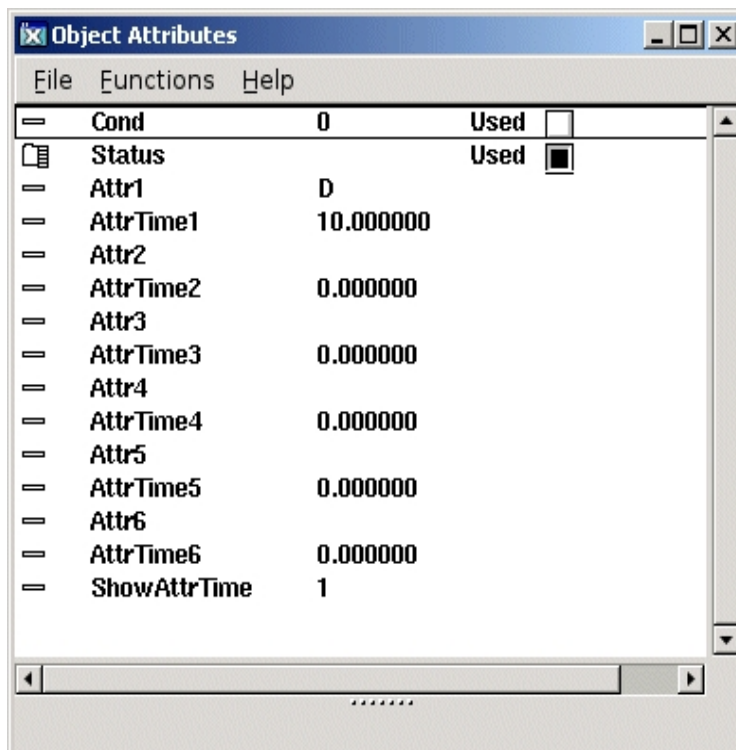
Fig DOrder Attributes

The selected order attributes are written in the order symbol.

The figure below illustrates how you can use an order object with delay to make a step active for a certain time.
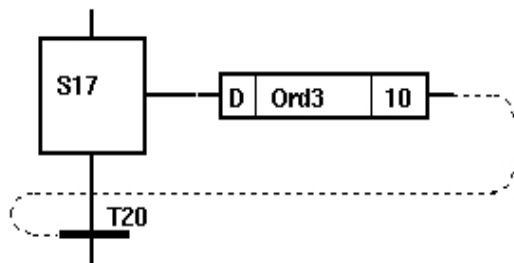
Fig A Delayed Transition

Note! You must use a ConFeedbackDigital connection to connect the delayed order object with the transition object, otherwise the order of execution will be ambigous.
See Feedback Connection

Subsequence - SubStep

When you are creating complex Grafcet programs, it is often suitable to use subwindows, and to place subsequences in these to get a better layout of the program.
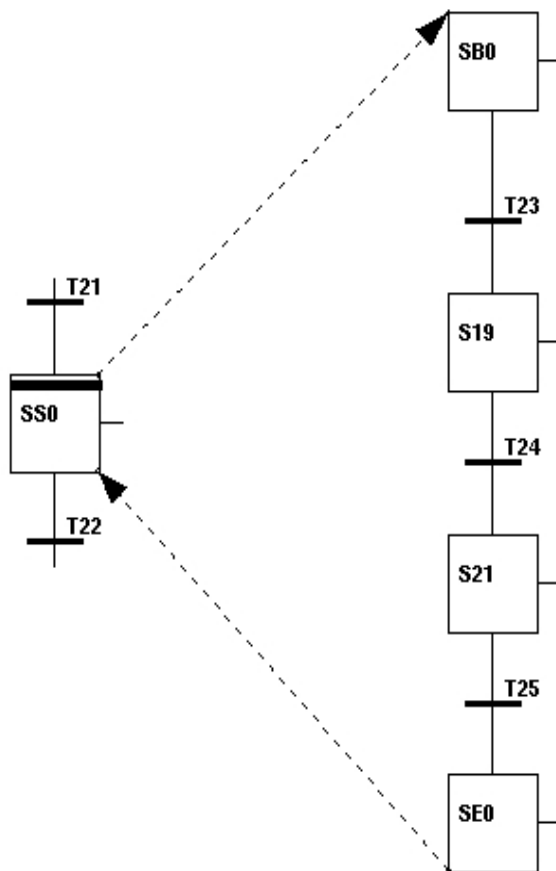
Fig Subsequence

The above figure shows the sub sequence of a SubStep. A sub sequence always starts with an SsBegin object, and ends with an SsEnd object. In its turn a subsequence can contain subsequences.

Building Grafcet sequences

Grafcet sequences are easily built in the plc editor by starting with a Step or InitStep object. By drawing a connection from the lower connection point of the step, and releasing the connection in an empty space in the work area, a connected Trans object will be created. In the same way Order objects are created from the right connection point, and from the Trans object, new Step objects will be created from the upper and lower connections points.

An Introduction to Function Block Programming

Blocks to fetch and store values

Blocks to fetch and store are used to read and write values. There are fetch and store blocks for each type of signal. In the figure below a number of these blocks are displayed. They are found in the 'Signal' folder in the palette.

| Ai | | |
| Ao | | |

| | StoAo | |

| Av | |
| Ap | |

| | StoAv | |
| | StoAp | |

| Di | | |
| Do | | |

	StoDo	
	SetDo	
	ResDo	
	StoDv	
	SetDv	
	ResDv	

| Dv | |

| Dp | |

	StoDp	
	SetDp	
	ResDp	

| Ii | | |
| Io | | |

| | StoIo | |

| Iv | |
| Ip | |

| | StoIv | |
| | StoIp | |
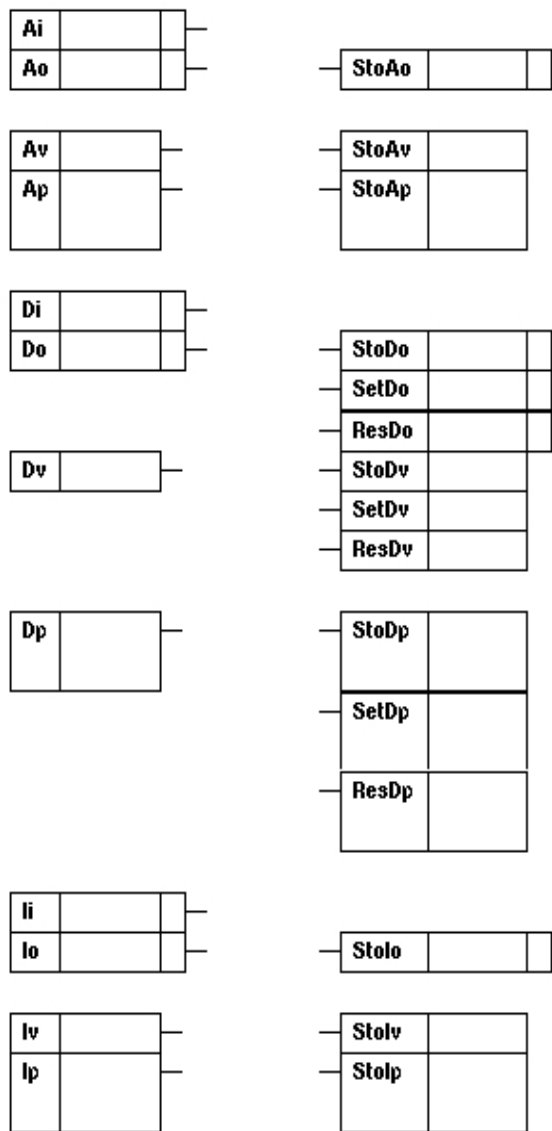
Fig Blocks to fetch and store values

To read signals you use blocks like GetAi, GetIi, GetDv or GetAo. When you want to give a value to a signal, you use for instance StoAv, StoDo, SetDv or ResDo.

Digital values can be written in two ways:

- 'Sto' stores the input value, i.e. if the input is 1 the signal becomes 1, and if the input is zero the signal becomes zero.
- 'Set' sets the signal to one if the input is true, Res sets the signal to zero if the input is true. For instance, if you set a digital output signal with a SetDo object, this will remain set until you reset it with a ResDo object.

To read, respectively assign attribute values other than the ActualValue attribute, you use the following:

- analog attributes, GetAp and StoAp
- integer attributes, GetIp and StoIp
- digital attributes, GetDp and StoDp, SetDp or ResDp
- string attributes, GetSp and StoSp
- time attributes, GetAtp, GetDtp, StoAtp and StoDtp

Logic Blocks

A number of objects are available for logical programming, for instance And-gate (And), Or-gate (Or), inverter or timer. For the logical programming digital signals are used. The objects are placed in the folder Logic.
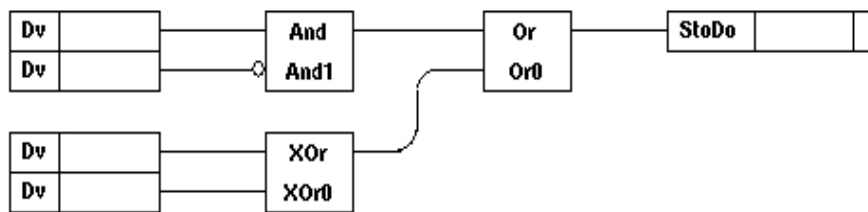
Fig Logic Blocks

The figure below shows an And-gate. For this object the following applies:

- Inputs to the left
- Output to the right
- Class name is written at the top
- The object name is written at the very bottom (can be changed by the user)
- You can use a variable number of inputs, default is 2
- The inputs can be inverted, indicated by a ring on the symbol's input

And-gate
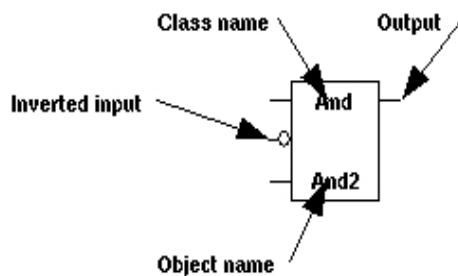
The attributes of the And-gate are changed with the Object Editor.
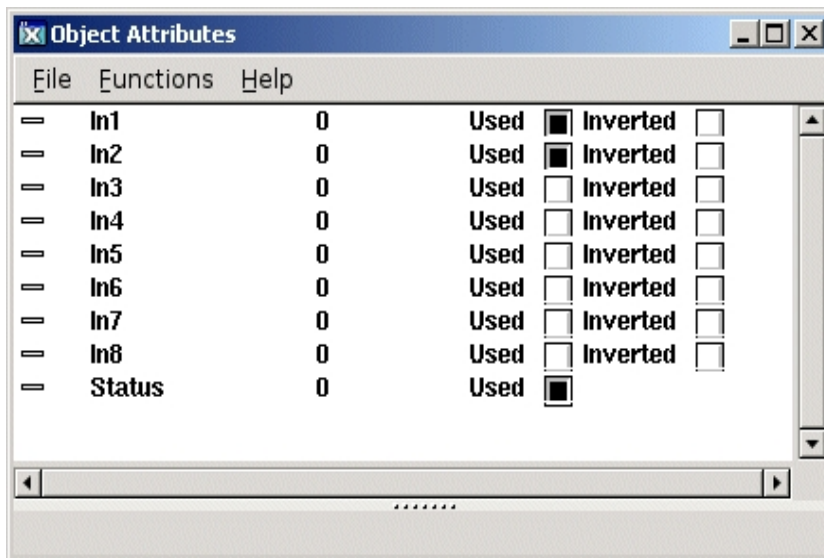
Fig Attributes of the And-Gate

The other objects in the 'Logic' folder have similar parameters, see ProviewR Objects Reference Manual .

Calculation Blocks

The folder 'Analog' contains a number of objects for handling analog signals, for instance filters, summation blocks, and integrators.
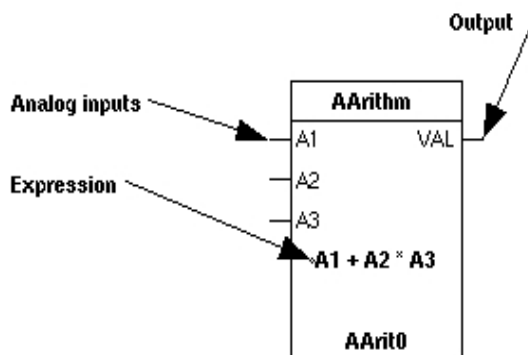
Fig Arithmetical Calculation Block

In this guide we do not describe the function of the objects, but it may be expedient to comment on the use of arithmetic blocks. The blocks are used for calculation of user defined expressions. These are written in the C language.

In the figure below the block will calculate the expression (A1 + A2 * A3) and give the output this value. A1, A2 and A3 are representing analog values, for instance signals supposed to be connected to the inputs of the object.
When writing these expressions it is important to use space before and after the operators, otherwise the expression may be misinterpreted at the execution.

The expression can contain advanced C code with arrays and pointers. When you write these, you should be aware that indexing outside arrays, or erroneous pointers might cause the execution of the plc-program to terminate.

Alarm Supervision

In ProviewR it is possible to supervise analog and digital signals. Supervision of analog signals is made against a limit value. If the supervised limit is exceeded, an alarm is sent to the Message Handler, which in turn sends the alarm to the out unit, e.g. an operator dialog.

See ProviewR Objects Reference Manual regarding the attributes of the objects.

Supervision of Digital Signals

For supervision of a digital signal or attribute, you use the DSup object (Digital supervisory), which is in the folder Logic.

The desired signal or attribute, is fetched with a Get-object that is connected to the DSup object. Outputs of logical blocks can be directly connected to the DSup object.

| Dv | ServiceMode | | DSup | DSup0 | P1 is in service mode |
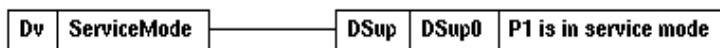
Fig Digital Supervisory Objects

Figure above illustrates supervision of a Dv signal and a digital attribute.

You also have an attribute in the DSup object, 'CtrlPosition', that indicates whether the alarm will be activated when the supervised signal/attribute becomes true or false.

Supervision of Analog Signals

For supervision of an analog signal or attribute, you use the ASup object (Analog supervisory), which is in the folder 'Analog' in the palette.

Supervision takes place in the same way as for DSup objects with the exception that you can choose whether the alarm will be released when the value is above or below the supervision limit.

Execute order

The execute order is determined by the how the function objects are coupled together. An object which output is connected to the input of another object, is executed before the object it is connected to. You can see the execute order by activating 'View/Show Execute Order' in the plc editor menu. For objects and nets that are not coupled together, the execute order is determined by the position. Object higher up is execute first.

If you want to influence the execute order between two objects, you can draw a special execute order connection beween the objects. This is choosen in the connection palette and has an arrow in one end. The object that the connection is pointing at will be executed after the object that the connection emanates from.

If a net contains a feedback connection, the execute order can't be determined and the error 'Amigous execute order' is messaged. You then have to exchange one connection to a feedback connection, ie a connection that doesn't determine the execute order. The feedback connection is dashed and is selected in the tool panel or in the connection palette.

I/O copying

If a signal is used in several places in a net of function objects, there is a chance that the signal value will be changed during the execution which can result in lockings and other fenomena that is hard to predict. Therefor a mechanism called I/O copying is used. The signal values for signals of type Ai, Ao, Av, Di, Do, Dv, Ii, Io, Iv, Co, Bo and Bi are gathered in special area objects. Before a plc thread starts to execute, a copy of the area objects are made, and during the execution all the readings are made from the copy, while writings are made into the original area objects. This ensures that this kind of fenomena is avoided, but also that you can get delays. If a signal value is set, and then read in the same plc thread, the modification will not be registred until the next scan.

Compile the plcpgm

Before starting to compile, you have to state on which platform (or platforms) the volume of the plc should run. Open the volume attributes editor from the navigator menu: 'File/Volume Attributes', and enter the OperatingSystem. Note, that more than one operating system can be chosen. The volume can at the same time be run in the production system, in a simulation system and in a educational system, and the systems can have different platforms.

The plcgpm is compiled by activating 'File/Build' in the plc editor menu. Any warning or error messages will be displayed in the message window. When building the node, any new or modified plcpgm will also be compiled.

11 Call functions from the plc program

The functionobject programming in the plc editor has its limitations, and some tasks can be done much easier and nicer in c-code. c programming can be achieved in CArithm and DataArihm where you can put an amount of c-code, but the number of characters are limited to 1023 (8191 for the DataArithmL), and occasionally this is not enough. Then you have two possibilities; to write a detached application, or to call a c-function from a CArithm or DataArithm. The advantage with calling a c-function is that all initialization and linking to objects and attributes are handled by the plc program. The execution of the function is also synchronous with the execution of the plc thread calling the function.

Write the code

The code is put into a c file, created somewhere under $pwrp_src. We create the file $pwrp_src/ra_myfunction.c and insert the function MyFunction() that performs some simple calculation.

```
#include "pwr.h"
#include "ra_plc_user.h"

void MyFunction( pwr_tBoolean cond, pwr_tFloat32 in1, pwr_tFloat32 in2,
                 pwr_tFloat32 *out)
{
  if ( cond)
    *out = in1 * in2;
  else
    *out = in1 + in2;
}
```

Prototype declaration

In the include file ra_plc_user.h a prototype declaration is inserted.

```
void MyFunction( pwr_tBoolean cond, pwr_tFloat32 in1, pwr_tFloat32 in2,
                 pwr_tFloat32 *out);
```

ra_plc_user.h is included by the plc program, and the function can be called from a CArithm or DataArithm object. You should also include ra_plc_user.h in the function code to ensure that the prototype is correct.

ra_plc_user should be placed on $pwrp_src and copied to $pwrp_inc, from where it is included by the plc program and the function code.

Compile the code

The c file is compiled, for example with make. Below a makefile is shown, that compiles ra_myfunction.cpp and puts the result, ra_myfunction.o on $pwrp_obj. Note that there is also

a dependency on ra_plc_user.h, which causes this file to be copied from $pwrp_src to $pwrp_inc.

```
ra_myfunction_top : ra_myfunction

include $(pwr_exe)/pwrp_rules.mk

ra_myfunction_modules : \
  $(pwrp_inc)/ra_plc_user.h \
  $(pwrp_obj)/ra_myfunction.o

ra_myfunction : ra_myfunction_modules
 @ echo "ra_myfunction built"

#
# Modules
#

$(pwrp_inc)/ra_plc_user.h : $(pwrp_src)/ra_plc_user.h

$(pwrp_obj)/ra_myfunction.o : $(pwrp_src)/ra_myfunction.c \
                              $(pwrp_inc)/ra_plc_user.h
```

Call in the plc program

The function is called from a CArithm or DataArithm.
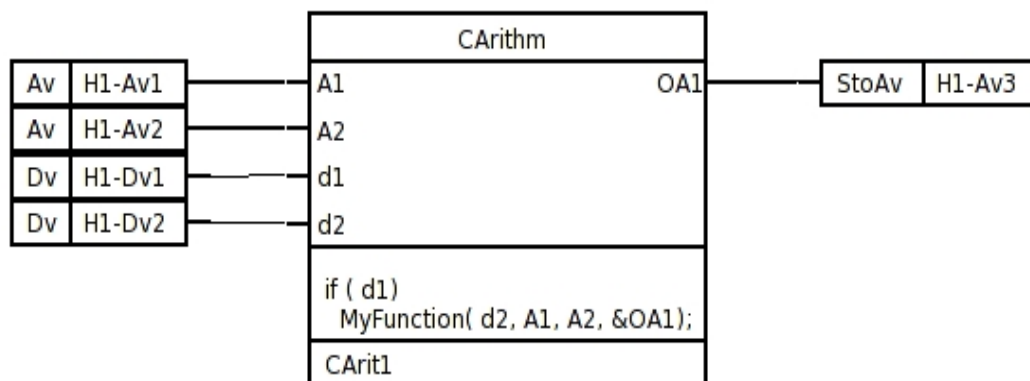
Fig Function call from the plc code

Link the plc program

When the source code of the function was compiled, the object module $pwrp_obj/ra_myfunction.o was created. This has to be added to the link command when the plc program is built, which is achieved by creating a BuildOptions object in the directory volume under the NodeConfig object. Insert the name of the object module in the ObjectModules array. When the directory volume is saved, an opt-file is created on $pwrp_exe that will be included by the linker when the node is built.
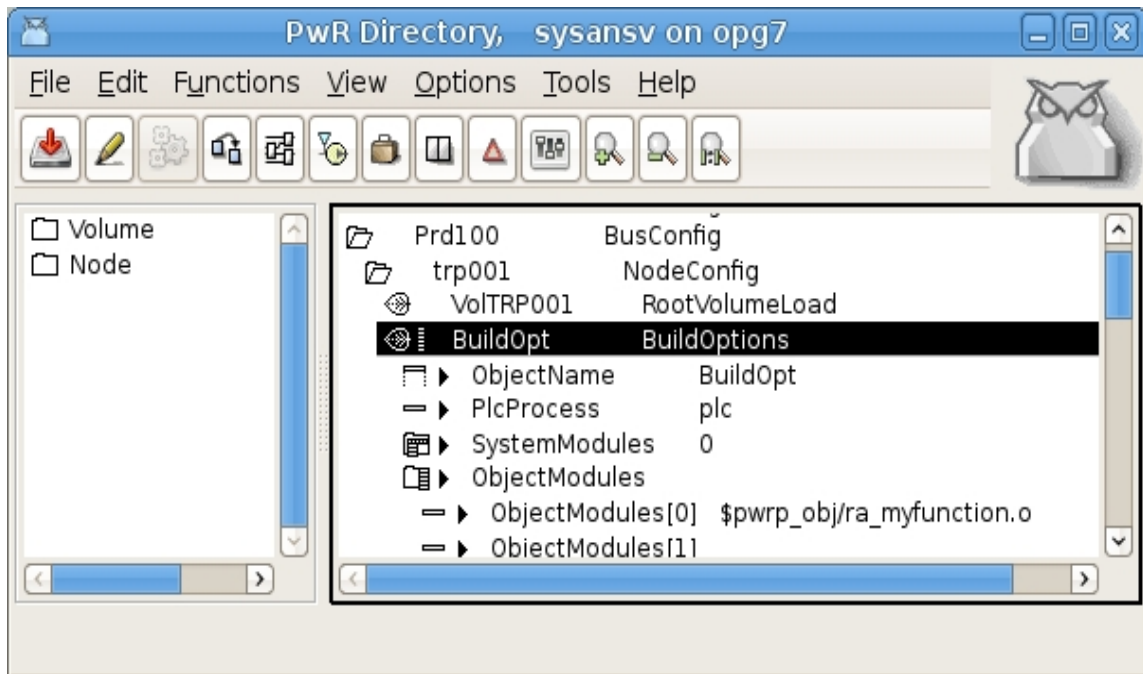
Fig The object module inserted in BuildOptions

We can now build the node and startup ProviewR runtime.

Debug
One disadvantage when you leave the graphic programming and call c-functions is that you can't use trace any more for debugging. If you suspect some error in the function code, you occasionally have to start the plc program in debug, set a breakpoint in the function and step forward in the code.

First you have to build the plc program with debug, by opening Options/Setting from the configurator and activate Build/Debug, and then build the node.

After that you start ProviewR runtime and attach the debugger, gdb, to the plc process by starting gdb with the pid for the process. pid is viewed by 'ps x'

```
> ps x
...
5473 pts/0    Sl      0:18 plc_mynode_0999_plc
```

where 5473 is pid for the plc process, and we start the debugger, set a breakpoint in the function and let the program continue to execute

```
> gdb -p 5473 plc_mynode_0999_plc
(gdb) b MyFunction
(gdb) c
```

When the program enters the function it stops in the debugger, and we can step (s) and examine the content in variables (x) etc.

If the plc program is terminated immediately after start, you can restart in debug.

```
> gdb plc_mynode_0999_plc
```

You can also kill the current plc process and start a new one in debug.

```
> killall plc_mynode_0999_plc
> gdb plc_mynode_0999_plc
```

12   Components and Aggregates

This chapter is about how to program with components and aggregates.

A component is one (or a number of) objects that handles a component in the plant. A component can be for example a valve, a contactor, a temperature sensor or a frequency converter. As these components are very common and exist in many different types of plants, it is a great advantage if we can construct an object that contains all that is needed to control and supervise the component, and is so general that it can be used in most applications.

A component in ProviewR can be divided into a number of objects:
- a main object containing configuration data and data needed to supervise and operate
  the component. It also contains the signal objects for the component.
- a function object that is placed in the plc program and that contains the code to
  control the component.
- an I/O object that defines possible communication with for example a profibus module.
- a simulate object, used to test and simulate the system.

Furthermore an object graph, documentation, trends etc are included in the component.

An aggregate is a larger part in the plant than the component, and contains a number of components. An aggregate can for example be a pump drive, consisting of the components pump, motor, contactor and safety switch. In other respects, the aggregate is built as a component with main object, function object, simulate object, object graph, documentation etc.

Object orientation
ProviewR is an object oriented system, and components and aggregates are a field where the benefits of object orientation are used. In the components, one can see how an object is built by other objects, and that an attribute, besides from being a simple type as a float or boolean, also can be an object, which in its turn is composed by other objects. An attribute that is an object is called an attribute object. It is not quite analogous to a free-standing object, as it lacks object head and an object identity, but apart from that it contatins all the properties of a free-standing object in terms of methods, object graph etc.

One example of an attribute object can be seen in the component object of a solenoid valve. Here all the signal objects, two Di objects for limit switches, and a Do object for order, are placed internally as attribute objects. Thus, we don't have to create these signals separately. When the valve object is created, also the signals for the valve are created. Another example of attribute objects is a motor aggregate that contains the component objects for frequency converter, safety switch, motor etc. in the shape of attribute objects.

Another important property in object orientation is inheritance. With inheritance means that you can create a subclass derived from an existing class, a superclass. The subclass inherits all the properties of the superclass, but has also the possibility to extend or modify some of the properties. One example is a component for a temperature sensor that is a subclass to a general sensor object for analogous sensors. The only difference between the temperature sensor class and its superclass is the object graph, where the sensor value is presented in the shape of a thermometer instead of a bar. Another example is a pump aggregate derived from a motor aggregate. The pump aggregate is extended by a pump attribute object and also has a modified object graph that apart

from the motor control also displays a pump.
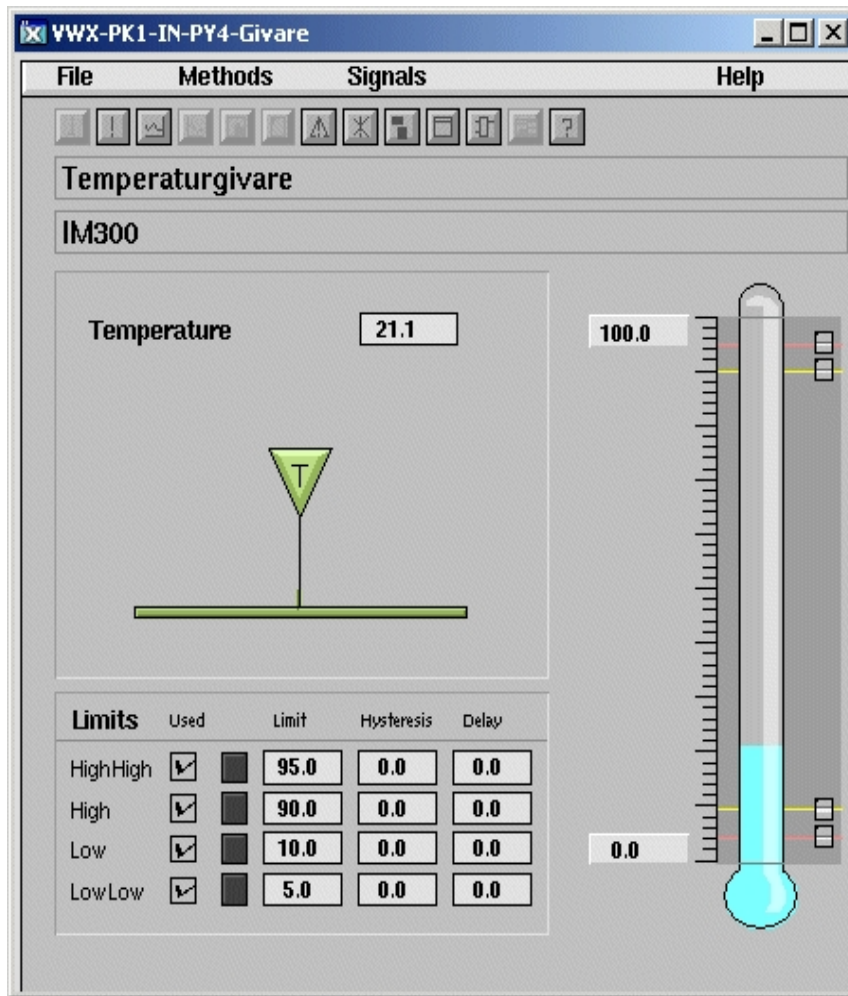
Fig Object graph for the class BaseTempSensor

Another property that we have introduced is the possibility to disable attributes. The reason for this is that the component objects have to be as general as possible, to be able to handle all variants of the plant component. A solenoid valve can, for example, have a limit switch indicating valve open, but there are also solenoid valves with a limit switch indicating valve closed, or valves with both switches or without switches. Of course we could create four different component classes, one for each limit switch alternative, but problems will arise when you start building aggregates of the components. The number of variants of an aggregate will soon be unmanageable if you want to cover all the variants of the components. If we, for example, want to create an aggregate containing four solenoid valves, and there are four variants of each valve, there will be 64 variants of the aggregate. If we want to build an aggregate containing four valve aggregates the number of variants is 4096. The solution is to build a valve component that contains both switches, but where you can disable one or both switches to be able to handle all four limit switch variants. In this case attribute objects of class Di are disabled, which means that they are not viewed in the navigator, ignored by the I/O handling. Also the in code for the valve component and in the object graph this is taken into consideration. The configuration is made in the configurator from the popup menu where you under 'ConfigureComponent' can choose a configuration from the alternatives.
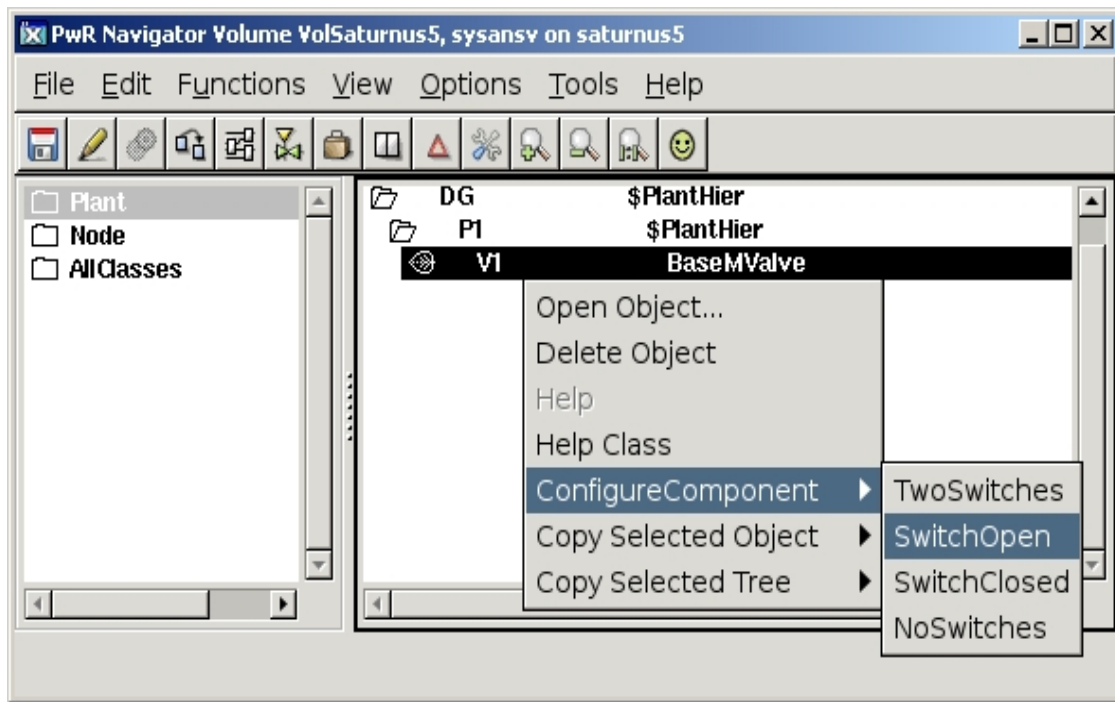
Fig The ConfigureComponent method for a BaseMValve.

Basecomponents

ProviewR contains a number of component and aggregate objects for common plant components, e.g. temperature sensor, pressure sensor, pressure switch, solenoid valve, filter, motor and fan drives. They are gathered in the classvolume BaseComponent. A basecomponent can be used directly, and this is probably the usual way to use them, but the idea is also that you from the basecomponents create libraries and classes for the specific components you are using in your plant.

For solenoid valves there are the baseclass BaseMValve. If you have a solenoid valve of type Durholt 100.103 you create a subclass with BaseMValve as superclass, Durholt_Valve_100_103 and insert the configuration that is valid for this valve. You also add a link to a datasheet and fill in the Specification attribute, which makes it possible to identify and order spareparts to the valve. When using a Durholt_Valve_100_103 object you don't have to do so much configuring and adaptations because this is already made in the class. In this way, component archives can be built for the types of component you use in your plant.

A problem arises when you use aggregates. An aggregate contains basecomponents from the BaseComponent volume, and if there are specific subclasses for a component, you want to use these. The solution is the Cast function. A basecomponent in an aggregate can be casted to a subclass, given that the subclass is not extended with new attributes. The casting means that the component fetches initial values, configurations, methods, object graph etc. from the subclass, i.e. in all situations it acts as the subclass it is casted to. The casting is performed from the popup menu in the configurator, where you in the 'Cast' alternative get a list of all the available subclasses. By selecting a subclass the component is casted to this.

Pressure switch

Let's have a look at a relatively simple component, a pressure switch, to examine how

it is built and how to configure it. For pressure switches there is the base component BasePressureSwitch, that is a subclass to BaseSupSwitch. As temperature, pressure and limitswitches are quite alike, they have a common superclass. BaseSupSwitch has also a superclass, Component, that is common for all component classes. The class dependency for the pressure switch class can be written

```
Component-BaseSubSwitch-BasePressureSwitch
```

The Component class

Component contains the attributes Desciption, Specification, HelpTopic, DataSheet, CircuitDiagram, Note and Photo that thus are present in all components. In Description there is place for a short description, in Specification you enter the model specification, the others are used to configure the corresponding methods in the operator environment.

BaseSubSwitch
From the superclass BaseSupSwitch the attributes Switch, AlarmStatus, AlarmText, Delay, SupDisabled and PlcConnect are inherited.

- Switch is a Di object for the pressure switch. It should be connected to a channel
  object in the node hierarchy.
- AlarmStatus shows the alarm status in runtime.
- AlarmText contains the alarm text for the alarm sent at alarm status. The alarm text
  has the default value "Pressure switch, ", but can be changed to some other text.
  Note that if the default text is kept, this will be translated if another language
  is selected. If it is replaced by another text, the translation will fail.
- Delay is the alarm delay in seconds, default 0.
- SupDisabled indicates that the alarm is disabled.
- PlcConnect is a link to the function object in the plc code.

To the main object BaseSupSwitch there is a corresponding function object, BaseSupSwitchFo, which also is inherited by the subclass BasePressureSwitch.

BasePressureSwitch
BasePressureSwitch doesn't have any attributes beyond those inherited from the superclasses. The unique properties in BasePressureSwitch in the graphic symbol, Components/BaseComponent/PressureSwitch, and the object graph where the switch symbol includes a P for pressure.

Configuration
We open the configurator and create the main object, BasePressureSwitch, in the plant hierarchy. In a suitable PlcPgm we insert a function object BaseSupSwitchFo and connect it to the main object with the connect function. Select the main object and click with Shift/Doubleclick MB1 on the function object. The function object contains the code for the component, which for a BaseSubSwitch is an alarm that is sent when the switch signal is lost.
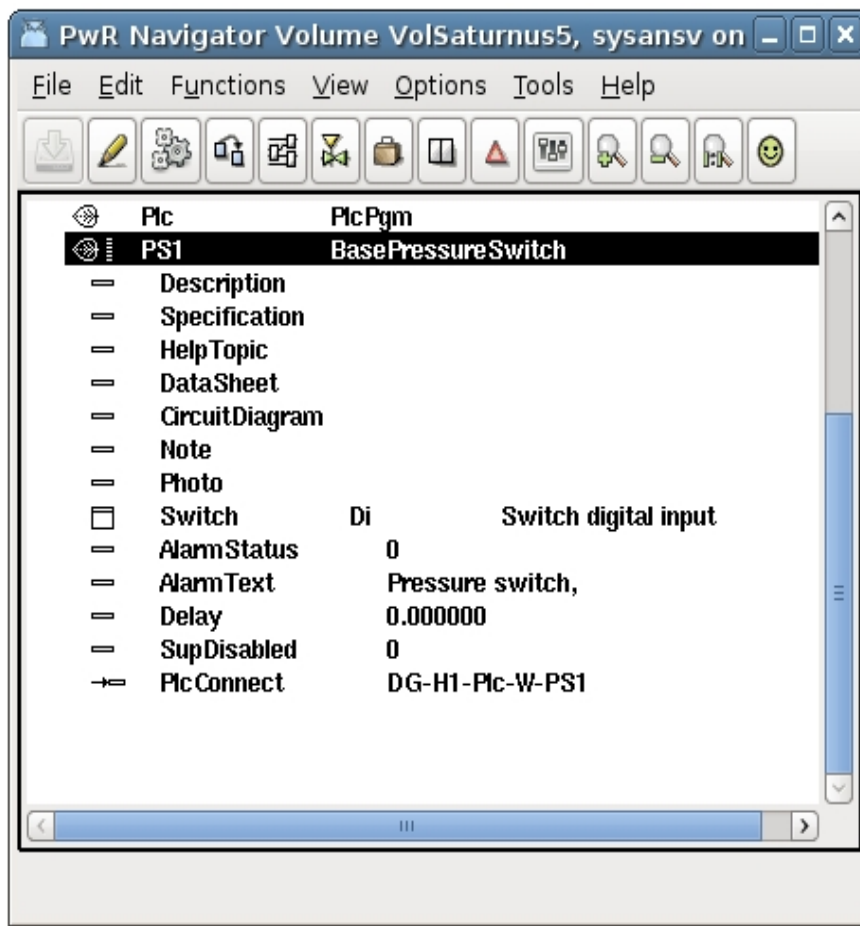
Fig Main object with corresponding function object

The pressure switch is to be viewed in a Ge graph. We open the Ge graph and fetch the subgraph BaseComponent/SupSwitch from the palette. The subgraph is adapted to a BaseSubSwitch object, and all we have to do is to connect it to the main object. Select the main object in the plant hierarchy and click with Shift/Doubleclick MB1 on the subgraph.
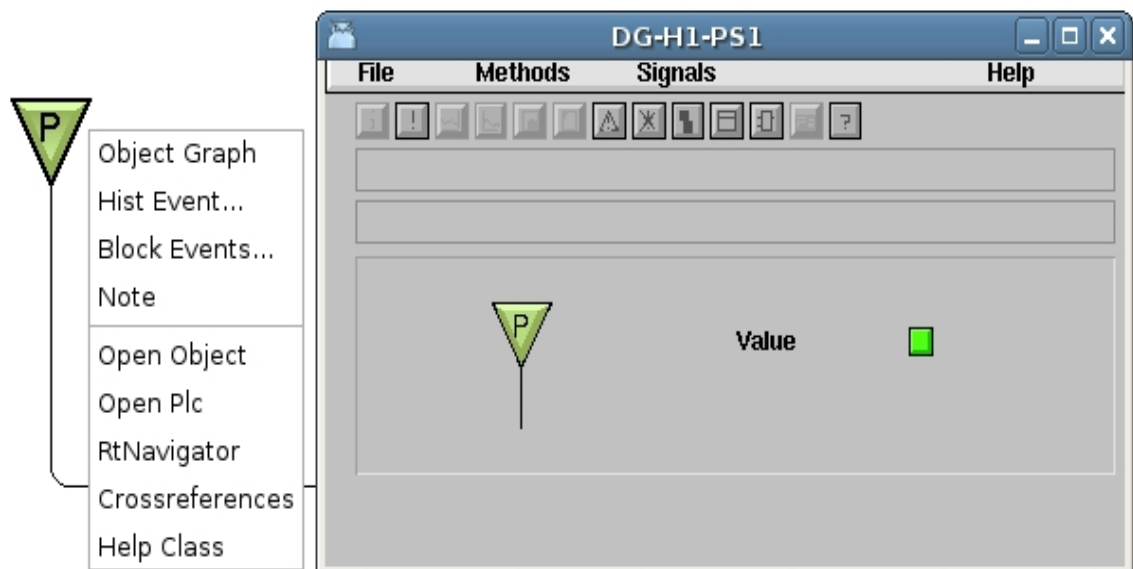
Fig Graphic symbol, popup menu with methods and object graph

We have now accomplished a working component. We can of course also continue to configure the method attributes with helptexts and links to photos, circuit diagrams and datasheet.

Control Valve

Let's have a closer look at a bit more complicated component, BaseCValve, that manoeuvers a control valve. In contrast to the pressureswitch above, you also have to use ConfigureComponent to configure the object, and there is also a simulate object that is used to test the component.

Suppose now that we have a control valve, that is controlled by an analog output, and that gives back the valve position in an analog input. Here we can use a BaseCValve. It has the analog output 'Order' and the analog input 'Position', i.e. the signals we request. Apart from these, there are two digital input signals for switch open and switch closed, but these can be disabled by a configuration.

Configuration
We place the main object BaseCValve in the plant hierarchy and the function object BaseCValveFo in a PlcPgm, and link them together with the Connect function. The functionobject has an order input pin that we connect to a PID object. We also have to state that our valve does not have any switches, and we do this by activating 'ConfigureComponent/PositionNoSwitches' in the popupmenu for the main object. When we open the main object, and in this the Actuator object, we shall find a Position signal, but no input signals for the switches.

File Edit Functions View Options Tools Help

```
    H2              $PlantHier
    Plc             PlcPgm
    CV1             BaseCValve
        Description
        Specification
        HelpTopic
        DataSheet
        CircuitDiagram
        Note
        Photo
        GraphConfiguration   PositionTwoSwitches
        Actuator        BaseActuatorA
            Description
            Specification
            HelpTopic
            DataSheet
            CircuitDiagram
            Note
            Photo
            GraphConfiguration   PositionTwoSwitches
            Order           Ao
            Position        Ai
            SwitchOpen      Di
            SwitchClosed    Di
            Mode            CompModeAM
            MaxPosDiff          5.000000
            DiffAlarmDelay      10.000000
            DiffAlarmText       Position error,
            SwitchAlarmDelay  10.000000
            SwitchAlarmText   Switch error,
            PosSwitchOpen     95.000000
            PosSwitchClosed   5.000000
            IndError          0
            IndWarning        0
            SwitchError       0
            DiffPosError      0
            PosEnum           Between
            SupDisabled       0
            PlcConnect        DG-H2-Plc-W-CV1-W-Actuator
            SimConnect
        Valve           BaseValve
        PlcConnect        DG-H2-Plc-W-CV1
        SimConnect        Sim-PlcSim-W-CV1
```

| CValveFo |
| Av | H2-Order |──| Ord |
| | | CV1 |

Fig Main object with function object
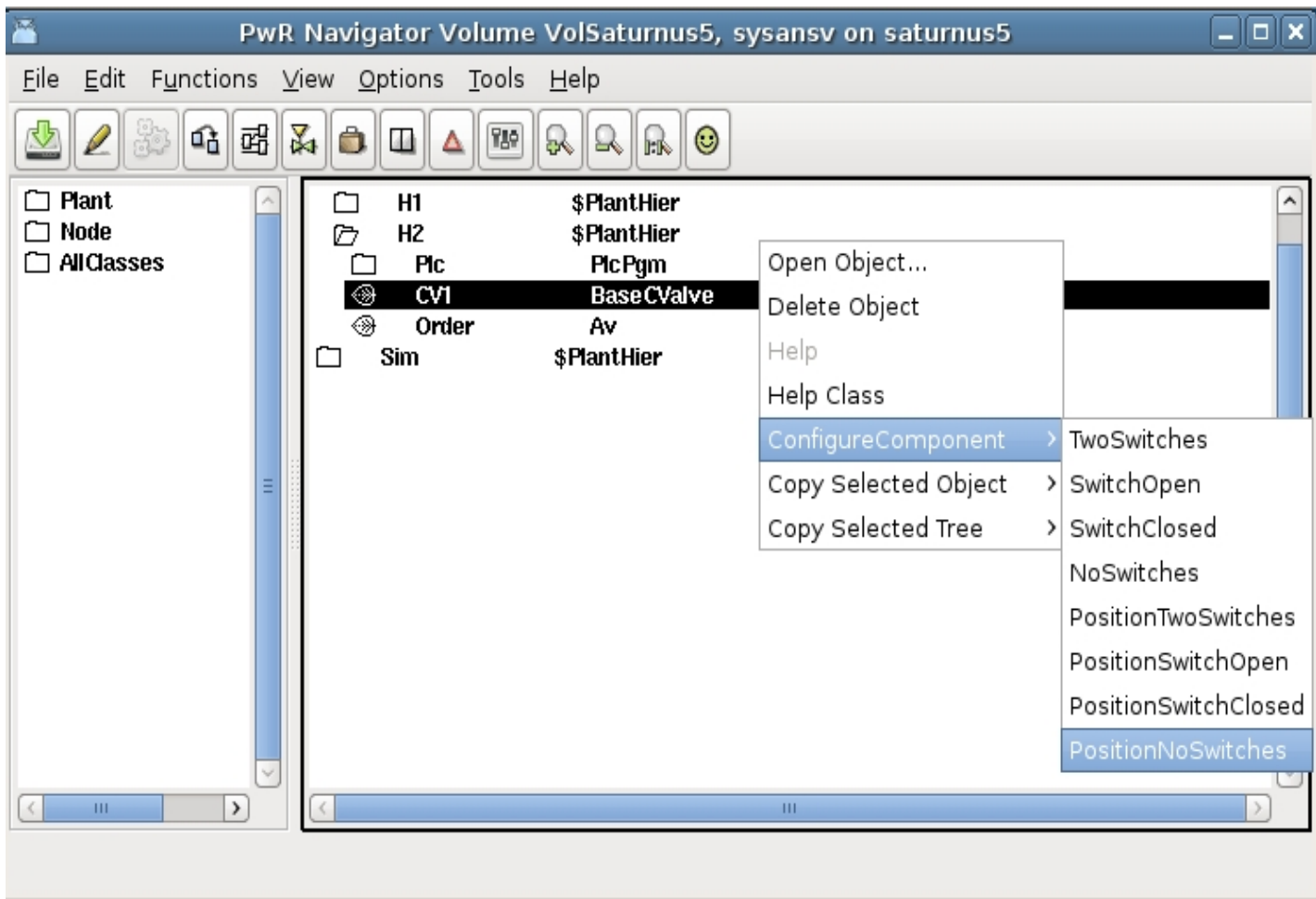
Fig Configuring the main object with ConfigureComponent/PositionNoSwiches

In the HMI we place the subgraph BaseComponent/CValve in a Ge graph and connect it to the main object.

We also want to be able to simulate the valve, to see that it works the way we want. For simulation there is the function object BaseCValveSim that we place in a specific PlcPgm for simulation, as this PlcPgm is not to be executed in the production system. The function object is connected to the main object with the Connect function.
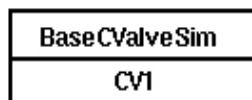
Fig Simulate object for the control valve

The configuration is finished and after building the simulation node we can test the system and examine the result.

Pump drive

The next example is an aggregate, a pump drive with a frequency converter that communicates via Profibus with the protocol PPO3. We will see how a component object in the aggregate, in addition to the usual main object, function object and simulation object, also contains I/O objects to fetch and send data via Profibus.

The class we use is BaseFcPPO3PumpAggr, and the class dependency for this class is

```
Aggregate-BaseFcPPO3MotorAggr-BaseFcPPO3PumpAggr
```

All aggregates have the superclass Aggregate that corresponds to the Component class for components. The next superclass, BaseFcPPO3MotorAggr contains all the functionality for the control. The pump class BaseFcPPO3PumpAggr extends the motor aggregate with a pump object representing the mechanical pump, but this doesn't contain any signal or any additional functionality. The pump aggregate also has its own object graph and graphical symbol.

Configuration
The main object BaseFcPPO3PumpAggr is placed in the plant hierarchy, and the function object, that is inherited from the motor aggregate, BaseFcPPO3MotorAggrFo, is placed in a PlcPgm, and they are linked together by the Connect function.

The main object has no less than 24 different configuration alternatives to choose between, dependent on which of the components Fuse, Contactor, SafetySwitch, StartLock and CircuitBreaker that are present in the construction. In our case we only have a contactor and a frequency converter and we the choose the ConfigureComponent alternative CoFc.

Some components in an aggregate can have their own configurations. In this case the contactor and the motor can be configured individually. Our contactor has two signals, a Do for order and a Di for feedback, and this applies to the default configuration, i.e. we don't have to change this. The motor, however has a temperature switch, and thus we select the motor component and activate ConfigureComponent/TempSwitch in the popup menu.

The next step is to connect the signal objects to the channel objects. The contactor has a Do for order and a Di for feedback that is to be connected to suitable channels in the node hierarchy. The motor has a temperature switch in the shape of a Di that also should be connected. The frequence converter contatins four signals, StatusWordSW (Ii), ActSpeed (Ai), ControlWordCW (Io) and RefSpeed (Ao). These signals are exchanged with the frequency converter via Profibus with the protocol PPO3. There is a specific Profibus module object for PPO3, BaseVcPPO3PbModule, that contains the signals for PPO3 communication. The module is configured by the Profibus configurator (see Guide to I/O System) and is connected to the FrequencyConverter component in the pump aggregate. As the component object and module object are adapted to each other, you don't have to connect each signal, you connect the component to the module instead. By selecting the module object and activating ConnectIo in the popupmenu for the FrequenceConverter component, the connection is made.

We also put the subgraph Component/BaseComponents/FccPPO3PumpAggr in an overview graph and connect this to the main object. Furthermore we place the simulate object BaseFcPPO3MotorAggrSim in specific simulate PlcPgm and connect it to the main object.

Mode

The pump aggregate contains a Mode object in which you configure where the pump is controlled from. It can have the modes Auto, Manuel or Local which means:

- Auto: the pump is controlled by the plc program.
- Manual: the pump is controlled by the operator from the object graph.
- Local: the pump is controlled from a pulpet.

In the mode object you can configure the modes that apply to the pump in question.

12.1 A Component case study

In this section we are going to use components and aggregates to program the control of the level in a water tank,

Process
The process that we will control is shown in figure 'Level control'. The water is pumped from a reservoir to a tank. Our task is to control the level in the tank. To our disposition we have a level sensor and a control valve. In the system there is also a return pipe with a solenoid valve that is always open during the control.

We note that the tank is 1 meter high, which will be reflected in various min and max values in the configuration.
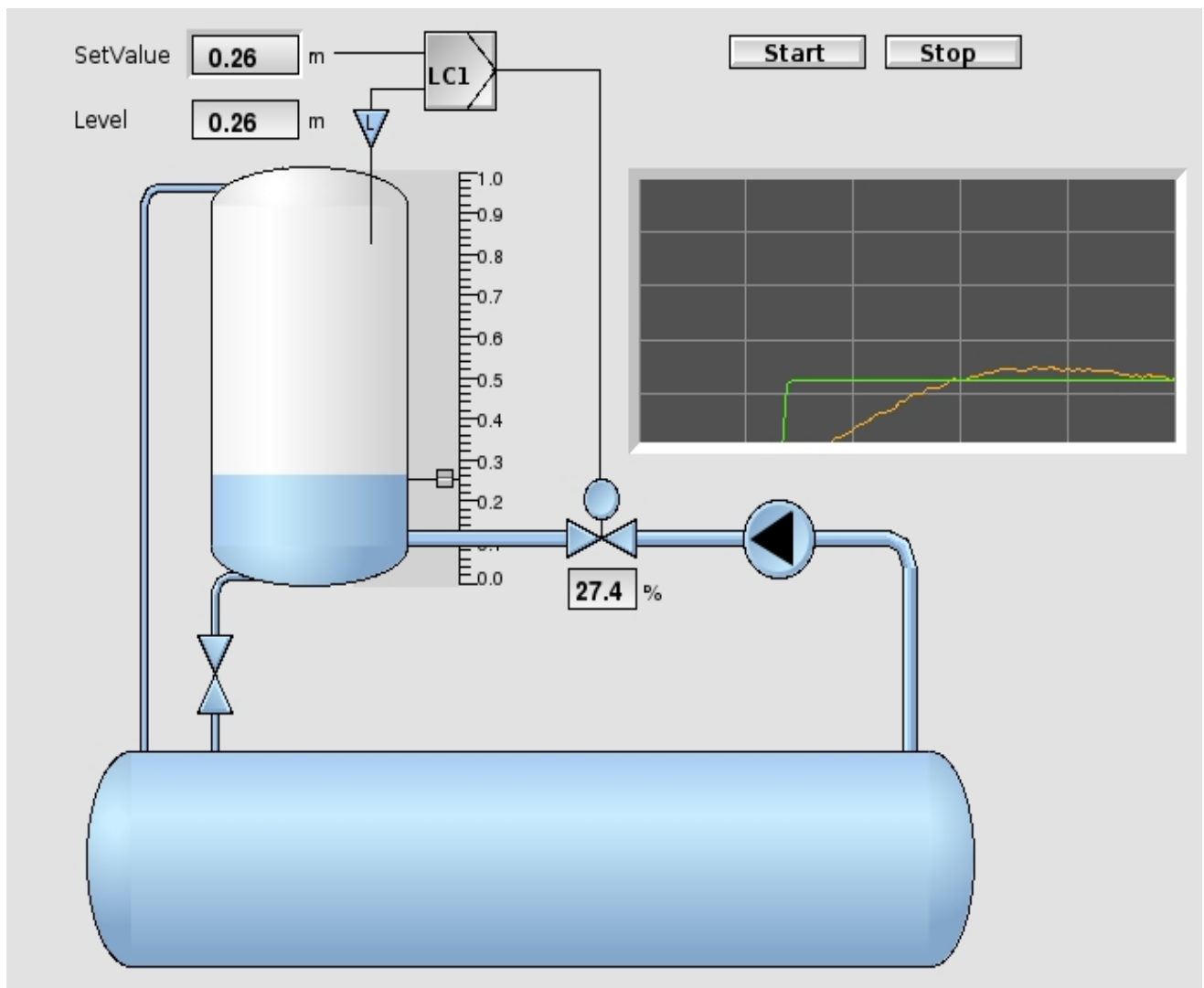
Fig Level control

We can identify the following components and aggregates:
- a contactor operated pump, with circuitbreaker, contactor, overload relay and safetyswitch
- a control valve with two limit switches for open and closed
- a level sensor
- a solenoid valve with two limit switches for open and closed valve

The following component object correspond to the components in the plant:

- Pump	BasePumpAggr
- Control valve	BaseCValve
- Level sensor	BaseLevelSensor
- Solenoid valve	BaseMValve

Configuration in the plant hierarchy

The components are created under the hierarchy LevelControl. Under this we place a PlcPgm 'Plc' that will contain the code for the control, and a PlcPgm 'Simulate' that will contain the code needed for simulation and testing of the program. We also create three Dv objects to start, stop and reset the control program, Start, Stop and Reset.
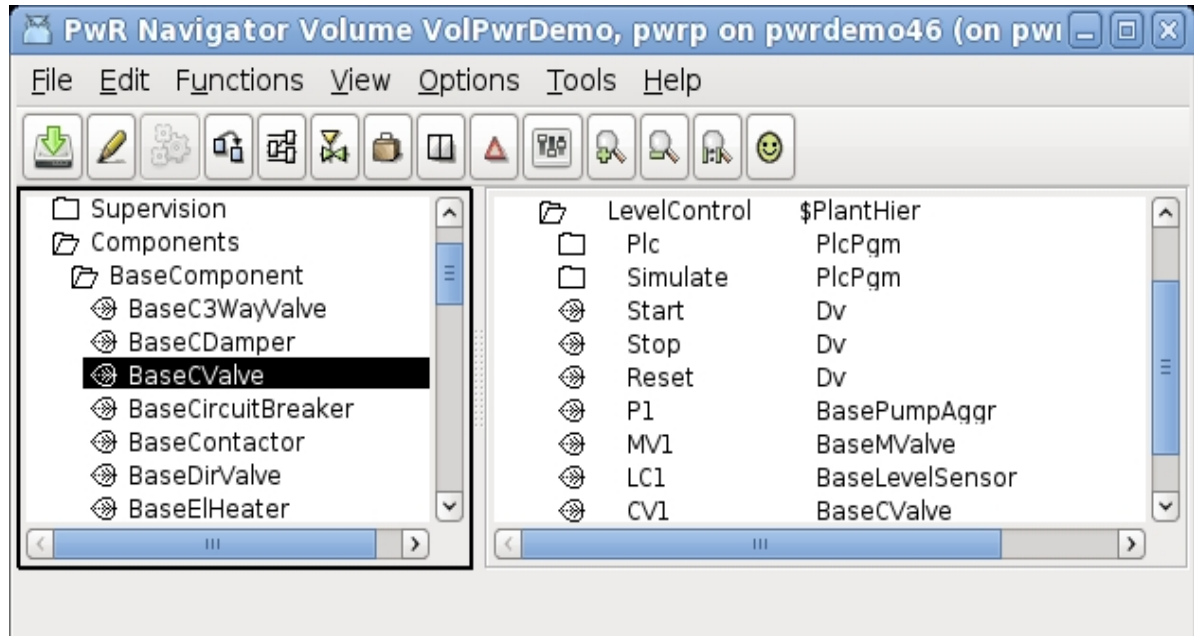
Fig Plant configuration

Pump
The pumpdrive consists of
- a circuitbreaker with one Di
- a contactor with a Do for order, and a Di for feedback
- a safety switch with one Di
- a motor without any signals

For the pump, a BasePumpAggr object is created with the name P1.

The ConfigureComponent alternative that corresponds to the construction is CbCoOrSs (CircuitBreaker, Contactor, Order, SafetySwitch) and we select P1 and activate this alternative in ConfigureComponent from the popupmenu.
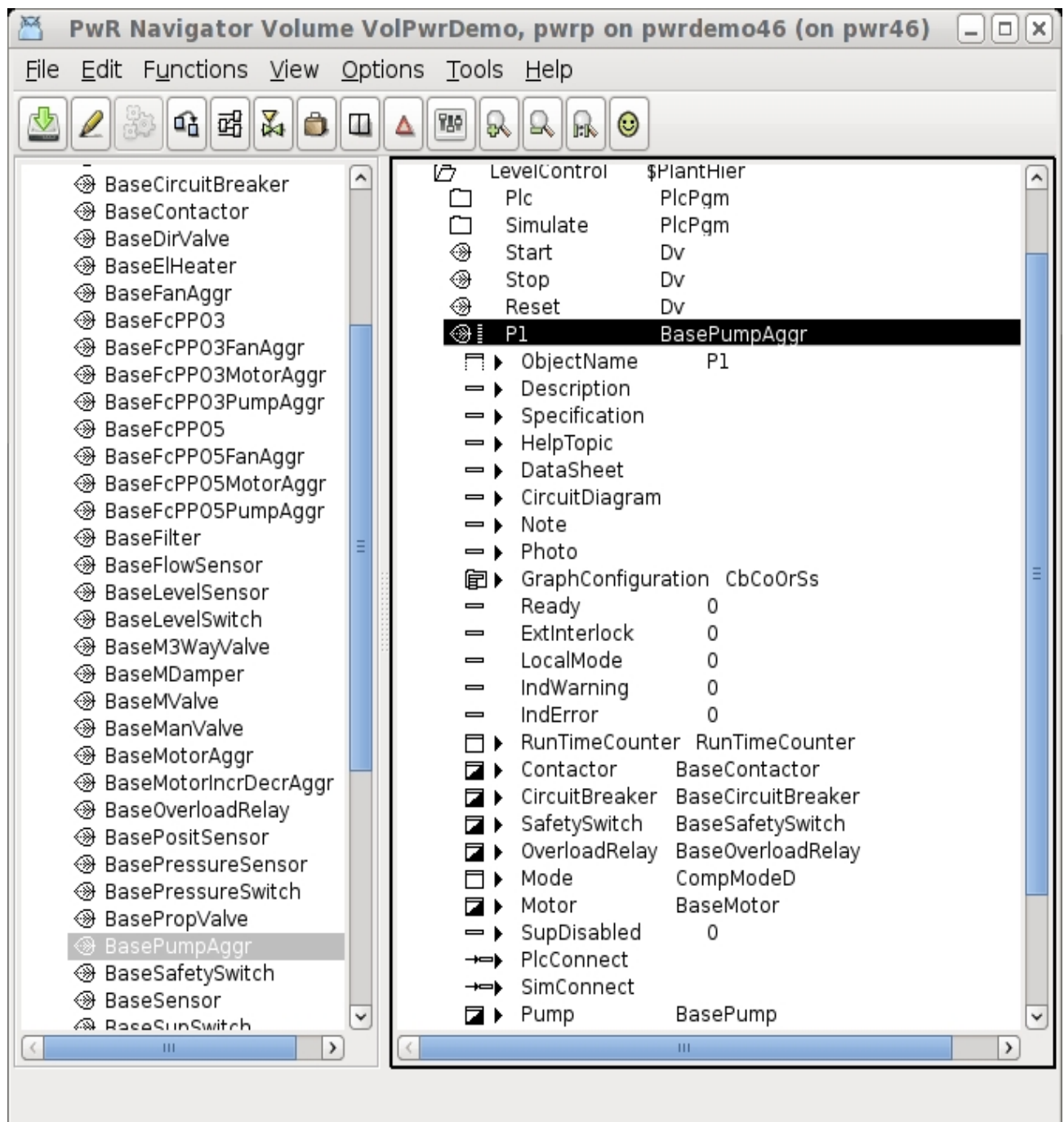
Fig Pump configuration

The components Contactor and Motor have their own configurations, and for them we also have to activate ConfigureComponent. We select Contactor and select ConfigureComponent/OrderFeedback as we have one signal for order and one for feedback. For the Motor object we activate ConfigureComponent/NoTempSwitchOrSensor, as the motor doesn't have any signals.

All the signals in the pump aggregate have to be connected to channel objects in the node hierarchy. We find the following signal objects and connect them to suitable channels:

P1.Contactor.Order   Do contactor order
P1.Contactor.Feedback   Di contactor feedback
P1.CircuitBreaker.NotTripped   Di circuitbreaker not tripped
P1.SafetySwitch.On   Di safety switch on

P1.OverloadRelay.Overload   Di overload relay tripped

Control valve
The control valve has an actuator that is controlled by an analog output signal, and limit switches for open and closed valve.

We create a BaseCValve object with the name CV1. The ConfigureComponent alternative that corresponds to our construction is TwoSwitches.

The following signals are connected to suitable channels in the node hierarchy:

CV1.Actuator.Order   Ao for order
CV1.Actuator.SwitchOpen   Di for switch open
CV1.Actuator.SwitchClosed   Di for switch closed

Level sensor
For the level sensor we create a BaseLevelSensor object with the name LC1. This doesn't have any ConfigureComponent method, but there are some other properties to configure.

The sensor object has alarm limits for HighHigh, High, Low and LowLow and these are stated in the attributes LC1.LimitHH.Limit, LC1.LimitH.Limit, LC1.LimitL.Limit and LC1.LimitLL.Limit. We set the limit values to 0.95, 0.90, 0.10 and 0.05. We also set the upper limit for presentation of the value in LC1.Value.PresMaxLimit to 1. This will affect the range in bars and trends.

Solenoid valve
The solenoid valve is controlled by a digital output and has feedback in the shape of digital inputs for limitswitch open and limitswitch closed.

For the solenoid valve we create a BaseMValve object with the name MV1.

In ConfigureComponent we activate TwoSwitches that corresponds to the current configuration with both limitswitches present.

The signals that are to be connected to channels in the nodehierarchy are:

MV1.Order   Do for order to open the valve.
MV1.SwitchOpen   Di for limitswitch open.
MV1.SwitchClosed   Di for limitswitch closed.

Plcprogram

The next step is to write the plc program for the level control, in which the function objects for the components will be inserted:

- BaseMotorAggrFo for the pump P1
- BaseCValveFo for the control valve CV1
- BaseMValveFo for the solenoid valve MV1
- BaseSensorFo for the level sensor LC1

We create the function objects and connect them to their main objects, by selecting the main object and activate Connect in the popupmenu for the function object.

We will also use a PID controller to control the level in the tank. The controller will have the value from the level sensor as process value, and set out the outvalue to the

control valve. The inflow to the tank will then be adjusted to reach the desired level. The controller is created with the functionobjects Mode and PID.

The program is built around a Grafcet sequence with four steps. See Fig Plc program

1. The initial step IS0 is the resting position when the pump is turned off and all the valves are closed.

2. When the Start Dv is set from a button in the operator graph, step S0 is activated. Here the pump is started as the sorder Ord0 is connected to the start inputpin of the functionobject for the pump P1. When the pump is started, the On outputpin of the pump object is set, and the activity is moved to the next step S1. Note that the sorder Ord0 continues to be active, i.e. the pump is turned on until the reset in step S2.

   The value of the error outputpin Err of the pump object is set into the Reset Dv. Reset is stated as ResetObject in the PlcPgm object, and all the sequences in the PlcPgm will be reset when Reset is set, i.e. the sequence will return to the initial position and the pump and the control will be turned off.

3. S1 is the working step, where the controller is active, controlling the level in the tank. As long is S1 is not active, the force input of the mode object is set, and the controller has 0 as outvalue. When the step is active, the controller fetches the processvalue from the output of the sensor object LC1, and the setpoint is set into the mode object LC1_Mode.SetVal from the operator graph. The outvalue of the controller is connected to the order inputpin of the control valve. The order Ord1 is also connected to the order input of the solenoid valve, which opens the valve.

4. Step S1 is active until the Stop Dv is set from a button in the operator graph. When the step is left, the controller is again forced to zero and the control valve is closed. Also the solenoid valve is closed. The step S2 is active for a moment, reseting the sorder Ord0, thus stopping the pump. Then the sequence returns to the resting position IS0.

The Grafcet sequence also requires a reset object, the Reset Dv, to be inserted in the ResetObject attribute of the PlcPgm.

The mode object LC1_Mode and the controller LC1_PID requires som additional configuration. In the mode object
- OpMode = Auto to start the controller in auto mode
- MaxSet = 1, maximum setpoint value is the height of the tank, 1 m
- SetMaxShow = 1, also the height of the tank
- SetEngUnit = m, meter
- PidObjDid = LevelControl-Plc-W-LC1_PID, the name of the PID object

In PID object
- PidAlg = PID
- Gain = 100
- IntTime = 10
- MaxGain = 200
- SetMaxShow = 1
- SetEngUnit = m
- ModeObjDid = LevelControl-Plc-W-LC1_Mode, the name of the mode object
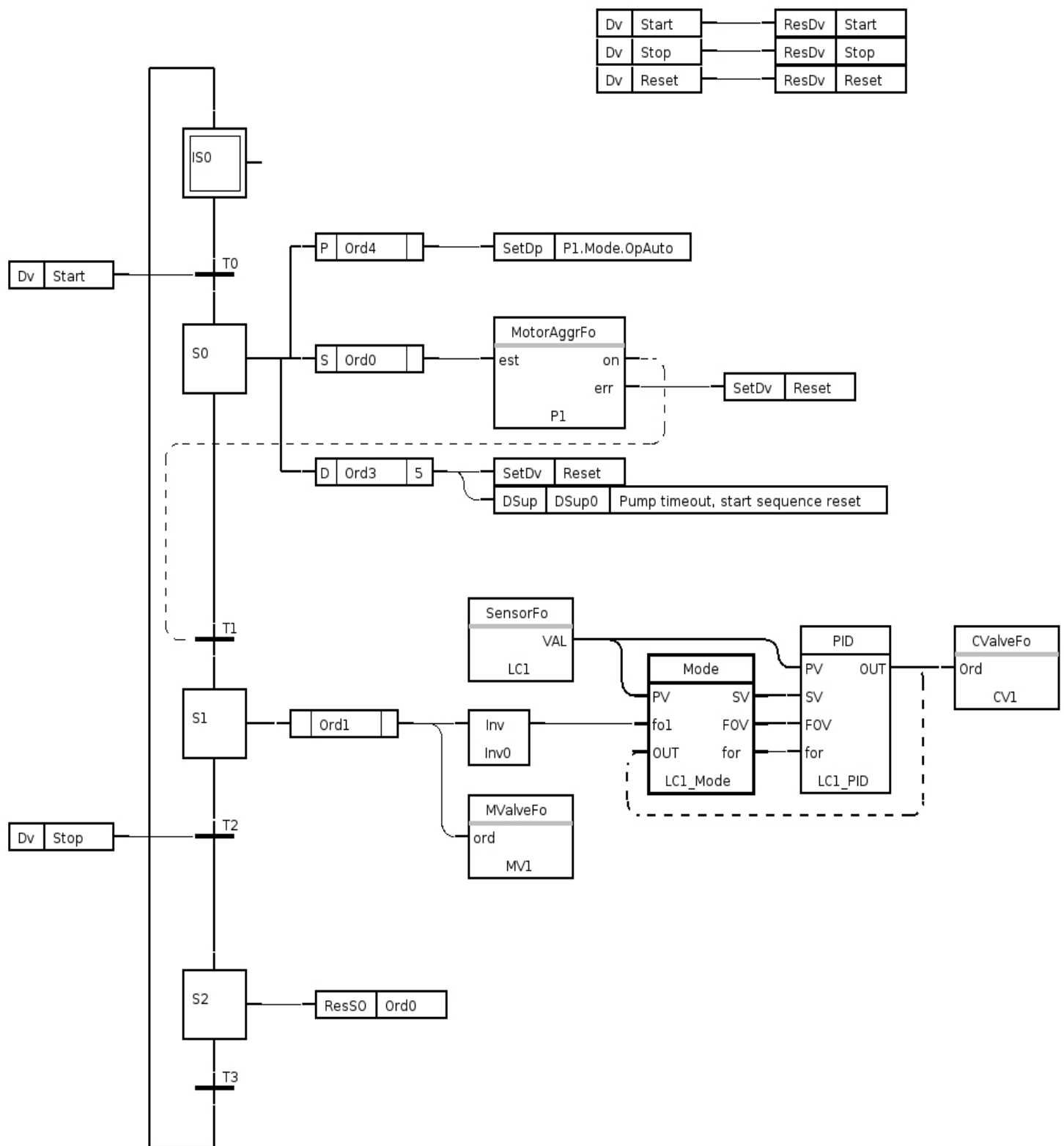
Fig Plc program

Simulate program

We create the simulate program to be able to test all the functions in the program, alarm handling and operator graphs before the commissioning. You can also use it for education and

demonstration.

The simulate code is put in a separate program 'Simulate', that should not execute in the production system. In this program, the simulation objects for the components are created:

- BaseMotorAggrSim for the pump P1
- BaseCValveSim for the control valve CV1
- BaseMValveSim for the solenoid valve MV1
- BaseSensorSim for the level sensor LC1

The function objects are connected to their main objects, by selecting the main object and activating Connect in the popupmenu of the function object.

The simulate objects for the pump, control valve and solenoid valve don't have any in or output pins, they work solely against data in the main object where they read output signals and set suitable values into the input signals. The simulate objects also have an object graph, from which you can influence the simulation and cause different faults in order to check that the errors are handled in a proper way, and that the operator is informed via process graphs and alarms.

The simulate object for the level sensor, however, has an input pin, and for this we have to calculate a simulated level in the tank. A change of the level is determined by the input flow minus the output flow divided by the area of the tank. If we assume that the input flow is proportional to the order output to the control valve (CV1.Actuator.Order), and subtracts the out flow throgh the solenoid valve when this is open. The change in flow is accumulated in the OA1 output of the CArithm, which is sent forward to the simulate object of the level sensor LC1. In LC1, some noise is added to the signal to get a more realistic appearence. This is achieved by setting LC1.RandomCurve to 1, and LC1.RandomAmplitude to 0.01.
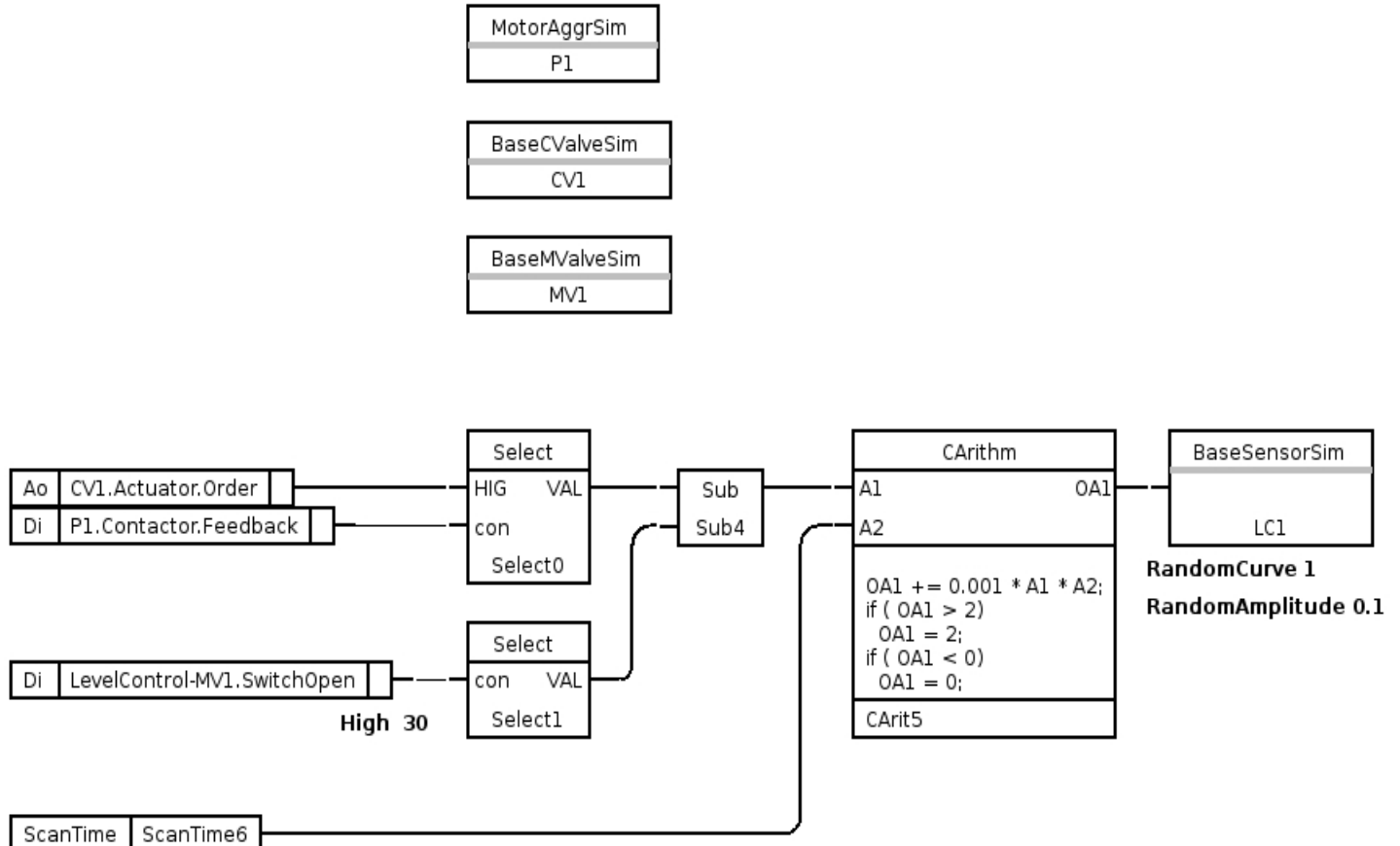
Fig Simulate program

Process graph

The process graph for the level control is drawn in Ge. We find the graphic symbols for the components in the Ge palette:

- Component/BaseComponent/PumpAggr for the pump
- Component/BaseComponent/CValve for the control valve
- Component/BaseComponent/MValve for the solenoid valve
- Component/BaseComponent/LevelSensor for the level sensor

The symbols have the dynamic HostObject, which means that they have a preprogrammed dynamic that is connected to different attributes in the object. You only need to insert the object name of the main object in HostObject.Object, or connect them by selecting the main object and click with DoubleClick Ctrl/MB1 on the symbol.

The default dynamic doesn't include opening the object graph when clicking on the symbol. We add this function by opening the attribute editor for the symbol and adding OpenGraph in action (if no OpenGraph.GraphObject is stated, the object graph is opened).

We also assemble a tank, from a rectangle and two halfellipses, on which we set fill and gradient properties and also group. On the group, the dynamic FillLevel is set, and FillLevel.Attribute is connected to the value of the level sensor, LevelControl-LC1.Value.ActualValue. The min and max values for FillLevel are set to 0 and 1.

To the left of the tank a slider is placed, from which the setpoint of the level is set. It consists of a Slider/SliderBackground3 and a Slider/Slider3. The slider is connected to the setpoint in the mode object LC1_Mode, i.e. LevelControl-Plc-W-LC1_Mode.SetVal. The min and max values for the slider are set to 0 and 1.

For the setpoint, also an input field 'SetValue' is created, which is connected to the same value as the slider above. The process value of the level is displayed in the 'Level' field, which is connected to the value of the level sensor.

The controller symbol is fetched from Process/PID_Controller in the palette. It has no dynamic as default, but we want the object graph for the mode object to be opened when clicking on the object, so we add Command to Action with the command

```
open graph /class/instance=LevelControl-Plc-W-LC1_Mode
```

The pushbuttons for Start and Stop are of type Buttons/SmallButton. SmallButton has ToggleDig as default action, but we want SetDig instead, as the Start and Stop signals are reset by the plc program. We remove Inherit in Actions to avoid ToggleDig and add SetDig instead, and connect to the Start and Stop Dv.

The trend curve is fetched from Analog/Trend in the palette. The process and set values are to be displayed here, i.e.
- Trend.Attribute1 is set to LevelControl-LC1.Value.ActualValue (the value of the level sensor).
- Trend.Attribute2 is set to LevelControl-Plc-W-LC1_Mode.SetVal (the set value in the mode object).

Finally we draw some pipes and lines and the graph is finished.

We also enter File/Graph attributes and insert the coordinates for the upper left and lower right corners in x0,y0 and x1,y1. DoubleBuffered is set to 1 and MB3Action to PopupMenu.
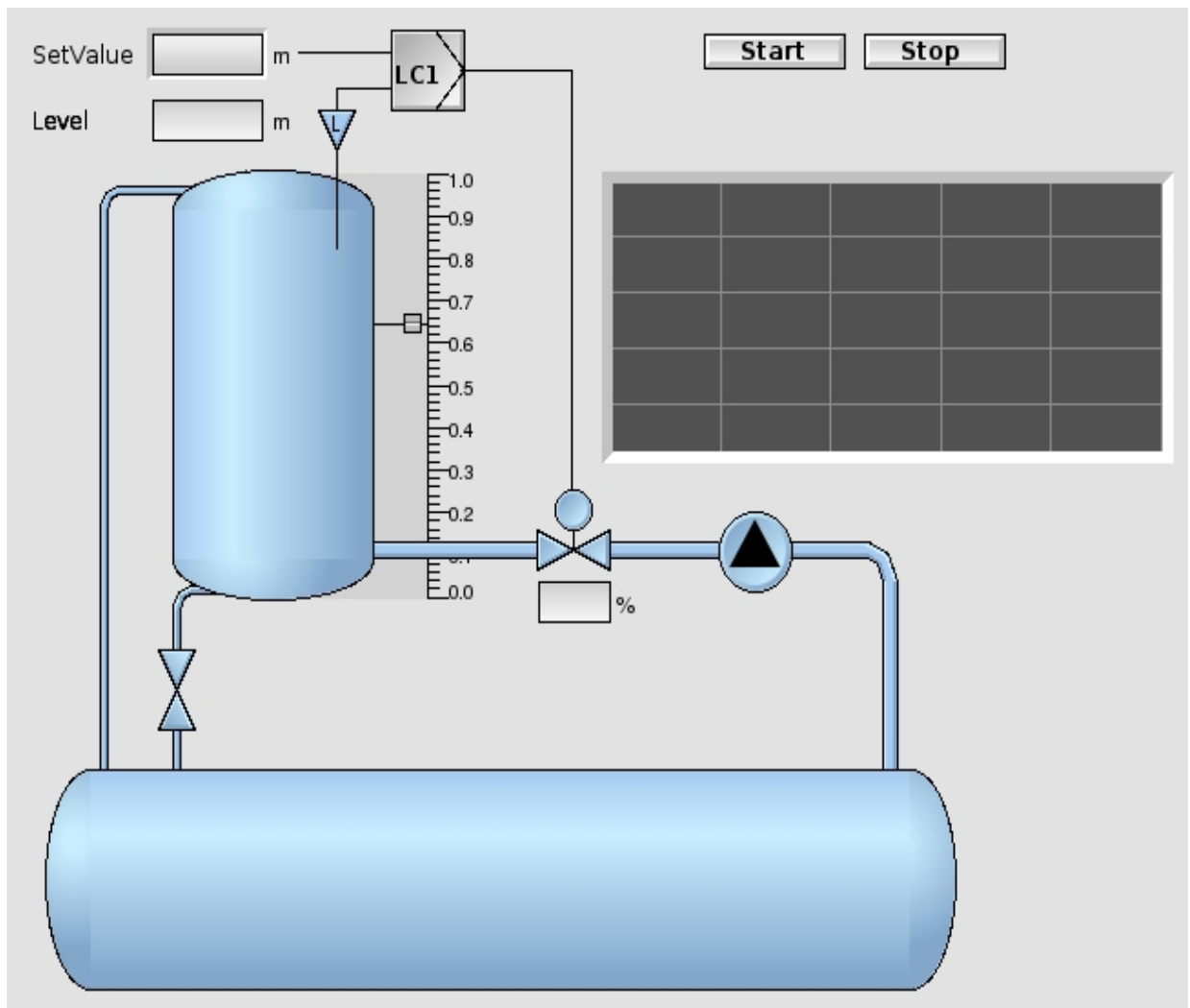
Fig Graphic editor

Simulation

To look at the result of our programming effort, we start the simulation.

When we open the graph, both valves are colored white, which marks that they are closed.
The pump is not started, which is marked with gray color and the triangle in the pump
symbol doesn't point in the flow direction. The tank is colored white, i.e. it is empty, and
the levelsensor flashes red as the level is beneath the LowLow level in the level sensor object.

By pressing the start button, we leave the resting step in the Grafcet sequence, and activate
the step that starts the pump (S0). When the pump is started, it is colored blue and the
triangle points in the flow direction. In this step, also the solenoid valve is opened and
colored blue. When the pump has started, the sequence proceeds to the working step S1.

We set a set value with the slider to approximately 0.3 and hopefully the controller starts
to work. Eventually, some adjustment of the controller parameters are needed, and the controller
graph is opened by clicking on the controller symbol, which opens the object graph for the
Mode object. In this, we click on the PID button to open the object graph of the PID object.
Here we can adjust the gain (Kp) and integration time (Ti).
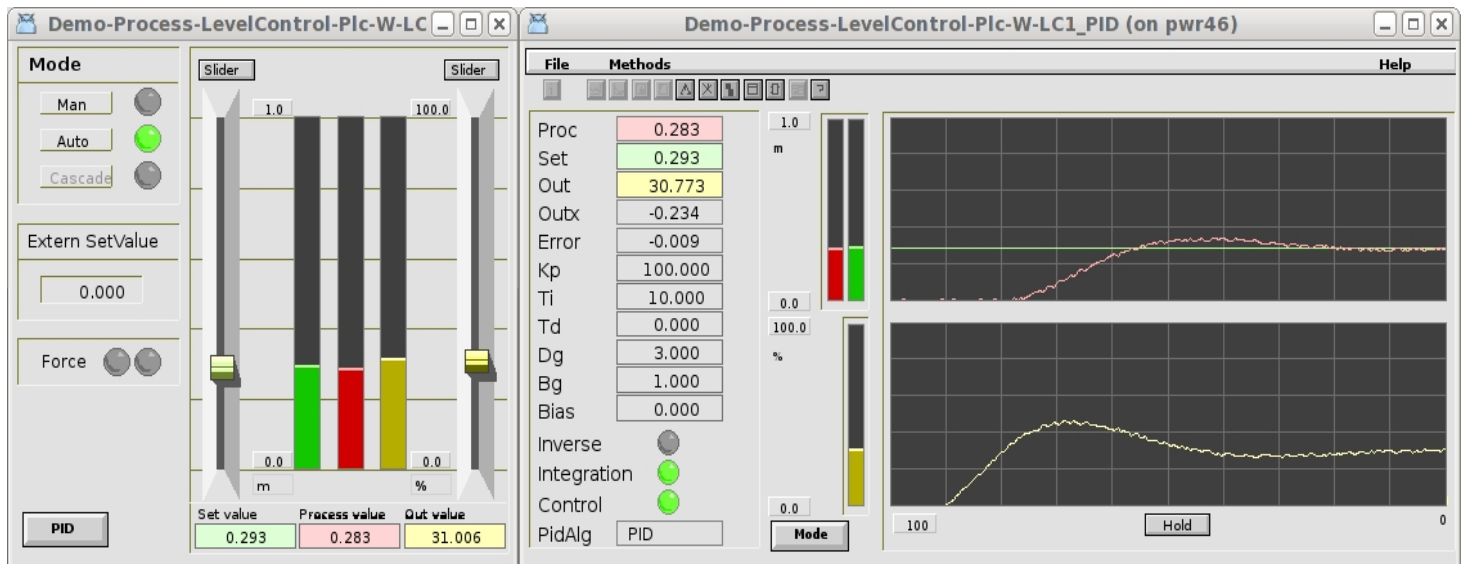
Fig Object graph mode and PID object.

Let's have a look at what we can do with the components in the process graph.

Level sensor

If you rightclick on the level sensor, a popup menu is opened with the methods that are defined for the sensor. With OpenPlc you open plc trace for the function object of the component. With RtNavigator the object is looked up and viewed in the navigator, with Trend a trend curve for the level is displayed and with OpenGraph the object graph is opened. The object graph can also be opened by clicking on the symbol.

The upper part of the object graph for components and aggregates has a similar appearance. There is a menu where you under 'Methods' can activate the methods of the component. Under 'Signals' you can see the signals in the component and open the object graph for them. For aggregates you can also see the components and open the object graph for them under 'Components'. There is also a toolbar with pushbuttons for the methods, and two text fields that display Description and Specification for the component. In the lowest row in the graph the Note message is viewed if such a message is inserted (by the Note method).

Furthermore the level is displayed as a number and with a bar, and the alarm limits are also viewed. The alarm limits can be adjusted with sliders and enabled or disabled by check boxes.

In the upper right corner of the graph there is a button marked with an 'S'. It is only visible in simulation mode (i.e. IOSimulFlag in the IOHandler object is set) and it opens the simulate graph. From the simulate graph you kan influence the simulated signal. We have already configured Random with amplitude 0.01, but you kan also add a sinus curve or a sawtoothed curve to the signal.
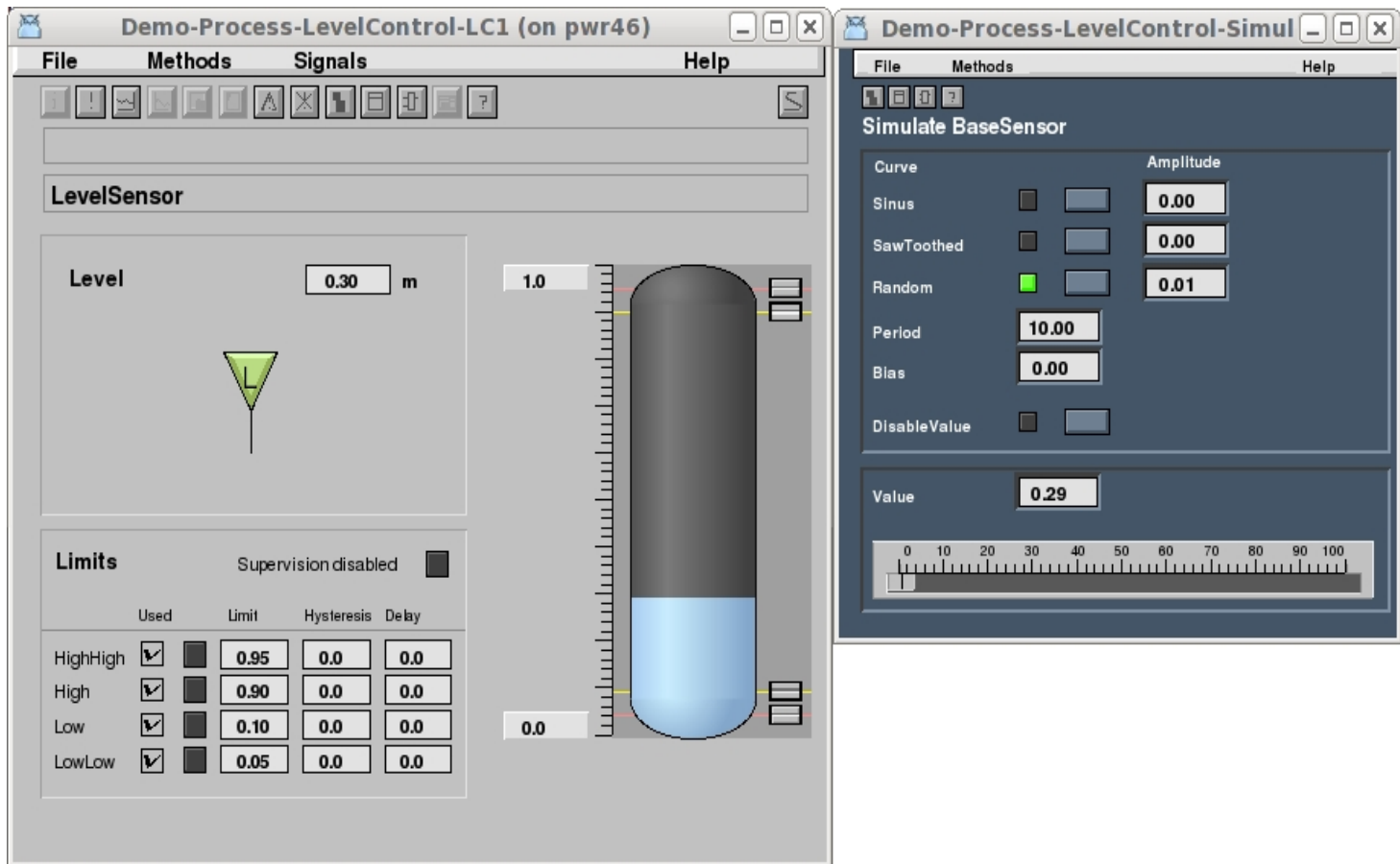
Fig Object graph for the levelsensor and simulate graph

Control valve

The object graph for the control valve has indicators for the limit switches and shows the order output to the valve as a bar.

You can take over the valve in manual mode, i.e. the valve position is now adjusted with the slider 'Manual' instead of beeing fetched from the outsignal from the controller. The control loop is now out of order and the water flow is adjusted from the slider.

From the simulate graph you can for example influence the simulation of the limit switches. If order is 0 and switch closed are not affected, you receive a limit switch alarm and a red flashing symbol. At simulation the simulate objects set the correct values into the limitswitches, but this can be overridden from the simulate graph. By pushing 'Manual Control' the switch is controlled from the graph instead, and by zeroing the limit switch you can check that the limitswitch supervision works.
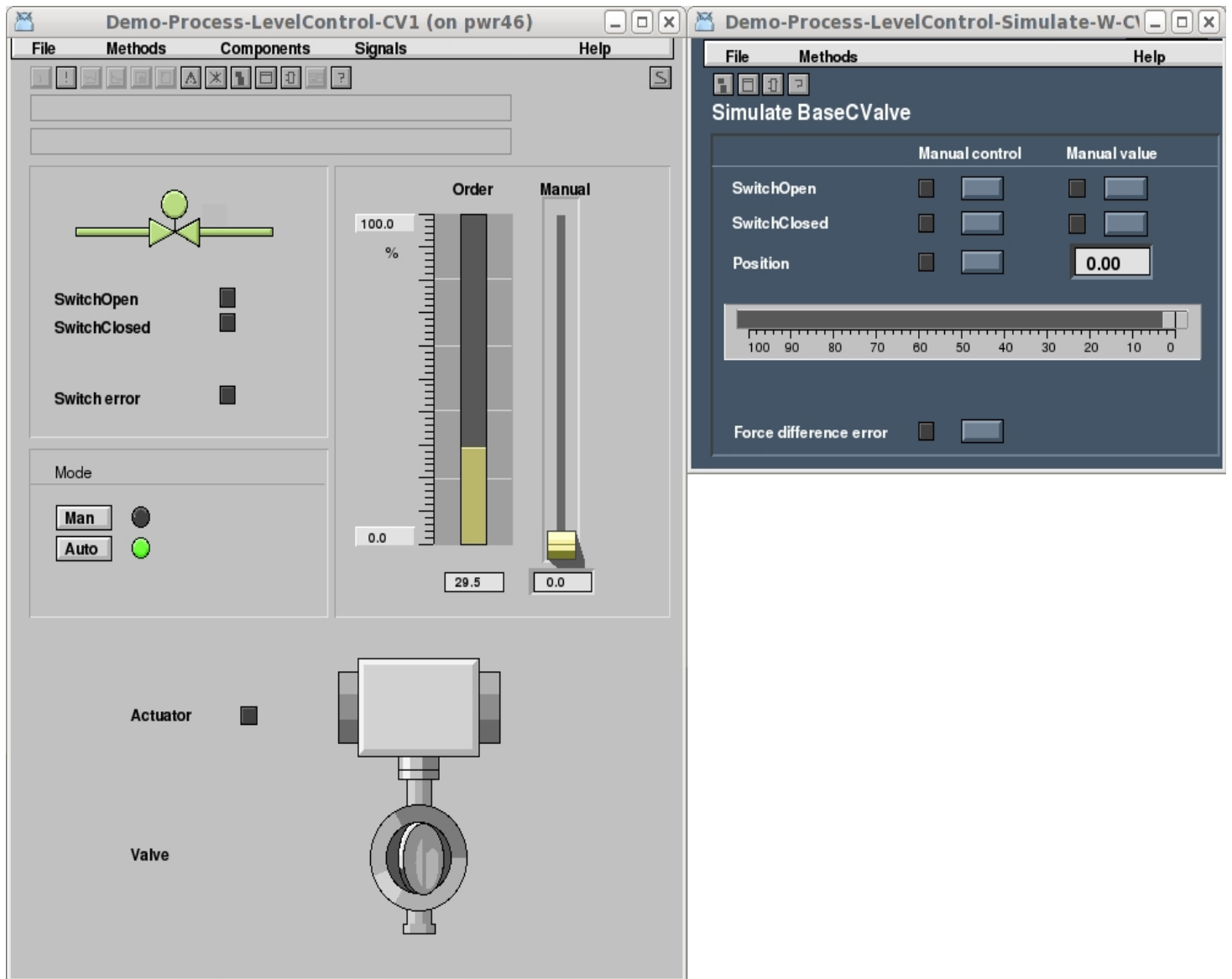
Fig Object graph for the controlvalve and simulate graph

Solenoid valve

The object graph for the solenoid valve displays limit switches and order signal
with indicators. The valve is switched to manual control by clicking on the Man button,
and can now be maneuvered by the Open and Close buttons.

From the simulate graph you can, as for the control valve, influence the simulation of the
limitswitches and trigger a switch error alarm.
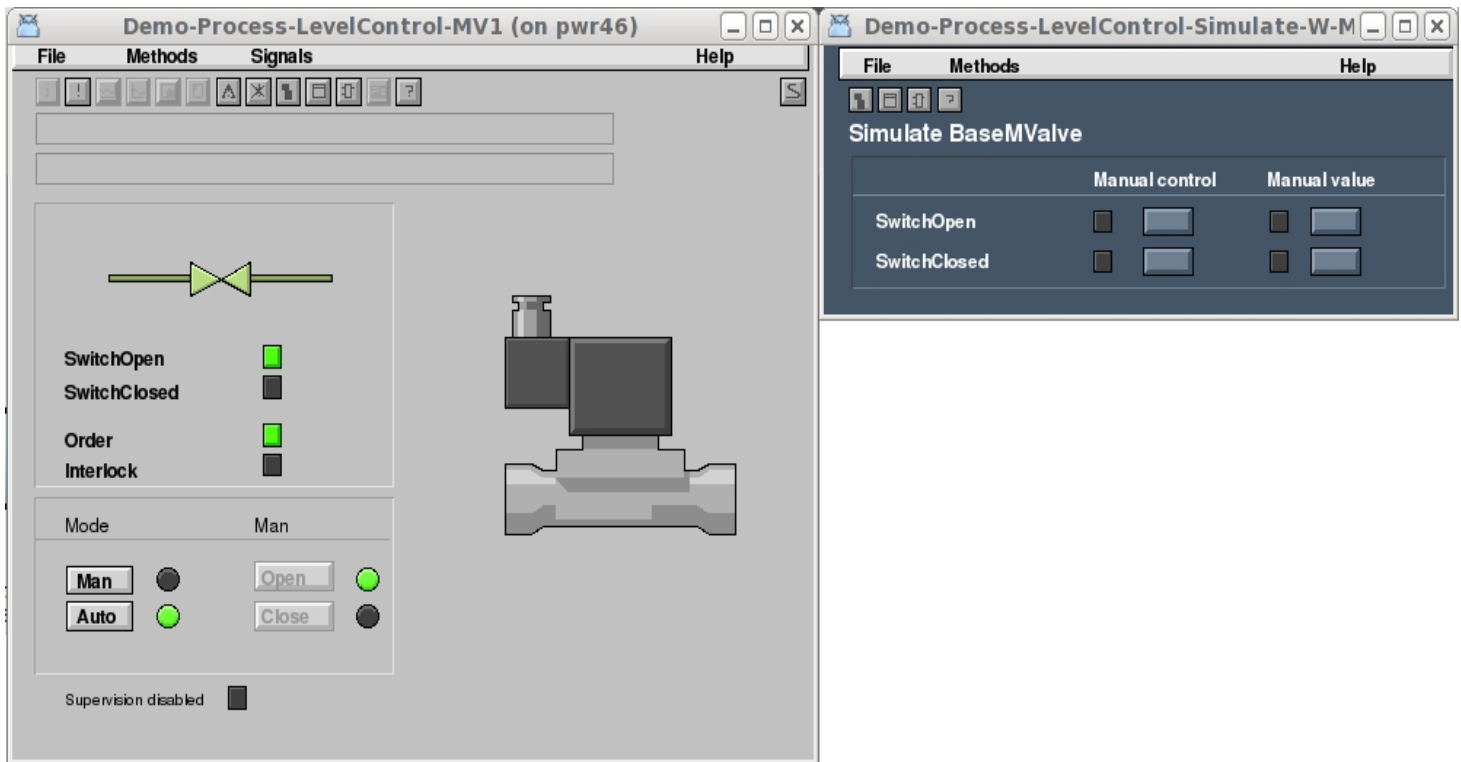
Fig Object graph for the solenoidvalve and simulate graph

Pump

The object graph for the pump shows a schematic drawing of the components in the pump, and also a status indicator for each component. By clicking on a component, you open the object graph for the component.

With the 'Man' button you can switch the mode to manual and start and stop the pump from the Start and Stop buttons in the graph.

From the simulate graph various events can be simulated.
- 'SafetySwich on' simulates that someone activates the safety switch. This causes the pump to turn off, and the pump symbol is colored yellow. Also the Err outputpin of the function object is set. As This is connected to the Reset Dv the sequence is reset and the control is turned off.
- 'CircuitBreaker tripped' simulates that the circuitbreaker has tripped.
- 'Contacor feeback lost' simulates that the contactor feedback is lost.
- 'OverloadRelay tripped' simulates that the overloadrelay has tripped.

Demo-Process-LevelControl-P1 (on pwr46)

File	Methods	Components	Signals	Help

Ready

Interlock

Mode

Man   Start

Auto   Stop

Local

Starts   10

Run   4 1:39:53

Circuit breaker

Contactor

Overload relay

Safety switch

Demo-Process-LevelContro

File	Methods	Help

Simulate BaseMotorAggr

CircuitBreaker tripped

Contactor feedback lost

OverloadRelay tripped

SafetySwitch on
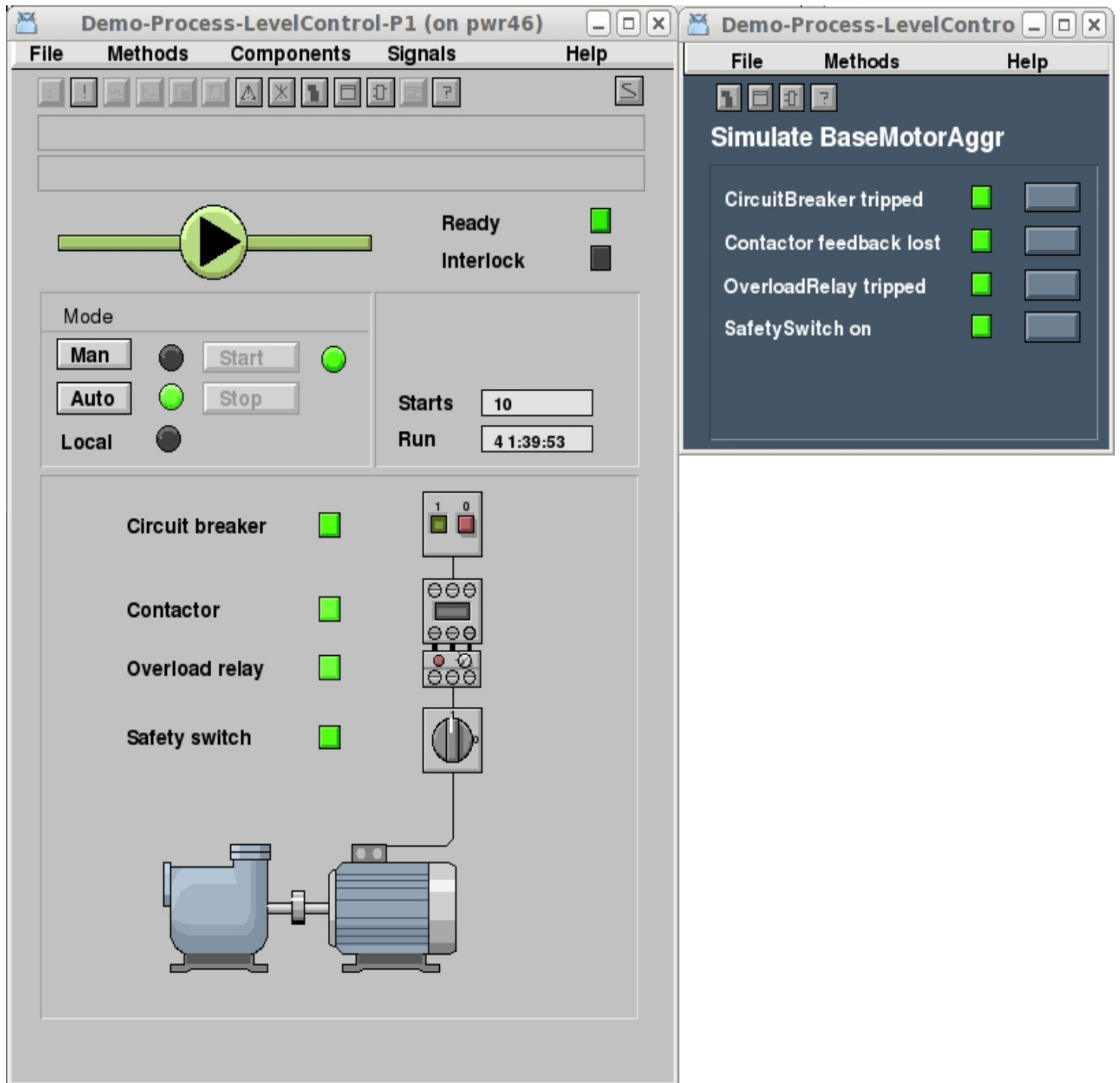
Fig Object graph for the pump and simulate graph

13   Alarms and events

The event handling in ProviewR displays alarms, information messages and events in two lists, the alarm list and the event list. The alarm list contains active or unacknowledged alarms and info messages, and supplies operators and maintainers with information about the current state of the process. The event list contains events, for example when an alarm is activated, and gives a picture of what has happened in the system.

Alarms and events are generated by supervision objects. The most common are DSup and ASup objects that supervises digital and analog signals. The supervision objects are scanned by the event monitor that sends messages about alarms and events to different outunits, eg the alarm and event list Xtt.

13.1   Alarms

An alarm is a state changed in the process or control system that requires attention from operators. The alarm is presented in an alarm list, often with a audible signal, and the operator has to acknowledge the alarm before the audible signal disappears and the alarm is removed from the list. Also the alarm state has to return before the alarm is removed.

An alarm consists of three events
- the alarm enters active state.
- the alarm state is returned from active to normal state.
- the alarm is acknowledged by an operator.

Priority
The priority of an alarm can be A, B, C or D, where A has the highest priority. The priorities have different color marking. A alarm are indicated with red, B alarms with yellow, C alarms with blue and D alarms with violet.

Type
There are seven different types of alarms the can be used to address alarms to different categories of receivers.

Alarm	Ordinary alarm for plant operators.
MaintenanceAlarm	Alarm for maintenance personnel.
SystemAlarm	Alarms for system manager.
UserAlarm1-4	User defined alarm types.

The alarm type is matched with the EventSelectType in the OpPlace object, and the alarm list will only display the selected alarm types.

13.2   Info messages

The info messages are similar to alarms but doesn't have any priority. It can also be configured to only be displayed in the event list, not in the alarm list.

Type

There are two types of Info messages, Info that is indicated with white color, and InfoSuccess that is indicated with green.

13.3   Events

An event is a change in the process or control system that is logged in an event list. They are used to see the history of process and for example draw conclusions about the chain of events if something has gone wrong.

One type of events are alarms that becomes active, returns and are acknowledged, but also other events can be logged in the event list.

13.4   Supervision objects

Alarms and events are configured with supervision objects, primarily DSup and ASup objects. A DSup object supervises a digital object, and generates an alarm or event when the digital signal indicates alarm state. An ASup object supervises an analog signal, and generates alarms or events when the analog value exceeds an alarm limit. Other supervision objects are CycleSup the supervises cyclic processes, NodeLinkSup that supervises network links to other nodes, and SystemSup the supervises functions in the control system.

DSup and ASup

DSup and ASup are the most common supervision objects and supervises digital and analog signals. They also exists in the form CompDSup and CompASup used in components and aggregates.

The DSup and ASup objects can either be placed in the plc code, and connected to the digital or analog value that is to be supervised, or placed in the plant hierarchy, for example under a signal. If the Sup object is positioned below a signal, ActualValue will be supervised as this is automatically inserted into Attribute in the Sup object. If the Sup object is placed under another object Attribute has to be filled in manually to indicate which attribute in the object should be supervised.

The DSup and ASup object can generate alarms, info messages and events. The configuration of the attributes EventType, EventPriority and EventFlags will decide which types of alarms and events are generated.

EventType

EventType configures which type of alarm or info messages are generated. The different alarm types are Alarm, MaintenanceAlarm, SystemAlarm and UserLarm1-4. Info messages can be Info or InfoSuccess.

EventPriority

EventPriority configures the priority for an alarm. The priority can be A, B, C or D. For info messages priority has no significance.

EventFlags

EventFlags is a bitmask with several options that decides which events will be generated and where they will be distributed. See Object Reference Manual for more info.

Various types of alarms and events

A alarm with beep

EventType   Alarm
EventPriority   A
EventFlags   Return, Ack, Bell

B alarm with beep

EventType   Alarm
EventPriority   B
EventFlags   Return, Ack, Bell

Info message displayed in the alarm list

EventType   Info
EventPriority   -
EventFlags   Ack, Bell, InfoWindow, Returned

InfoSuccess message displayed in the alarm list

EventType   InfoSuccess
EventPriority   -
EventFlags   Ack, Bell, InfoWindow, Returned

Info message displayed only in the event list

EventType   Info
EventPriority   -
EventFlags   Returned

NodeLinkSup

A NodeLinkSup object supervises the network link to another node. If link is disconnected
an alarm is generated.

CycleSup

A CycleSup object supervises a cyclic process or thread. If the scan of the process is delayed
an alarm is generated or the emergency break is set. CycleSup are used to supervise plc
threads but can also be used for other processes.

SystemSup

In the MessageHandler object there is an array of SystemSup objects supervising functions in
the control system. An alarm or info message is generated at the following events

- A network link is disconnected.
- At system start.
- At restart.
- At restart of an outunit for the message handling.
- System emergency break.
- When system status indicates warning or error.
- When the errorcount of an IO device has exceeded soft or hard limit.
- When alarm quota is exceeded.

The SystemSup object can be edited to change alarm texts, priorities etc, or to disable unwanted alarms.

13.5   Event monitor

The server process handling alarms and events are called the event monitor and has the name rt_emon. It's configured with a MessageHandler object in the node hierarchy.

Configurations in the MessageHandler objects are for example EventListSize, that is the number of internally stored events, and EventLogSize that states the size of the event log, i e the number ov events stored on disk.

The event monitor scans all sup objects and send information about alarm and events to different outunits, for example the alarm and event list. When an outunit is started, all events in the internal list is sent to the outunit, and the new events are sent when when they occur. Also a status message is sent whit all active or unacknowledged alarms to ensure that the alarm list is correctly updated.

13.6   Alarm blocking

If a part of the plant is closed, the alarms from that part can be blocked. This is done from the navigator by opening the popup menu for an hierarchy and activate 'Block events'. A priority is selected for the blocking, and alarms with this priority and lower priority will be blocked. A list of active blockings can be opened from a button in the operator window, and from here blockings also can be removed.

13.7   Suppression of alarms

A common alarm system problem is that one fault give raise to a cascade of alarms, that makes is impossible for the operator to find the root cause. This can be avoided by suppression of alarms that are not relevant in a specific situation.

Suppression is made with the function object SuppressSup. A supervision object is stated in the object, and when the input is high, the alarm from the supervision object is blocked. The input can for example be connected to the Active output of a DSup object as in the example below.

DSup	DSup0	Emergency break		SuppressSup	Plc-W-DSup2
				SuppressSup	Plc-W-DSup3
				SuppressSup	Plc-W-DSup4
				SuppressSup	Plc-W-DSup5
				SuppressSup	Plc-W-DSup6
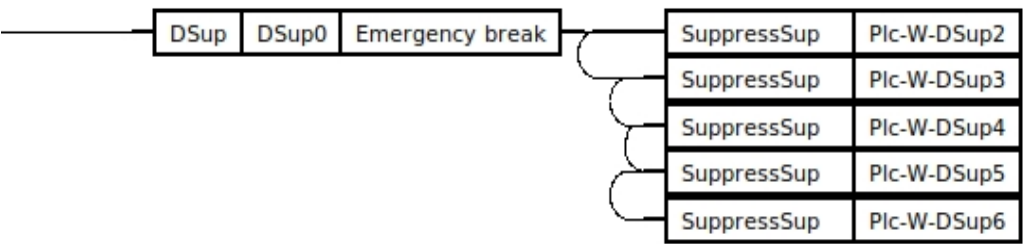
Fig Alarm suppression

13.8   Outunits

13.8.1   Alarm and event list in Xtt

The alarm and event list in the operator environment can be opened from buttons in the operator window, and also with the xtt commands

```
xtt> show alarmlist
xtt> show eventlist
```

In the EventSelectList attribute of the OpPlace object you state hierarchies from where alarm and events should be displayed.

Alarm and event list in an XttMultiView

There is a function that creates copies, so called satellites, of the alarm and event lists. These can be usee to display the alarm or event list in a XttMultiView. In the XttMultiView configuration, Action.Type is set to AlarmList or EventList.

AlarmView

An alarm list satellite can also be combined with an AlarmView, that will categorize the alarms and display them under different maps in the list. The maps are configured by creating AlarmCategory objects under the AlarmView object, one for each map. In the AlarmCategory object the hierarchies, priorities or event types for the alarms in the map. A map containing active or unacknowledged alarms are marked with a color that shows the priority of the alarms in the map. Also the number of unacknowledged alarms under the map is displayed.

An AlarmView object has to be stated in the AlarmView array of the OpPlace object.

An alarm list with an AlarmView can be opened with the command

```
xtt> show alarmlist satellite /alarmview=
```
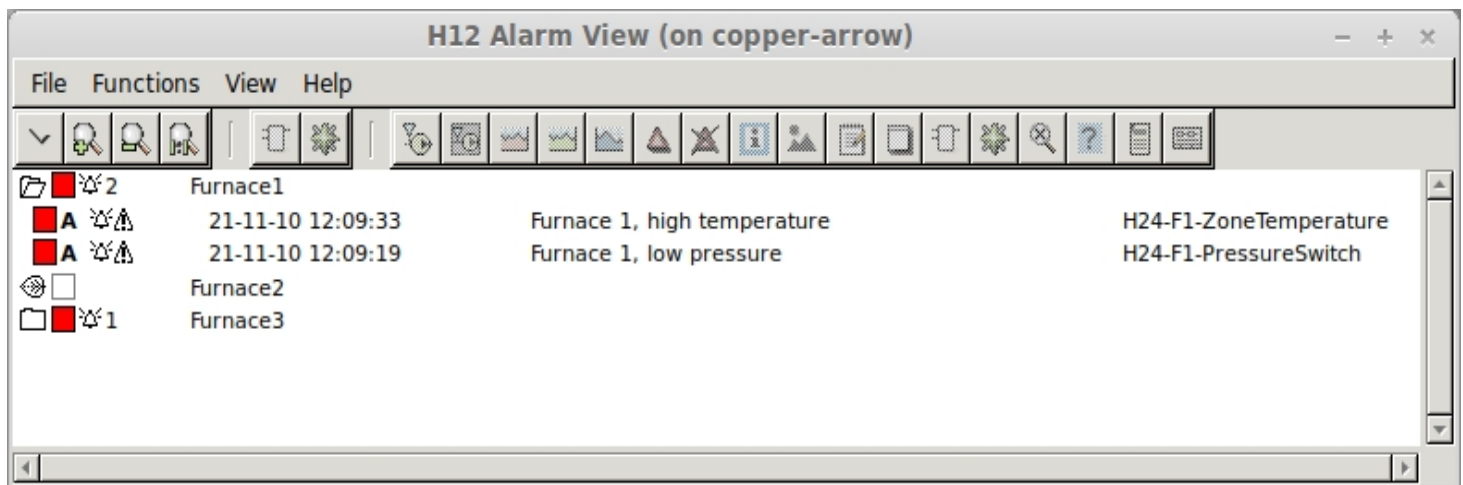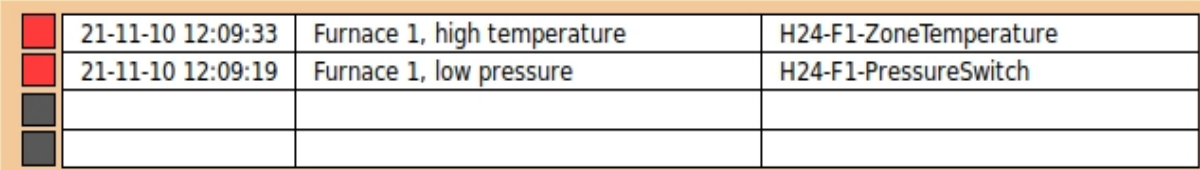

Fig Alarm list with an AlarmView

It can also be displayed in an XttMultiView by stating the AlarmView object in Action[].Object[0] in the XttMultView object.

AlarmTable

The AlarmTable object makes it possible to display the latest alarms in a process graph.

The AlarmTable object i placed under the OpPlace object and the alarm list for the OpPlace is copied to arrays in the AlarmTable object. There are arrays for EventText, EventName, Time, EventPrio etc, that can be displayed in a table in the process graph. With the Member attribute a selection of alarm from specific hierarchies can be made, but the selection is limited by the EventSelectList of the OpPlace object.

	21-11-10 12:09:33	Furnace 1, high temperature	H24-F1-ZoneTemperature
	21-11-10 12:09:19	Furnace 1, low pressure	H24-F1-PressureSwitch

Fig An AlarmTable object displayed in a process graph.

13.8.2   Alarm and event list in the web interface

In the menu of the web operator window there are buttons to open the alarm and event lists. The hierarchy selection of alarms is stated in EventSelectList in the WebSocketServer object.

13.8.3   Event log

In the event log all events are stored in a database. This is configured by setting the size of the event log in EventLogSize in the MessageHandler object. From the operator window, a dialog for viewing and searching for events, can be opened.

The event log is handled by the server process rt_elog and is placed in $pwrp_db/rt_eventlog.db.

13.8.4   History storage of events

Events can also be stored in the history database. The configuration is made with a SevHistEvent object where you state which event to store and for how long.

Store event is viewed by the Alarm and Event Analyser.
Read more in Guide to storage enviroment

14   Communication

14.1   Internal communication

The internal communication in ProviewR sends information about volumes, objects, alarms, events, history data etc. between processes and nodes. There are three different protocols for alarm handling, net handling and history data, the all are based on Qcom.

Qcom

Qcom is a message bus that sends queued messages between processes. Communicating nodes all have to share the QCom bus, and a node can only attach one bus. The bus is configured in the BusConfig object by stating the bus identity, a value between 1 and 999.

You also have to configure which nodes are to communicate with each other. With the default configuration, all nodes in the same project will communicate, and additional nodes in other project is configured with FriendNodeConfig objects. This can for example be process stations that are mounted by operator stations.

If you have a project with several nodes, and don't want them all to communicate with each other, you can set QComAutoConnectDisable in the BusConfig object, and specify the communcation links with FriendNodeConfig objects.

All Qcom messages are sent with acknowlegement. If the acknowlege doesn't arrive, the message is resent with double timeout. After a number of resends without reply the link is regarded to be down. Dependent on the type of net and transmission speed, the timeouts might have to be adjusted. This can be done with the ResendTime attributes in the NodeConfig and FriendNodeConfig objects.

See the document QCOM for more information about Qcom.

Nethandler

The nethandler sends information about volumes and objects between nodes, for example the volumes in a node, the parent, children or siblings of an object, or the content of an object or attribute. Also subscriptions are set up via the nethandler, that is data cyclically sent from one node to another, usually from process stations to operator stations. The processes for the net handling are rt_neth, rt_neth_acp and rt_tmon.

Eventhandler

The event handler scans all the supervision object in a node and sends alarms and

events to outunits, for example to the alarm and event lists in the operator enviroment.
They send acknowlegdements back to the event handler. Exactly which alarms and event that
is sent to an outunit is configured in the select list. The select list for the operator
environment is found in the the OpPlace object. Only events and alarms below the stated
hierachies are sent. The process of the event handling, rt_emon, is configured with a
MessageHandler object.

Historical data storage

The historical data storage means that data is sent from process stations and stored in
a database in a storage server. The server process rt_sevhistmon in the process station
scans all attributes marked for storage, fetches the corresponding values and sends them
to the sev_server process that stores them into the database. When curves of historical data
are to be displayed, a request is sent from the operator environment and a suitable
selection of points are fetched from the database and sent to the operator environment.
See chapter Data Storage

Web and app communication

The web interface and Andriod application fetches information from the ProviewR realtime
database through the server processes rt_webmon, rt_webmonmh and rt_webmonelog. This
communication is configured by the WebHandler object.

Status server

The Runtime monitor and Supervision center fetches information from the Status server.
This communication is based on http and soap, and is configured with the StatusServerConfig
object.

14.2   Remote

The Remote concept in ProviewR is a way to standardize the methods of communication with other
systems. It describes a number of transport programs and ProviewR objects used to implement
a variety of different communication protocols and to handle incoming and outgoing messages.
Remote is designed to use different transport protocols such as TCP/IP or BEA Message Queue,
and different hardware media such as ethernet or serial lines.
The main purpose of Remote is to provide the programmer with an interface to communication.

14.2.1   Introduction

There are some different classes and objects that are used to handle the communication.

RemoteConfig

Required to have any remote communication at all. Without this no remotehandler is started.
Place one object in the node-hierarchy.

RemNode

Defines a link of some type to a remote node over a specific protocol.
Several different protocols are supported and there is one specific class for each protocol.
The supported protocols are:

TCP/IP
UDP/IP
RabbitMQ
MQTT
MQ (BEA Message Queue)
ALCM (an old Digital protocol, supported for historical reasons)
Serial
Modbus/RTU Serial
3964R (serial protocol from Siemens)
Webspere MQ

Configuration of each protocol is described further down.
Place RemNode-objects below the RemoteConfig-object.

RemTrans

Generic class that defines a specific message to or from a specific remote node on a
specific protocol. Should be placed below the RemNode-object. The size of a message to be sent
is specified in the RemTrans-object. The data to be sent however resides in a buffer that is
configured as a child to the remtrans-object. When a message is to be sent, data of the length
specified in the remtrans-object is fetched from the buffer. Sometimes however a header of some
length is added to the message.

Buffer

Defining the send- and/or receive data area for each message. Exists in different sizes.
Should be placed below a RemTrans-object. The buffer must be of at least the size of the
message that will be received or sent. If size is not enough message will be cut in the end.

RemTransSend

Function object used in a plc-program for sending messages.

RemTransRcv

Function object used in a plc-program for receiving messages.

Logging of transactions

Remote transactions can be logged to text file. The extent of the logging is configured
in the LoggLevel attribute of the RemTrans object. The log file is configured with a
LoggConfig object.

14.2.2   Protocols

Which protocol to use is defined by the type of Remnode-object you configure. For each configured
Remnode-object a transport job is started. That is, a program to handle the specific protocol is
started as a process. This process will handle all RemTrans-objects that are configured as children
to the Remnode-object.

14.2.2.1   UDP

UDP uses socket communication without connection (datagram), compared to TCP wich is a connected protocol.

In the RemnodeUDP-object you specify the name and ip-address of the node to communicate with as well as the port-numbers for both ends. The local port-number needs to be unique on your node not to conflict with other communications.

As default, all messages are sent with a special header that is not included in the user data buffer. This header is added at the beginning of the message. The purpose for the header is to give information about the message that is sent. This helps identifying what type of message that is received and to which buffer the data will be unpacked. The header looks like:

```
char        RemId1;  /* STX (Hex 02) */
char        RemId2;  /* ETB (Hex 0F) in data message without acknowledge
                        ENQ (Hex 05) in data message with acknowledge
                        ACK (Hex 06) in acknowledge message */
short int   Length;  /* Number of bytes in message including this header */
short int   MessId1; /* Message identity part 1 */
short int   MessId2; /* Message identity part 2 */
```

All of the integers in the header will be sent as big endian values, which means the most significant byte first in the datagram. The user data is the responsibility of the user to switch, if he wants integers to be sent with big endian. Intel (x86), VAX and Alpha all use little endian! To send messages without headers, the attribute DisableHeader should be set to TRUE. When communicating between two ProviewR-systems the header should be kept on. MessId1 and MessId2 are fetched from attributes RemTrans.Address[0] and RemTrans.Address[Through the header it is also possible to request an acknowledge of a sent message. If there is no acknowledge the message will be resent with a cyclicity specified by the RetransmitTime-attribute in the remtrans-object.

Since UDP/IP is a connectionless protocol there is a possibility to watch the connection using keepalive-messages. This is set through the attribute UseKeepalive.

Sending messages

The transport will send a message to the remote port, consisting of header + data.
MessId in the header is taken from RemTrans.Address[0,1], byte-switched to send as big endian.
If MaxBuffers in remtrans-object > 0, the message is sent with type "want acknowledge" and is stored in the retransmit queue for the remnode. When a corresponding acknowledge message is received, the message is deleted from the retransmit queue. This is done automatically by the transport process.

Receiving messages

When we receive a buffer, we first check the header to see that this is a correct RemTrans message.
Then we search for RemTrans.Address[0,1] that matches the byte-switched MessId. If the data object for this message is big enough to contain the message, the message will be stored, and the DataValid flag will be set.
If the Remnode is marked to be used without header (DisableHeader-attribute set) then a RemTrans marked as a receiving remtrans with a enough large buffer will be searched.

14.2.2.2 TCP

RemnodeTCP is configured much in the same way as RemnodeUDP. The big (only) difference is that TCP is a connected protocol which acts in client/server fashion. Thus you have to either connect to a remote socket (act like a client) or await a connection (act like a server). When acting like a server only one client will be accepted. The ConnectionMode-attribute in the remnode-object defines if you are a client or a server. Setting it to zero (default) means client and setting it to one means server.

14.2.2.3 RabbitMQ

RemnodeRabbitMQ configures a transport on the open source broker RabbitMQ.

Exchange, SendQueue and ReceiveQueue are configured in the remnode object. If the queues doesn't exist they will be created on the server. If Exchange is left empty, the default exchange is used.

Username and password are specified to gain access to the RabbitMQ broker. The user has to be define on the broker and given appropriate permissions. This can be done with rabbitmqctl on the server node, eg

```
> rabbitmqctl add_user myuser mypasswd
> rabbitmqctl set_permissions -p / myuser .* .* .*
```

In the RemTrans object, Address[0] and Address[1] is used to address a sent message to the corresponding RemTrans in the target node. Address[3] states the message delivery mode where 2 is persistent and other values not persistent. Address[2] is an option bitmask where bit 1 is KeepAll and bit 2 MsgOrder.

KeepAll means that messages are requeued if the remtrans is occupied. By default they are discarded. If there are several remtrans objects on the same queue, the order for requeued messages might be changed unless the MsgOrder bit is set. MsgOrder will set the prefetch count to 1.

If a message should be recovered after a server failure, Durable should be set in the RemoteRabbitMQ object, and delivery mode in RemTrans.Address[3] should be set to 2 (persistent).

If messages should be able to survive network failures, a suitable method is to send messages to a local broker and receive messages from a broker on the sending node. In this case you need separate RemnodeRabbitMQ objects for sending and receiving, and you also need to start the broker in both nodes.

14.2.2.4   MQTT

RemnodeMQTT configures messages with MQTT.
Topics for subscribing and publishing are configured in the remnode object.

Username and password are specified to gain access to the MQTT server. The user has to be define on the server and given appropriate permissions. For Mosquitto, a password file can be created with the mosquitto_passwd utility, and the file should be copied to the /etc/mosquitto directory.

In the RemTrans object, Address[0] and Address[1] is used to address a sent message to the corresponding RemTrans in the target node.

The specification of topics differs if the remote header is disabled or not.

If the header is present

 - Sending: publishing is made with the topic in PubishTopic in the RemnodeMQTT.
   Address[0] and Address[1] in the RemTrans object is used to match RemTrans objects.

 - Receiving: Subscriptions are med with the topic in SubscribeTopic in the RemnodeMQTT object.
   The message is directed to the RemTrans with matching Address[0] and Address[1].

If header is disabled

 - Sending: publishing is made with the topic in RemTrans.TransName.

 - Receiving: A generic topic is set in SubscribeTopic in the RemnodeMQTT object, eg 'lab57/rcv/#'.
   A more narrow topic is set in RemTrans.TransName, eg 'lab57/rcv/msg1'.

14.2.2.5   MQ

RemnodeMQ is a transport for sending messages on BEA Message Queue (BMQ). It requires that you to have BEA Message Queue installed on your node. This message queue is good for the safe delivery of messages to a remote node even if it is not up at the moment you send your message. Vice versa,

messages will safely be delivered to you.

This documentation expects you to have basic knowledge on BEA Message Queue. In basic the communication runs on a specific bus. Each node has group-number and can only communicate to other groups on the same bus. On each group several queues can be configured.

To be able to start this transport of course the Message Queue software needs to be running. You also need some environment variables to be set. These are:

```
DMQ_BUS_ID
DMQ_GROUP_ID
DMQ_GROUPNAME
```

In the RemnodeMQ-object you configure on which BMQ-queue to receive messages in the attribute MyQueue. You also configure the remote nodes group-number and queue to send to in the attributes TargetGroup and TargetQueue.

Sending messages

Similarly to UDP and TCP-transports RemTrans. Address[0,1] are used to identify the message. Address[0] represents the message-class and Address[1] represents the message-type (according to the BMQ-nomenclature). Address[2,3] are used to define what type of delivery mode (Address[2]) that should be used and what action should be taken when a message cannot be delivered (Address[3]).

Possible delivery modes are:

```
PDEL_MODE_WF_SAF  25
PDEL_MODE_WF_DQF  26
PDEL_MODE_WF_NET  27
PDEL_MODE_WF_RCM  28
PDEL_MODE_WF_MEM  29
PDEL_MODE_AK_SAF  30
PDEL_MODE_AK_DQF  31
PDEL_MODE_AK_NET  32
PDEL_MODE_AK_RCM  33
PDEL_MODE_AK_MEM  34
PDEL_MODE_NN_SAF  35
PDEL_MODE_NN_DQF  36
PDEL_MODE_NN_NET  37
PDEL_MODE_NN_RCM  38
PDEL_MODE_NN_MEM  39
PDEL_MODE_WF_DEQ  40
PDEL_MODE_AK_DEQ  41
PDEL_MODE_WF_CONF 42
PDEL_MODE_AK_CONF 43
PDEL_MODE_WF_ACK  44
PDEL_MODE_AK_ACK  45
```

and possible actions are:

PDEL_UMA_RTS 1
PDEL_UMA_DLJ 2
PDEL_UMA_DLQ 3
PDEL_UMA_SAF 4
PDEL_UMA_DISC 5
PDEL_UMA_DISCL 6

Not all combinations are possible (see BEA Message Queue documentation for more information).

Recommended combinations are,

for safe delivery of message
Address[2] = 26
Address[3] = 4

and for discarding the message if it cannot be delivered
Address[2] = 39
Address[3] = 5

If Address[2,3] are both set to zero, a default setting will be used. The default is to discard the message if it cannot be delivered.

Receiving messages

Address[0,1] are used to identify the message. Address[0] represents the message-class and Address[1] represents the message-type (according to the BMQ-nomenclature).

14.2.2.6   Serial

RemoteSerial is an attempt to generalize the use of a simple serial line communication protocols.
Its useful when we have a one-way sending of messages from some equipment to the control system.
You can specify up to eight termination characters in the attribute TermChar[0-7].
These termination characters are used to detect the end of a received message (if a character matches any of the termination characters).
Specify also the settings for the serial link, that is - DevName , Speed , Parity (0 = none, 1 = odd, 2 = even), StopBits and DataBits. These could be for example /dev/ttyS0, 9600, 0, 1, 8.

14.2.2.7   3964-R

3964R is a simple serial line communication protocol that is developed by Siemens.

You specify the settings for the serial link as with RemnodeSerial, except that there are no stp bits to specify. You must also specify the character timeout (the maximum time between received characters) in the attribute CharTimeout. The AckTimeout-attribute specifies the time to wait for an answer.

Messages will be sent straight on without any header. ACK, NAK, DLE and BCC is handled according to the 3964R protocol.

Receiving messages

There can be only one RemTrans object for receive-messages because of the lack of header in this protocol. Every received message will be put in the first found RemTrans-object below the RemNode-object. If the data object for this message is big enough to contain the message, the message will be stored, and the DataValid flag will be set.

Sending messages

The transport will send a 3964R message to the serial without adding any header.
If we don't have contact with the other node the message will be buffered
if there are still free buffers for this message.

14.2.2.8   Modbus Serial

The format of MODBUS that is implemented is RTU. For identification of messages we use the fields known as slave address and function code in the MODBUS header. The RemTrans.Address[0] and [1] define these fields in the ProviewR environment. Se MODBUS specifications documents for more information.

Modbus Serial is not yet implemented as an I/O-system in ProviewR. You have to configure the messages yourself by using RemnodeModbus and specifying RemTrans-object for the various operations you want to perform. Modbus works in a request/reply manner, so for each operation you want to perform you specify one RemTrans-object for sending and one for receiving. Except for the Modbus-header of the message and the checksum handling you have to specify the content of the message to send in the send-buffer. In the same way you must decode the content of a received message yourself.

Modbus TCP is implemented as an I/O-system in ProviewR. See more information about this in the document "Guide to I/O-systems". With Modbus TCP you don't have to care about the encoding of the messages.

Receiving messages

When we receive a buffer, we search for RemTrans.Address[0] and [1] that matches the fields slave address and function code in the message header.
If the data object for this message is big enough to contain the message, the message will be stored, and the DataValid flag will be set.

Sending messages

Messages will be sent using the contents of RemTrans.Address[0] and [1] as the fields slave address and function code in the message header.

14.2.2.9 Websphear MQ

RemnodeWMQ is a transport for sending messages on Websphear Message Queue (WMQ). It requires that you to have Websphear MQ installed on your node.

RemnodeWMQ configures communication through a message queue using Websphere MQ as a client. All regarding Server, channel, queue manager and queues to connect to is configured in the RemnodeWMQ-object.

Configurations for the different messages is configured in the RemtTrans-object that define each in- and outgoing message. For each RemTrans-message the following can be configured:

TransName	a string that defines the MsgId (message identity of the message).
Address[0]	defines whether the message should be sent as a persistent message or not. 1 means that the message will be sent as a persistent message. 0 not.

14.2.3   An example

To show how to work with the classes that are briefly described above we will start with a little example. The classes are described in more detail below.

In our example we have one ProviewR-system communicating with an other node via UDP/IP. We will send a few messages in both directions. My node is named 'dumle' and the remote node is named 'asterix'.

The messages we will send are:

d_a_RequestData	4 Byte
d_a_Report	20 Byte

and the messages we will receive are:

a_d_Data	365 Byte (as an answer to the d_a_RequestData-message)
a_d_Error	10 Byte

The configuration in the node-hierarchy looks like this.

The RemoteConfig-object has to be there. I have added a RemnodeUDP object below this and configured address and nodename as well as the port-numbers to communicate on.
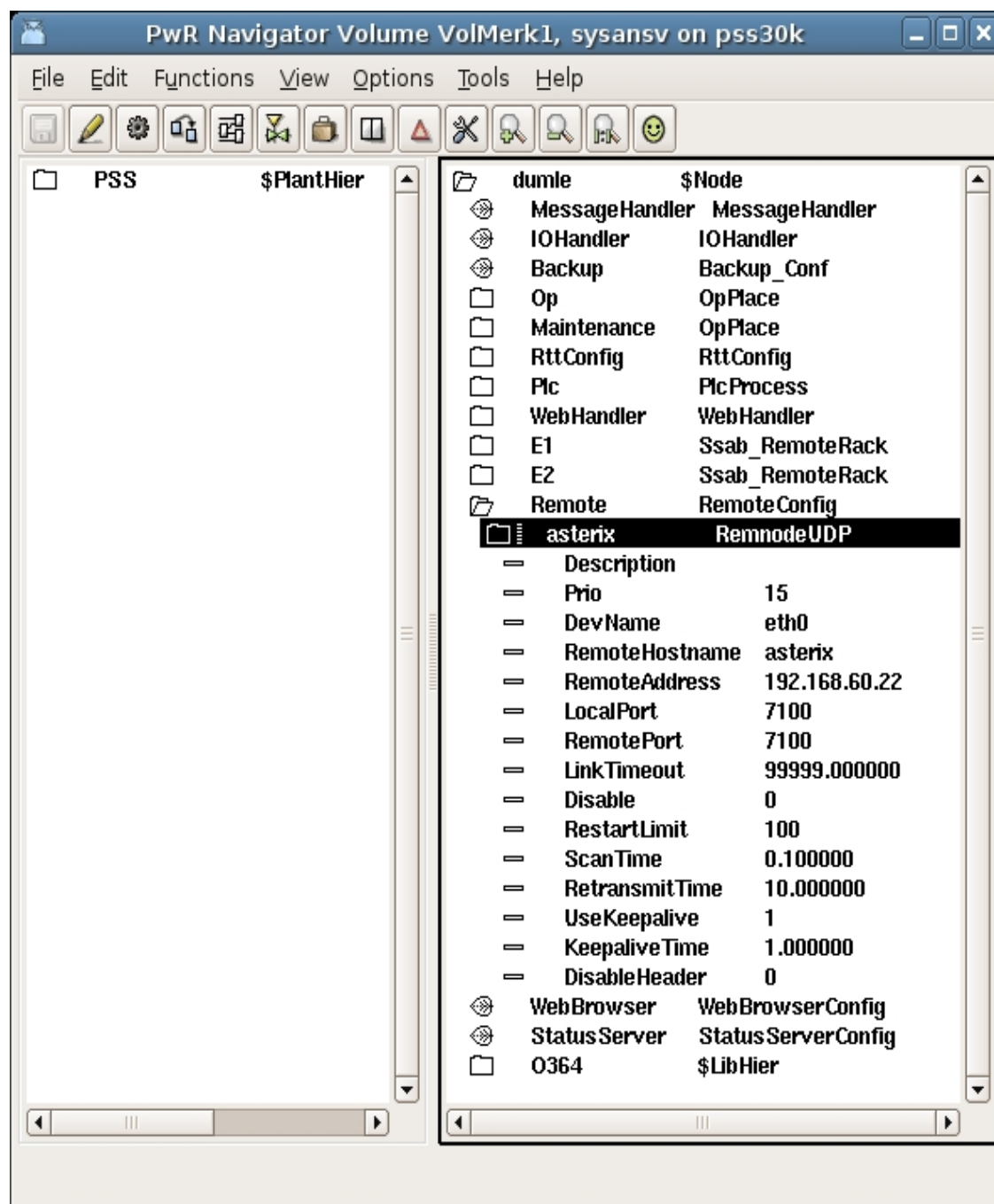
Below the RemnodeUDP-object I have added four RemTrans-objects, one for each message. In the remtrans-objects I have configured the direction (send or receive)and numbered the addresses so I can distinguish between the messages and set the sizes on the sending messages.
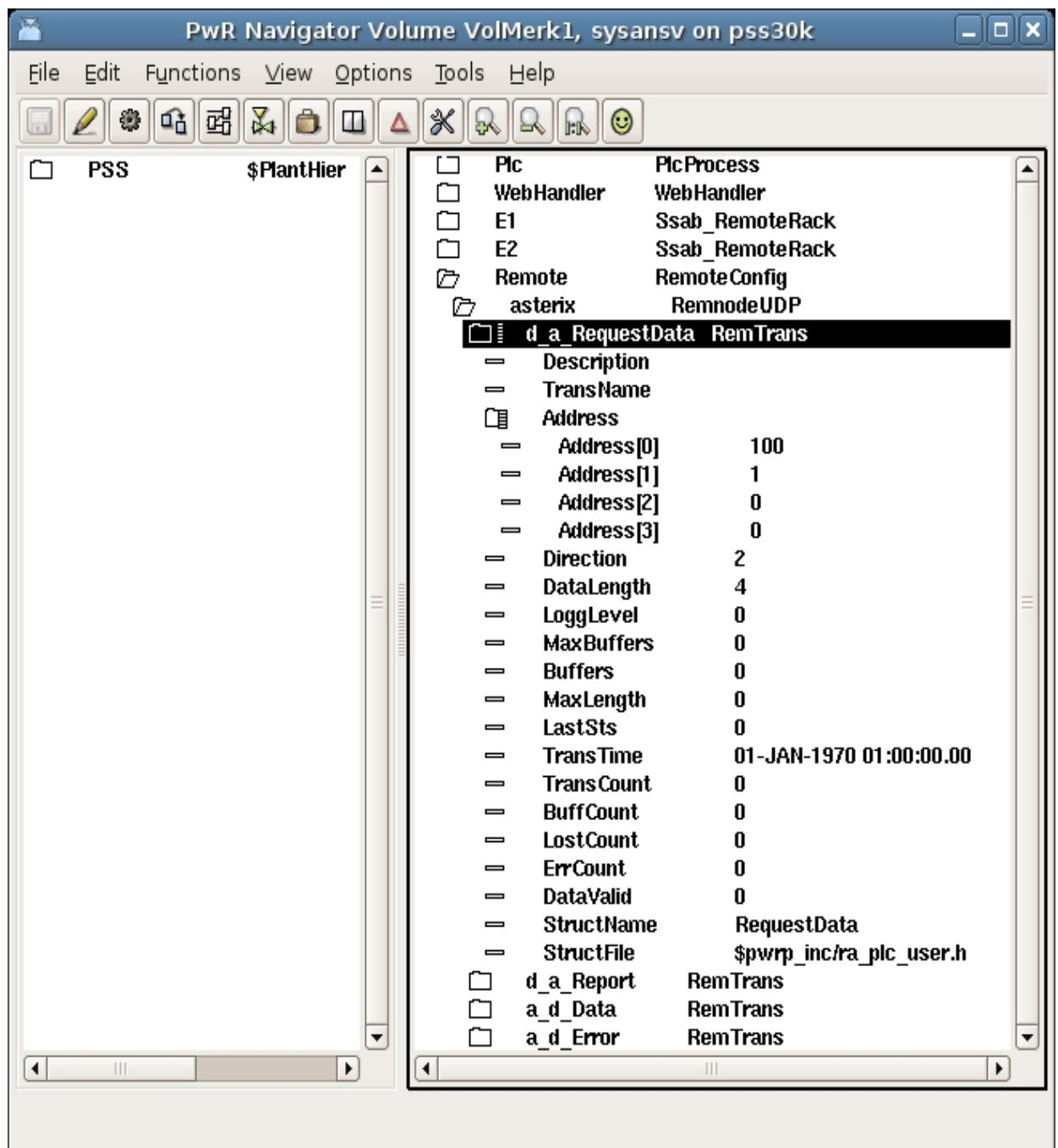
Below the remtrans-objects I have put the data-buffers. For the smaller messages I have chosen the small Buff256-buffer. The Data-message is larger and I have therefore chosen the Buff1440-buffer for this message.
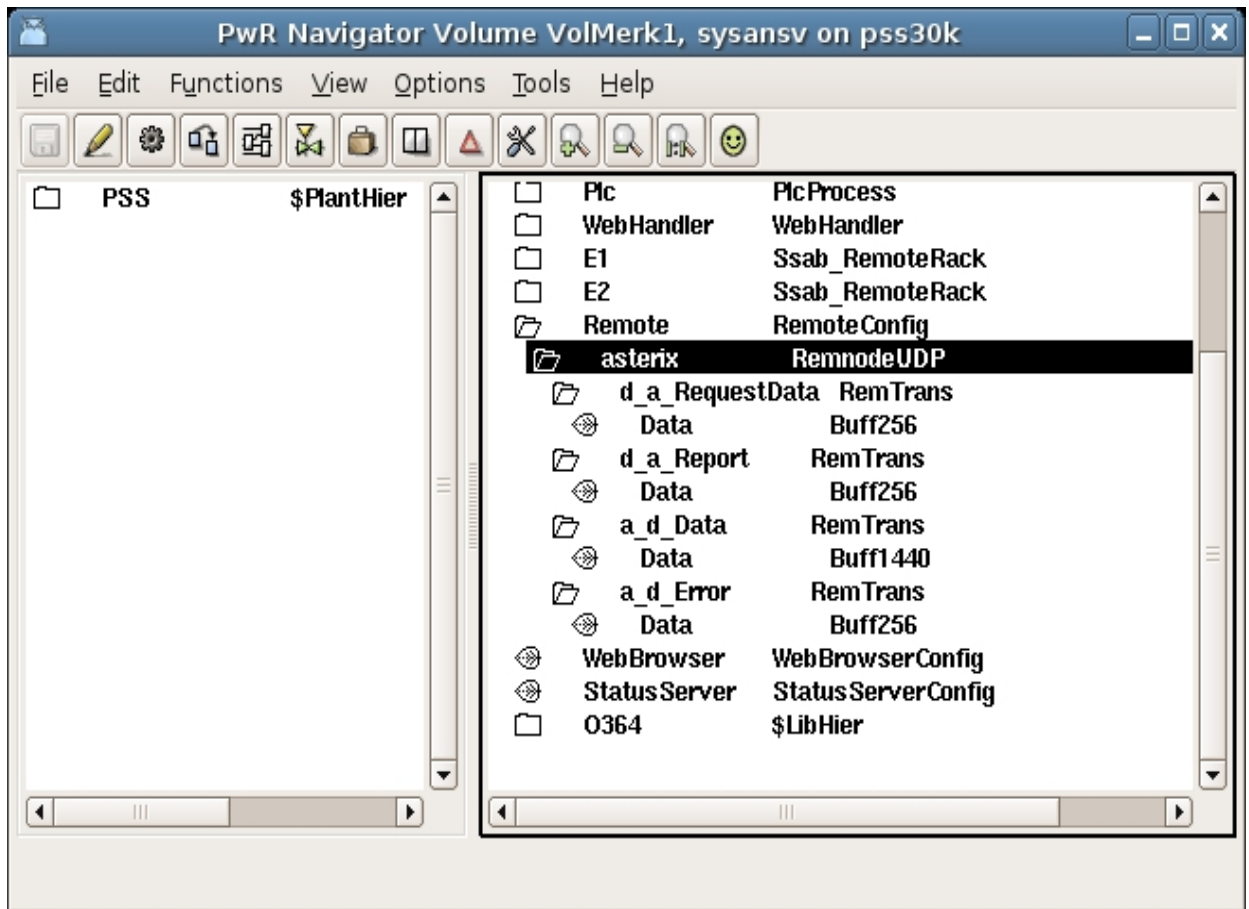
The data structures

The data structures for the messages are defined in the file ra_plc_user.h in $pwrp_inc-directory. This file is automatically included when you compile the plc-code. The structures look like:

```
typedef struct {
  pwr_tUInt32  Id;
} d_a_RequestData;

typedef struct {
  pwr_tUInt32   Id;
  pwr_tFloat32  data_1;
  pwr_tInt32    data_2;
  pwr_tInt32    data_3;
  pwr_tInt32    data_4;
} d_a_Report;

typedef struct {
  pwr_tUInt32   Id;
  pwr_tFloat32  data_1;
  ...
  ...
  pwr_tInt32    data_xx;
} a_d_Data;

typedef struct {
```

```
    pwr_tUInt32    Id;
    pwr_tInt32     func_no;
    pwr_tInt16     err_code;
} a_d_Data;
```

By default the gcc compiler will align elements in data stuctures on 32 or 64-bit boundaries. When creating data structure for communications, this will cause confusion as different compilers and platforms have different alignment rules. To avoid this use attribute packed when declaring the stucture.

```
typedef struct {
    pwr_tUInt32    Id;
    pwr_tInt32     func_no;
    pwr_tInt16     err_code;
} __attribute__((__packed__)) a_d_Data;
```

The plc code

I have a plc program named Comm. In this program I have placed one RemTransSend-object and one RemTransRcv-object. These objects are found below the "Other"-hierarchy in the plc-editor palette.
To the RemTransSend-object I have connected the RemTrans that I want to send; in this case the d_a_RequestData-message. The message will be sent when the Dv-signal "RequestData" is set. Similarly I have connected the a_d_Data-message to the RemTransRcv-object, which will be the answer to my request.
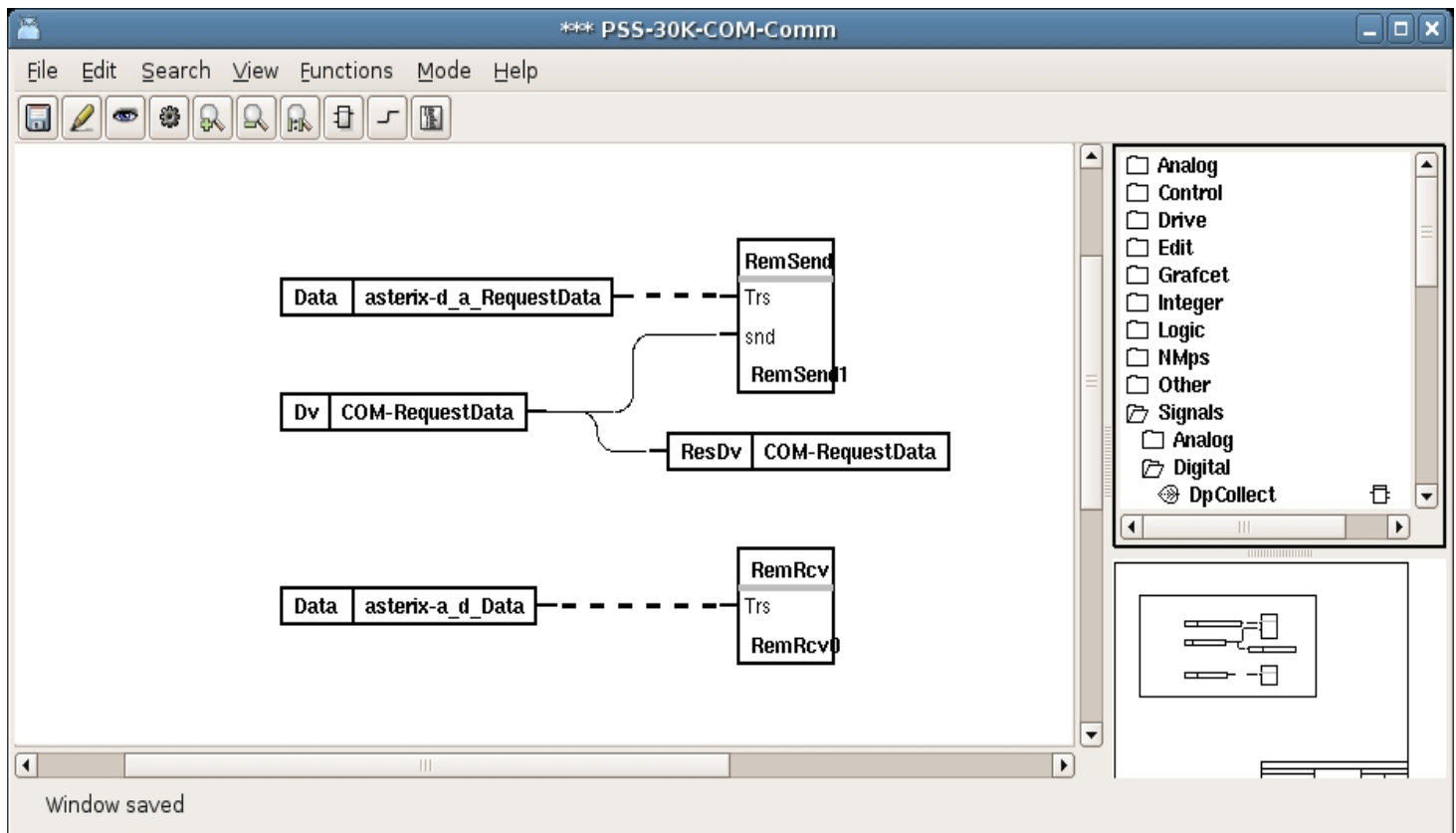
Both the RemTransSend and the RemTransRcv-objects have a subwindow. For the RemTransSend this subwindow will be executed when there is a flank on the snd-pin. When the subwindow has been executed, the DataValid-attribute of the connected RemTrans-object is set. The transport job for this Remnode sends the message and sets the DataValid-flag to zero.

For the RemTransRcv the subwindow will be executed when the DataValid-attribute in the connected RemTrans-object is set. When the transport job for this Remnode receives a message, it fills the data buffer with the received data and then sets the DataValid-flag.
After execution the flag will be reset.

Send subwindow

In the send subwindow we fill in the data in the send-buffer. The send-buffer for the message to send is connected to a DataArithm. The special 'structdef'-syntax casts Da1 to be a pointer to a d_a_RequestData-struct.
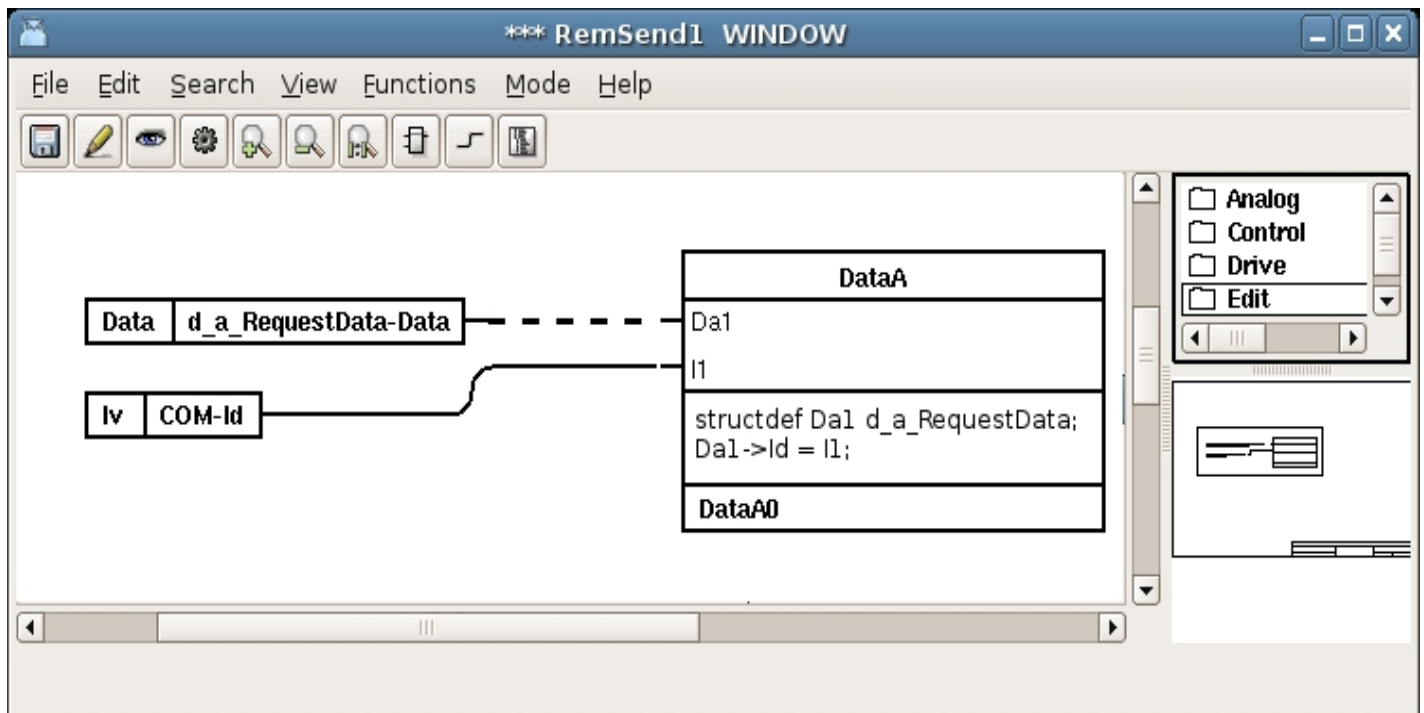
Receive subwindow

In the receive subwindow we unpack the data received in the receive-buffer. The receive buffer is connected to a DataArithm. Again we use the classdef-statement to cast the Da1-pointer.
We unpack the data to the output pins of the DataArithm. If one DataArithm is not enough to unpack the parameters we just add more DataArithm's and continue in the same way.

File   Edit   Search   View   Functions   Mode   Help

Analog
Control
Drive
Edit
Grafcet
Integer
Logic
NMps
Other
Signals
TLog
Components

DataA

| Data | a_d_Data-Data |

Da1

OA1	StoAv	Rcv-data_1
OI1	StoIv	Rcv-Id
OI2	StoIv	Rcv-data_2
OI3	StoIv	Rcv-data_3
OI4	StoIv	Rcv-data_4
OI5	StoIv	Rcv-data_5
OI6	StoIv	Rcv-data_6

structdef Da1 a_d_Data;
OI1 = Da1->Id;
OA1 = Da1->data_1;
OI1 = Da1->data_2;
/* ... */
OI6 = Da1->data_xx;

DataA1

15   Data Storage

There are three different types of data storage in ProviewR, trends, fast curves and historical data storage. Trends are storing data cyclic during a shorter period of time in the realtime database. Fast curves are triggered by some event and stores data for a period of time after the trigg event, and in some cases also before. The historical storage is made on disk in a relation database and is able to cyclically store data in several years.

15.1   Trends

There are two types of trends
- DsTrend with an internal data buffer that can store about 500 samples with a minimum scan time of 1 s.
- DsTrendCurve, with configurable buffer size and with a minimum scan time of 20 ms.

15.1.1   DsTrend

The DsTrend object has an internal data buffer of 1912 byte that can for example store 478 samples of type Float32. Also other data types are possible. The configuration is a bit odd because the buffer is divided into two parts, but normally you just have to state the attribute that is to be stored in DataName, and possibly insert a value into Multiple to state the scan time.

See also DsTrend in Object Reference Manual.

15.1.2   DsTrendCurve
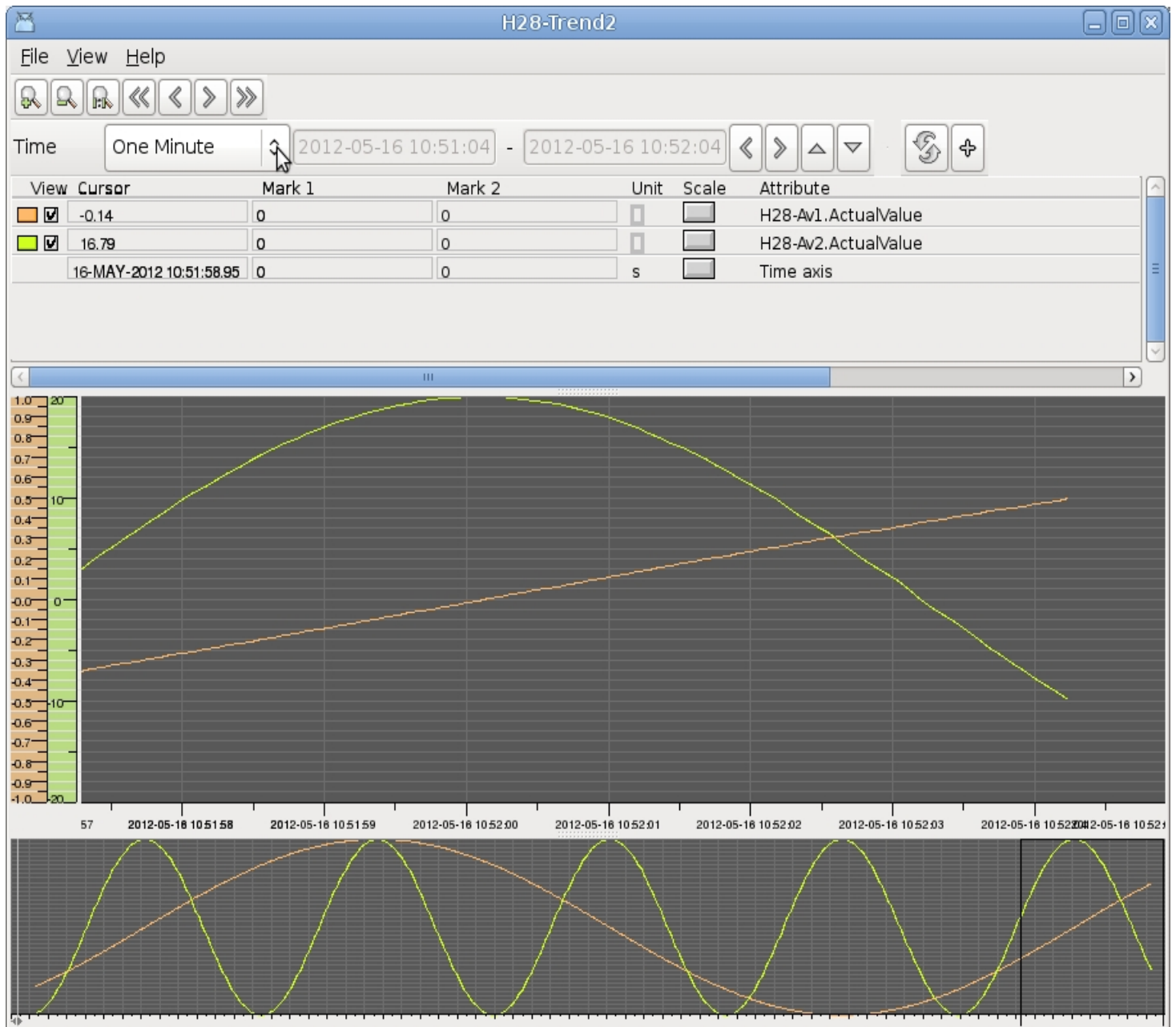
Fig Trend configured with DsTrendCurve

The DsTrendCurve object store trends with external buffer objects, where the size of the
Buffer objects limits the number of samples that can be stored. A CricBuff100k object, as
in the example below, can contain 25000 samples. The storing can be done with scantimes
down to 20 ms. There are also a snapshot function where the current trend is frozen and
displayed in a curve window where it can be analyzed in detail. The snapshot curve can be
stored to files and opened at a later occasion.

The trend is configured with a DsTrendCure object. Up to 10 different attributes that is
to be stored, can specified in the Attribute array. For each attribute, a buffer object of
type CircBuffer is created and stated in the Buffer attribute. The size of the buffer object
should be adapted to the number of samples in the trend. Also a time buffer can be stated,
but this is only necessary if the snapshot function is to be used. The time resolution
determines the size fo the time buffer. For a resolution of 1 s, 4 bytes per sample is
used, and for a resolution of 1 ns 8 bytes is needed.
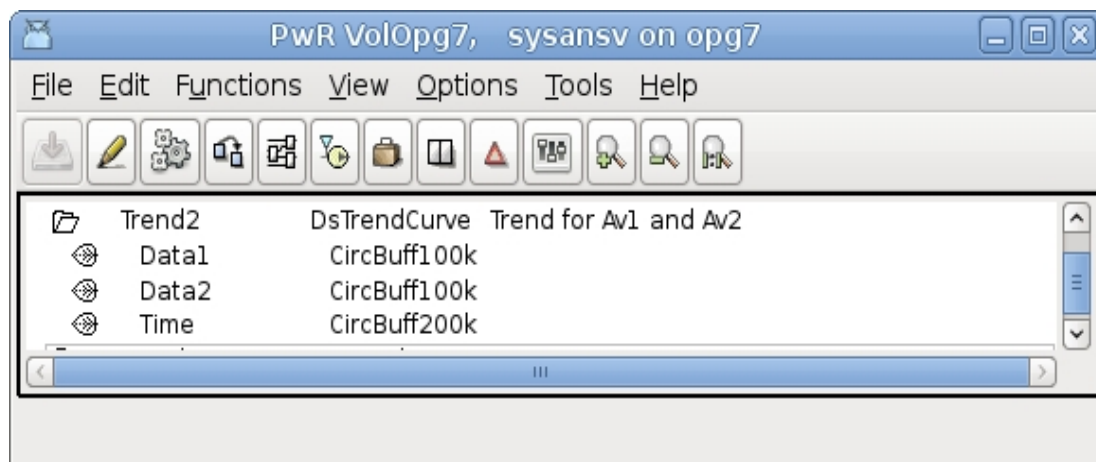
Fig Configuration of DsTrendCurve with buffer objects
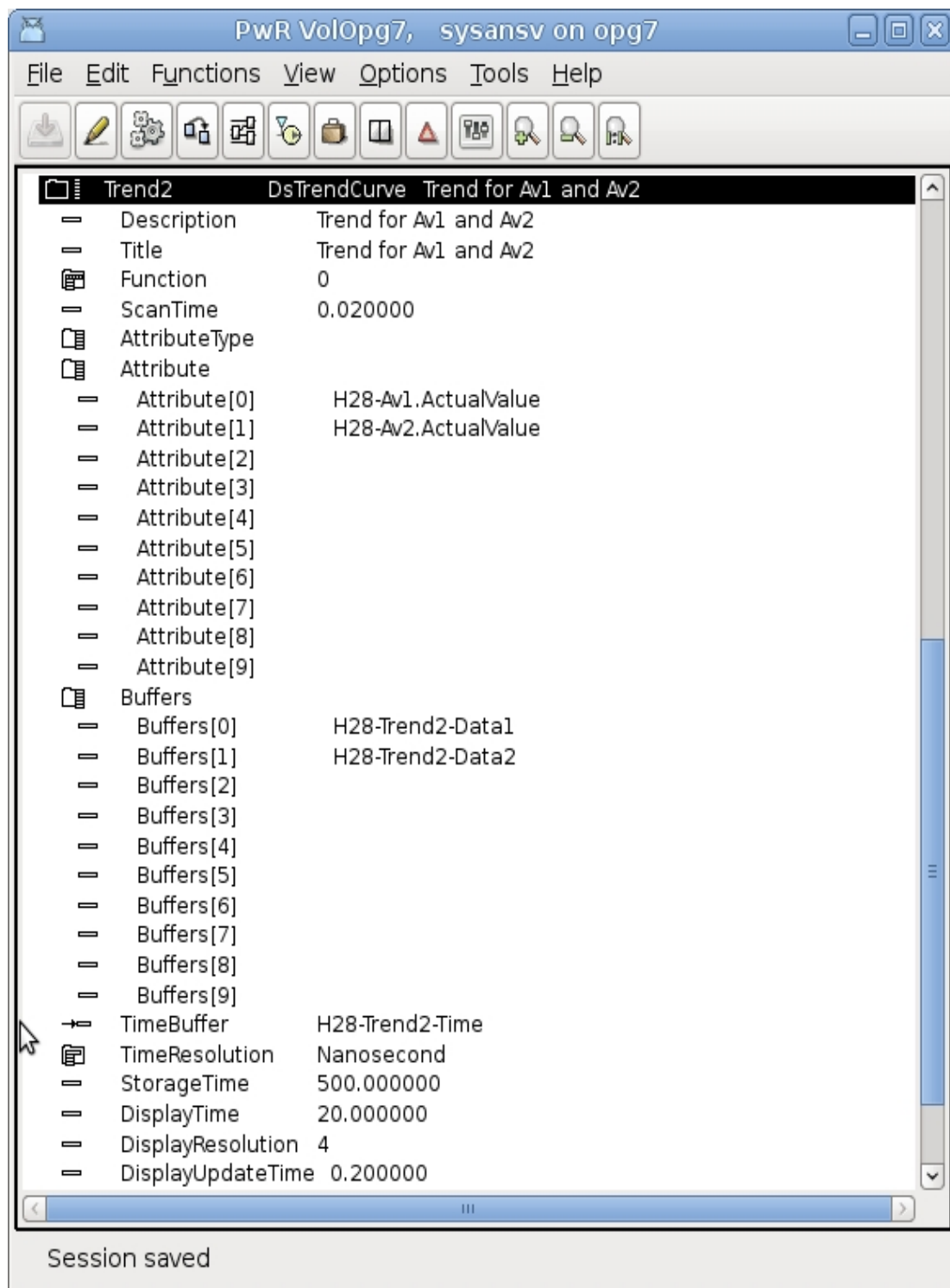
Fig Configuration of DsTrendCurve

Snapshot means that you take a copy of the data, and display it in a separate window. You can configure the trend to show only a limited part of the stored data and that data is stored for a longer period of time, and also with higher resolution. In the snapshot window it is then possible to go further back in time or to zoom in and increase the resolution of the curve.

The attributes DisplayTime and DisplayResolution in the DsTrendCurve object is used to specify the part of the data that is displayed in the trend window. In the example above the total storage time is 500 s, while only the last 20 s (DisplayTime) is shown in the

trend window. DisplayResolution is set to 4, which states that every forth sample is displayed. As the scan time is 20 ms a value i shown every 80 ms, and to totally 20 s/0.08 s = 250 samples is viewed. The total data contains 500 s/0.02 s = 25000 samples that thus are available in the snapshot window.

See also DsTrendCurve in Object Reference Manual.

15.2   Fast curves

A Fast curve is triggered by a certain event and then stores data for a certain period of time than is then displayed in a curve window. The trigger event can be a digital signal that goes high, an analog value reaching a limit value, or a manual triggering. The fast curve can be configured to continuously store data so that also data before the trigger point can be viewed.

A fast curve is configured with a DsFastCurve object. Up to 10 attributes can be handled by one DsFastCurve object and the attributes are stated in the Attribute array. For each attribute a buffer object should be created and stated in the Buffers array. Also a buffer object for the time should be created and stated in the TimeBuffer attribute. The size of the buffer objects should be adapted to the data quantity that is to be stored.

A fast curve can be viewed in the ordinary curve window, that is opened for example from the Fast item in the popup menu. It can also be viewed in a Ge graph with a FastCurve component.

See also DsFastCurve in Object Reference Manual.

15.3   Historical data storage

Sev is the storage environment, where historical data is stored in a database. It is a complement to the other environments in ProviewR, the development, runtime and operator environment. Sev contains server processes that handles fetching and storage of historical data. Sev can be installed as a separate unit on a storage station, but it is also included in the runtime package and can be started in the runtime environment.

Read more in Guide to Storage Environment.

16 Application programming

This chapter is about how to write application programs, i.e. programs in c, c++ or java, that attaches ProviewR. It is assumed that the reader has basic knowledge in the c programming language.

In many ProviewR applications, coding everything in the plc editor with function object programming works excellent. However, there are applications that with graphic programming will be unnecessarily complex, for example advanced models, handling of databases and material planning. In this case you write an application program in c, c++ or java, that attaches the realtime database, rtdb. The program reads input data from rtdb, makes its calculations, and sets outdata to rtdb, where the data is further processed by the plc program, sent to the I/O system and viewed in operator graphs.

We will concentrate on c/c++, as this is the programming language that is most common in application programming and also has most functionality. The interfaces used are described in Programmer's Reference Manual (PRM).

16.1   Attach to the database and handle object and data

We start by writing a simple c++ program that attaches to the realtime database and links to some objects.

The cpp file should be created on the $pwrp_src directory, or subdirectory to this. We create the directory $pwrp-src/myappl and edit the file ra_myappl.cpp.

Datatypes

In the includefile pwr.h the basic datatypes in ProviewR are defined. The most common is pwr_tBoolean for digital signals and pwr_tFloat32 for analogous signals, but there are also c types for all the other ProviewR types, e.g. pwr_tInt32, pwr_tUInt32, pwr_tString80 etc.

Gdh initialization

The database is attached with a call to gdh_Init() which takes an identifier string for the application as an argument. First we include pwr.h that contains the basic types in ProviewR and rt_gdh.h that contains the API to the database.

```
#include "pwr.h"
#include "rt_gdh.h"

int main() {
  pwr_tStatus sts;

  sts = gdh_Init( "ra_myappl");
```

```
  if ( EVEN(sts)) {
    cout << "gdh_Init failure " << sts << endl;
    exit(0);
  }
}
```

The function returns a status variable of type pwr_tStatus. An even status implies that
something is wrong, an odd that the call was a success. The status can be translated to
a string that gives more information about what is wrong. This is achieved with the
errh interface which is described later.

Read and write attribute values

If we want to read or write an object attribute we can use the functions
gdh_SetObjectInfo() and gdh_GetObjectInfo().

A read and write of the Dv H1-H2-Start can look like this. Note that the value of the Dv
is fetched from the attribute ActualValue.

```
pwr_tBoolean value;

sts = gdh_GetObjectInfo( "H1-H2-Start.ActualValue", &value, sizeof(value));
if (ODD(sts)) {
  value = !value;
  sts = gdh_SetObjectInfo( "H1-H2-Start.ActualValue", &value, sizeof(value));
}
```


Direct link to attributes

Application programs are often put into an infinite loop, supervising attributes in the
database and reacting to certain changes. In this case you preferably direct link to the
attribute, i.e. get a pointer. This is done by gdh_RefObjectInfo(). In the example below
the program is split in an init() function direct linking to attributes, a scan()
function containing the supervision and control functions, and a close() function removing
the direct links.

```
class ra_myappl {
  pwr_tBoolean *start_ptr;
  pwr_tRefId dlid;
 public:
  ra_myappl() {}
  void init();
  void scan();
  void close();
 };

void ra_myappl::init()
{
  sts = gdh_RefObectInfo( "H1-H2-Start.ActualValue", &start_ptr, &dlid,
                          sizeof(*start_ptr));
  if ( EVEN(sts)) exit(0);
}
```

```
void ra_myappl::scan()
{
  for (;;) {
    if ( *start_ptr) {
      // Do something...
      cout << "Starting" << endl;

      *start_ptr = 0;

    }
    sleep(1);
  }
}

void ra_myappl::close()
{
  gdh_UnrefObjectInfo( &dlid);
}
```

In the init() function the pointer start_ptr is set to point to the value of the Dv
H1-H2-Start in the database.

Warning

Note that pointers in c requires caution. If you use pointer arithmetics or array indexes
it is easy to point at the wrong position in the database, and to write in the wrong position.
This can cause errors which are very hard to find the source for.

Direct link to objects

gdh_RefObjectInfo() can, besides direct link to individual attributes, also direct link to
objects and attribute objects.

Suppose that we will set points in a curve and display the curve in a graph. We direct link
to the object H1-H2-Curve of class XyCurve. The includefile pwr_baseclasses.hpp contains a
c++ class, pwr_Class_XyCurve, for the object.

```
#include <math.h>
#include "pwr.h"
#include "pwr_baseclasses.hpp"
#include "rt_gdh.h"

class ra_myappl {
  pwr_Class_XyCurve *curve_ptr;
  pwr_tRefId dlid;
 public:
  ra_myappl() {}
  void init();
  void scan();
  void close();
 };

void ra_myappl::init()
```

```
{
  pwr_tStatus sts;
  pwr_tOName name = "H1-H2-Curve";

  // Connect to database
  sts = gdh_Init( "ra_myappl");
  if ( EVEN(sts)) exit(0);

  // Direct link to curve object
  sts = gdh_RefObjectInfo( name, (void **)&curve_ptr, &dlid, sizeof(*curve_ptr));
  if ( EVEN(sts)) exit(0);
}

void ra_myappl::scan()
{
  for ( unsigned int i = 0;;i++) {
    if ( i % 5 == 0) {
      // Calculate x and y coordinates for a sine curve every fifth second
      for ( int j = 0; j < 100; j++) {
        curve_ptr->XValue[j] = j;
        curve_ptr->YValue[j] = 50 + 50 * sin( 2.0 *  M_PI * (j + i) / 100);
      }
      // Indicate new curve to graph
      curve_ptr->Update = 1;
    }
    else if ( i % 5 == 2)
      curve_ptr->Update = 0;
    sleep(1);
    if ( i > 360)
      i = 0;
  }
}

void ra_myappl::close()
{
  gdh_UnrefObjectInfo( dlid);
}


int main()
{
  ra_myappl myappl;

  myappl.init();
  myappl.scan();
  myappl.close();
}
```

The program is compiled and linked with

```
> g++ -g -c  ra_myappl.cpp -o $pwrp_obj/ra_myappl.o -I$pwr_inc -DOS_LINUX=1
      -DOS=linux -DHW_X86=1 -DHW=x86
> g++ -g -o $pwrp_exe/ra_myappl $pwrp_obj/ra_myappl.o $pwr_obj/pwr_msg_rt.o
      -L$pwr_lib -lpwr_rt -lpwr_co -lpwr_msg_dummy -lrt
```

Later we will see how to use make for compiling and linking.

When opening the object graph for the H1-H2-Curve object, we can study the result.
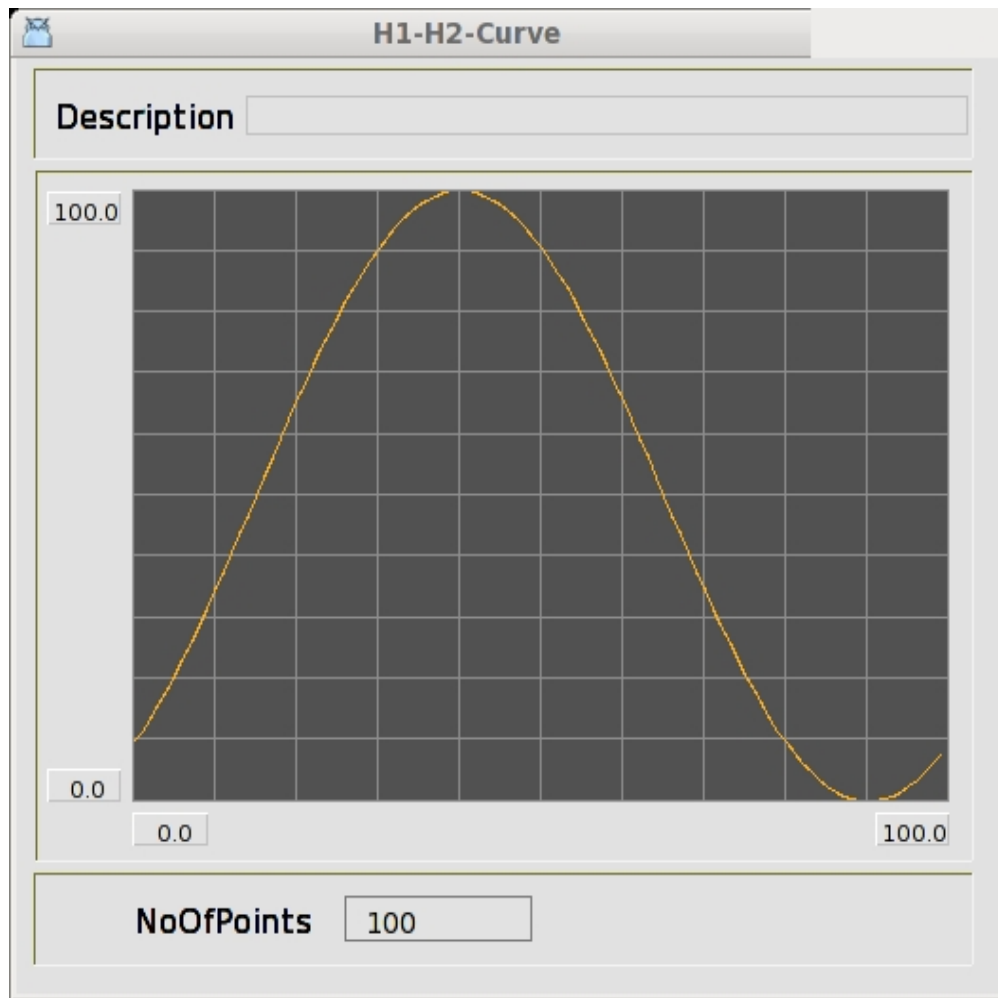
Fig Object graph for the curve object.

16.2   Console log

Log on the console log

Console log

The console log contains log messages from system processes. If there is something wrong with the system, you should look in the console log to examine if any process logs error messages. The error log is a text file on $pwrp_log, pwr_'nodename'.log, which can also be opened in rt_xtt from System/SystemMessages. The loggings have five severity levels, fatal, error, warning, info and success. Fatal and error are colored red, warning colored yellow, info and success colored green.

Also applications can write on the console log. First you attach to the console log with errh_Init(), then you kan write messages with different severity with errh_Fatal(),

errh_Error(), errh_Warning(), errh_Info() and errh_Success(),

errh_Init() is called before gdh_Init() and has as arguments a name of the application and an application index supplied as errh_eAnix_Appl1, errh_eAnix_Appl2 etc. Every application should have a unique application index within the node.

```
#include "rt_errh.h"

sts = errh_Init( "ra_myappl", errh_eAnix_Appl1);
```

To the log functions you send the string that is to be written in the log, e.g.

errh_Error("Something went wrong");

The string can also work as a format statement containing %s to format strings, %f for float and %d for integer, see printf for more info.

errh_Error("Number is to high: %d", n);

The format %m translates a status code to corresponding text

```
catch ( co_error e) {
  errh_Error( "Error status: %m", e.sts());
}
```

Application status

Every application has a status word in the $Node object. It is found in the attribute ProcStatus[] in element applicationindex + 20. The status should reflect the condition of the application and is set by the application itself by the function errh_SetStatus().

errh_SetStatus(PWR__ARUN);

PWR__ARUN is defined in rt_pwr_msg.h and linked to the text "Application running".

Other useful status codes are

PWR__APPLSTARTUP "Application starting up" (info)
PWR__APPLRESTART "Application restaring" (info)
PWR__APPLTERM   "Application terminated" (fatal)

In the node object there is also a SystemStatus that is a kind of sum of all the status of the server and application processes. The most severe server or application status is placed in the systemstatus.

Watchdog

An application that has called errt_Init() is supervised by the system. It should call aproc_TimeStamp cyclic, or the application status is set to "Process timeout" (fatal). The timeout for applications is 5 s.

Application object

An application object can be created for the applications. It is placed in the node hierarchy under the $Node object and is of class Application.

The application object is registered by the function aproc_RegisterObject() that has the object identity for the object as argument.

```
pwr_tObjid aoid;
pwr_tOName name = "Nodes-MyNode-ra_myappl";

sts = gdh_NameToObjid( name, &aoid);
if (EVEN(sts)) throw co_error(sts);

sts = aproc_RegisterObject( aoid);
if (EVEN(sts)) throw co_error(sts);
```

Status graph

The application is viewed in the status graph for the node if the application has attached errh and registered the application object. It will be shown under 'Application' on the row corresponding to the application index. In the graph the status of the application and the last/most severe log message are displayed.
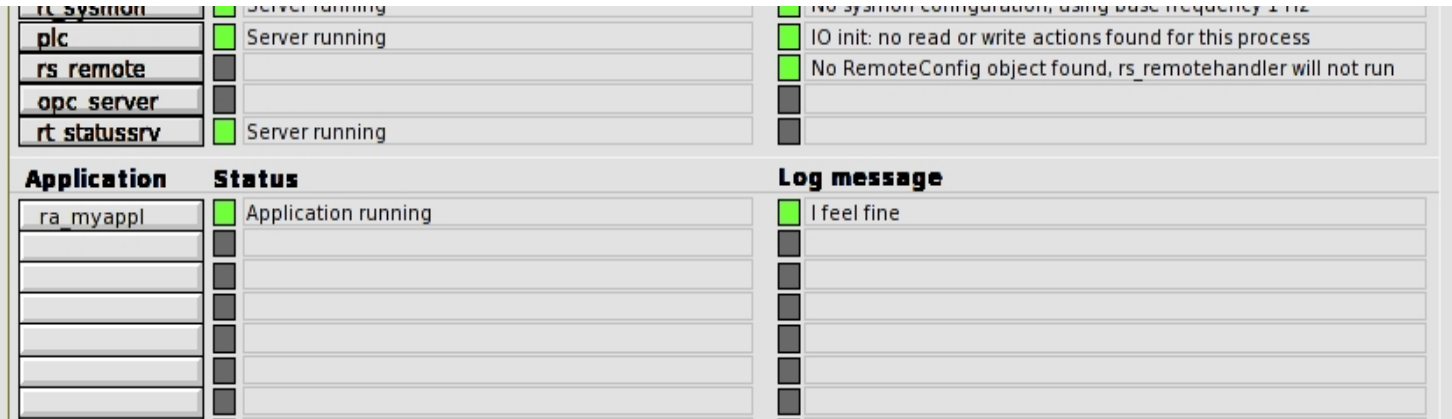
Fig Detail for the status graph displaying status and log message for the application.

If the process is halted, status is set to timeout. This will also affect the system status.
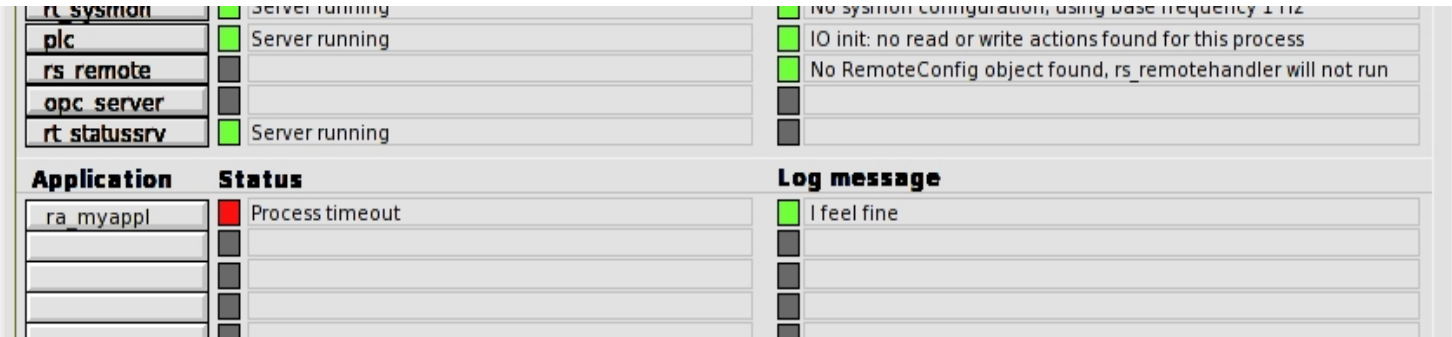
Fig The application is halted.

Example
In the example we have extended the program with the xy-curve above, and inserted

calls to set application status, log on the console log and register the application object.

```cpp
#include <math.h>
#include <iostream>
#include "pwr.h"
#include "pwr_baseclasses.hpp"
#include "rt_gdh.h"
#include "rt_errh.h"
#include "rt_aproc.h"
#include "rt_pwr_msg.h"
#include "co_error.h"

class ra_myappl {
  pwr_Class_XyCurve *curve_ptr;
  pwr_tRefId dlid;
 public:
  ra_myappl() {}
  void init();
  void scan();
  void close();
};

void ra_myappl::init()
{
  pwr_tStatus sts;
  pwr_tOName name = "H1-H2-Curve";
  pwr_tObjid aoid;

  // Init errh with anix 1
  sts = errh_Init( "ra_myappl", errh_eAnix_appl1);
  if ( EVEN(sts)) throw co_error(sts);

  // Write message to consolelog and set application status
  errh_Info( "I feel fine");
  errh_SetStatus( PWR__APPLSTARTUP);

  // Connect to database
  sts = gdh_Init( "ra_myappl");
  if ( EVEN(sts)) throw co_error(sts);

  // Register application object
  sts = gdh_NameToObjid( "Nodes-Saturnus7-ra_myappl", &aoid);
  if ( EVEN(sts)) throw co_error(sts);

  aproc_RegisterObject( aoid);

  // Directlink to curve object
  sts = gdh_RefObjectInfo( name, (void **)&curve_ptr, &dlid, sizeof(*curve_ptr));
  if ( EVEN(sts)) throw co_error(sts);

  errh_SetStatus( PWR__ARUN);
}

void ra_myappl::scan()
```

```
{
  for ( unsigned int i = 0;;i++) {
    // Notify that we are still alive
    aproc_TimeStamp();

    if ( i % 5 == 0) {
      for ( int j = 0; j < 100; j++) {
        curve_ptr->XValue[j] = j;
        curve_ptr->YValue[j] = 50 + 50 * sin( 2.0 *  M_PI * (j + i) / 100);
      }
      curve_ptr->Update = 1;
    }
    else if ( i % 5 == 2)
      curve_ptr->Update = 0;
    sleep(1);
    if ( i > 360)
      i = 0;
  }
}

void ra_myappl::close()
{
  gdh_UnrefObjectInfo( dlid);
}


int main()
{
  ra_myappl myappl;

  try {
    myappl.init();
  }
  catch ( co_error e) {
    errh_Fatal( "ra_myappl terminated, %m", e.sts());
    errh_SetStatus( PWR__APPLTERM);
    exit(0);
  }
  myappl.scan();
  myappl.close();
}
```

16.3   Start the application

An application that is to be started at ProviewR runtime startup is inserted into the application file. This resides on $pwrp_load and is named ld_appl_'nodename'_'qbus'.txt, e.g.

```
$pwrp_load/ld_appl_mynode_999.txt
```

In the file you insert one line for each application that is to be started

```
# id        name        [no]load [no]run file      prio [no]debug  "arg"
ra_myappl, ra_myappl, noload,  run,   ra_myappl, 12,  nodebug,    ""
```

16.4   Receive system events

ProviewR transmits messages at certain events, e.g. when a soft restart proceeds or when
the runtime environment is stopped. An application can listen to these messages, for
example to terminate when ProviewR is terminated. The messages are received from Qcom. You
create a Qcom queue and bind this queue to the queue that submits the messages.

```
#include "rt_qcom.h"
#include "rt_ini_event.h"
#include "rt_qcom_msg.h"

qcom_sQid qid = qcom_cNQid;
qcom_sQid qini;
qcom_sQattr qAttr;

if ( !qcom_Init(&sts, 0, "ra_myappl")) {
  throw co_error(sts);

// Create a queue to receive stop and restart events
qAttr.type = qcom_eQtype_private;
qAttr.quota = 100;
if ( !qcom_CreateQ(&sts, &qid, &qAttr, "events"))
  throw co_error(sts);

// Bind to init event queue
qini = qcom_cQini;
if ( !qcom_Bind(&sts, qid, &qini))
  throw co_error(sts);
```

In each scan you read the queue with qcom_Get() to see if any messages have arrived. You
can also use the timeout in qcom_Get() to wait to next scan. In the example below, the
terminate event is handled, but also the oldPlcStop and swapDone events that indicates
the start and end of a soft restart. You only have to do this if you want the application
to discover new objects of new configurations after a soft restart.

```
int tmo = 1000;
char mp[2000];
qcom_sGet get;
int swap = 0;

for (;;) {
  get.maxSize = sizeof(mp);
  get.data = mp;
  qcom_Get( &sts, &qid, &get, tmo);
  if (sts == QCOM__TMO || sts == QCOM__QEMPTY) {
    if ( !swap)
      // Do the normal thing
      scan();
```

```
    }
    else {
      // Ini event received
      ini_mEvent  new_event;
      qcom_sEvent *ep = (qcom_sEvent*) get.data;

      new_event.m  = ep->mask;
      if (new_event.b.oldPlcStop && !swap) {
        errh_SetStatus( PWR__APPLRESTART);
        swap = 1;
        close();
      } else if (new_event.b.swapDone && swap) {
        swap = 0;
        open();
        errh_SetStatus( PWR__ARUN);
      } else if (new_event.b.terminate) {
        exit(0);
      }
    }
  }
}
```

If you only are interested in stopping the process when ProviewR is taken down, there is a more simple way to kill it. You can put a scripfile, pwrp_stop.sh, on $pwrp_exe where you kill the process.

```
killall ra_myappl
```

16.5   Baseclass for applications rt_appl

The baseclass rt_appl contains many of the initializations and supervision of events described above. By subclassing rt_appl you don't have to supply any code for this, it is done by rt_appl. rt_appl contains three virtual functions that are to be implemented by the subclass, open(), close() and scan(). open() is used at initialization to direct link to attributes anad object, scan() is called cyclic with supplied cycletime, and in close() you remove the direct links.

rt_appl handles this:

- Initialization of gdh, errh and qcom
- Setting of application status at startup and restart
- Handling events for soft restart and termination
- Timestamps to avoid timeout

This example shows the application ra_appl subclassing rt_appl.

```
class ra_appl : public rt_appl {
public:
 ra_appl() : rt_appl( "ra_appl", errh_eAnix_appl1) {}
 void open();
 void close();
 void scan();
};
```

```
void ra_appl::open()
{
 // Link to database objects
}

void ra_appl::close()
{
 // Unlink to database objects
}

void ra_appl::scan()
{
 // Do something
}

int main()
{
 ra_appl appl;

 appl.init();
 appl.register_appl( "Nodes-MyNode-MyAppl");

 appl.mainloop();
}
```

16.6   Send alarms and messages

From an application you can send alarms and messages to the alarmlist and eventlist of
the operator. First you have to connect to the event monitor with mh_ApplConnect() which
takes the object identity for the application object as first argument. For subclasses to
the rt_appl class this identity is fetched with apploid().

```
#include "rt_mh_appl.h"

pwr_tUInt32 num;
pwr_tOid aoid = apploid();
sts = mh_ApplConnect( aoid, mh_mApplFlags(0), "", mh_eEvent_Info, mh_eEventPrio_A,
    mh_mEventFlags_Bell, "", &num);
  if (EVEN(sts)) throw co_error(sts);
```

We then can send alarms with mh_ApplMessage().

```
mh_sApplMessage msg;
pwr_tUInt32 msgid;

memset( &msg, 0, sizeof(msg));
msg.EventFlags = mh_mEventFlags(mh_mEventFlags_Returned |
      mh_mEventFlags_NoObject |
      mh_mEventFlags_Bell);
time_GetTime( &msg.EventTime);
```

```
strcpy( msg.EventName, "Message from ra_myappl");
strcpy( msg.EventText, "I'm up and running now !");
msg.EventType = mh_eEvent_Alarm;
msg.EventPrio = mh_eEventPrio_B;

sts = mh_ApplMessage( &msgid, &msg);
if (EVEN(sts)) throw co_error(sts);
```
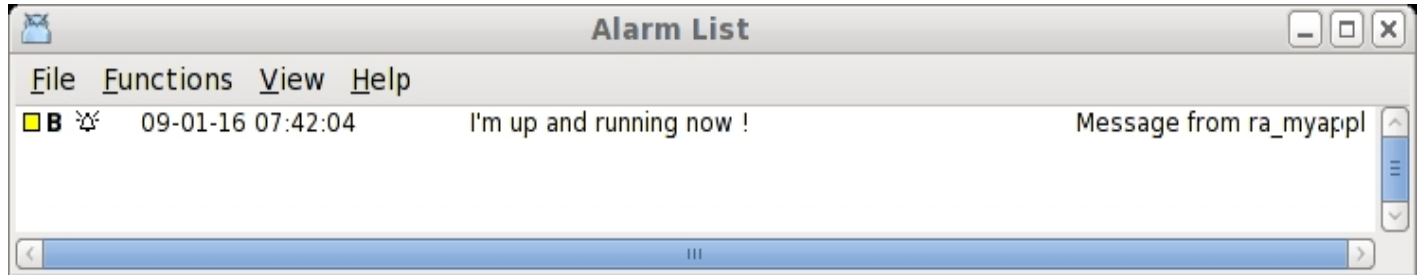

Fig The alarm in the alarmlist.

16.7 Communicate with other processes

ProviewR's protocol for communication between processes can be used also by applications.
The communication can be

- between processes in the same node
- between processes in different nodes that belong to the same project
- between processes in nodes that belong different projects, if the project has the same
  Qcom bus. In this case the nodes have to be configured by FriendNodeConfig objects where
  Connection is set to QcomOnly.

Read more about Qcom in Qcom Reference Guide.

16.8 Fetch data from a storage station

Data stored in a ProviewR storage station can be fetched by the client interface sevcli.
First you initiate sevcli with sevcli_init() and state which storage station you want to
fetch the data from with sevcli_set_servernode().

```
sevcli_tCtx sevctx;
char server_node[40] = "MyStorageStation";

if ( !sevcli_init( &sts, &sevctx))
  throw co_error(sts);

if ( !sevcli_set_servernode( &sts, sevctx, server_node))
  throw co_error(sts);
```

Then you can fetch data with sevcli_get_itemdata(). Data is identified by object identity and
attribute name. You also state the time range for the data that is to be fetched and
maximum number of points.

```
pwr_tTime *time_buf;
```

```
void *value_buf;
pwr_tTime from = pwr_cNTime;
pwr_tTime to = pwr_cNTime;
int rows;
pwr_eType vtype;
unsigned int vsize;
pwr_tOName name = "H1-H2-Temperature";
pwr_tOName aname = "ActualValue";
pwr_tObjid oid;
char timstr[40];

sts = gdh_NameToObjid( name, &oid);
if (EVEN(sts)) throw co_error(sts);

if ( !sevcli_get_itemdata( &sts, sevctx, oid, aname, from, to, 1000, &time_buf, &value
                           &rows, &vtype, &vsize))
  throw co_error(sts);

for ( int i = 0; i < rows; i++) {
  time_AtoAscii( &time_buf[i], time_eFormat_DateAndTime, timstr, sizeof(timstr));

  cout << timstr << " " << ((pwr_tFloat32 *)value_buf)[i] << endl;
}

free( time_buf);
free( value_buf);
```

Finally you call sevcli_close() to disconnect the server node.

```
sevcli_close( &sts, sevctx);
```

16.9   I/O handling

If an application requires fast and synchronized I/O data it can work directly against the
I/O system and call the I/O routines to read and write I/O on its own.

Initialization is done with the function io_init(), to which a process argument is supplied.
Process identifies which I/O units (agent, rack or card) are handled by a specific process.
Each I/O object has a Process attribute and if this corresponds to the process sent as
argument to io_init(), the unit will be handled by the application. If a card is handled
by an application, also the rack and agent of the card have to be handled by the application.

As the Process attribute is a bitmask, a unit can be handled by several processes by
setting several bits in the mask. If you, for example, have several cards in a rack, and some
of the cards should be handled by the plc-process and some by an application, the rack unit
has to be handled by both the plc and the application. Whether it works, to handle a
unit from several processes, depends on how the I/O methods for the unit are written.
For example for Profibus, you can't divide the handling of slaves into different
processes.

```
#include rt_io_base.h
```

```
io_tCtx io_ctx;

sts = io_init( io_mProcess_User, pwr_cNOid, &io_ctx, 0, scantime);
if ( EVEN(sts)) {
  errh_Error( "Io init error: %m", sts);
  throw co_error(sts);
}
```

Reading is executed with io_read() which reads data from the I/O units and places
the data in the signals connected to the unit. The application preferably links directly
to these signals, and also to the signals of the output units. The output units are written
to with the function io_write().

```
sts = io_read( io_ctx);

sts = io_write( io_ctx);
```

16.10   Thread safe strings and times

Times and string attributes in the realtime database can not be accessed in an atomic
operation and thus needs special consideration when handled in two different threads or
processes. The problem is that when one thread is writing a time or string attribute, another
thread can read the attribute before the write is finished, and thus read a value that
contains parts of both the previous and the new value. To avoid this the read and write
operations should be protected by a lock. There are two locks, one for time attributes and one
for string attributes.

Time lock

The time lock is used for attributes of type pwr_tTime and pwr_tDeltaTime. When a time
attribute is read or written, the lock should be set. The following plc function objects will
use the lock to protect the reading or writing of the time value.
StoATv, StoDTv, CStoATv, CStoDTv, StoDTp, StoATp, CStoDTp, CStoATp, GetATp, GetDTp, GetATv,
GetDTv, CurrentTime.

Note that other time objects are not protected by the lock. It's not safe to use the output
from for example a AtAdd in another Plc thread or an application program. In this case the
time should first be stored in an ATv object.

To read and write times, applications should use these gdh functions
```
void gdh_GetTimeDL( pwr_tTime *atp, pwr_tTime *time);
void gdh_SetTimeDL( pwr_tTime *atp, pwr_tTime *time);
void gdh_GetDeltaTimeDL( pwr_tDeltaTime *dtp, pwr_tDeltaTime *time);
void gdh_SetDeltaTimeDL( pwr_tDeltaTime *dtp, pwr_tDeltaTime *time);
pwr_tStatus gdh_GetObjectInfoTime( char *name, pwr_tTime *time);
pwr_tStatus gdh_SetObjectInfoTime( char *name, pwr_tTime *time);
pwr_tStatus gdh_GetObjectInfoDeltaTime( char *name, pwr_tDeltaTime *time);
pwr_tStatus gdh_SetObjectInfoDeltaTime( char *name, pwr_tDeltaTime *time);
```

Before using any of these functions, the time lock has to be initialized by the application
with a call to lck_Create(), eg
```
lck_Create(&sts, lck_eLock_Time);
```

String lock

The string lock is used for string attributes. The following plc function objects will use the lock. StoSv, CStoSv, StoSp, CStoSp, StoNumSp, CStoNumSp, GetSp, GetSv.

Other string objects are not protected by the lock and the string values in these objects should not be read or written in other Plc threads or in application programs.

To read and write string, applications should use these gdh functions

```
void gdh_GetStrDL( char *sp, char *str, int size);
void gdh_SetStrDL( char *sp, char *str, int size);
pwr_tStatus gdh_GetObjectInfoStr( char *name, char *str, int size);
pwr_tStatus gdh_SetObjectInfoStr( char *name, char *str, int size);
```

Before using any of these functions, the string lock has to be initialized by the application with a call to lck_Create(), eg
```
lck_Create(&sts, lck_eLock_Str);
```

16.11   Build an application

A c++ application has to be compiled and linked, and you can use make to do this. ProviewR contains a rule file, $pwr_exe/pwrp_rules.mk, that contains rules for compilation.

A makefile for the application ra_myappl on the directory $pwrp_src/myappl can look like this ($pwrp_src/myappl/makefile):

```
ra_myappl_top : ra_myappl

include $(pwr_exe)/pwrp_rules.mk

ra_myappl_modules : \
  $(pwrp_obj)/ra_myappl.o \
  $(pwrp_exe)/ra_myappl

ra_myappl : ra_myappl_modules
 @ echo "ra_myappl built"

#
# Modules
#

$(pwrp_obj)/ra_myappl.o : $(pwrp_src)/myappl/ra_myappl.cpp \
                          $(pwrp_src)/myappl/ra_myappl.h

$(pwrp_exe)/ra_myappl : $(pwrp_obj)/ra_myappl.o
 @ echo "Link $(tname)"
 @ $(ldxx) $(linkflags) -o $(target) $(source) -lpwr_rt -lpwr_co \
   -lpwr_msg_dummy -lrpcsvc -lpthread -lm -lrt
```

The makefile is executed by positioning on the directory and writing make

```
make
```

You can also insert the build command into the Application object for the application in the attribute BuildCmd. In this case the build command is

```
make --directory $pwrp_src/myappl -f makefile
```

This command is executed when the node is built from the configurator. This is a way to ensure that all applications are updated when the node is built.

16.12   Java applications

Some API also exist for java in the shape of the classes Gdh, Errh and Qcom. Below is an example of a java application attaching the realtime database and reading and writing an attribute.

```java
import jpwr.rt.*;

public class MyJappl {
  public MyJappl() {
    Gdh gdh = new Gdh( null);

    CdhrBoolean rb = gdh.getObjectInfoBoolean( "H1-H2-Start.ActualValue");

    PwrtStatus rsts = gdh.setObjectInfo( "H1-H1-Start.ActualValue",
          !rb.value);
  }

  //Main method
  public static void main(String[] args) {
    new MyJappl();
  }
}
```

To compile and execute you have to put $pwr_lib/pwr_rt.jar and the working directory into CLASSPATH, and $pwr_exe into LD_LIBRARY_PATH

```
> export CLASSPATH=$pwr_lib/pwr_rt.jar:$pwrp_src/myjappl
> export LD_LIBRARY_PATH=$pwr_exe
```

Compile with
```
> javac MyJappl.java
```

and execute with
```
> java MyJappl
```

For auto start of the application you create a shellscript that exports CLASSPATH and LD_LIBRARY_PATH, and starts the java application. The script is inserted into the appl-file in the same way as a c application.

17   Creating Process Graphics

This chapter describes how you create process graphics.

Process graphics are drawn and configured in the Ge editor.

The Ge editor

Ge is opened from the menu in the navigator: 'Functions/Open Ge'. It consist of

- a tool panel
- a work area
- a subgraph palette
- a color palette
- a window displaying the plant hierarchy
- a navigation window

Background picture

A background image is drawn with base objects such as rectangles, circles, lines, polylines
and text. These are found in the tool panel. Create a base object by activating the pushbutton
in the tool panel and dragging or clicking MB1 in the work area. If the base object should be
filled, select the object and activate fill in the tool panel. Change the fillcolor by
selecting the object and click on the desired color in the color palette. Change the border
color by clicking with MB2 in the color palette, and the text color by clicking Shift/Click MB1.

Subgraphs

A subgraph is a graphic component, e.g. a valve, a motor, a pushbutton. To create a subgraph,
select a subgraph in the subgraph palette and click MB2 in the work area.

Groups

Base objects and subgraphs can be grouped together by selecting them and activating
'Functions/Group' in the menu.

Dynamics

Subgraphs and groups have dynamic properties, i.e. they can be connected to signals in the
runtime database, and change color, position or shape depending on the values of the signals.
A subgraph often has default dynamic behavior, for example an indicator shifts between two
colors. You only have to connect the indicator to a digital signal to make it work. This is
done by selecting a signal in the plant hierarchy window, and click on the valve with
Ctrl/DoubleClick MB1.

A pushbutton has an action property; it sets, resets or toggles a signal in the database.
A button with a set action is created by selecting a ButtonSet in the subgraph palette and
clicking MB2 in the work area. The signal that should be set is connected as above, by selecting
the signal and clicking with Ctrl/DoubleClick MB1 on the button. In the object editor, a button
text can be assigned.

Connect a subgraph to a signal
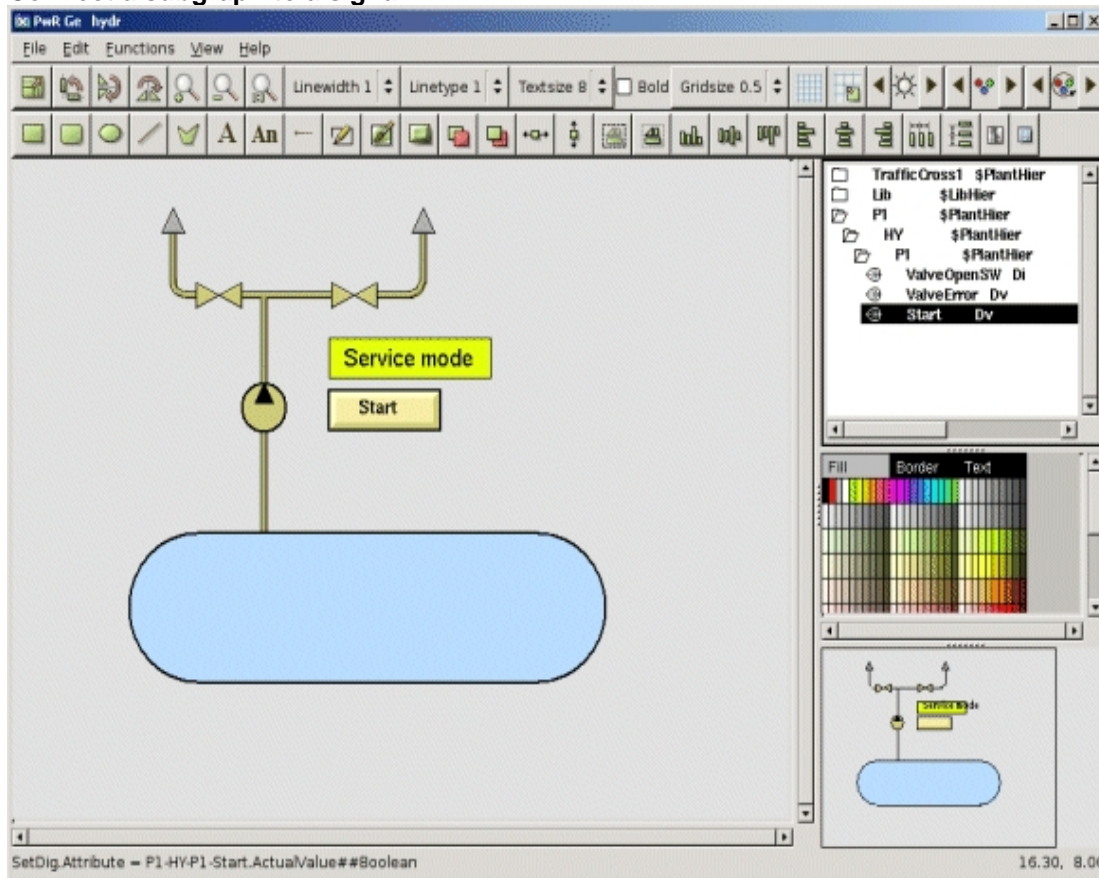
Groups also have a dynamic property, i.e. they can shift color, move, or perform some action. They don't have any default action or default color, as the subgraphs. You have to assign this for each group.

The Object editor

Base objects, subgraph objects and groups have properties, that are changed from the object editor. The object editor is opened by selecting the object, and activating 'Function/Object attributes' in the menu. By opening the object editor for the pushbutton mentioned above, you can for example enter the text that is displayed in the button in the attribute 'Text'.

If a subgraph has more advanced dynamics, for example shift between several colors, you often have to connect it to several signals. If you open the object editor for a valve, you see that it can be connected to two attributes, 'DigError.Attribute' and 'DigLowColor.Attribute'. The DigError attribute indicates that something is wrong, and if this signal is true, the valve is colored red. The DigLowColor attribute is connected to the open switch of the valve. If this signal is false, the valve is colored in the color stated in 'DigLowColor.Color'. If the signal is true, it keeps the color given in the editor. The signals of the two attributes are inserted by selecting each signal in the plant hierarchy respectively, and clicking with Ctrl/Doubleclick MB1 on the attribute row of the attribute. The color 'DigLowColor' is stated by opening the attribute and selecting one of the 300 colors. The colors have the same order as in the color palette, and with a little practice they can be identified by the name.

The Object Editor for a valve

```
X Object Attributes                              _ □ X
 File   Functions   Help
   📄   SubGraph            pwr_valve           ▲
   ▭   DigError.Attribute
   ▭   DigLowColor.Attribute
   📄   DigLowColor.Color   Inherit
   📄   Cycle               Inherit
   📄   DynType             Inherit
   📄   Action              Inherit




                                                ▼
 ◄ |                                        | ► |
              ▲▲▲▲▲▲▲
```
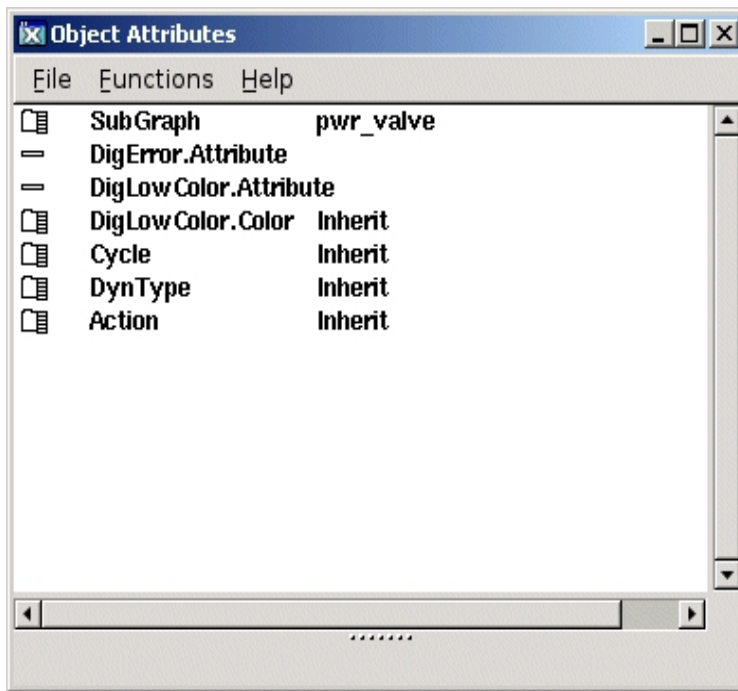
Graph borders

The drawing area in Ge is unlimited in every direction, so before saving the graph, you have
to state the borders of the graph. Open the graph attributes with 'File/Graph' attributes in
the menu. Measure the coordinates of the upper right corner, and insert as x0 and y0, then
measure the coordinates of the lower left corner and insert as x1 and y1. The measurement is
done by placing the cursor in position and reading the coordinates in the the message row.

Graph Attributes

```
X Object Attributes                               _ □ X
  File   Functions   Help
   ▭   subgraph            0                      ▲
   ▭   x0                  2.500000
   ▭   y0                  -0.500000
   ▭   x1                  26.000000
   ▭   y1                  16.500000
   ▭   Scantime            0.500000
   ▭   FastScantime        0.500000
   ▭   AnimationScantime   0.500000
   ▭   JavaWidth           0
   ▭   IsJavaApplet        0
   ▭   IsJavaFrame         0
   ▭   BackgroundImage
   ▭   BackgroundTiled     0
   ▭   DoubleBuffered      0
   📄   MB3Action           Close
   ▭   Translate           0



                                                  ▼
 ◄ |                                          | ► |
               ▲▲▲▲▲▲▲
```
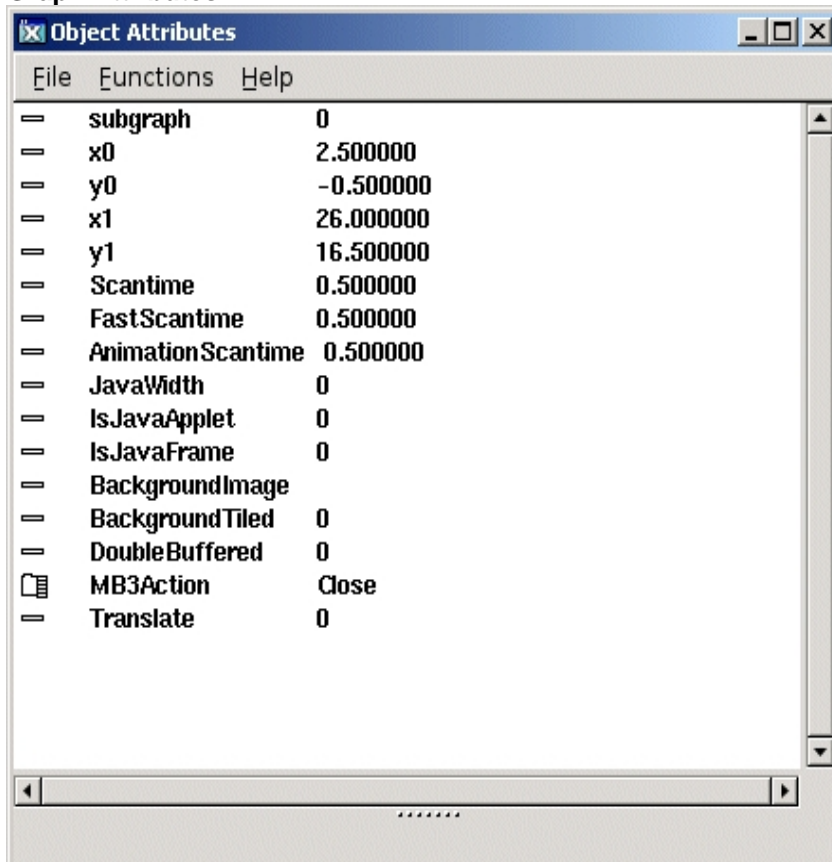
Configuration in the workbench

The XttGraph object

To each plant graphics an XttGraph object is belonging. This object is usually a child of the operator place object (OpPlace) for the node, on which the graphics will be displayed. It is necessary to create an XttGraph object for each node, on which the graphics will be displayed. However, you only need to have one graph file. The following attributes in the graph object must be set to appropriate values:

- Action, the name of the pwg file with file type, e.g 'hydr.pwg'
- Title, the title of the graph window
- ButtonText, text of the button in the operator window

The XttGraph object contains other attributes which e.g. helps you to customize the position and size of plant graphics. These attributes are described in detail in ProviewR Objects Reference Manual.
See XttGraph in Object Reference Manual

18   Web operator environment

Besides the ordinary operator environment in X windows, ProviewR also contains an operator environment written in HTML5 and javascript. The environment is runnable in most webbrowsers on phones, tablets and PCs.

The interface consists of three parts
- an operator window opened in the web browser.
- a server process that serves the operator window with information via a web socket.
- a directory where files are available for the web.

Operator window

The operator window displays a menu to the left and a start page to the right.
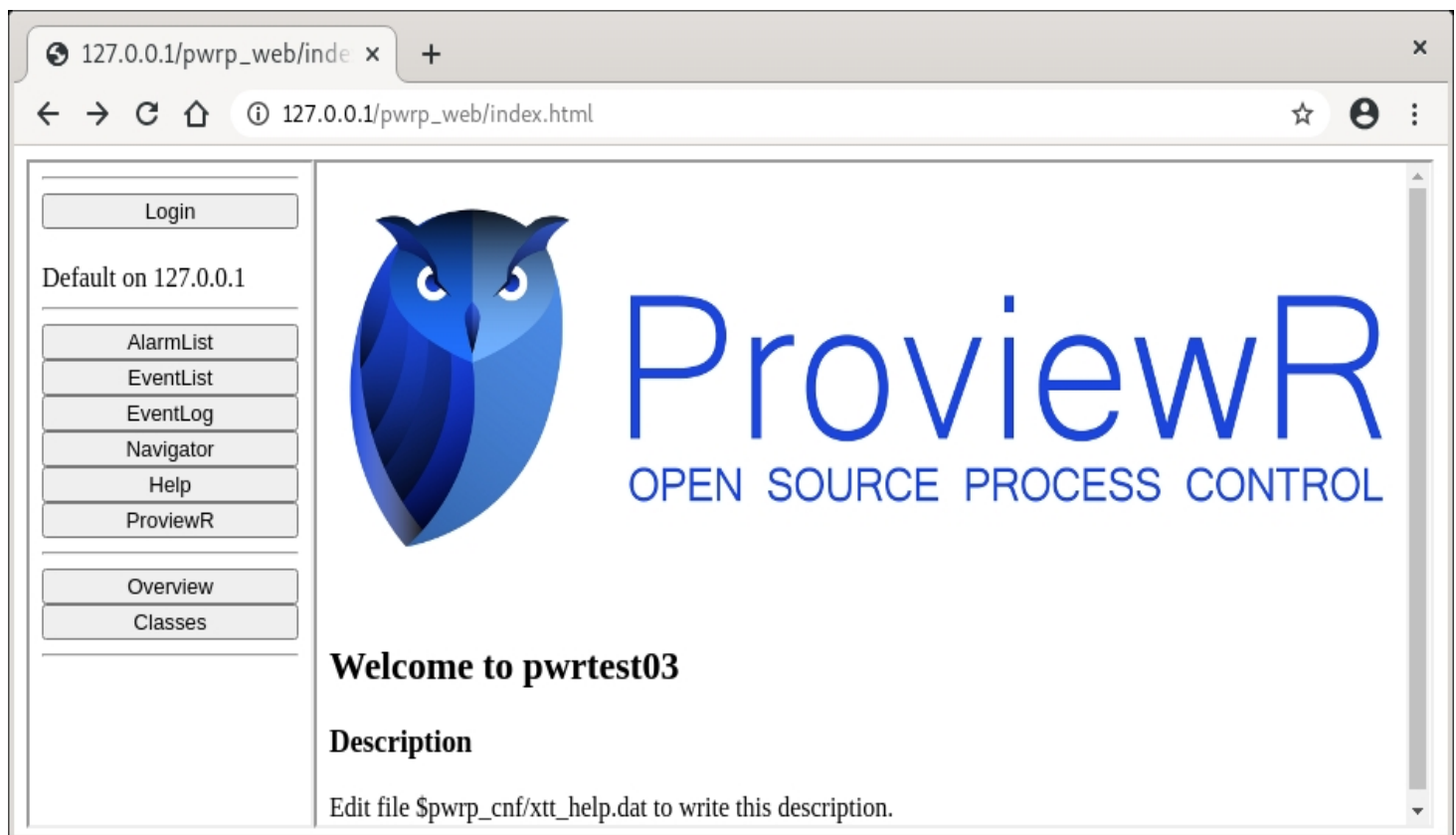
Fig Web operator window

The operator window is configured with an OpPlaceWeb object in the node hierarchy.

The menu is divided in three sections. The first contains a number of standard functions
- language selection.
- login and logout.
- open alarm and event list.
- open event log.
- open runtime navigator.

- link to project help texts.
- link to ProviewR documentation.

These buttons can be enabled or disabled with the Enable/Disable attributes in the OpPlaceWeb object.

The second section contains buttons to open graphs configured with WebGraph objects. The WebGraph objects is placed as children to the OpPlaceWeb object and contains the name of the Ge graph pwg-file.

The third section contains links defined with WebLink objects. The WebLink objects contains an URL and is positions below the OpPlaceWeb object.

The right page will by default contain the help text for the project, written in the file $pwrp_cnf/xtt_help.dat. If another page is prefered, this can be stated in the StartURL attribute.

There can be serveral OpPlaceWeb objects in one node, configuring web pages with different settings. The OpPlaceWeb objects should have different filenames stated in the FileName attribute, and can be opened with an URL to the specified file. The default file is index.html for the first OpPlaceWeb object, and index2.html, index3.html etc, for the preceeding OpPlaceWeb objects.

The opplace with the FileName index.html can be opened with the URL

```
http://'hostname'/pwrp_web/index.html
```
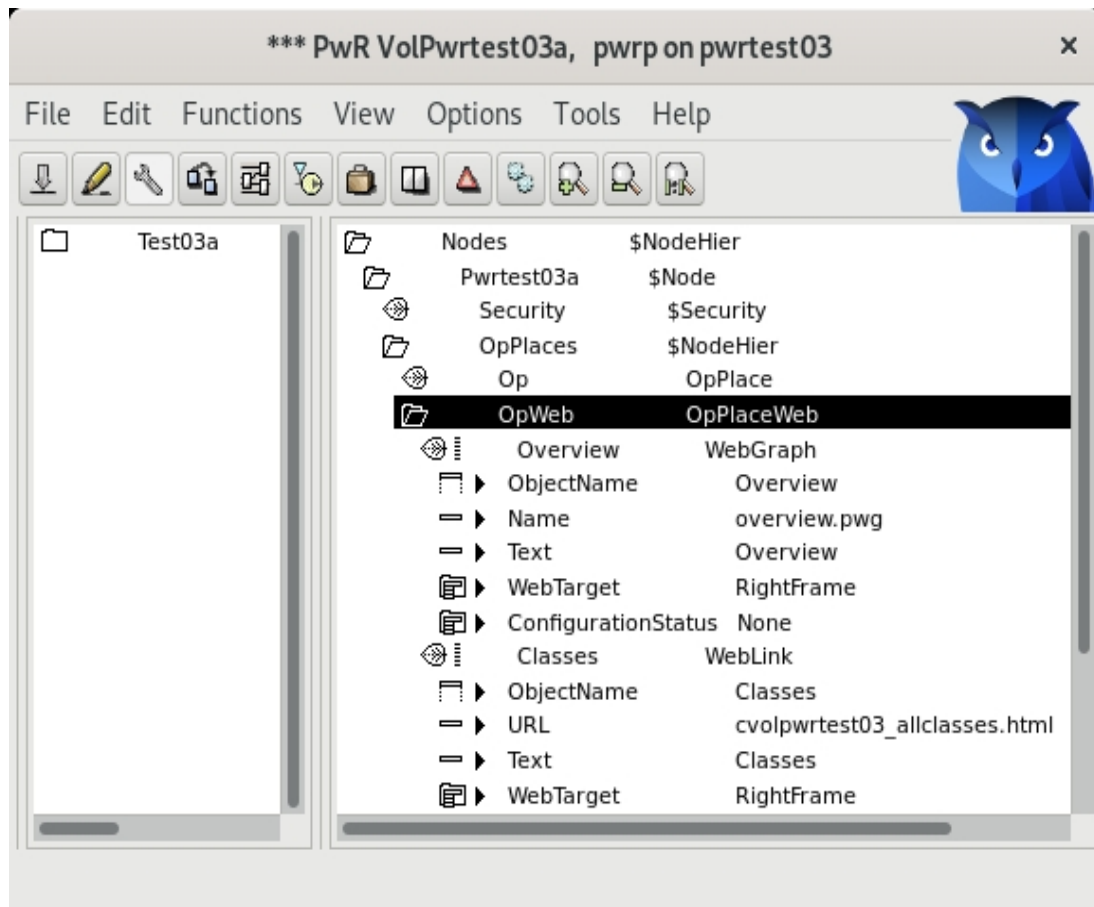

Fig Web operator configuration

Server process

The server process is configured with a WebSocketServer object in the node hierarchy. The server is a java process and requires that java is installed. The server listens for requests from operator windows to open a web socket, and supplies information about the realtime database to the operator window.

Alarm and event list

The alarm and event list requires that the EventSelectList in the WebSocketServer object is filled in with hierarchies from which alarms and events should be displayed.

Web directory

The operator window requires a number of files, and these are gathered on the web directory, pwrp_web. This directory is exposed by the web server and available from the web.

Here are some files that should be present on the directory

- pwg files for Ge graphs, used in WebGraph objects. Should be copied from $pwrp_pop.
- pwg files for object graphs, can be copied from $pwr_exe.
- flw files for plc trace, shoud be copied from $pwrp_load.
- crossreference files, rtt_crr*.dat should be copied from $pwrp_load.
- xtt help text html files. These are generated when the node is built.
- documentation of local classes are generated when the class volume is built.
- All files on $pwr_web are copied to $pwrp_web at installation of the pwrrt package.

Some files are generated or copied at installation, but some have to be copied manually or preferably automatically by creating Build objects in the directory volume.
In the example below there are BuildCopy objects configured to copy

- pwg files from $pwrp_pop.
- flw files from $pwrp_load.
- crossreference files from $pwrp_load.
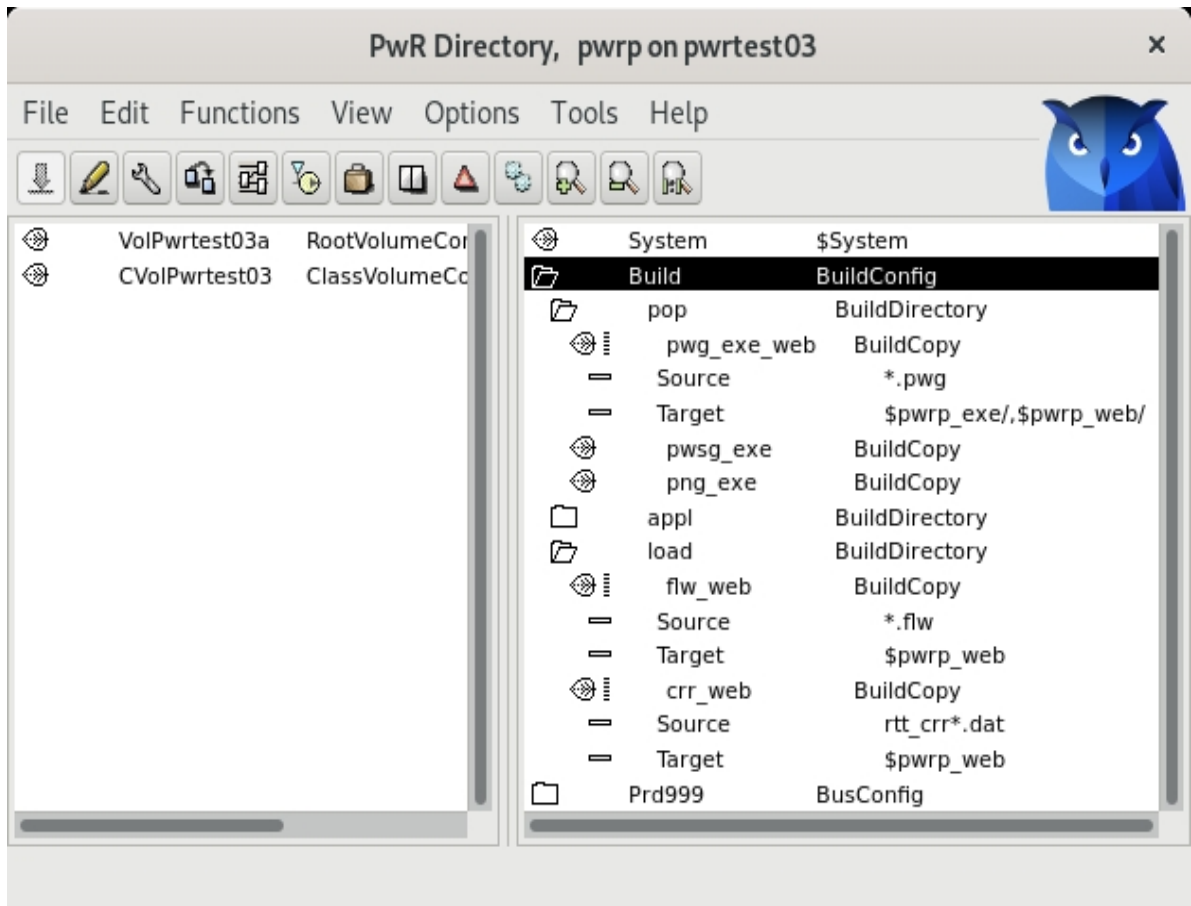
Fig Directory pwrp_web build configuration

Help texts

The help texts for the project in the file xtt_help.dat, is exported to html files on
$pwrp_web by the build method of the OpPlaceWeb object. The start page for the help texts are
$pwrp_web/xtt_help_index.html. This is showed as default in the right frame of the home page
if no other URL is stated in the StartURL attribute of the OpPlaceWeb.

If the helptext contains image tags, the png or gif files have to be copied to $pwrp_web.

Distribution

The files on $pwrp_web should be part of the distribution package, and distributed to
$pwrp_web on the process or operator station. This is done by setting WebFiles in the
Distribution object under the NodeConfig object in the directory volume.

Web server configuration

The pwrp_web directory has to be avaiable from the web, and inserted in the web server
configuration. For nginx and apache this is done at installation of the pwrrt package, but
if you want to view a project on a development station this has to be done manually. Hhere are
some examples.

nginx
Add these lines in /etc/nginx/sites-enabled/default (replace /pwrp/common/web/ with
the real location of $pwrp_web)

```
location /pwrp_web/ {
  alias /pwrp/common/web/;
}
```

 apache
Add these lines in /etc/apache2/apache2.conf (replace /pwrp/common/web/ with
the real location of $pwrp_web)

```
Alias /pwrp_web/ /pwrp/common/web/

<Directory /pwrp/common/web>
    AllowOverride None
    Require all granted
</Directory>
```

Security

If the login frame is enabled, a user can login with a valid username and password, and gain
the privileges granted for the logged in user. Valid users are members or the system group
specified in WebSystemGroup in the Security object.

In the example below, the system group 'Common' is used that contains the ordinary users
pwrp, system etc, but you are recommended to create a specific system group with special
users for the web.

For not logged in users, the privileges are determined by DefaultWebPriv in the Security
object. It is recommended that DefaultWebPriv is RtRead or zero (no privileges). Users that
have no privileges can only open the login frame.
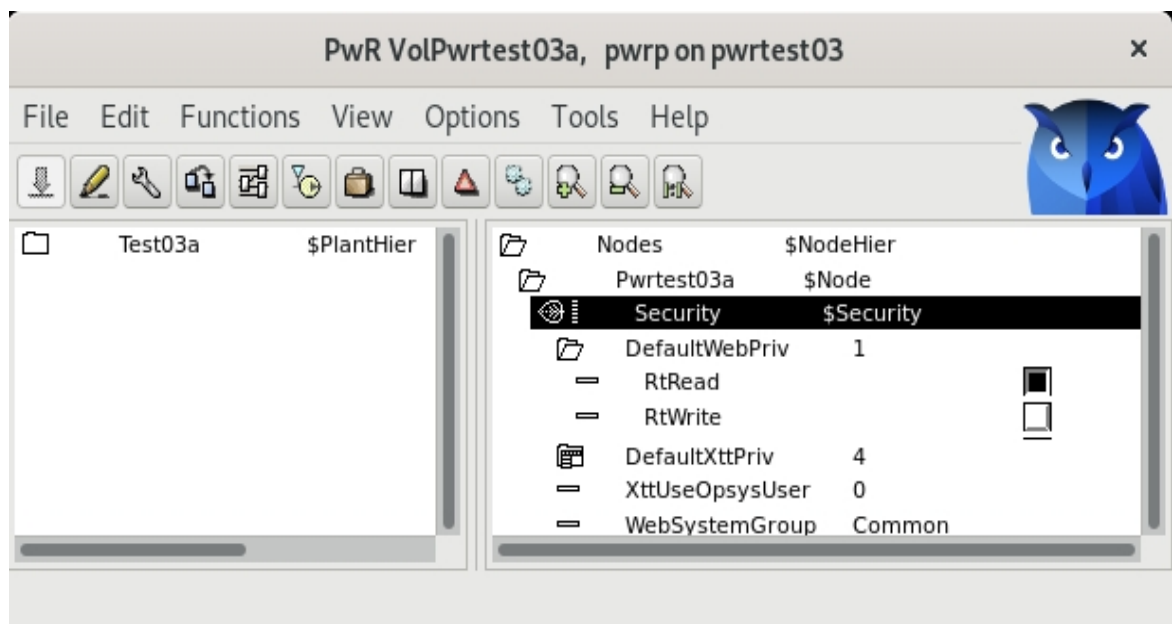
Fig Security configuration

19   Starting and testing a ProviewR system

In preceding chapters we have described how to configure a ProviewR system, how to create
PLC programs and how to create plant graphics. Now it is time to run and test the system.

This chapter shows how to:

- build the system.
- start the simulate environment
  and the runtime monitor
- distribute
- start the runtime environment

Syntax check

Some classes have a syntax check method, checking that the object is correctly configured.
The methods of the signals objects check for example that they are connected to a channel, and
the method for a PlcPgm checks that it is connected to a plc thread object etc. The syntax methods
captures a great deal of errors, but does not guarantee that everything will work faultlessly.

The syntax methods are executed from 'Functions/Syntax Check' in the Configurator menu.

19.1   Build

Before you have a running system for a node, you have to build the node. It means that you
generates what you need in the runtime environment, e.g. a boot file specifying which volumes
are to be loaded, loadfiles for the volumes containing info about the objects in the volume,
an executable for the plc program etc.

You build by activating 'Functions/Build Node' in the Configurator menu, or the corresponding
button in the tools panel. If there are several nodes configured in the project, you also
have to select a node from the viewed list of nodes.
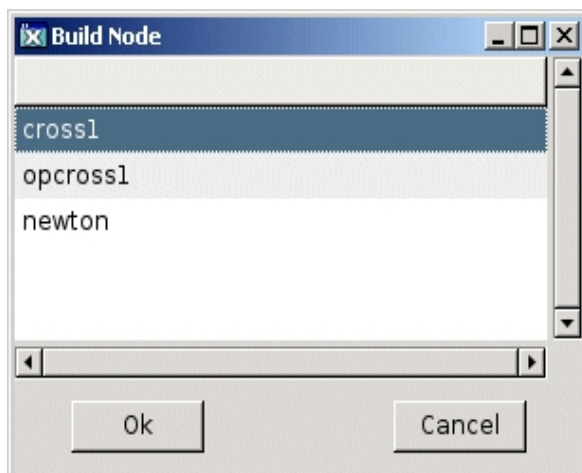
Fig Node selection

Also the build is divided into methods for different classes. The build method for a PlcPgm generate code for the modified windows and compiles this code. The build method for a volume creates a loadfile for the volume, and calls the build mehtods for all objects in the volume. The build method for a node calls the build method for the volume, creates a bootfile for the node, and to link the plc executable. The build methods for an object can be called by selecting the object and activating 'Functions/Build Object' in the menu, and the build method for the current volume by activating 'Functions/Build Volume'.

Below follows a description of some build methods.

XttGraph	Copies the pwg file for the graph from $pwrp_pop to $pwrp_exe.
WebGraph	Copies the pwg file from $pwrp_pop to $pwrp_web.
OpPlaceWeb	Generates html-files for the homepage of the node, and converts xtt helpfiles to web format.
PlcPgm	Generates c-code that is compiled by the c compiler.
RootVolume	Calls the build method for all objects in the volume, and creates a load file with info about the objects in the volume. Also creates crossreference files if specified in Options.
ClassVolume	Generates include files with c structs for the classes of the volume, and creates a loadfile with the type and class definitions.
Node	Calls the build method for the rootvolume, if the volume is available. Creates a bootfile with info about which volumes are to be loaded, and links the plc program of the node.

Normally the build method first checks if anything is modified, and only performs the build if it finds a modification. In some cases you want to force a build, and then set 'Force' in the Build column in options opened from 'Options/Settings' in the menu. It is also possible to mark that you want to create crossreference files ag build, or that compile and link of the plc should be performed with debug.

If a node contains subvolumes or shared volumes, these have to be built by 'Build Volume' before building the node.

19.2   Simulate

Simulate means starting the system for a process or operator station on the development station. This is a fast way to test programs and process graphics during the development phase. Also when a system is in production, you can test modifications of the system before downloading them to the production system.

When simulating, the input data from the process has to be simulated. This is done by creating a PlcPgm that reads output data to the process, i.e. Do an Ao values, and from these set values to Ai and Di objects. There are special StoDi and StoAi objects for this purpose, only used at simulation. You have to assure that simulate program only executes at simulation, e.g. by setting the ScanOff attribute of the PlcWindow object if IOSimulflag in the IOHandler object is 1.

To be able to simulate on the development station, it has to be configured in the directory volume. The simulation is performed on a separate QCom bus, which is configured by a BusConfig object on the top level in the right window of the directory volume. In this you state the bus identity, e.g. 999. Below the BusConfig object you place a NodeConfig object for the development station, and fill in the node name and IP address. You can use the loopback address 127.0.0.1 as long as you don't plan to communicate to other nodes. You also have to specify which volume you want to simulate by setting the name of the RootVolumeLoad object below the NodeConfig object, to the name of the volume.

Note that the configuration guide for the directory volume normally creates a simulate bus and a simulate node.
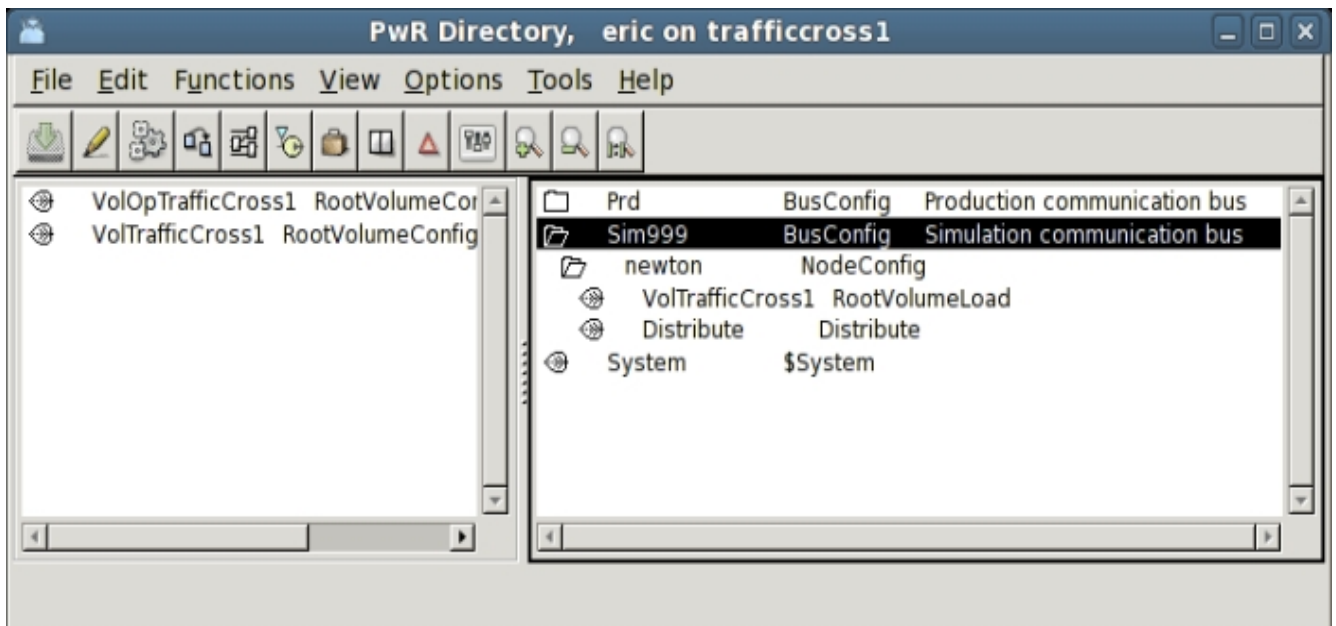
Fig Directory volume with configuration of the development station newton

When the development station is configured, you build by opening the configurator for the volume that is to be simulated, and activate 'Build Node' and select the development station in the viewed list of nodes.

Before starting the runtime you have to define the environment variable PWR_BUS_ID to the identity fo the simulation bus. A default value for PWR_BUS_ID is set in the file /etc/proview.cnf, parameter QcomBusId. With the command

```
> echo $PWR_BUS_ID
```

you check the bus identity, and with the command

```
> export PWR_BUS_ID=999
```

you assign another value. This command you can for example insert in $pwrp_login/login.sh.

Now you can start the ProviewR runtime with

```
> rt_ini &
```

and stop with

```
> . pwr_stop.sh
```

If ProviewR doesn't start you can add -i to the start command to see error messages.

```
> rt_ini -i
```

Note that you always have to reset by executing '. pwr_stop.sh' before a new start attempt.

When the runtime environment is running you can explore the system by starting the runtime

navigator

```
> rt_xtt
```

You can also use the runtime monitor to start the runtime environment. See below.

19.2.1   Simulate Server

A simulate server can be started to control the execution of the plc program, and to store the
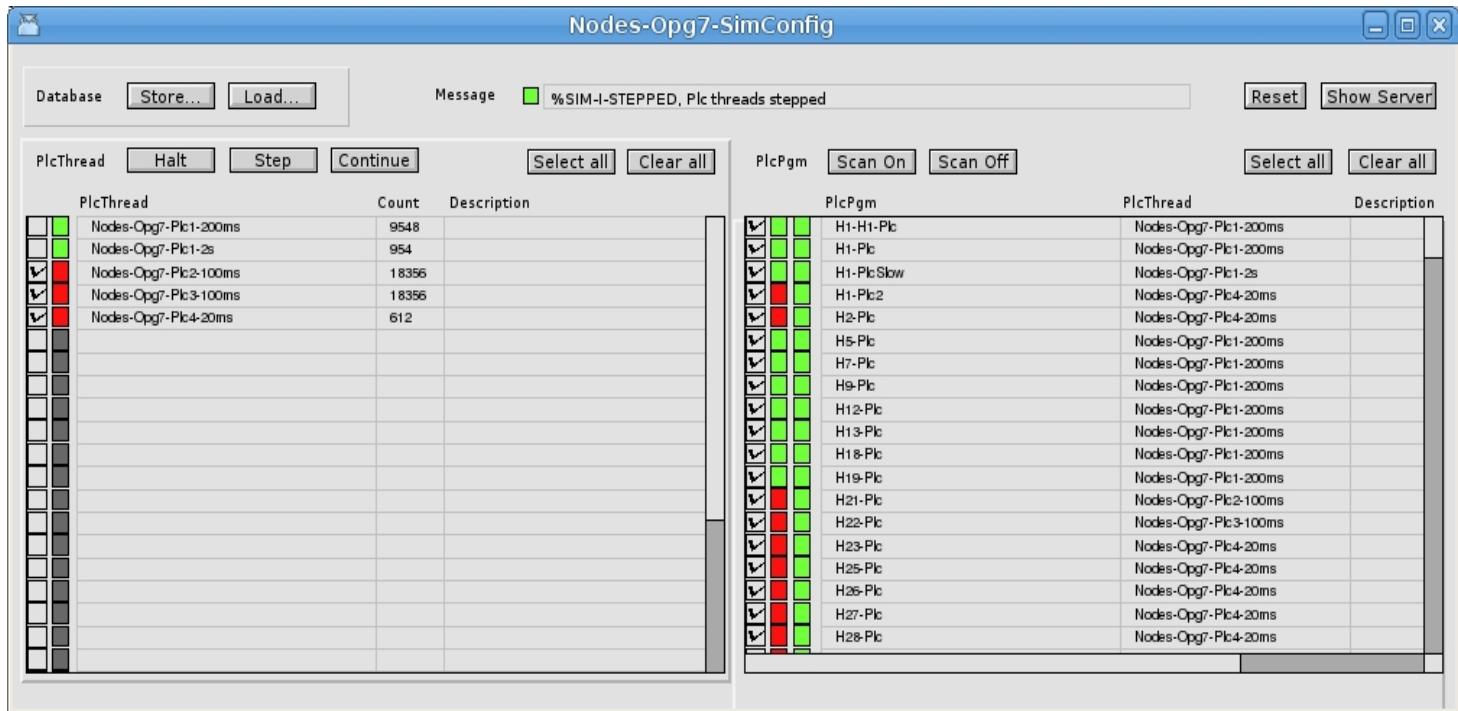
19.3   Runtime Monitor

Often you want to start the runtime environment on the development node, for example if you have made a change in the system that you want to test, before sending it down to the production system.

The Runtime Monitor is used to start and stop the runtime environment on the development station.
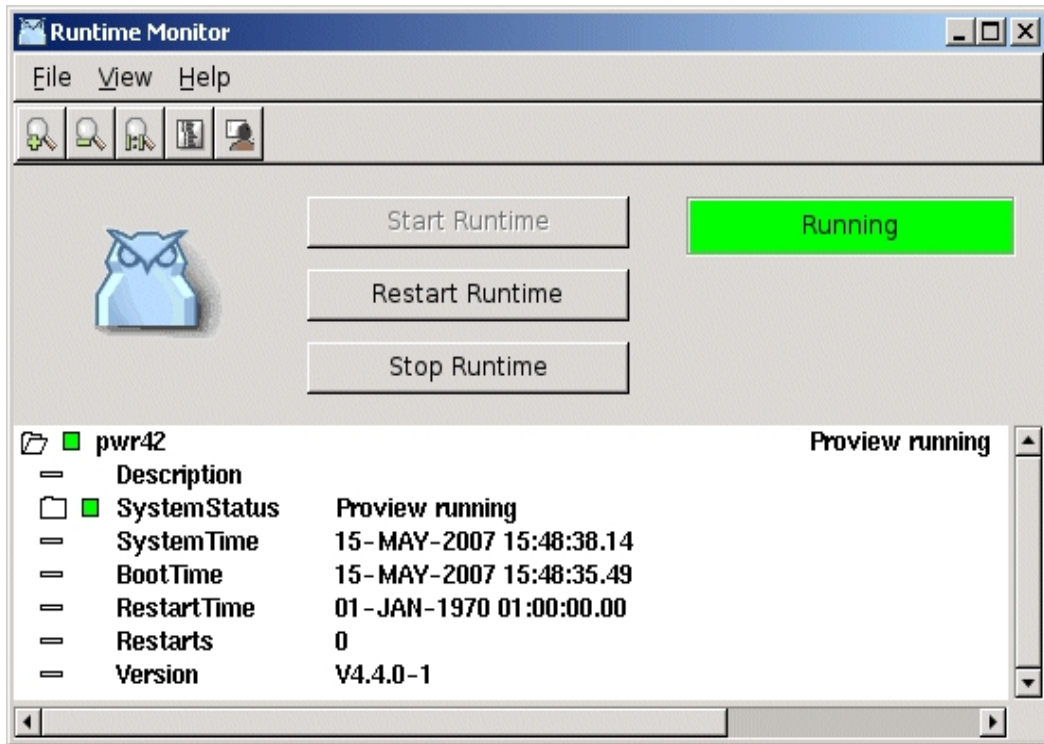
To start the runtime environment on the development station, the following requirements have to be fulfilled

- the node should be configured with a NodeConfig object in the project volume.

- the correct comunication buss should be set. To do this you set the bash environment variable PWR_BUS_ID to the buss stated in the BusConfig object in the project volume, for example
```
export PWR_BUS_ID=999
```

The runtime monitor also requires a StatusServerConfig object to be configured below the $Node object in the volume to start.

The Runtime Monitor is started from Tools/Runtime Monitor in the menu. There are buttons to start and stop the runtime environment ('Start Runtime' and 'Stop Runtime'). In the colored square, the status of the runtime environment is displayed ('Running' or 'Down'). The color indicates the status of the system, red for error status, yellow for warning, and green for OK.

The button 'Restart Runtime' performs a soft restart, and can be used if the runtime is started already.

19.4   Process and operator stations

In this section we go through the steps to start ProviewR on a process or operator station.

- configuration of the node in the directory volume
- build the node
- installation of the runtime package
- distribute to the node

- start the runtime environment on the node

Directory volume configuration

The node has to be configured in the directory volume by a NodeConfig object. This is placed below a BusConfig object that contains the bus identity for the production bus. In the NodeConfig object the node name and IP address are configured, and below the NodeConfig object there is a RootVolumeLoad object that states which root-volume is to be started on the node.

The node is built from the rootvolume configurator by activating 'Functions/Build Node' and selecting the process station in the node list (if there is only one single node configured the node list is not viewed).

Installation of the runtime package

The runtime package is installed on the process/operator node by installing the pwrrt package of the present Linux distribution. Read the installation guide on the Download page on www.proview.se for more information.

19.4.1   Distribute

Distribute, means collecting the files that are created when the node is built, and that is needed to execute the runtime environment, into a package. The package is copied to the process or operator station and unpacked there.

Which files are going to be a part of the package, is configured in the directory volume with a Distribute object beneath the NodeConfig object of the node. The Distribute object contains the attribute Components where you specify what types of components or files that are selected. If there are components specified that are not generated, a warning message will comply with the distribution.
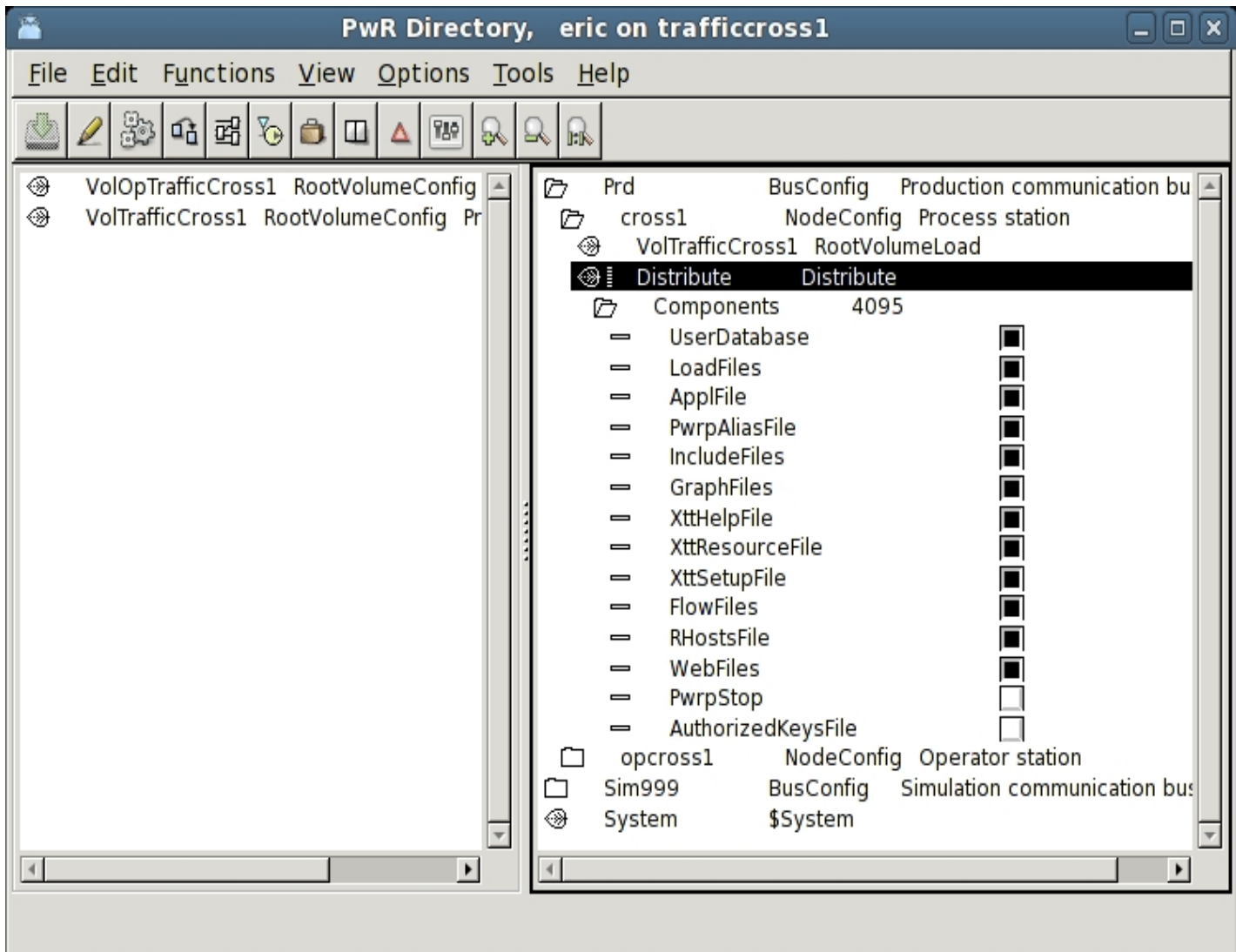
Fig The Distribute object

If there are other files, e.g. application programs, that are to be a part of the package, you add an object of type ApplDistribute below the Distribute object. In the ApplDistribute object you can state which files are to be added (specification with wildcard is allowed) and where they are to be copied.

All files needed at runtime should be a part of the package. It is important that a package represents a complete version of the system, making it possible to restore the runtime environment if you for example want to go back to a previous version. If a disk crash occurs, it is also important to be able to restore the system on a new disk without any manual copying and modifications of files.

The distribution is performed from the Distributor that is opened from 'Functions/Distribute' in the Configurator menu. Select the node you want to distribute to, and select 'Functions/Distribute' in the Distributor menu.
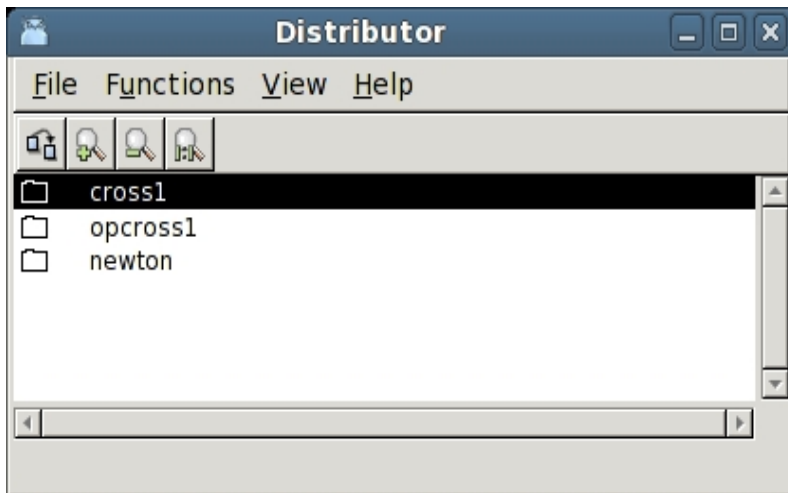
Fig The distributor

The distributor will now collect the specified files into a package and copy the package to the user 'pwrp' on the process node with ssh. ssh requires password, and this has to be typed two times in the terminal window from which the configurator is started. At installation the user 'pwrp' is given the password 'pwrp', but this may have been changed for security reasons.

If you don't have network contact with the process station, you can create a package and move it to the process station for example with an USB stick. Select the node in the Distributor and activate 'Functions/Create Package'. The package is stored on $pwrp_load with the name pwrp_pkg_'nodename'_'version'.tgz, e.g. pwrp_pkg_cross1_0002.tgz. On the process station you unpack the package with the script pwr_pkg.sh -i which takes the name of the package as argument.

```
> pwr_pkg.sh -i pwrp_pkg_cross1_0002.tgz
```

Restore a previous version

Sometimes a modification of the system doesn't work as planned, and you want to restore a previous version. This is easily done with pwr_pkg.sh. The packages are stored in the home directory of user pwrp, /home/pwrp. By finding the package for the previous version and start pwr_pkg.sh -i with this package, you restore the version.

```
> pwr_pkg.sh -i pwrp_pkg_cross1_0001.tgz
```

19.4.2 Bus identity

The QCom bus identity for the node is set in the file /etc/proview.cnf.

```
# Default QCOM Bus Id
qcomBusId 517
```

This value has to correspond to the configured bus id for the node in the directory volume. The current bus id is stored in the environment variable $PWR_BUS_ID.

19.4.3 Start the runtime environment

The runtime environment on a process or operator station is started with the command

```
> pwr start
```

and stopped with

```
> pwr stop
```

or

```
> pwr kill
```

The command 'pwr stop' requires that all processes are alive, while 'pwr kill' clears all in all situations.

If proview runtime doesn't start correctly, you can start with the command

```
> rt_ini -i
```

to view console loggings in the terminal window.
Always reset with 'pwr kill' before a new start attempt.

20 The Configurator

The configurator is used to navigate in, and configure the Workbench.
The configurator displays the objects in one volume. The objects are usually separated in two windows, a left and a right, and how the separation is done depends on what type of volumes are handled.

- For rootvolumes and subvolumes, the plant hierarchy is displayed in the left window, and the node hierarchy in the right.

- For the directory volume, volumes are displayed in the left window and buses and nodes in the right.

- For class volumes, classes are displayed in the left window and types in the right.

From 'View/TwoWindow' you can choose whether to display two windows or only one. If only one window is displayed, every second time you activate 'TwoWindow' the upper window will be displayed, else it will be the lower window.

From 'Edit/Edit mode' you enter edit mode, and a palette with various classes is displayed to the left. You can now create new objects, move objects, change values of attributes etc.

Volume representation

Volumes are stored in various formats, in a database, in a loadfile or in a textfile. The configurator can display a volume in all these formats, and it has four different representations of volumes:

- db, a database. Rootvolumes and subvolumes are created and edited in a database. Before you can start the runtime environment, loadfiles are generated from the volumes. The loadfiles are read at runtime startup. The db representation i editable.

- wbl, a textfile with extension .wb_load. The classvolumes are stored as wbl, and root and sub volumes can be dumped in a wbl-file, for example when upgrading, and later reloaded. The wbl representation is not editable. When editing a class volume you import the wbl representation to a mem-represention, and then save it as wbl again.

- dbs, a loadfile. From rootvolumes, subvolumes and classvolumes, in db and wbl representation, loadfiles are cerated and used in the runtime environment. The configurator also reads the dbs-files of the classvolumes to be able to interpret the classes, and the dbs-files of the root and subvolumes to be able to translate references to external objects. The dbs representation is not editable.

- mem, a volume the configurator keeps internally in memory. Copy/Paste buffers consist of mem-volumes. The classeditor imports the classvolume, which originally is a wbl, to a mem volume, as the mem representation is editable.

As we see above, the same volume can exist both as a database or as a loadfile. When staring the configurator, you specify a volume as an argument. For this volume, the database is opened, i.e. it is represented as a db, for the other volumes in the project, the loadfiles are opened, i.e. they are represented as dbs. This makes it possible to display the other volumes in the project, and to solve references to them, but they are not editable. If the database of the

volume is locked, because someone else has opened it, an error message is displayed and the loadfile is opened instead of the database.

In the figure below the volume list is displayed, which is opened from 'File/Open' in the menu. It shows all volumes opened by the configurator. We can see that the database for the root volume VolTrafficCross1 is opened, while the other root volume, VolOpTrafficCross1 is opened as a loadfile. Also the class volumes are opened as loadfiles.
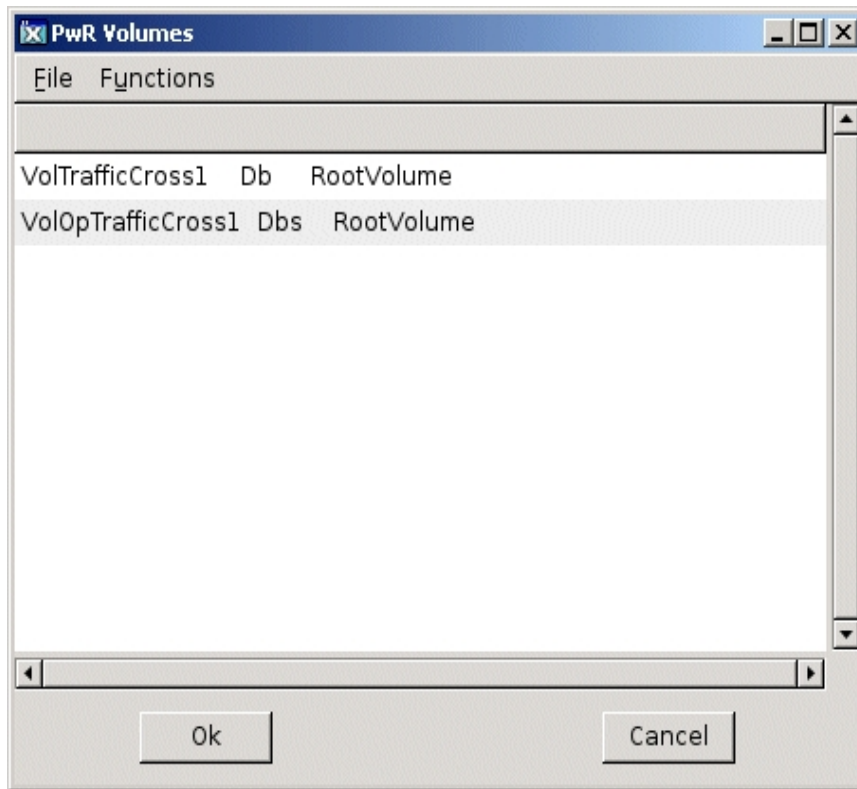
If no volume is given as argument when starting the configurator, the database of the directory volume is opened, and the other volumes are opened as dbs-volumes.

Navigate
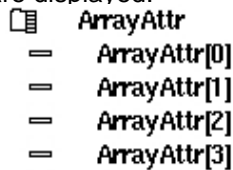
The objects of the current volume are displayed in the configurator. The objects are ordered in a tree structure, and objects with children are displayed with a map, and objects without children with a leaf. For each object the object name, class and possible description is displayed as default(the description is fetched from the Description attribute in the object).

By clicking with MB1 on a map, the map is opened and the children of the object are displayed. Is the map already open, it will be closed. You can also open a map with a doubleclick anywhere in the object row.

If you want to see the content of an object, click with Shift/Click MB1 on the map or leaf, or Shift/Doubleclick MB1 anywhere in the object row. Now the attributes of the object are displayed, together with the value of each attribute. The attributes are marked with various icons dependent of type.

Bitmaps for different types of attributes

```
⊛┊  BitmapDemo
  ▭     OrdinaryAttr
  ▤     ArrayAttr
  ▤     EnumAttr
  ▤     MaskAttr
  →▢    ObjidAttr
  →▭    AttrRefAttr
  ▢     ObjectAttr
```
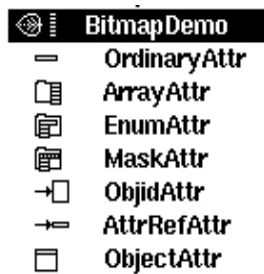
- An ordinary attribut is marked with a long narrow rectangle.

- An array is marked with a map and a pile of attributes. The array is opened with Click MB1
  on the map, or Doubleclick anywhere in the attribute row. Now the elements of the array
  are displayed.

```
  ▤   ArrayAttr
  ▭      ArrayAttr[0]
  ▭      ArrayAttr[1]
  ▭      ArrayAttr[2]
  ▭      ArrayAttr[3]
```

- An attribute referring another attribute or object, i.e. of type Objid or AttrRef, is marked
  with an arrow pointing to a square.

- Enumeration types, Enum, is marked with am map and some long narrow rectangles. By clicking
  MB1 on the map the different alternatives of the enumeration are displayed. The alternatives
  are displayed with check boxes, and the chosen alternative is marked. You can also
  Doubleclick MB1 in the attribute row to display the alternatives.

```
  ▱   EnumAttr          Apple
  ▭      Apple                        ■
  ▭      Orange                       ▢
  ▭      Banana                       ▢
  ▭      Strawberry                   ▢
  ▭      Grape                        ▢
```

- Mask types, Mask, is marked similar to Enum, an the different bits are displayed with Click
  MB1 on the map, or Doubleclick MB1 in the attribute row.

```
  ▱   MaskAttr          10
  ▭      Apple                        ▢
  ▭      Orange                       ■
  ▭      Banana                       ▢
  ▭      Strawberry                   ■
  ▭      Grape                        ▢
```
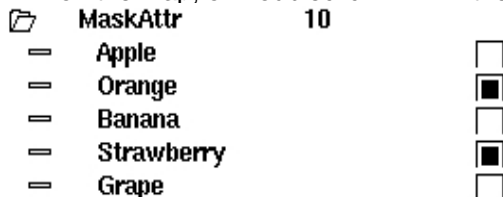
- Attribute objects, i.e. attributes that contain the datastructure of an object, are marked
  with a square with a double line on the upper side. The attribute object is opened with
  Click MB1 on the square, or Doubleclick MB1 in the attribute row.

An object or attribute is selected with Click MB1 in the object/attribute row (not in the map
or leaf). With Shift/MB1 you can select several objects. With Drag MB1 you can also select
several objects.

From an ergonomic point of view, it is often better to navigate from the keyboard. You mainly
use the arrow keys. First you have to set input focus to the window, by clicking on it. Input
focus between the left and right window is shifted with TAB.

With ArrowUp/ArrowDown you select an object. If the object has children, you open the children with ArrowRight, and close with ArrowLeft. The content of the object, i.e the attributes, are displayed with Shift/ArrowRight and closed with ArrowLeft.

An attribute that is an array, enum, mask or attribute object, is opened by ArrowRight and closed by ArrowLeft.

When you feel at home in the object tree, you can set yourself as 'advanced user'. Additional function is the placed in the arrow keys. ArrowRight on an object, displayes for example the attributes of the object, if it has no children. If it has children, you have to use Shift/ArrowRight as before.

Editing

When editing a volume, you create new objects, copy objects, remove objects and change values of attributes.

Create an object

You create an object by selecting the class of the object in the palette. The palette is divided in the folders Plant, Node and AllClasses. Under Plant you find the most common classes in the plant hierarchy, under node the most common in the node hierarchy. If the class is not found here, all the classes are available under AllClasses. Here, all the class volumes are listed, and under each volume, the classes of the volume. After that, you click with the middle mousebutton on the future sibling or parent to the new object. If you click on the map/leaf in the destination object, the new object is placed as the first child, if you click to the right of the map/leaf, it is placed as a sibling.

You can also create an object from the popup menu. Select a class in the palette and open the popup menu by Click MB3 on the destination object. Activate 'Create Object' and choose where to put the new object, relatively the destination, before, after or as first or last child.

The Configurator in edit mode
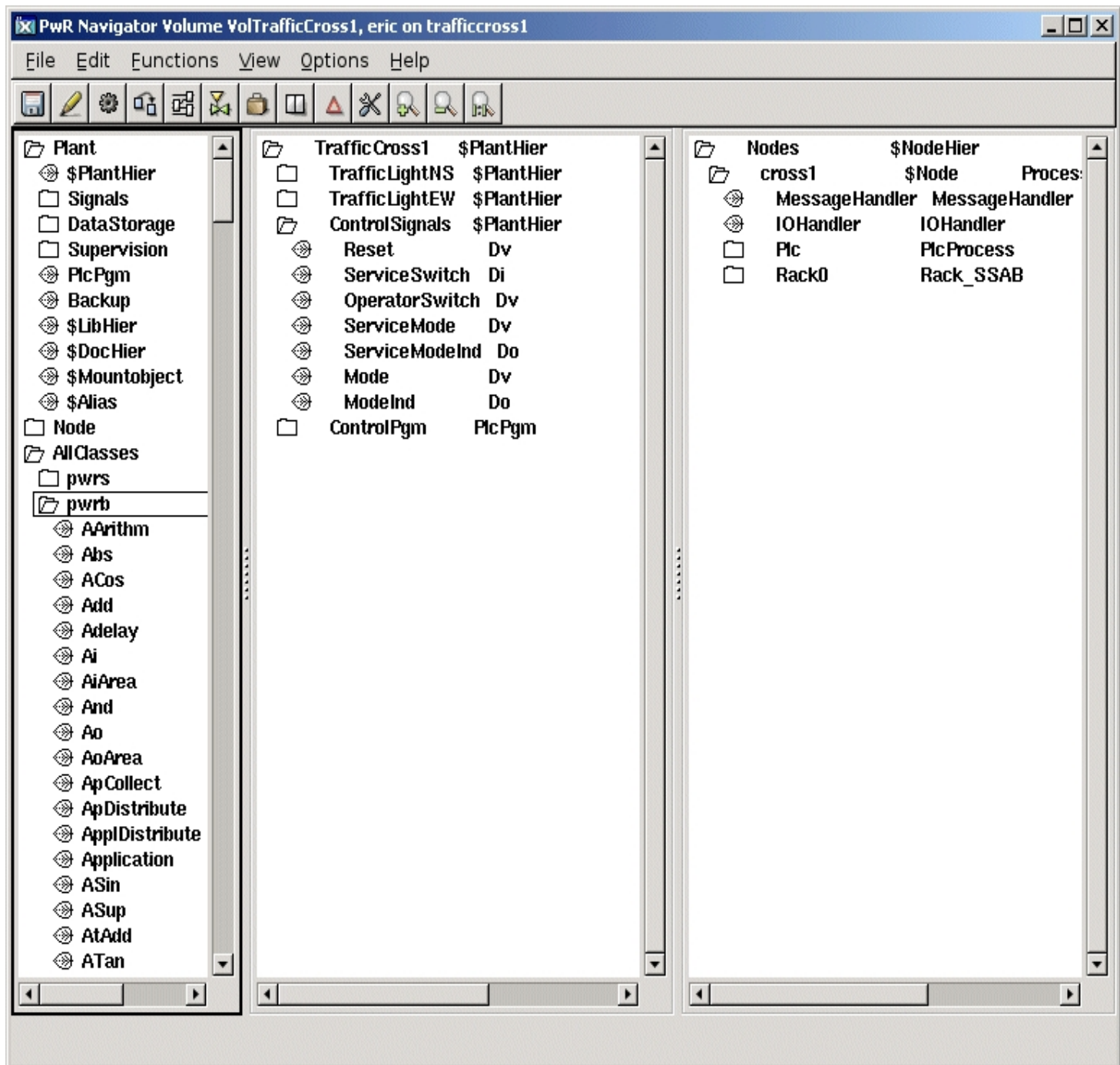
Delete an object

An object is deleted from the popup menu. Click MB3 on the object and activate 'Delete Object'.

Move an object

You can also move an object from the popup menu, but it is often easier to use the middle mouse button: select the object that is to be moved and click with the middle button on the destination object. If you click on the map/leaf on the destination object, the object is placed as first child, else as a sibling.

Note! Avoid using Cut/Paste to move an object. This will create a copy of the object with a new object identity, and references to the object might be lost. You can use the command

paste/keepoid to keep the identity.

Copy an object

You can copy an object with copy/paste or from the popup menu.

- copy/paste. Select the object or objects that are to be copied and activate 'Edit/Copy' (Ctrl/C) in the menu. The selected objects are now copied to a paste buffer. Select a destination object, and activate 'Edit/Paste' (Ctrl/V). The objects in the paste buffer are now placed as siblings to the destination objects. If you instead activate 'Edit/Paste Into' (Shift+Ctrl/V) the new objects are placed as children to the destination object. If the copied objects have children, the children are also copied by copy/paste.

- from the popup menu. Select the object or objects that are to be copied, open the popup menu from the destination object, and activate 'Copy selected object(s)'. You now have to choose where the new objects are to be placed, relative to the destination object, as first or last child, or as next or previous sibling. If the copied objects have ascendants, and they also are to be copied, you activate 'Copy selected Tree(s)' instead.

Change object name

The name of an object is changed by selecting the object, and activating 'Edit/Rename' (Ctrl/N) in the menu. An input field is opened in the lower region of the configurator, where the new name is entered. An object name can have max 31 characters.

You can also change the name by displaying the object attributes. In edit mode, the object name is displayed above the attributes, and is changed in the same way as an attribute.

Change an attribute value

Select the attribute to be changed, and activate 'Functions/Change value' (Ctrl/Q) in the menu. Enter the new value in the input field. If you want to terminate the input, you activate 'Change value' again.

Not all attributes are editable. It depends on the function of the attribute, if it is to be assigned a value in the development environment or not. Editable attributes are marked with an arrow.

You can also change the value of an attribute from the object editor, opened from the popup menu (Open Object). An attribute of type multiline text, can only be edited from the object editor.

As 'advanced user' you can open the input field with 'ArrowRight', as a faster alternative to 'Change value'.

Symbol file

The symbolfile is a command-file that is executed at configurator startup.
It can contain definitions of symbols and other configurator commands.
The default filename of the symbolfile is $pwrp_login/wtt_symbols.pwr_com.
Here are some examples of useful commands.

Shortcut to somewhere in the database hierarchy:

```
define rb9 "show children /name=hql-rb9"
```

20.1   Object Editor

The quantity of data for an object is divided in attributes. The Object Editor displays the
attributes of an object and value of each attribute. If you are in edit mode, you can also
change the values of the attributes.

The attributes are displayed in the same way as in the configurator, the main difference is
that they are displayed in a separate window.

Navigate

Navigation and assignment of values is also done in the same way as in the configurator.

Start

The Object Editor is opened from the Configurator or the Plc editor. Activate 'OpenObject' in
the popup menu for an object, or select the object and activate 'Functions/Open Object' in the
menu. From the Plc editor you can also start the object editor by doubleclickning on the
object. If the Configuration/Plc editor is in edit mode, the Object Editor is also opened in
edit mode.

Menu

File/Close   close the object editor.

Functions/Change value   open the input field for the selected attribute.
                                                 This is only allowed in edit mode.
Functions/Close change value   Close the input field.

Object Editor
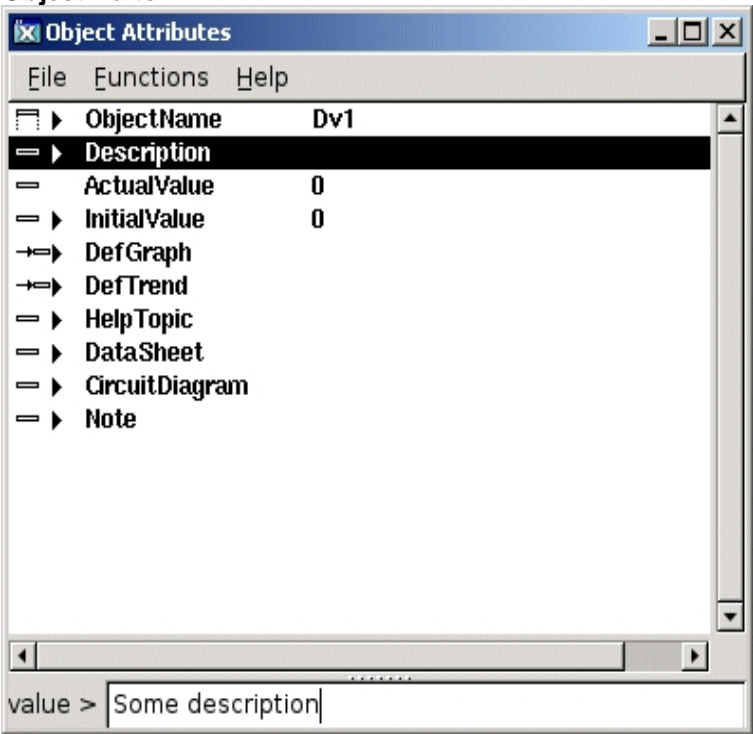
The Object Editor has an input field to enter multiline texts. In this case you can not terminate the input with 'Enter', as for singleline texts. You either click on the 'Ok' button or activate 'Functions/Close change value' (Ctrl/T) to terminate.

A Multiline text
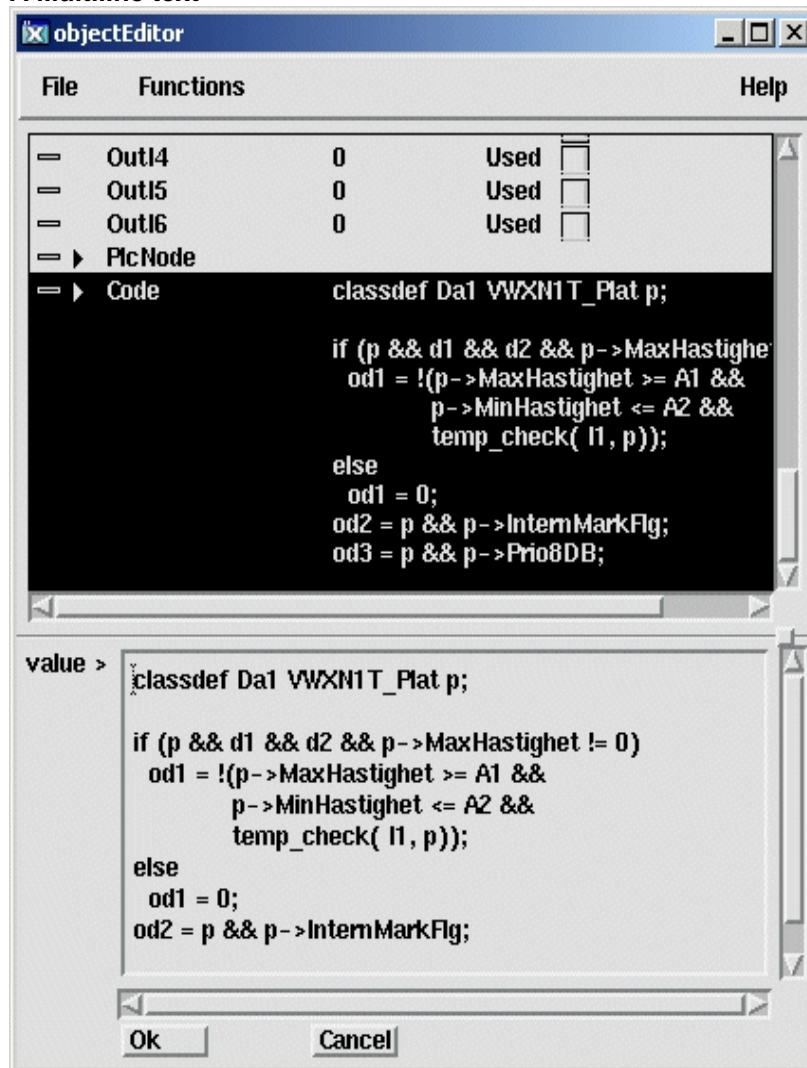
The Object Editor for plc objects has functions to state which inputs or outputs that are to be displayed in the function block. You can also choose if digital inputs are to be inverted. This is chosen with check boxes for each attribute respectively ('Used' and 'Inverted'). The check box for 'Used' can also be changed from the keyboard with 'Shift/ArrowRight', and the check box for 'Inverted' can be changed with 'Shift/ArrowLeft'.

Plc object with check boxes
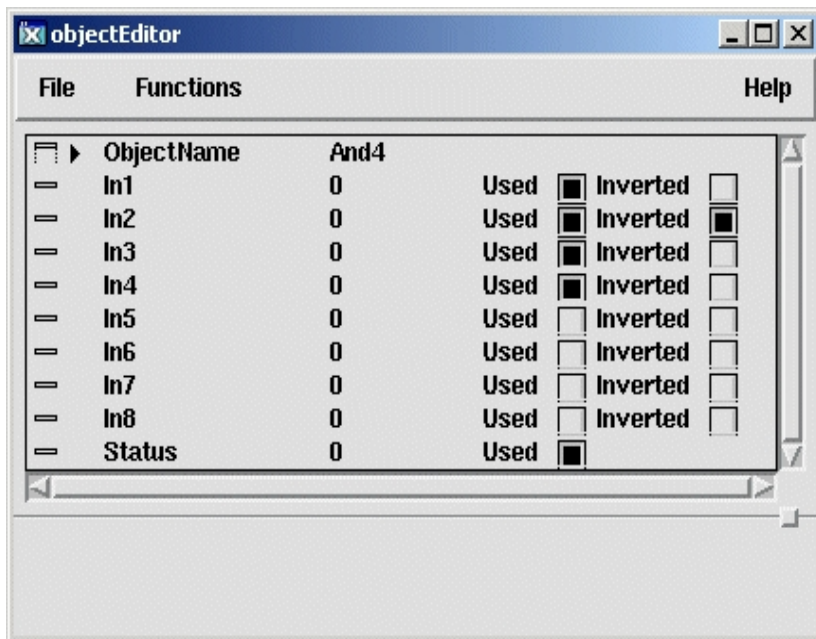
20.2   Object Text Editor

For text attributes in objects as BodyText, HelpText, CArithm and DataArithm a special
text editor is available. The editor is opened by right-clicking on the object in the
plc editor and activating EditText or EditCode in the popup menu.

Menu

File/Close   close the text editor.

File/Save   Save the text to the object attribute. Note that the text is not saved
                                           in the database until the plc editor session is saved.

Edit/Copy   Copy selected text to the clipboard.
Edit/Cut   Cut selected text.
Edit/Paste   Paste text from the clipboard into the current cursor position.

Object Text Editor

20.3   The Spreadsheet Editor

The Spreadsheet editor is used to view or configure several objects of the same class simultaneously. Objects for a certain class, below a specified object in the object tree, are displayed in a table in the editor. In the table also the values of an attribute in the objects are displayed, and you can easily shift between different attributes.

The Spreadsheet Editor is opened from the configurator: 'Functions/Spreadsheet' in the menu. If the configurator is in edit mode, the Spreadsheet Editor is also opened in edit mode.

When the spreadsheet editor is started, you first have to state which objects are to be displayed, i.e. which class they belong to and under which hierarchy the are placed.This is done by activating 'File/Select Class' in the menu. Enter class, hierarchy and state if attribute objects, i.e. objects that reside as attributes in other objects, are to be displayed. It's also possible to select objects by name with a wildcard search pattern.

Choose class and hierarchy
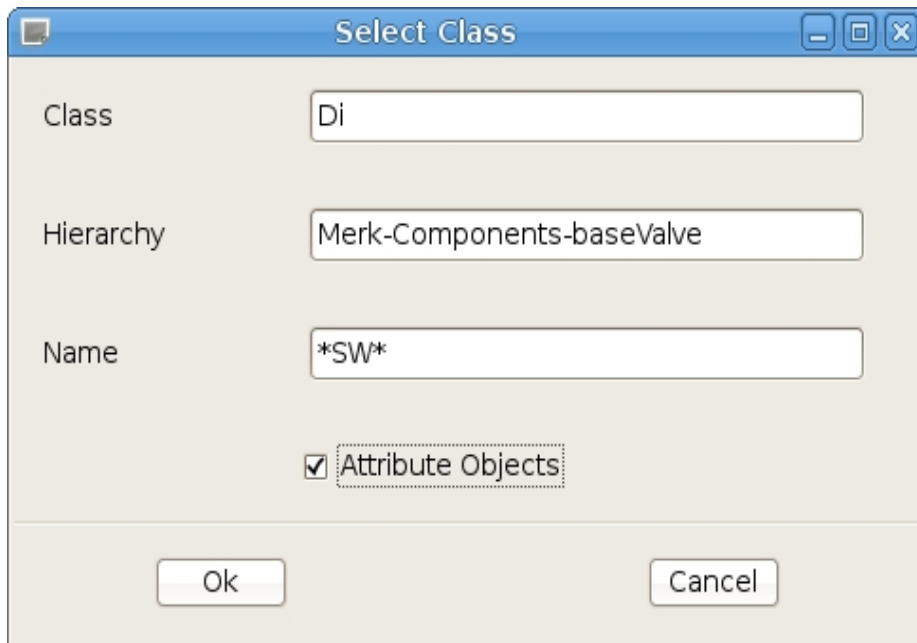
After this you choose which attribute is to be displayed. Select an attribute in the attribute list and click on 'Ok', or doubleclick on an attribute.

Choose attribute
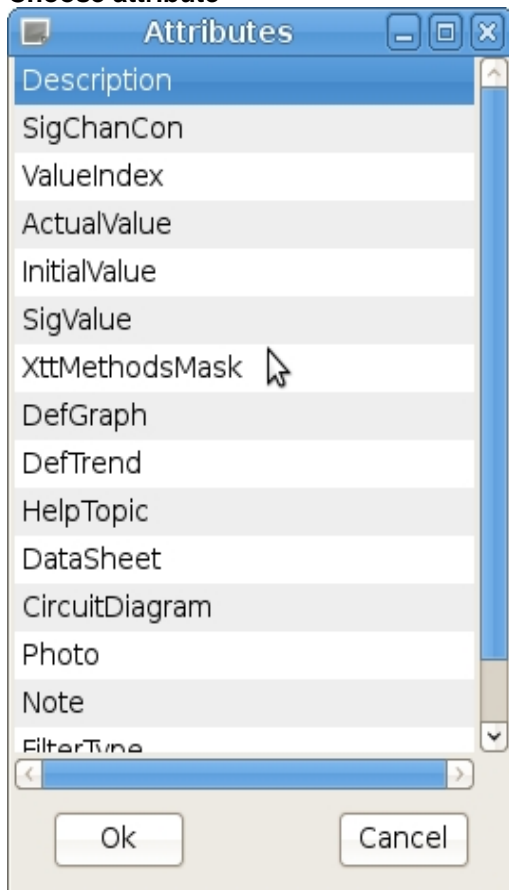
The result is shown in the figure below. Here, the attribute 'Description' was chosen. You can easily view the other attributes in the object by activating 'File/Next Attribute' (Ctrl/N) and 'File/Previous Attribute' in the menu.

Spreadsheet Editor

```
*** Spreadsheet Editor                                    [_][□][X]
File  Functions  Help
━ ▶ Merk-Components-baseValve-V1.SwitchOpen.Description      Open switch
━ ▶ Merk-Components-baseValve-V1.SwitchClosed.Description     Close switch
```
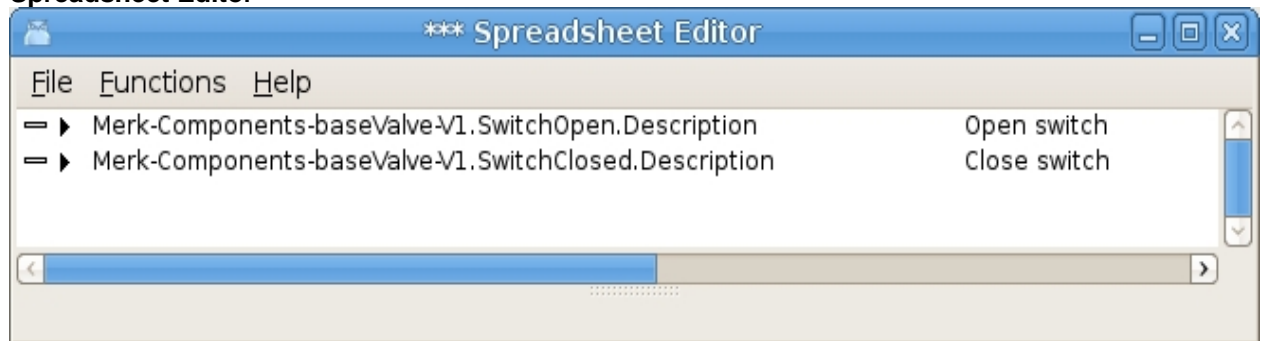
Menu

File/Select Class	State class and hierarchy for the objects that are to be displayed.
File/Select Attribute	State which attribute is to be displayed.
File/Next Attribute	Display the next attribute for the object in the table.
File/Previous Attribute	Display the previous attribute for the objects in the table.
File/Print	Print the table.
File/Close	Close the Spreadsheet Editor.
Functions/Change value	Open an input field for the selected object.
Functions/Close change value	Close the input field.

20.4   Help window

The helpwindow is used to view and navigate in help texts. The help texts can be various manuals and guides that comes with ProviewR, or helptexts written by the constructor to describe the plant and to give assistance to the operators.

20.5   Message window

The message window displays messages from ProviewR that are transmitted at various operations. The messages can have five levels of severity, that are marked with different colors:

S	Success	green
I	Information	green
W	Warning	yellow
E	Error	red
F	Fatal	red

If an arrow is displayed in front of the message, the message contains a link to an object.
By clicking the arrow, the object is displayed.

20.6   Utilities

The utilities window is a graphic interface to different configurator commands.

For more information about the commands, see chapter Commands.

20.7   Backup utility
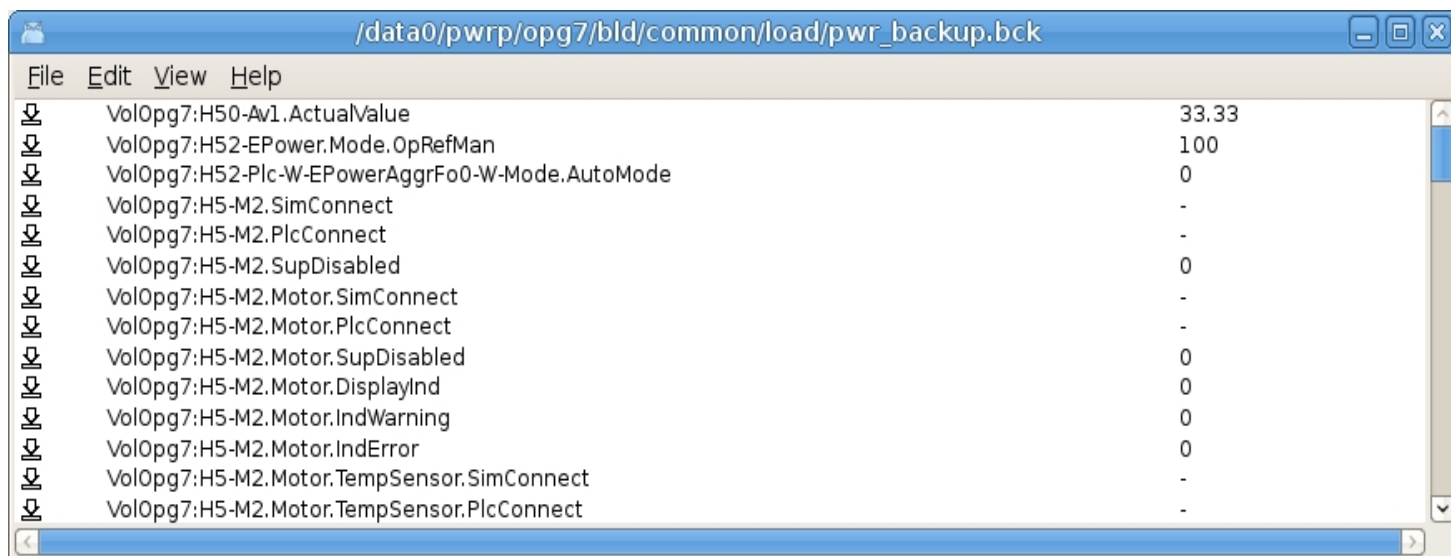
The backup utility analyses a runtime backup file. It makes it possible to

- inspect the content of the backup file.
- see the difference between two backup files stored and different times.
- see the difference between the backupfile and the corresponding values in the
  development database.
- Transfer selected values from the backup file to the development database.

The backup function in the ProviewR runtime enviroment, stores the values of
objects and attributes specified with Backup objects, in the backup file. The
is read at ProviewR startup and the previously stored values are inserted into
the realtime database.

Display the content of a backup file

Open the backup utility from the root volume configurator of the node from which the
backup file is fetched. The backup utility is opened with the command 'backup show'.
Activate File/Open in the menu and select a backup file. The backup file can be a copy
of the current backup file or a copy from a previous occation. All the attributes and
their values are displayed in the backup window.

/data0/pwrp/opg7/bld/common/load/pwr_backup.bck		
File Edit View Help		
⊻	VolOpg7:H50-Av1.ActualValue	33.33
⊻	VolOpg7:H52-EPower.Mode.OpRefMan	100
⊻	VolOpg7:H52-Plc-W-EPowerAggrFo0-W-Mode.AutoMode	0
⊻	VolOpg7:H5-M2.SimConnect	-
⊻	VolOpg7:H5-M2.PlcConnect	-
⊻	VolOpg7:H5-M2.SupDisabled	0
⊻	VolOpg7:H5-M2.Motor.SimConnect	-
⊻	VolOpg7:H5-M2.Motor.PlcConnect	-
⊻	VolOpg7:H5-M2.Motor.SupDisabled	0
⊻	VolOpg7:H5-M2.Motor.DisplayInd	0
⊻	VolOpg7:H5-M2.Motor.IndWarning	0
⊻	VolOpg7:H5-M2.Motor.IndError	0
⊻	VolOpg7:H5-M2.Motor.TempSensor.SimConnect	-
⊻	VolOpg7:H5-M2.Motor.TempSensor.PlcConnect	-

Fig Display backup file

Compare two backup files

Open the first backup file for display as described above. Then activate
File/Compare Backup File in the menu and select the other backup file. The attributes
with values that differs between the files are now displayed.

Compare a backup file with the development database

Open the backup file for display, and then activate File/Compare Database in the menu.
The attributes with values that differs between the backup file and the development
database in now displayed.

Transfer values from a backup file to the development database

Enter edit mode in the configurator and then open the backup utility. Open the backup file for display, and then compare this with the values in the development database from File/Compare Database in the menu. The attributes with values that differs between the backup file and the database are now displayed with checkboxes. Check the values you want to transfer, and activate File/Transfer to database in the menu. Finally activate Save in the configurator to store the modifications.

20.8 Build Directories

The Build Directories window displays the directories with configured build actions. The check box for each directory shows if the directory will be built or not, ie if anything has to be updated in the directory.

Note! If a file is stored after the Build Directories window is opened, the update button has to be pressed to show the correct status. Otherwise the stored file will not be updated when the directory is built.

The directory folders can be opened to view the files, makefiles or scripts that are to be updated. By activating 'Edit/Show all' in the menu also files and makefile that are not to be updated are viewed.

Only directories and files that are checked are executed. Individual directories or files can be built by clearing the check boxes for other directories and files.

By pressing the 'Build Directories' button in the toolbar the build is executed.

The directories and the build actions are configured in the directory volume.

20.9 Build Export and Import

The Build Export window displays files that should be exported to other projects or modules. Common files between projects can for example be dbs-files for class volumes or h-files for transactions between nodes. Normally files are exported from a project to a common directory, from where they later are imported by other projects. It's not recommended to import or export directly from or to other projects.

By pressing the 'Export files' button in the toolbar the export is executed.

The Build Import window displays files that should be imported from other projects and modules.

By pressing the 'Import files' button in the toolbar the import is executed.

21 Plc Editor

In The Plc Editor you create plcprograms in a graphical programming language.

Programming with function block is made in a horizontal net of nodes and connections from left to right in the document. Signals or attributes are fetched on the left side of the net, and the values are transferred via connections from output pins to input pins of functions blocks. The function blocks operate on the values, and on the left side of the net, the values are stored in signals or attributes.

Grafcet sequences consist of a vertical net of nodes and connections. A state is transferred between the steps in the sequence via the connections. Grafcet and function block nets can interact with each other and be combined to one net.

Start

The Plc editor is opened from the configurator. Select an object of class PlcPgm and activate 'Functions/Open Program' (Ctrl/L) in the menu, or activate 'Open Program' in the popupmenu for the PlcPgm object. The configurator should not be in edit mode.

Working mode

The Plc editor can be in four different modes: View, Edit, Trace and Simulate. The mode is selected under 'Mode' in the menu.

View
In View you can look at the program, but not create or modify objects. The menu alternatives for edit functions are dimmed.

Edit
If you have edit privileges you can enter the edit mode. Now it is possible to create and modify objects.

Trace and Simulate
If you want to trace the program you enter the trace mode. This requires that the ProviewR runtime environment is started in the development station. Simulate works as trace, but you can also set values to signals.

Trace is easier and faster performed from Xtt. We recommend that you use PlcTrace in Xtt instead.

Editing

The Plc editor consist of

- a working area.
- tow palettes, one for function objects and one for connections (only one palette at a time is visible).
- a navigation window, from which the working area can be scrolled and zoomed.

The Plc editor
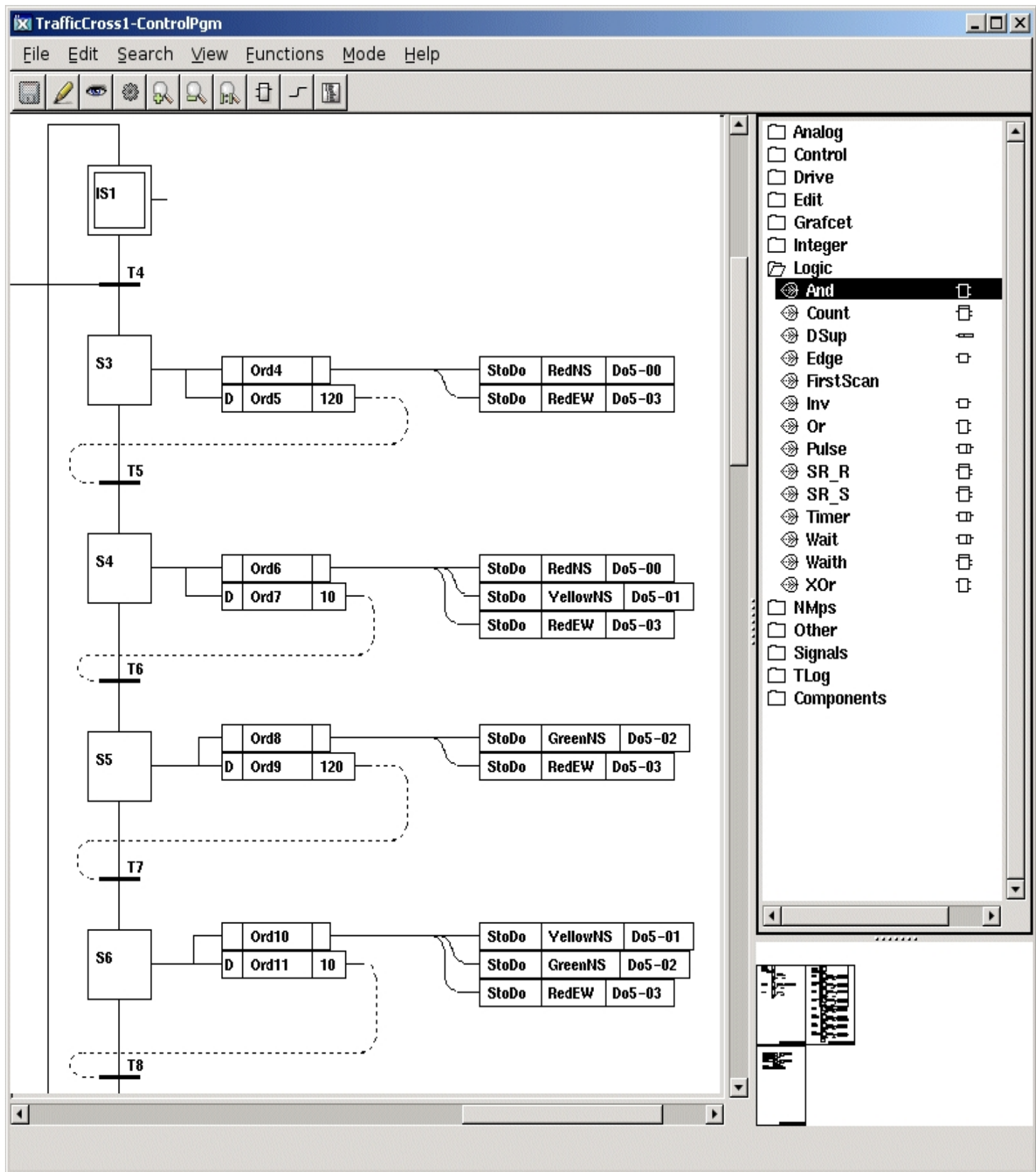
The Palettes

The Object Palette
When you start the Plc editor, the function object palette is displayed. When creating a
function block in the work area, you choose a class in the palette.

The Connection Palette

When you create connections between objects, the editor chooses a suitable type of connection. Though, in some cases the constructor has to influence the choice of connection type. This is done in the connections palette that is displayed by activating 'View/Palette/Connection' in the menu. When the palette is closed, by activating 'View/Palette/Object' or 'View/Palette/Plant', the editor is again responsible for choice of connection type.

Plant Hierarchy

You can view the plant hierarchy by activating 'View/Palette/Plant' in the menu. When connecting function objects to signals, for example when fetching signal values, it is possible to indicate which signal is to be fetched. You can also select the signal in the configurator, which in many cases is a smoother alternative.

Navigation window

Down to the left there is a view of the program in reduced scale. The part of the working area that is displayed in the main window, is marked with a rectangle. By moving the rectangle (Drag MB1) you scroll the main window. You can also zoom with Drag MB2.

Function objects

Create object

To create objects the editor has to be in edit mode. Enter edit mode from 'Mode/Edit' in the menu.

To create an object, you select a class in the palette, and click with MB2 (the middle button) in the working area.

Modify an object

An object is created with certain default values. This applies also to which inputs and ouputs are viewed in the plc editor and can be connected to other objects. If a value is to be changed the object editor is opened for the object. The object editor is opened in following ways:

- doubleclick on the object
- activate 'Open Object' in the popup menu for the object.
- select the object and activate 'Functions/Open object' in the menu.

From the object editor you can change the values of various attributes. The attributes for a plc object are separated into input attributes, internal attributes and output attributes.

Inputs

The value of an input attribute is fetched from another function block, via a connection. The attribute is displayed in the function block as an input pin. In some cases the input is not used, an and-gate has for example 8 inputs but often only two of them are used. This is controlled by the 'Used' check box in the object editor. If 'Used' is marked, the attributes are displayed with an input pin, else they are hidden.

Some input attributes, especially of analog type, can be assigned a value in the object editor. If 'Used' isn't marked for the attribute, the assigned value is used. However, if 'Used' is marked the value is fetched from the output the attribute is connected to. This is for example the function for the limit values 'Min' and 'Max' in a Limit object. You can choose whether to fetch the value from another function block, or to assign a value. The assignment

works in runtime as an initial value, that later can be modified in various ways.

Some digital inputs can be inverted. To do this you mark the check box 'Inverted' in the object editor. In the function block this is displayed with a circle on the input pin.

Internal attributes
Internal attributes can contain configuration values that are assigned in the development environment, or values that are calculated in runtime. The latter type is not changeable, and maybe not even visible in the development environment.

Outputs
The value of an output attribute is transferred to an input via a connection. As for an input, you can choose whether to display an output pin or not with the 'Used' check box in the object editor.

Select an object

Objects are selected in the following ways

- click with MB1 on the object.
- Shift/Click MB1 adds the object to the list of selected objects, or removes it if the object already is selected.
- by Drag MB1 you can select one or several objects. Objects that have some part within the marked rectangle are selected.
- by pressing the Shift key and Drag MB1 you add the objects in the marked rectangle to the selectlist.

Selected objects are drawn with red color.

Move objects

A single object is moved by placing the cursor on it and drag with MB1.
Several objects are moved by selecting them and dragging one of the objects with MB1.

Connections

Create connections

An output pin and an input pin is connected in the following way

- place the cursor on the pin, or in an area in the function object close to the pin, and push MB2 (the middle button).
- drag the cursor to the other pin. or to an area in the function object close to the pin, and release MB2.

A connection is now created between the objects.

Two inputs are connected in the same way, but some of the connected inputs have to be connected to an output, and from this output the value is fetched to all the connected inputs.

Data types
The values that are transferred between different objects via the connections can be digital, analog, integer or string values. Inputs and outputs that are connected have to be of the same type. If they are of different type you have to use an object that converts between the types, e.g AtoI or ItoA. These conversion objects are found under 'Signals/Conversion' in the palette.

Analog and integer connections are marked with slightly thicker lines, and digital connections with thinner lines.

Furthermore there is a connectiontype for transfer of an object reference. These are drawn with a thick, dashed line.

Reference connections
If the editor has difficulties to find a path for the connection between the input and output pin, because there are too many objects in the way, or because they reside in different documents, the connections are drawn as reference connections. Reference connections can also be drawn by activating 'View/Reference connection' in the menu.

Reference connection
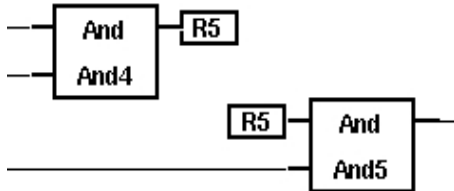
Execute order

Besides transferring a signal value, the connections also determine the execute order between different function blocks. If two objects are connected trough an output and an input, normally the output-object is to be executed before the input-object. But sometimes a feedback is needed in the net, and then you face an execute order loop. To determine the execute order you have to specify the feedback with a connection of type ConFeedbackDigital or ConFeedbackAnalog. These are selected in the connection palette, viewed by activating 'View/Palette/Connection' in the menu. Under the folder 'ConPlc' you can find the feedback connections. They are drawn with dashed lines.

Feedback connection
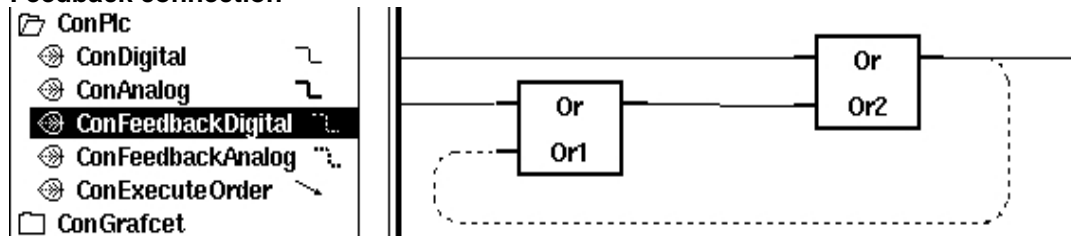
Here you can also find the connection type 'ConExecuteOrder'. In some cases you want to control the execute order between to function blocks, though they are not connected to each other. Then you can draw a ConExecuteOrder between them (between which input or output doesn't matter). The connection is to be drawn from the object that is to execute first, to the object that is to execute last. In the figure below, the storage of the attribute 'Temperaturer' is done before the storage of the attribute 'Tjocklek'.

Execute order connection
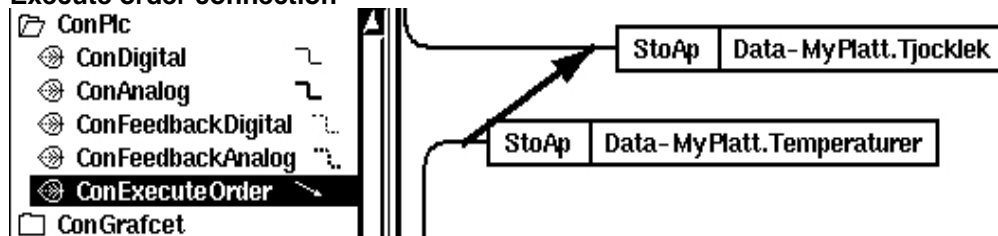
Fetch and store signal values

Fetch signal and attribute values

In the left side of the net of function blocks, values of signals and attributes are fetched.
The fetching is performed by objects as GetDi, GetDo, GetDv, GetIi etc. Fetching of attribute
values is performed by GetDp, GetIp, GetAp and GetSp. These objects you find under the folder
'Signals' in the palette. When an object of this type is created, you have to state which
signal, or which attribute that is to be fetched. The easiest way to do this, is to select the
signal/attribute in the configurator, and click with Ctrl/Doubleclick MB1 on the object.
The signal/attribute is then displayed in the function block, and if the signal is an input
signal, the channel of the signal is also displayed.

There is a faster way to create these objects. If you draw a connection from an input pin in
a function object, and release it in an empty space in the working area, a generic Get object
is created with the datatype of the input, i.e. a GetDgeneric, a GetIgeneric, a GetAgeneric
or a GetSgeneric. When you specify the signal or attribute, the Get object is to fetch, the
generic Get object is converted to a Get object of the the correct type for the signal or
attrbute. If you choose a Dv in the configurator, a GetDgeneric will be converted to a GetDv
when clicking with Ctrl/Doubleclick MB1 on it.

Store signal and attribute values

In the right side of the net calculated values are stored in signals and attributes. The
storage is performed by objects as StoDo, StoDv, StoDp, StoIo etc. The method to specify the
signal or attribute to connect is the same as for Get objects, i.e. by selecting the
signal/attribute in the configurator and click with Ctrl/Doubleclick MB1 on the object.

If you draw a connection from an output pin in a function block, a generic Sto object is
created, that is converted to a Sto object of suitable type when connected to a signal or
attribute. If you want to store values with Set or Reset (for example SetDo or ResDo), you
can't use this method. You have to create the objects from the palette.

Generic Get and Sto objects
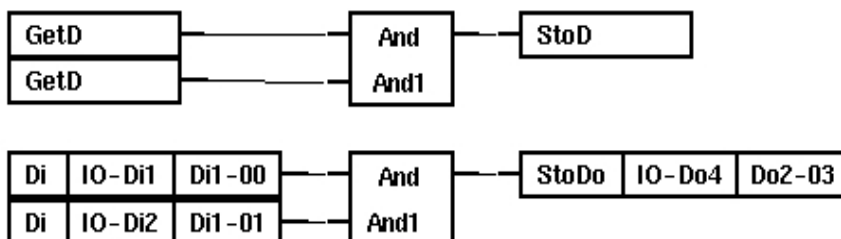
Subwindows

Some objects contain subwindow, e.g. CSub, SubStep, Trans, Order. An object with a subwindow
is marked with a thick gray line somewhere in the function block. A subwindow is opened in
different ways:
- by selecting the object and activate 'Function/Subwindow' in the menu.
- by activating 'Subwindow' in the popup menu for the object.
- by clicking on the object with Shift/Doubleclick MB1.

You create a new subwindow in the following way (the fact that only one editing session can

be open at a time, makes it a bit complicated)

- create the object that is going to contain the subwindow
- save
- open the subwindow
- leave edit mode in the main window
- enter edit mode in the subwindow

Control the execute order

You normally don't have to consider the execute order of different function blocks in a window. As signals are I/O copied, i.e. every timebase in the plc program, makes a copy of all signal values before the execution that is not changed throughout the execution. The storing and fetching of signal values will not be affected by the execute order between individual storing or fetching objects.

However, if you store and fetch the value of an attribute, that is not I/O copied, the execute order can be of importance for the function.

The execute order is determined by the connections between the function blocks. The common connections are both signal transferring and executeorder determining. If you make a feedback you then have to choose a connectiontype that is signal transferring, but not executeorder determining. The different feedback connections are of this type. Furthermore there is a connection that is executeorder determining but node signal transferring, ConExecuteOrder. With this you can control the execute order between different function blocks without transferring any signals values.

The execute order for the function blocks in a plc window is displayed with 'View/Show execute order' in the menu. The number displayed for each function block states the order in which they are executed. The object without a number doesn't have any executable code.

The execute order between different PlcPgm is controlled by the attribute ExecuteOrder in the PlcPgm object. Exectute order determines the order within a thread. Lower values of ExecuteOrder are executed before higher.

Compile

Before a plc window can be executed, is has to be compiled. At the same time, a syntax control of the plc code is performed. If the syntax in not correct, a message is displayed in the message window. The error message can be of type Error or Warning. Error is a more severe error that has to be attended to. By clicking on the arrow in front of the message in the message window, the erroneous object is displayed in the plc editor.

After the syntax check, c-code is generated and sent to the c compiler. If there is an object with user defined c-code, e.g CArithm or DataArithm, the c compiler can find errors that are written in the terminal window. Always look in the terminal window to check that the compilation succeeded.

The compile is executed from 'File/Build' in the menu.

If you want to check the syntax without generating any code, you activate 'File/Syntax'. The c compiler is not activated, thus possible c code errors are not detected.

Cut and Paste

The plc editor contains a paste buffer. The paste buffer is common for all windows, which makes it possible to copy between different windows. With the functions 'Edit/Copy' and 'Edit/Cut' in the menu, the selected objects are copied to the paste buffer (Cut also removes them from the working area). The function 'Edit/Paste' copies the paste buffer to the working area. The copied objects are now moved with the cursor, and you place them on the correct position by clicking MB1 to lock them.

Cut, Copy and Paste can also be activated from the keyboard with Ctrl/X, Ctrl/C and Ctrl/V.

Special Plc objects

Here a number of objects that have special functions in the plc program are described.

Document
The document object is used to divide the code in pages, when printing the code. When you open a new window, it contains a document object. From the object editor you can change the dimension of the document, and enter signature and page number. Other information in the document header is filled in automatically. The document object is found under the folder 'Edit' in the object palette.

ShowPlcAttr
ShowPlcAttr can be used as an extension of the document header. In the object is displayed information about volume, scantime and reset object for Grafcet sequences.

Head, Title, Text and BodyText
These objects are used to write informational text in the document. Head, Title and Text contains singleline texts of different size with max 79 characters. Bodytext contains a multiline text with max 1023 characters. The objects are found under 'Edit' in the palette.

Point
The point object is a free connection point that is used to branch a connection or to control the layout of a connection. Point is found under 'Edit' in the menu.

Grafcet
Grafcet sequences are built with specific Grafcet objects as InitStep, Step, Trans and Order. The connections between the objects follow specific rules. The vertical pins in a Step object are for example connected to Trans objects, and the horizontal pin is connected to an order object. Here is an example of how to create a Grafcet sequence.

Start by creating an InitStop object. Draw a connection from the lower pin, and release it in the working area below the InitStep object. Now a Trans object is created, that is connected to the InitStep object. Draw a connection from the lower pin of the Trans object and release it in the working space below the Trans object. A Step object is now created there. If you draw a connection from the Step objects lower pin, another Trans object is created. If you want a branch of the sequence, you draw an additional connection from the lower pin of the Step object. Now a step divergence is created with specific StepDiv connections. If you in the same manner create a branch from a Trans object, by drawing to two connections from the lower pin, a parallel branch, with TransDiv connections marked with double lines is created. If you draw a connection from the horizontal pin of a Step an Order object is created, and so on. As you can see this is a fast way to build complex sequences.

ScanTime
ScanTime feches the actual scantime, i.e. the time since the last lap.

FirstScan

FirstScan is true the first lap of the plc execution after ProviewR startup. It is also true after a soft restart.

Menu

File/Save	Save
File/Print/Documents	Print all documents
File/Print/Overview	Print an overview
File/Print/Selected documents	Print selected documents
File/Syntax	Perform a syntax check of the code
File/Build	Compile the program
File/Plc Attributes	Open the Object editor for the PlcPgm object
File/Delete Window	Delete the plc window
File/Save Trace	Save trace objects
File/Restore Trace	Restore previously saved traceobjects
File/Close	Close the window
Edit/Undo Delete	Undo the last delete action
Edit/Undo Select	Reset the select list
Edit/Cut	Cut selected objects
Edit/Copy	Copy selected object to the paste buffer
Edit/Paste	Copy the paste buffer to the work area
Edit/Connect	Connect selected object to the selected signal or attribute in the configurator
Edit/Delete	Delete selected objects
Edit/Change Text	Change text in the selected text object
Edit/Expand Object	Expand the selected object
Edit/Compress Object	Compress the selected object
Search/Object	Search for an object name
Search/String	Search for a string
Search/Next	Search further with the same string
View/Palette/Object	Display the functions object palette
View/Palette/Connection	Display the connection palette
View/Palette/Plant	Display the plant hierarchy
View/Reference connections	Create connections as reference connections
View/Grid Size	Set grid size
View/Show Grid	Show the grid
View/Zoom/In	Zoom in
View/Zoom/Out	Zoom out
View/Zoom/Reset	Reset to original zoom factor
View/Show Execute Order	Show execute order for the functions objects
View/Redraw	Redraw connections and redraw the window
Functions/Open Object	Open the object editor for the selected object
Functions/Subwindow	Open the subwindow for the selected object
Mode/View	View mode
Mode/Edit	Edit mode
Mode/Trace	Trace mode
Mode/Simulate	Simulate mode

Mouse functions

Working area

Click MB1	Select an object. Click in an empty space will reset the select list
Shift/Click MB1	Add object to the select list
DoubleClick MB1	Open object editor
Shift+Ctrl/DoubleClick MB1	Copy to past buffer. Click in an object copies the object, click in empty space copies selected objects
Drag MB1	On an object: move object or move selected objects In empty space: select objects inside the marked rectangle
Shift/Drag MB1	Add objects inside the marked rectangle to the select list
Click MB2	Create object
DoubleClick MB2	Delete. Click in object deletes the object, Click in empty space deletes all the selected objects
Shift+Ctrl/Click MB2	Paste. Copy the paste buffer to the working area
Shift+Ctrl/DoubleClick MB2	Cut. Click in an object deletes the object, click in an empty space deletes selected objects. Deletet object are put in the paste buffer
Press MB3	Popup menu

Navigation window

Drag MB1	Scroll working area
Drag MB2	Zoom working area

22   Helpfile

Helptexts are displayed in the help window that can be opened from the configurator and the operator environment. Helptexts are written in a file $pwrp_exe/xtt_help.dat. The helptexts are divided in topics, and each topic has a key, that is specified when the help text for the topic is to be displayed. Links in the helptext that point to other topics, makes it possible to navigate in the helptexts.

The topic 'index' is the root topic that is displayed from different utilities

- 'Help/Project' in the configurator menu.
- 'Help/Project' in the runtime navigator.
- The 'Help' button in the operator window.

Specific help topics can be opened from Ge graphs by buttons (actiontype Help), or from the popup menu for an object in the operator environment (method 'Help').

22.1   Conversion

The helptext can be converted to html, PDF and PostScript format. When converted to html, each topic is converted to one html page. When converted to PDF and PoscScript, there are a number of additional tags available, to create a document of the helptext with chapters and headers.

The conversion is done by 'co_convert'.

Conversion to html

A helpfile is converted to html with the command

```
co_convert -f [-d outputdirectory] 'helpfile'
```

Example
```
co_convert -f -d $pwrp_web $pwrp_exe/xtt_help.dat
```

Conversion to postscript

A helpfile is converted to PostScript with the command

```
co_convert -n [-d outputdirectory] 'helpfile'
```

Example
```
co_convert -n -d $pwrp_lis $pwrp_exe/xtt_help.dat
```

Conversion to PDF

A helpfile is converted to PDF with the command

```
co_convert -f [-d outputdirectory] 'helpfile'
```

Example
```
co_convert -f -d $pwrp_lis $pwrp_exe/xtt_help.dat
```

22.2 Encoding

The default encoding of the help file is ISO 8859-1. UTF-8 can be specified with a Coding statement on the first line of the help-file, eg

```
Coding:UTF-8
```

Supported values for Coding are UTF-8 and ISO8859-1.

22.3 Syntax

There are a number of different tags that influence the search
and the conversion of the helpfile.

topic	Defines the helptext for a topic
bookmark	Defines a position inside a topic
link	Link to a topic or an URL
index	List of topics
h1	Header 1
h2	Header 2
b	Bold text
c	Code
t	Tab
hr	Horizontal line
include	Include other helpfiles

PDF and PostScript tags

The following tags are used to format the helptexts when converted to PDF and PostScript

chapter	Divide topics in chapters
headerlevel	Increase or decrease header level
pagebreak	New page
option	Options
style	Specific text style
Titlepage and document info	

Example

22.3.1 Topic

<topic>
<topic> begin a topic and should be placed in the first position
of a line. The topic-tag should be followed by the key that
the help function will search for. All the following lines until
 a </topic> tag will be displayed as text for the topic.

<topic> 'key'

</topic>

End a topic. </topic> should be placed in the first position
of a line.

Example
<topic> start engine
The engine will be started by...
</topic>

The command

```
wtt> help start engine
```

will display the text of this topic.

22.3.2   Bookmark

<bookmark>
Bookmark is a line inside a topic which can be found by a link-tag or the /bookmark qualifier
in the help command. The bookmark tag should be placed at the end of the line and should be
followed by a name.

'some text' <bookmark> 'name'

Example
This is a bookmark. <bookmark> first_engine

The command

```
wtt> help start engine/bookmark=first_engine
```

will display the text of the topic and scroll to the bookmark.

22.3.3   Link

<link>
The <link> tag is a link to another help topic. The <link> tag should be placed at the
end of the line. When the line of the link is activated the topic of the link will be
displayed. The link tag should be followed by the topic, and can also be followed by a
bookmark and the helpfile where the topic resides, separated by comma. If a line contains
a link, it will be marked with an arrow.

'some text'   <link> 'topic'[,'bookmark'][,'helpfile']

Example
Link to first engine <link> show engine, first_engine

22.3.4   Index

<index>
The <index> tag is a special link that will display an index of
the helpfile, that is a list of all the topics in alphabetical order.

'some text' <index>

22.3.5   Header1

<h1>
The <h1> tag will display a line as a header with larger text size.
The tag should be placed at the beginning of the line. A header line

can't contain any links.

<h1>'header text'

Example
<h1>This is a h1 header
will be displayed as

This is a h1 header

22.3.6 Header2

<h2>
The <h2> tag will display a line as a header with bold text surrounded
by gray lines. The tag should be placed at the beginning of the line.
A header line can't contain any links.

Example
<h2>This is a h2 header
will be displayed as

This is a h2 header

22.3.7 Bold

<b>
The <b> tag will display a line with bold text.
The tag should be placed at the beginning of the line.

Example
<b>This is a bold line
will be displayed as
This is a bold line

22.3.8 Code

<c>
The <c> tag will display a line with the code font Courier.
The tag should be placed at the beginning of the line.

Example
<c>for (i = 0; i < 10; i++)
will be displayed as
```
for ( i = 0; i < 10; i++)
```

22.3.9 Tab

<t>
The <t> tag makes it possible to write columns. Only three columns
(two <t> tags) are allowed.

Example
Col1 <t> Col2 <t> Col3
will be displayed as

| Col1 | Col2 | Col3 |

22.3.10 Horizontal line

<hr>
The <hr> tag will display a horizontal line.

The tag should be placed at the beginning of the line.

Example
<hr>
will be displayed as

22.3.11 Include

<include>
Includes another helpfile. The <include> tag should not be placed
inside a topic.

<include> 'filename'

22.3.12 Chapter

<chapter>
This tag divides the topics in chapters. A chapter begins with <chapter> and ends with
</chapter>. The title of the first topic in the chapter will be the header of the chapter.

</chapter>
Ends a chapter.

Example
<chapter>
<topic>
Introduction
...
</topic>
</chapter>

22.3.13 Headerlevel

Divides the topics in a chapter in header levels.

<headerlevel>
Increases the header level

</headerlevel>
Decreases the headerlevel

22.3.14 Pagebreak

<pagebreak>
Forces a pagebreak

22.3.15 Option

<option>
Option can have the values

printdisable   Ignore the tags and text until the next 'printenable' in PDF
                      and PostScript files. Normally used for links that have no effect
                      in PDF and PostScript.
printenable   Reset the 'printdisable'.

Example
<option> disable

Some text
...
<option> enable

22.3.16 Style

<style>
Specifies that a topic should be written in a specific style.
Styles

function	Style used for functions and commands. Large title and pagebreak after each topic.

Example
<topic> MyFunction <style> function
...
</topic>

22.3.17 Title page and document info

The title page and page for document info can be created with two special topics.

__DocumentTitlePage

Topic for title page. Is place first in a help file.

<topic> __DocumentTitlePage
...
</topic>

__DocumentInfoPage

Topic for document information, eg copyright. Is placed after the titlepage.

<topic> __DocumentInfoPage
...
</topic>

22.3.18 Helpfile example

<topic> helpfile_example
Start and stop of engines.

Engine 1 <link> helpfile_example, bm_engine_1
Engine 2 <link> helpfile_example, bm_engine_2
Characteristics <link> helpfile_example, bm_char

<h1>Engine 1 <bookmark> bm_engine_1
Start engine one by pressing the start button.
Stop engine one by pressing the stop button.

<h1>Engine 2 <bookmark> bm_engine_2
Start engine two by pressing the start button.
Stop engine two by pressing the stop button.

<h2>Characteristics <bookmark> bm_char

<b><t>Engine1 <t>Engine2
Max speed <t> 3200 <t> 5400
Max current <t> 130 <t> 120

</topic>

This is the outlook of this example

22.3.18.1 Start and stop of engines.

Engine 1
Engine 2
Characteristics

Engine 1

Start engine one by pressing the start button.
Stop engine one by pressing the stop button.

Engine 2

Start engine two by pressing the start button.
Stop engine two by pressing the stop button.

Characteristics

	Engine1	**Engine2**
Max speed	3200	5400
Max current	130	120

23 Users

This chapter describes how to create a user in ProviewR, and how to grant privileges and access for the user.

The increasing availability of ProviewR system for different type of users, for example via the intranet, has resulted in increasing demands of possibilities to limit the possibility for various users to influence the system. ProviewR contains a user database, where you define the users for different systems, and where you have the possibility to group systems with common users. The database is designed to face the demands of increasing access control, and at the same time to keep the administration on a low level.

23.1 User database

The user database is populated by system groups and users. When a ProviewR utility is started, for example the opeator or development environment, there is a check that the user exists in the database, and the privileges of the user are registred. The privileges determine what a user is allowed to do in the system.

Systemgroup

The concept systemgroup is introduced not to have to define every system in the database. Instead you define system groups, and connect numbers of systems to each systemgroup. These systems will share users.

The database is built of an hierarchy of systemgroups. The hierarchy has two functions, to describe the connection between different systemgroups, and to introduce heritage between systemgroups. The system groups lower in the hierarchy, can inherit attributes and users from systemgroups higher in the hierarchy.

Whether a system group will inherit users or not, is determined by the attribute UserInherit. If the attribute is set, the systemgroup will inherit all users from its parent usergroup.
Also the users that the parent has inherited from its parent are inherited. A systemgroup can override an inherited user by defining the username in its own systemgroup.

A systemgroup is referred to by the 'path'-name in the hierarchy, where the names are separated by periods, e.g. 'ssab.hql.se1', where ssab is the root group, and se1 the lowest level in the hierarchy.

A ProviewR system is connected to a systemgroup by stating the systemgroup in the System object. If the systemgroup is not present in the user database, though a parent or ancestor is, it is supposed that the systemgroup inherits users from the ancestor.

Attributes

Attribute	Description
UserInherit	The systemgroup inherits users fro its parent systemgroup, also users

that the parent has inherited.

Users

A user is characterized by a username, a password and a set of privileges. A user is also connected to a systemgroup.

The privileges define what a user is allowed to do in ProviewR. Some privileges influence the access to make changes from ProviewR utilities, e.g. the navigator or plc-editor, some regards the construction of operators graphics, to control which input fields and pushbuttons a user can influence.

A username can be connected to several system groups, but from the database point of view, they are different users, with unique passwords and privileges. They just happen to have the same username.

Privileges

Privilege	Description
RtRead	Read access to rtdb. Default privileges for user that is not logged in
RtWrite	Write access to rtdb. Allows user to modify rtdb from xtt and Simulate mode in trace
System	Privilege for system manager
Maintenance	Privilege for maintenance technician
Process	Privilege for process technician
Instrument	Privilege for instrument technician
Operator1	Privilege for operator
Operator2	Privilege for operator
Operator3	Privilege for operator
Operator4	Privilege for operator
Operator5	Privilege for operator
Operator6	Privilege for operator
Operator7	Privilege for operator
Operator8	Privilege for operator
Operator9	Privilege for operator
Operator10	Privilege for operator
DevRead	Read access to the workbench
DevPlc	Write access in the plc editor
DevConfig	Write access in the configurator
DevClass	Write access in class editor (not yet implemented)

23.2   Example

ProviewR user database V1.0.0

```
ssab
. . . . . . sysansv    System DevRead DevPlc DevConfig (14680068)
. . . . . . skiftel    Maintenance DevRead (2097160)
. . . . . . 55         Operator1 (64)
. hql                  UserInherit
. . . . . . anna       RtWrite Operator4 (514)
. . bl2
. . . . . . anna       Operator4 (512)
. . bl1                UserInherit
```

```
. . . . . . 55          Operator1 (64)
. . . . . . carlgustav  Operator8 (8192)
. hst
. . . . . . magnus      Operator1 (64)
. . rlb                 UserInherit
. . . . . . amanda      Operator4 (512)
```

Look at the example above. This is a listing of a user database. To the left, you see the
system groups, and the number of periods mark their level in the hierarchy. In the same row
the attribute of the system group is written. Under each systemgroup, its users with
privileges are found. Thus the systemgroup ssab has the users sysansv, skiftel and 55.

The systemgroup sasb.hql.bl1 has the attribute UserInherit, which results in that it inherits
users from its parent. Also the parent ssab.hql has UserInherit, i.e. ssab.hql.bl1 also
inherits from ssab. The users of ssab.hql.bl1 is then, sysansv, skiftel, anna 55 and
carlgustav. Here the user 55 of sasb.hql.bl1 overrides the user 55 of ssab.

The systemgroup ssab.hql.bl2 lacks UserInherit and has only the user anna.

The systemgroup ssab.hst.rlb has UserInherit and inherits from its parent ssab.hst. Though,
this has not UserInherit and has not inherited from its parent ssab. The users for
ssab.hst.rlb is then amanda and magnus.

A system with the systemgroup sandviken.hql will be denied access because the systemgroup and
all its ancestors are missing.

A system with the systemgroup ssab.vwx.n2 will inherit users from the systemgroup ssab,
i.e. sysansv, skifel and 55. All systemgroups don't have to be present in the database, the
existence of an ancestor is enough. The ones that are not found are supposed to have the
attribute UserInherit.

23.3 Login

This sections describes how Login and access control works in different ProviewR environments.

Development environment

When starting the configurator, a login window is opened where you can state username and
password. You can also give the username and password as arguments to the workbench if you
want to avoid the login procedure. To open the configurator, you need the privilege DevRead,
and to enter edit mode, you need DevWrite. To edit in the plc editor, you need DevPlc.

Operator environment

When the operator environment is started with an OpPlace as argument, the user is fetched from
the UserName attribute in the corresponding User object. To make modifications in the
database from the runtime navigator, the privilege RtWrite is required. In the process graphics
there are pushbuttons, sliders etc. from which you influence the database. These objects have
an access attribute, that determines which privileges are required to activate the object.
These privileges are matched to the users privileges, and if he isn't granted any of them,
he is denied access.

From the runtime navigator, you can with the login/logut command, login as another user and thereby change your privileges.

Web access

For process graphics on the web there is a special login frame that can be added to the start menu of the project home page. This is enabled in OpPlaceWeb.EnableLogin attribute. The login frame checks the username and password.

It's possible to specify a specific system group for web users in the WebSystemGroup attribute in the $Security object. The default group is common.web, which doesn't exist in the template user database. The first existing parent group, common, is then used instead and the users in this group has access to the web. If you want to restrict the web access, either create the common.web group without UserInherit, or create another group and insert this into the $Security object. Also create proper users in the group.

Edit the user database

The user database is edited from the adminstrator or configurator. It is opened from File/Open/UserDatabase in the menu.
See chapter Administration

You can also edit the database by commands in pwr_user,
Man kan ï¿½ven editera databasen med kommandon i pwr_user, see pwr_user in chapter Tools.

24 Class Editor

This section describes how to create new classes in ProviewR.
There are a number of different cases when you might consider creating a new class.

Data objects

You want to store data in a data structure, for example to easy gain access to the data from applications. You can also create data objects that describe material that passes through a plant, where the data object contains properites for one material, e.g. length, width etc. The material can be moved between NMps cells to indicate the position of a material in the plant.

Plc function object

A function object used in plc programming consists of a class that defines the input and output pins of the function object, and possible internal attributes. This type of objects also consist of codes that are executed by the plc program. You can choose to create the code as plc code or c code.

Components

A component object reflects a component in the plant, and is often divided into two or three different classes, a main object, a function object and a bus object, possibly also a simulate object. The main object is placed in the plant hierarchy and contains the signals that are connected to the component, in addition to other configuration data. A function object, placed in a plc program, is connected to the main object and works partly with data from its own inputs and outputs, and partly with signals and other parameters in the main object. If the signal exchange is made via Profibus, you kan also create a special module object that contains channel objects for the data transported on the Profibus. It is sufficient to make one connection between the main and the module object, to connect all signals and channels in the component. The simulation object is a function object, that is connected to the main object, and that simulates the component when the system is run in simulation mode.

Subclasses of components

ProviewR contains a number of basecomponent classes for valves, motors etc. These are designed in a general fashion to cover a large number of components. Often, you create a subclass that is adapted to a specific component, and that, for example, contains a link to a data sheet, helptext etc. for this component. By creating a subclass of a basecomponent you inherit all the methods and attributes from this, but you also have the possibility to increase the functionality with more attributes and more plc-code.

Aggregates

An aggregate reflects a plant part that contains a number of components. In this case, you can create an aggregate class that contains the different components in shape of attribute objects. To the aggregate, there is also a function object, that calls the functions objects for the present components. Aggregates can also contain other aggregates and give rise to quite extensive object structures. In principle, you could build a plant is one single object, but in practice it is appropriate to keep the object structure on a fairly low level. It is

mainly when you have several identical aggregates that you benefit by creating an aggregate object of a plant part.

24.1   Database structure

Object

In the chapter Database structure there is a description of how objects are constructed. Now there is reason to go a little further in the subject.

An object consists of an object head and an object body. The object head contains information about the object name, class and relation to other objects. The object body contains the data of the object.

Object header

An object has a name with a maximum size of 31 characters that is stored in the object header.

In the object header there is also a link to the class description of the object. The class description contains information of how to interpret the data of the object, how it is divided into different attributes, and the type of the attributes. You also find the methods that work on the object.

An object is placed in a tree structure and the object head contains pointer to the closest relatives: father, backward sibling, forward sibling and first child.

The structure of an object head are common for all types of objects.

Object body

An object can have two bodies, one body that contains the data that is needed in runtime. It can also contain one additional body with data that only exist in the development environment.

A body is divided into attributes that contain data of a specific type, e.g a Boolean, a Float32 or an Int32. But an attribute can also have a more complex datatype, as an array or a class.

RtBody
RtBody is the body that exists in the runtime database. The body is also present in the development environment, to make it possible to set values to different attributes in the body.

DevBody
Some objects also have a DevBody, a body that exists only in the development database, and that is not loaded into the runtime database. This body is mainly used by plc objects, where devbody for example contains graphical data for the plc editor.

24.2   Class description

The layout of an object body is described in the class description of the object. Here you
also find methods and other properties of the class. The class description is built of
specific class definition objects that reside in a class volume. The class volume has a strict
syntax of how to build the class descriptions. A presentation of the the different objects
that are a part of the class description follows here.

Class volume

Class descriptions reside in a specific type of volume, a class volume. These can contain
two hierarchies, one hierarchy with class descriptions, and one with type descriptions.

$ClassHier

The class descriptions are found under the root object 'Class' of type $ClassHier. Below
the $ClassHier object, $ClassDef objects define the classes in the volume.

$ClassDef

A $ClassDef object with its descendants, describe a class. The name of the object gives the
name of the class. Below the $ClassDef, the following objects can be located

- an $ObjBodyDef object, 'RtBody', that describes the runtime body.
- an $ObjBodyDef object, 'DevBody', that describes the body in the development environment.
- a Template object, i.e. an object of the current class that contains default values for
  instance objects of the class.
- one or several body objects that contains data for a specific function.
- a PlcTemplate object, that can be opened by the plc editor, and that contains plc code for
  the class.
- menu objects that define the popupmenu in the navigator, configurator and xtt.
- method objects that link to methods that are called for example when objects are created
  or moved in the development environment.

$ObjBodyDef

An $ObjBodyDef object can either have the name 'RtBody', and then describe the runtime body,
or the name 'DevBody' and describe the development body. The attribute 'StructName' contains
the name of the c-struct of the class in the included file that is generated for the volume.
Below the $ObjBodyDef object, one attribute object for each attribute in the object body is
located. $Attribute objects are used for data objects, and $Input, $Intern and $Output for plc
functionobjects.

$Attribute

An $Attribute object describes an attribute in a body. The attribute can be of the following
type:

- a base type, e.g. Boolean, Float32, Time, Int16.
- a derived type, e.g. String80, Text1024, URL.
- an array of a base type or derived type.
- another class.
- an array of a class.
- an rtdb pointer, i.e. a pointer that can be interpreted by all processes.
- a private pointer, i.e. a pointer that is valid on one single process.

The type is stated in the attribute 'TypeRef'. In the attribute 'Flags' you state if the object describes an array, pointer, class etc. If the object describes an array, the number of elements is stated in 'Elements'.

$Input

$Input describes an input pin in a functions object in the plc program. The input can be of type Boolean, Float32, Int32, String80, Time, DeltaTime or of datatype (pointer to Float32). $Input gives rise to an attribute with two elements, one element of the stated type, and one element with a pointer to the stated type. If the input is connected, the pointer points to the connected output attribute, if the pointer is not connected it points to its first element, where you then can specify a value for the input.

The attribute 'PgmName' states the name of the attribute in the c-struct, and 'GraphName' the textstring that is written in the function object at the input pin.

$Intern

Defines an intern attribute in a function object, i.e. an attribute that is neither an input nor an output.

$Output

$Output describes an output pin in a function object. The same datatypes is valid for an $Output as for an $Input.

$Buffer

$Buffer specifies an attribute that contains data of a certain size that only a single function is able to interpret. The data is described by a class, but is not viewable in for example xtt. PlcNode, that is found in all plc objects, is an example of a $Buffer. Here you find graphic information that is only of interest for the plc editor.

Class body

A class can contain a class body object. The class body object contains data that is common for all instances of the class. One example of a class body object is $GraphPlcNode that reside in all plc classes. $GraphPlcNode contains data for code generation and graphic layout of the function object.

Menus

Menu objects are used to define popupmenus for objects in the development environment and in the operator environment. $Menu defines a popupmenu in the development environment and $RtMenu in the operator environment. Below the menu object, menu alternatives are defined by $MenuButton objects, and submenues with $MenuCascade objects. The menu objects are placed below the $ClassDef object.

The menu object calls methods, i.e. c functions that are linked with the development or operator environment. There is for the moment no possibility to do this from a project. This has to be done from the ProviewR source code.

$Menu

$Menu objects describe popup menus in the development environment. The object name specifies

the function, the first part states the tool (Navigator/Configurator). The five last letters
state during which conditions the menu is present, dependent on which objects are selected
or pointed at.

```
char 1: P stands for pointed, i.e. the object at which the pointer points.
char 2: states what should be pointed at: 'o' an object, 'a' an attribute,
        'c' a class in the palette.
char 3: 's' stands for selected, i.e. the object that is selected.
char 4: states what the selected object should be: 'o' an object,
        'a' an attribute, 'c' a class in the palette.
char 5: states if selected and pointed should be the same object:
        's' same object, 'n' different objects.
```

Example ConfiguratorPosos: 'Po' the pointer points at an object, 'so' one object is selected,
's' the object the pointer points at and the selected object is the same object.

$RtMenu

The menu objects that describe popup menus in the operator environment.

$MenuButton

Defines a menu alternative in a popup menu.

24.3   Type description

Type descriptions, as class descriptions, reside in a class volume. They are placed in a
separate hierarchy under a $TypeHier object. Types are divided into two categories, base types
and derived types.

Base types

Base types are defined in the system volume pwrs. Example of base types are Boolean, Float32,
Int32, String, Enum and Mask.

Derived types

Derived types can be defined in any classvolume. They consist of
- arrays of base types, e.g. String80.
- enumeration types, Enum, with defined characterstrings for various values.
- bitmasks, Mask, with defined strings for various bits.

$TypeHier
Type descriptions are placed under the root object 'Type' of class $TypeHier. The $TypeHier
object has $Type and $TypeDef objects as children.

$Type
Description of a base type. This object is reserved for the system volume pwrs.

$TypeDef
Description of a derived type. The attribute 'TypeRef' contains the base type. The most
common usages are strings and texts with specific size, and enumeration types and bitmasks.

To define an enumeration type, the basetype should be $Enum. Below the $TypeDef object, texts for different values are defined with $Value objects. When the value of an attribute of the type is to be displayed, the text that corresponds to the value is displayed. When the attribute is given a value, the different texts are viewed with check boxes and you select one alternative.

To define bitmasks the basetype $Mask is used. Below the $TypeDef object, texts are defined for different bits by $Bit objects. When the attribute is given a value, the texts are displayed with check boxes, as for enumeration types. For bitmasks, several alternatives can be chosen.

$Value
Defines a value in an enumeration type. The value corresponds to a text, that is viewed in the configuration and in xtt when the attribute is opened. In the includefile for the volume, an enum declaration is created that can be used in c-code.

$Bit
Defines a bit in a bitmask. The bit corresponds to a text that is viewed in the configurator and in xtt when an attribute of the type is opened. In the includefile for the volume, an enum declaration is created that can be used in c-code.

24.4   Create classes

24.4.1   Create a class volume

The classdefinition objects reside in a classvolume, and first the classvolume has to be registered and created.

The registration is made in the global volume list which is opened from File/Open/GlobalVolumeList in the navigator menu. Here you create a VolumeReg object with suitable volume name and volume identity. The volume identity for user classvolumes should be in the intervall 0.0.2-249.1-254. Use preferably the prefix CVol in the name to indicate that it is a class volume. Also state the current project.
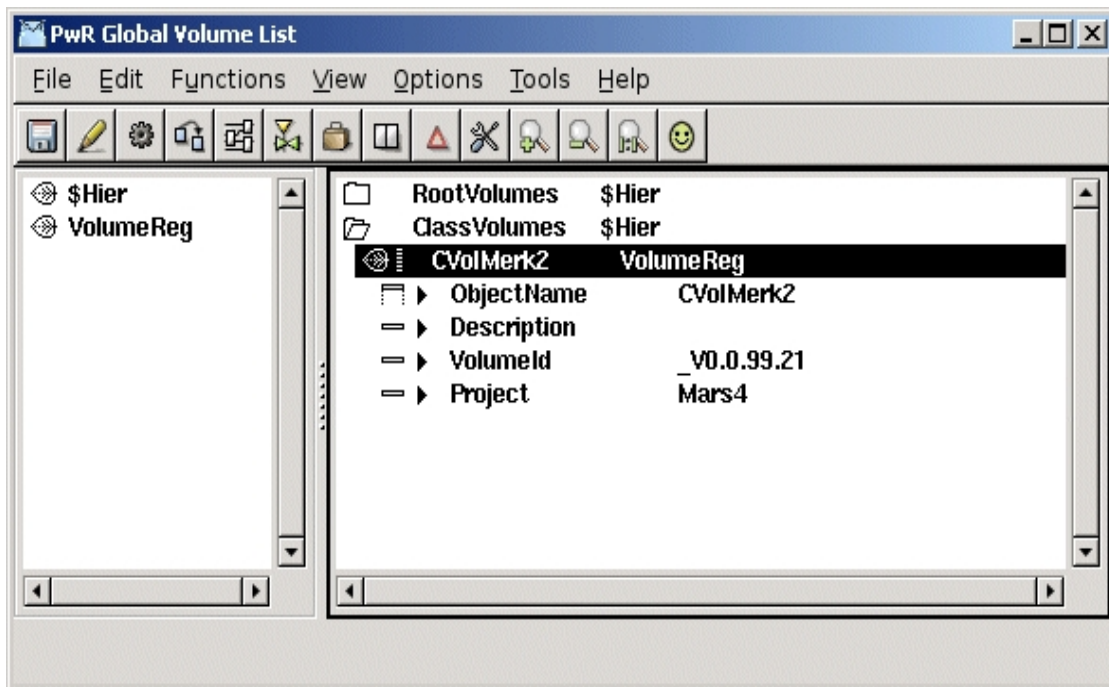
Registration of the class volume in GlobalVolumeList

Next step is to configure the classvolume in the directory volume of the project, with a ClassVolumeConfig object. Open the Directory volume with

```
$ pwrs
```

and create a ClassVolumeConfig object in the left window. The object should have the same name as the classvolume. After saving and leaving edit mode, the classvolume can be opened by rightclicking on the ClassVolumeConfig object and activating 'Open ClassEditor...'.

Configuration of th classvolume in the Directory volume

Now the Class Editor is opened, where you can create classdefinition objects. By entering edit mode, a palette is viewed, with the class and type description classes that are used to

define a class or type.

Begin by creating an object of type $ClassHier on the top level. This will automatically get
the name 'Class'. Below the $ClassHier objects, $ClassDef objects are created for each class
that is to be defined.

24.4.2   Data classes

Data classes are the most elementary classes, and usually used to store data. The classes
constist of a RtBody with attributes.

To create a class you put a $ClassDef object below the 'Class' object.
The name of the object states the class name.

Under the $ClassDef object you create a $ObjBodyDef object that automatically gets the name
RtBody.

Under the RtBody object, an $Attribute object is created, that defines an attribute in the
class. The name of the $Attribute object states the attribute name. In the object you should
state this in the attribute object:

- the attribute type is stated in TypeRef, e.g a 32-bit integer is stated with
  pwrs:Type-$Int32, a 32-bit float with pwrs:Type-$Float32 and a boolean with
  pwrs:Type-$Boolean. Actually it is the full name of a type definition object that is
  inserted. See the Object Reference Manual, pwrs/Types, which types are defined.
- if the attribute name contains national characters, in PgmName you have to state a name
without national characters, that is accepted by the c compiler.
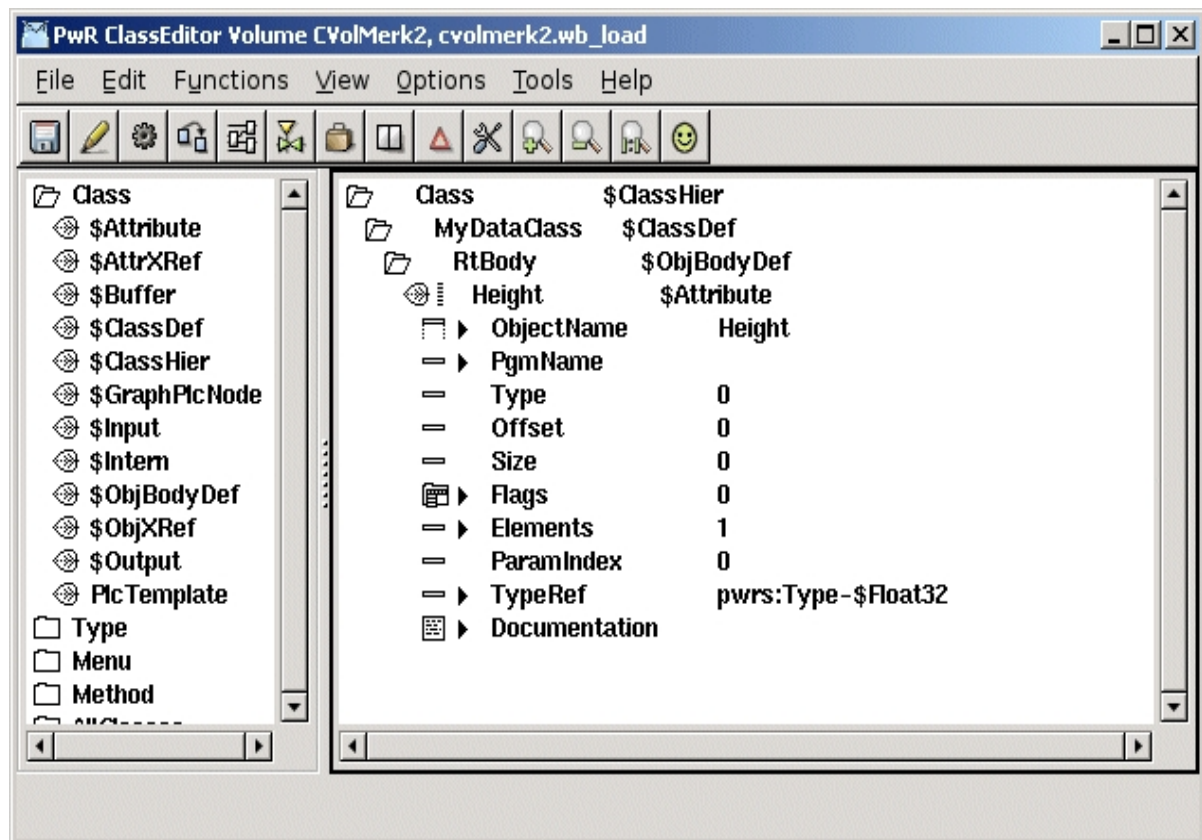
Definition of an attribute

When you save, an instance object of the current class with the name Template, is created under

the $ClassDef object. Here you can see the layout of the class, and also set template values for attributes. When other instances of the class are created, they are created as a copy of the Template object.
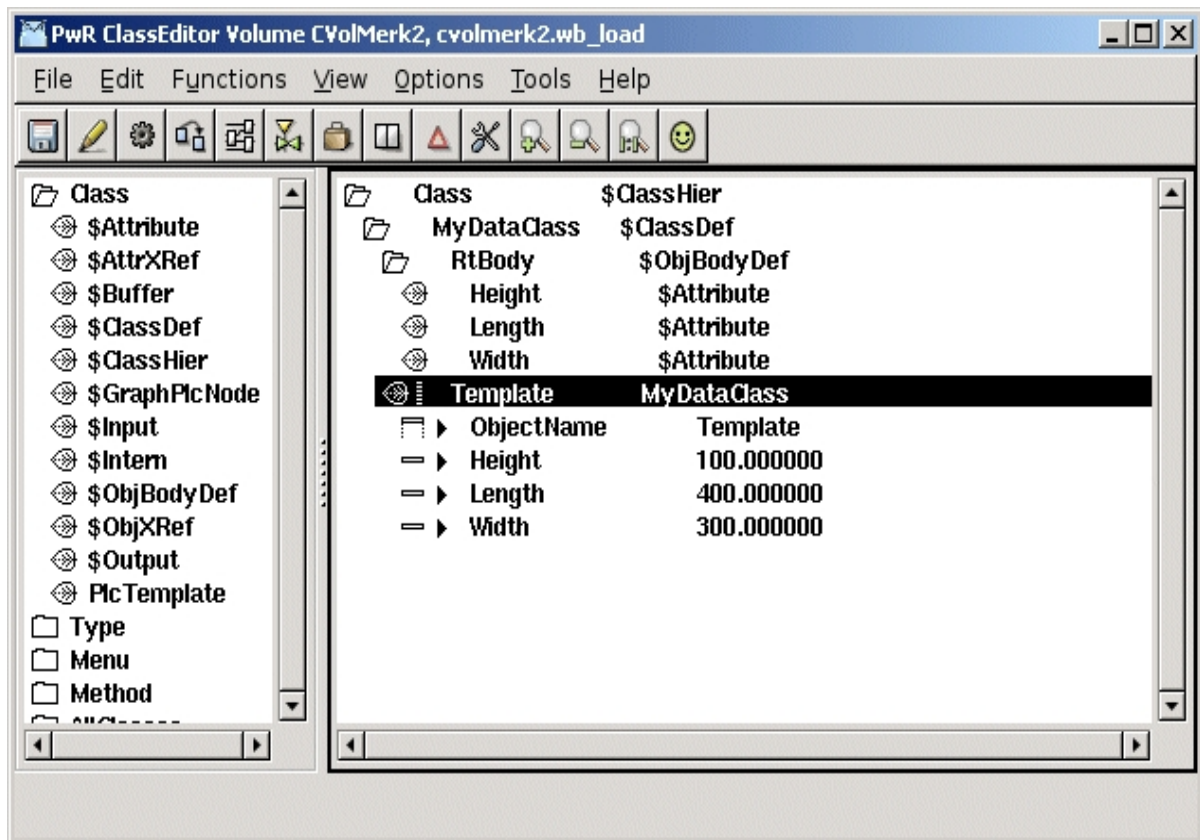
Template object with default values

Arrays

An array attribute is defined with an $Attribute object, as other attributes. Here you set the Array bit in Flags, and state the number of elements in the array in Elements.
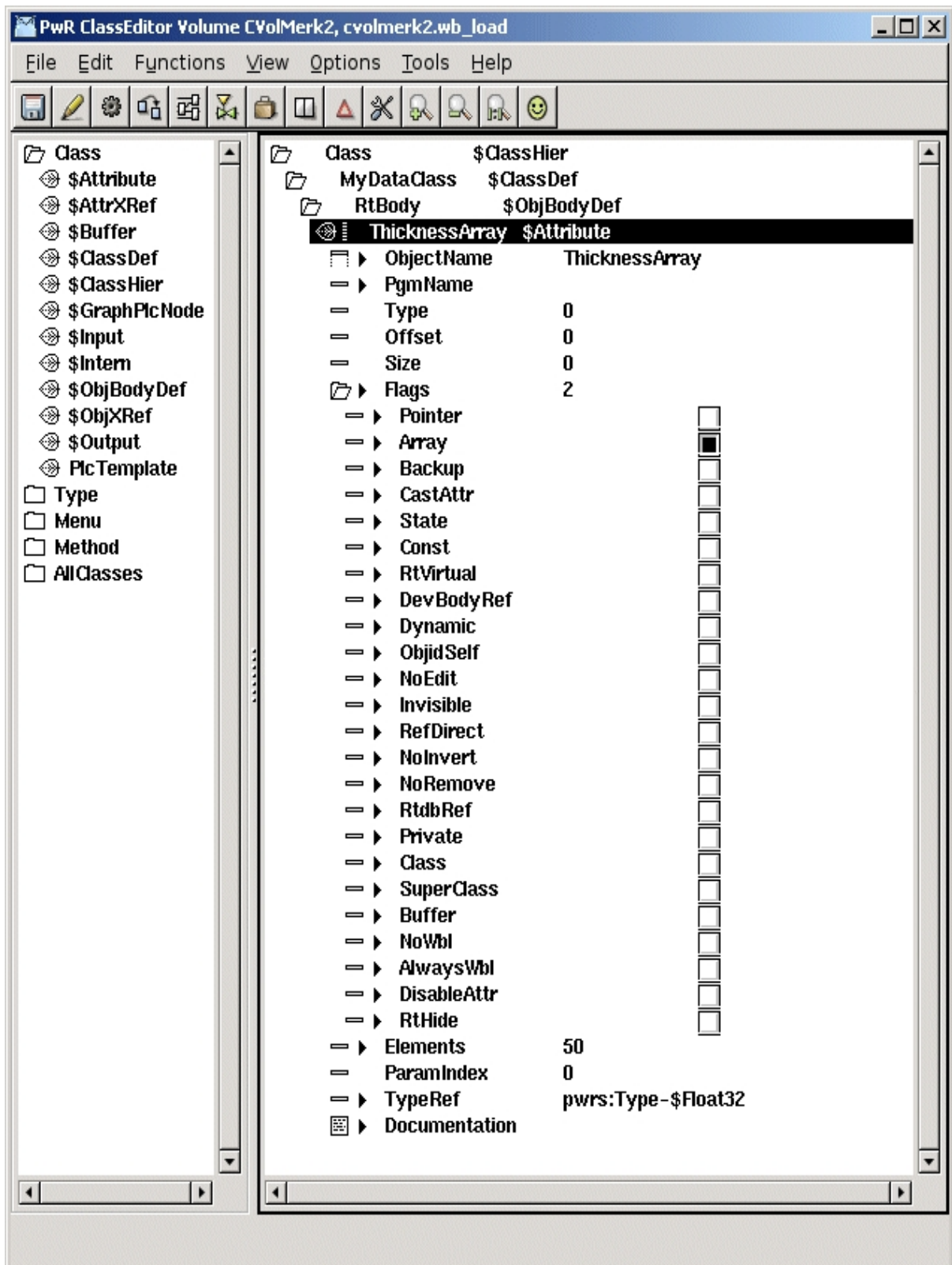
Definition of an array attribute with 50 elements

Pointers

There are two types of pointer attributes

- relative pointers that can be used by serveral processes. The value is set with the gdh_StoreRtdbPointer() function and converted to an absolute pointer with

gdh_TranslateRtdbPointer(). Set the Pointer bit in Flags. Note! The size of the element the pointer is pointing at should be set in Size (in bytes).

- absolute pointers. These can only be set and used by one single process. Set the Pointer bit and the Private bit in Flags.

Attribute objects
The term attribute objects refer to attributes that are described by a data structure. The reason to do this can be that you want to gather data under a map, or that the datastructure is repeated, and in this case you create an attribute object array.

The data structure of the attribute is defined in a separate class. The class should only contain a runtime body, and can not have a development body.

The attribute object is defined by an $Attribute object. In TypeRef the class describing the datastructure is stated, and in Flags the Class bit is set.

You can also create an array, by setting the Array bit in Flags, and state the number of elements in Elements.

Attribute objects can also contain attributes that are attribute objects. The number of levels are limited to 20, and the total attribute name is limited to 255 characters.

An attribute in an attribute object is referred to with periods as delimiter, i.e. the attribute Description in the attribute object Pump in object o, is referred to with the name 'o.Pump.Description'. If Pump also is an array of pumpobjects, the name of the Description attribute in the first pump object is 'o.Pump[0].Description'.
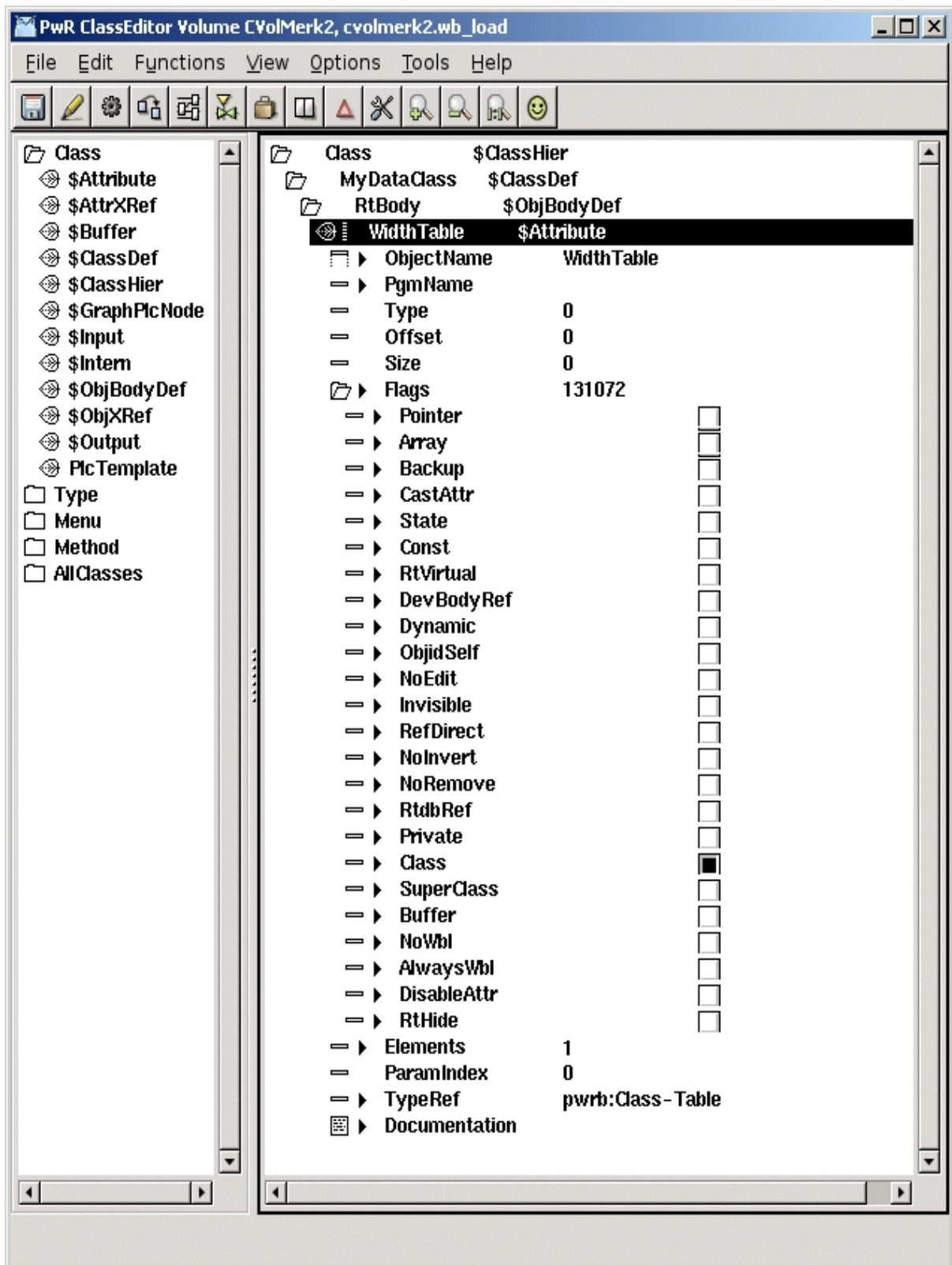
Definition of an attribute object of class Table

Subclass

You can also define a class as a subclass to another class. The subclass will inherit

attributes and methods from the other class, which is called the super class.

A subclass is defined by naming the first $Attribute object in the class to 'Super',
and setting the Class and SuperClass bits in Flags. The superclass is stated in TypeRef.

All the attributes that exist in the superclass will also be present in the subclass. The
subclass can also have additional attributes that are defined as usual by $Attribute objects.

A superclass can only contain a runtime body, not a development body.
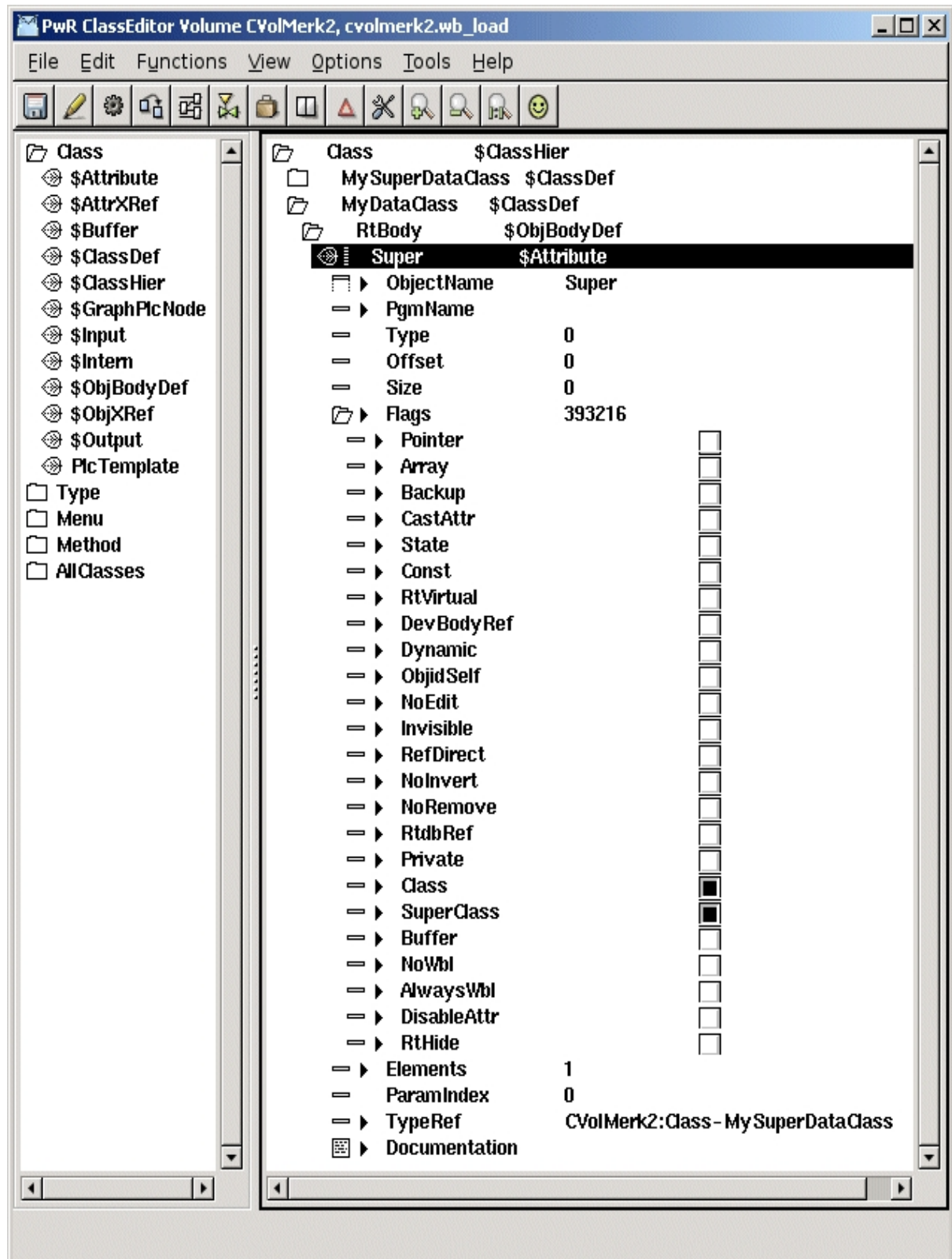
The Super attributes makes MyDataClass a subclass of MySuperDataClass

24.4.3 Function object classes

Function objects are used in the plc editor to program the plc program. A function object is also described by a class, usually a bit more complex than a data class. It defines, in addition to the data structure, the graphic layout with inputs and outputs, and the code that is to be generated for the plc program.

The code can be defined either by c-code, or by graphical programming in the plc editor.

24.4.3.1 Function object with c code

The function object class is defined by a $ClassDef object under the 'Class' object. Name the object and activate 'Configure-CCodeFo' from the popupmenu of the object. Now are created

- a RtBody object.
- a DevBody object with a PlcNode object that defines a buffer for graphic information in the instances.
- a GraphPlcNode object that contains information for graphic and code generation for the class.

Next step is to define the attributes of the class. The attributes are divided into inputs, internal attributes and outputs.

Inputs
The input attributes define the input pins of the function object, i.e. values that are fetched from output pins of other function objects. The inputs are defined by $Input objects that are placed below the RtBody object.

In TypeRef the datatype of the input is stated. Valid datatypes for an input are pwrs:Type-Float32, pwrs:Type-Int32 and pwrs:Type-String80.

In GraphName the text at the input pin in the function object is stated. Normally you use 2 - 4 characters, block letters for analog signals, lower-case for digital, and first character upper-case for other signal types.

An input attribute in an instance object, contains both a pointer to the output it is connected to, and a value that can be stated. You choose whether to use the input pin and connect an output, or to set a value, with a check box (Used). If you choose not to mark Used, the input pin is not displayed in the function object. In the Template object, you can set default values for the input, that will be used when the input is not connected.

Intern attributes
Intern attributes are attributes that are not inputs or outputs. They can be used for calculated values that need to be stored in the object, or values that are used to configure the object.

All common datatypes are valid for intern attributes.

Outputs
The output attributes define the output pins of the function object, i.e. values that are stored in the object, and can be fetched by inputs of other function objects. The outputs are defined by $Output objects that are placed below the RtBody object.

The datatype for the output is stated in TypeRef. As for $Input, Boolean, Float32, Int32 and String80 can be stated, and in GraphName the text for the output pin in the function object is stated.

Note !
$Input, $Intern and $Output have to be placed in this order below RtBody: $Input first, then $Intern and then $Output.

Default values
Defaultvalues of attributes can be set in the Template object.

If you want to state which inputs and outputs should be viewed as default, there is a mask in the GraphPlcNode object that controls this, default_mask. Bits in default_mask[0] correspond to input attributes, and bits in default_mask[1] to output attributes. If the bit that corresponds to a specific input or output is set, this will be viewed as default.
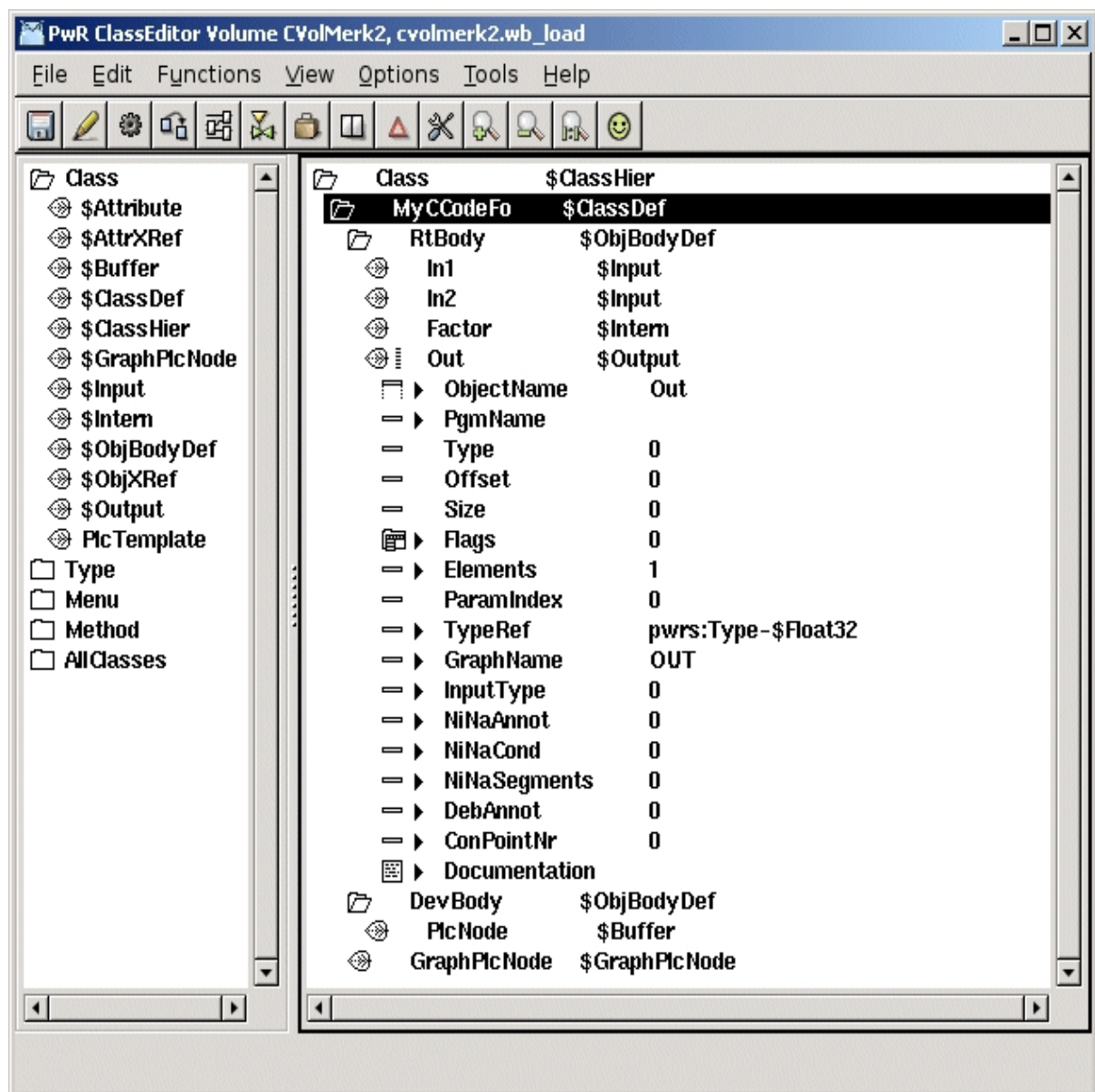
Function object with two inputs, one intern attribute, and one output
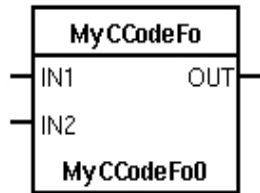
The function object for the class

Code

When the classvolume is built, an h-file with a c struct for the class is generated. The name of the struct is

```
pwr_sClass_'StructName'
```

where StructName is fetched from the StructName attribute in RtBody. As default, it is the same as the class name, but, for example if the classname contains national characters, another name can be specified.

Below an example of the struct for the class MyFo is viewed. MyFo contains two inputs In1 and In2, one intern attribute Factor, and an output Out, all of type Float32.

```
typedef struct {
  pwr_tFloat32          *In1P;
  pwr_tFloat32          In1;
  pwr_tFloat32          *In2P;
  pwr_tFloat32          In2;
  pwr_tFloat32          Factor;
  pwr_tFloat32          Out;
} pwr_sClass_MyFo;
```

Note that each input consist of two elements, a pointer with the suffix 'P', and an element that can be given a value if the input is not connected. If the input is connected, the pointer element will point to the output it is connected to, otherwise it will point to the value element. Therefore, in the c-code, you should use the pointer element to fetch the value of the input.

The code for the class is a function with this appearance

```
void 'StructName'_exec( plc_sThread *tp,
                        pwr_sClass_'StructName' *o) {
}
```

In the code, data is fetched from the inputs, and calculated values are put on the outputs. Also intern attributes can be used to store information to the next scan, or to fetch configuration data.

In the code example below In1 and In2 are inputs, Factor is an intern attribute and Out an output.

```
  o->Out = o->Factor * (*o->In1P + *o->In2P);
```

Note that the pointer element for the inputs In1 and In2 are used in the code.

You should also add prototype declaration of the exec function in ra_plc_user.h

```
void 'StructName'_exec( plc_sThread *tp,
                        pwr_sClass_'StructName' *o);
```

The module of the c-code is compiled and linked with the plc program. This requires a link
file to be placed on the directory $pwrp_exe. The file is named
plc_'nodename'_'busnumber'_'plcname'.opt, e.g. plc_mynode_0999_plc.opt. The content of the
file is input to the linker, ld, and here you add the modules of the plc-code. In the example
below these modules are placed in the archive $pwrp_lib/libpwrp.a

```
$pwr_obj/rt_io_user.o -lpwrp -lpwr_rt -lpwr_usbio_dummy -lpwr_usb_dummy -lpwr_pnak_dun
 -lpwr_cifx_dummy -lpwr_nodave_dummy -lpwr_epl_dummy
```

24.4.3.2   Function object with plc code

A function object, where the code is written in plc-code in the plc editor, is defined in a
similar way as the function object with c-code above.

The functionobject class is defined by a $ClassDef object below the 'Class' object. Name the
object and activate Configure-Fo in the popupmenu for the object. Now, in addition to the
objects created for the c-code functionobject, also a Code object of class PlcTemplate is
created. This object can be opened with the plc editor, and define the code for the class.

Inputs, intern attributes and outputs in the function object are defined in the same way as for
the c-code function object, by $Input, $Intern and $Output attributes.
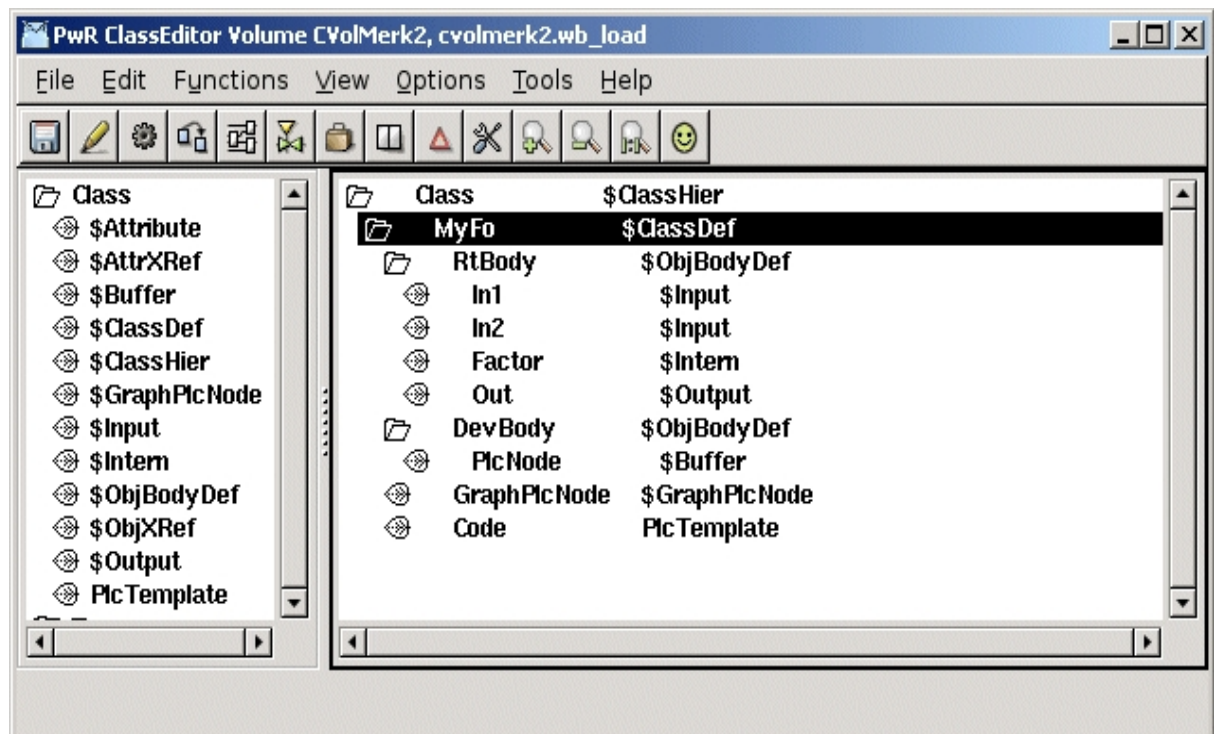
Definition of a function object with plc code.

Code
By activating 'Open Program...' in the popupmenu of the Code object, the plc editor is opened.
Here the code is written with function object programming. The code is created similar to an
ordinary program, but here you also have to fetch values from the input and intern attribute,
and to set values to the outputs.

Values of inputs, intern attributes, and also outputs, are fetched in the code with GetDp, GetIp, GetAp or GetSp objects. You connect the objects to attributes in the class by selecting the attribute in the Template object for the class, and activate the 'Connect' method for the Get object. A symbolic reference $PlcFo is put into the Get object. This will later be exchanged to a reference to the current instance, when the code for the instance is compiled.

Calculated values are stored in outputs or intern attributes with StoDp, StoIp etc. These are connected to attributes in the same way as the inputs, by selecting the attributes in the Template object and activating 'Connect'.
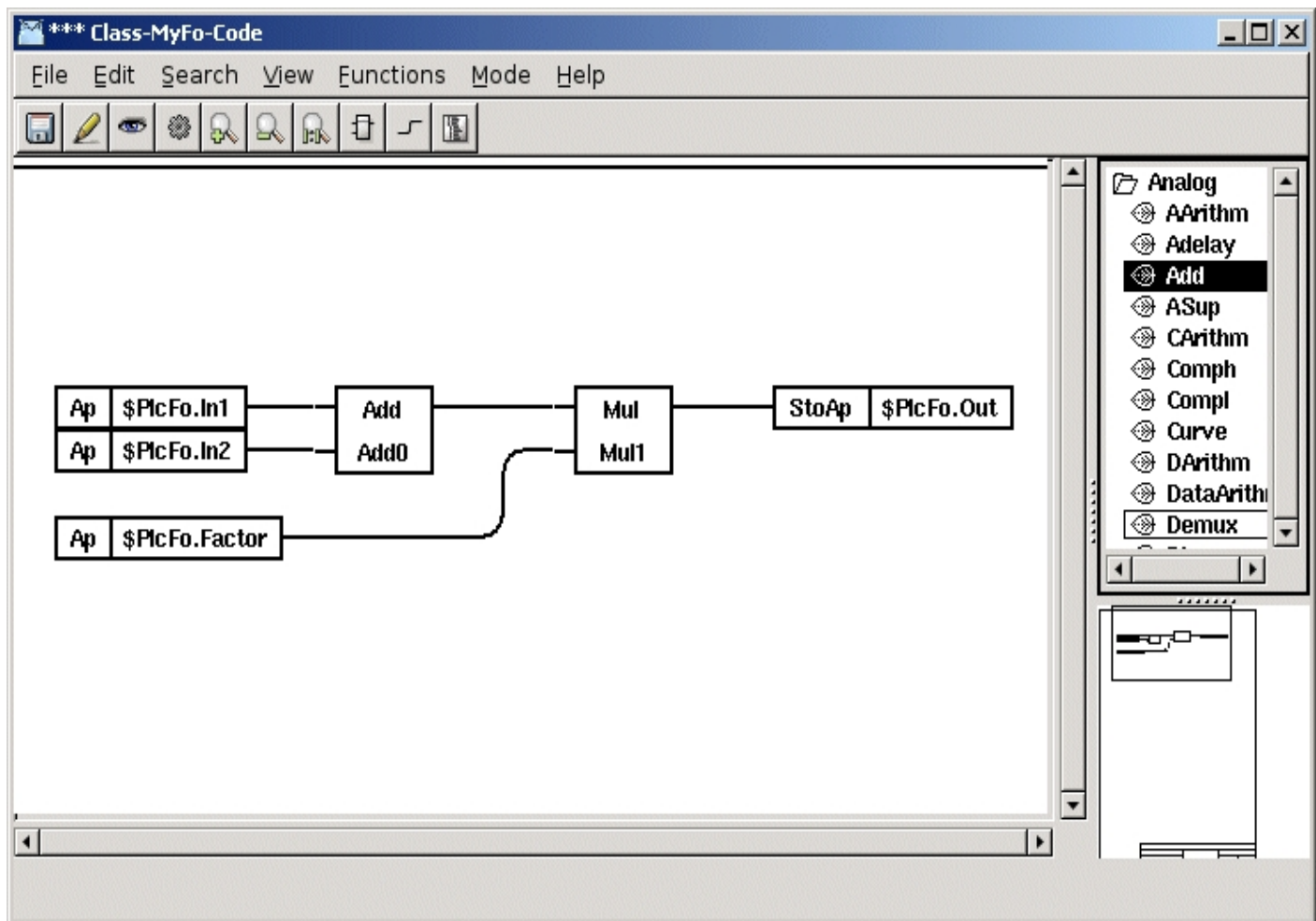
Example of plc code for a function object

The template code in the Code object should not be compiled or built. When an instance object is compiled for the first time, the template code is copied to the instance.

When the template code is changed, the code of the instances will be updated the next time they are compiled (the volume containing the instances, has to be updated with UpdateClasses first).

24.4.4   I/O classes

I/O objects are the objects handled by the I/O-handling in ProviewR. They are divided in Agent, Rack, Card and Channel objects. When adapting new I/O systems to ProviewR, you have to create new classes of types Agent, Rack and Card. I/O objects are defined by a $ClassDef object where the IoAgent, IoRack or IoCard bit is set in Flags.

A more detailed description of how to create I/O objects is found in Guide to I/O System.

24.4.5 Components

A component is an object, or a number of objects, that handles a component in the plant. It could be a valve, a motor, a frequency converter etc. The idea behind the component concept is that by creating one object (or a number of objects) you get all the functionality required to control the component: an object containing data and signals, a function object with code to control the component, an object graph for HMI, a simulation object, I/O objects to configure bus communication etc.

A component can include the following parts
- a main object
- a function object
- a simulation object
- one or more I/O bus objects
- object graph for the main object
- object graph for the simulation object
- graphic symbol for the main object

24.4.5.1 Main object

The main object contains all data needed to configure and make calculations. The object is placed in the plant hierarchy, as an individual object or as a part of an aggregate.

Often the class BaseComponent:Component is used as super class to a component class. It contains a number of attributes as Description, Specification, DataSheet etc.

All the input and output signals that are attached to the component should be placed in the main object. Di, Ii, Ai, Do, Io, Ao or Co object are inserted as attribute objects. When creating instances of the component, the signals have to be connected to channel objects. For profibus, for example, you can create a module object, that contains the channels, and preconnect the signals in the main object to these channels. For each instance, you then don't have to connect every channel individually, but can make a single connection between main object and module object.

Special attributes
PlcConnect
If there is any code that is to be created by the plc program, you create a function object for the class. This has to be connected to the main object, and this connection is stored in an attribute with name 'PlcConnect' of type pwrs:Type-$AttrRef.

SimConnect
If there is a simulation object, this is connected to the main object by a 'SimConnect' attribute of type pwrs:Type-AttrRef.

IoConnect
If there is a I/O module object, this is connected with an 'IoConnect' attribute of type pwrs:Type-AttrRef. The attribute is handled by the IoConnect method.

IoStatus
If you want to fetch the status from the I/O-module object, you create the attribute 'IoStatus' of type pwrs:Type-$Status, and set the Pointer bit in Flags. You also have to set 4 in Size (for relative pointers the size of what the pointer is pointing at, has

to specified in Size).

The attribute will be assigned a pointer to the Status attribute of the I/O-module in runtime when the I/O handling is initialized. The Status attribute is of type Status and can for example be displayed in an object graph with the dynamic type StatusColor. If you want to use IoStatus in the plc code for the object, you have to consider that the attribute is a pointer and fetch the value with GetIpPtr.

SequenceReset
It is possible to use Grafcet sequences in a component. One difference from an ordinary sequence are that the reset object should be defined as an attribute of class Dv in the main object, with the name 'SequenceReset'. SubSteps can not be used in the sequence.

GraphConfiguration
GraphConfiguration is of type Enum and used to decide which object graph is to be opened for the current instance. It is used by the 'ConfigureComponent' method (see below).

DisableAttr
The DisableAttr function makes it possible to disable an attribute in an instance. If an attribute is disabled, it will not be viewed in the navigator or object editor. If the disabled attribute is a signal, it will be ignored by the I/O handling.

The disable function is used for components that can exist in different configurations. A solenoid valve for example, can have one switch indicating that the valve is open, and one indicating that the valve is closed. Totally there are four configurations for the solenoid valve:

- no switches
- switch open
- switch closed
- both switch open and switch closed

You could create four different solenoid valve classes, but a problem will come up when building aggregates of the valve objects. An aggregate, containing a valve object also has to exist in four variations, and if the aggregate contains two valve objects, there has to be 16 variations. By using the DisableAttr function on the switch attributes we can create a solenoid valve class that covers all four configurations, and also can be used in aggregate classes.

DisableAttr for an attribute is configured in the following way.
- the DisableAttr bit in Flags is set for the attribute.
- before the attribute, an attribute of type pwrs:Type-$DisableAttr is placed, with the same name as the attribute, but with the prefix 'Disable'. The Invisible bit in Flags should be set for the DisableAttr attribute.

Example
In the solenoid valve class above, the switch closed is represented by the attribute SwitchClosed that is a digital signal of type pwrb:Class-Di. Immediately above the attribute an attribute with name 'DisableSwitchClosed' of type pwrs:Type-$DisableAttr is placed. For this attribute the Invisible bit in Flags is set, and for the SwitchClosed attribute the DisableAttr bit in Flags is set.

File Edit Functions View Options Tools Help

	MValve	$ClassDef
	RtBody	$ObjBodyDef
	DisableSwitchClosed	$Attribute

	ObjectName	DisableSwitchClosed
	PgmName	
	Type	0
	Offset	0
	Size	0
	Flags	2048
	Elements	1
	ParamIndex	0
	TypeRef	pwrs:Type-$DisableAttr

SwitchClosed $Attribute

	ObjectName	SwitchClosed
	PgmName	
	Type	0
	Offset	0
	Size	0
	Flags	4325376
	Pointer	☐
	Array	☐
	Backup	☐
	CastAttr	☐
	State	☐
	Const	☐
	RtVirtual	☐
	DevBodyRef	☐
	Dynamic	☐
	ObjidSelf	☐
	NoEdit	☐
	Invisible	☐
	RefDirect	☐
	NoInvert	☐
	NoRemove	☐
	RtdbRef	☐
	Private	☐
	Class	■
	SuperClass	☐
	Buffer	☐
	NoWbl	☐
	AlwaysWbl	☐
	DisableAttr	■
	RtHide	☐
	NewAttribute	☐
	Elements	1
	ParamIndex	0
	TypeRef	pwrb:Class-Di

Class panel (left):

- Class
  - $Attribute
  - $AttrXRef
  - $Buffer
  - $ClassDef
  - $ClassHier
  - $GraphPlcNode
  - $Input
  - $Intern
  - $ObjBodyDef
  - $ObjXRef
  - $Output
  - PlcTemplate
- Type
- Menu
- Method
- AllClasses

Attribute with disable function

Cast

Component classes are often built in a relatively flexible way to minimize the number of variants. Often you create a baseclass that makes use of the DisableAttr function to be able to cover a number of different configurations. In the example above a solenoid valve class can cover four different configurations by setting DisableAttr on the switch signals. You also create subclasses that are adapted to specific valves. For example, a Durholt 100.103 doesn't contain any switches, and an subclass is created where both switches are disabled in the Template object. You also set other adaptations in the Template object as a link to a datasheet. The result is a subclass that can be used for Durholt valves without any configurations for each instance.

If we now build a general aggregate, containing a solenoid valve, and want to be able to use the subclasses that exist for the solenoid valve, we use the Cast function. With the Cast function, an attribute object can be casted as a subclass of the original class, given that the subclass has the same size. When an attribute object is casted, default values, and thus configurations, are fetched from the subclass. Also classname and methods are fetched from the subclass.

The cast function for an attribute is entered in the following way:
- The CastAttr bit in Flags is set for the attribute.
- Before the attribute, an attribute of type pwrs:Type-$CastId is placed with the same name as the attribute, but with the prefix 'Cast'. The Invisible bit in Flags should be set for the cast attribute.
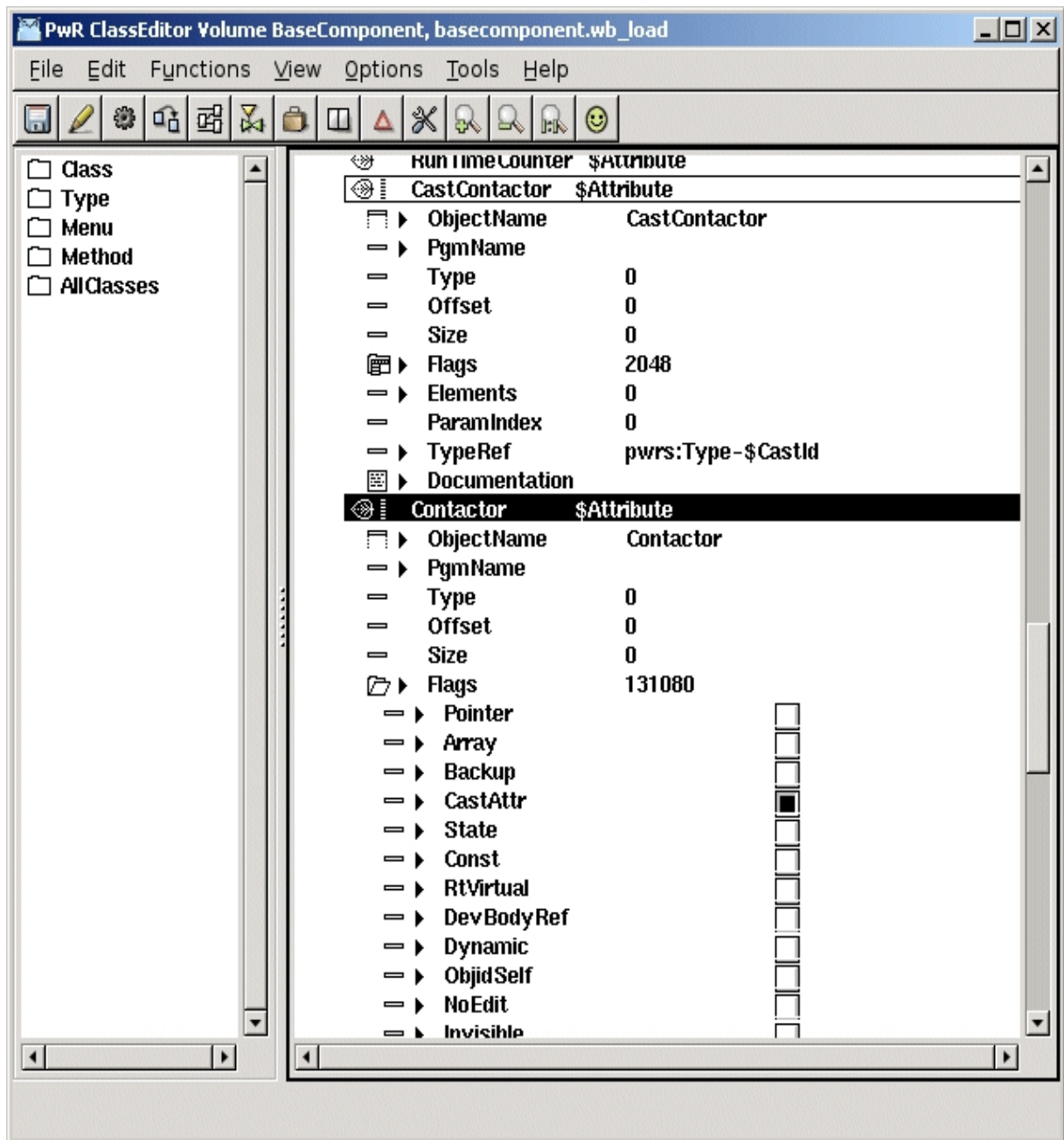
Contactor with cast attribute

Casting of an instance is executed by activating the 'Cast' method in the popupmenu for the attribute. A list with the baseclass and all subclasses are displayed, where a suitable cast class can be selected.

If an attribute has both cast and disable attributes, the cast attribute should be placed before the disable attribute.
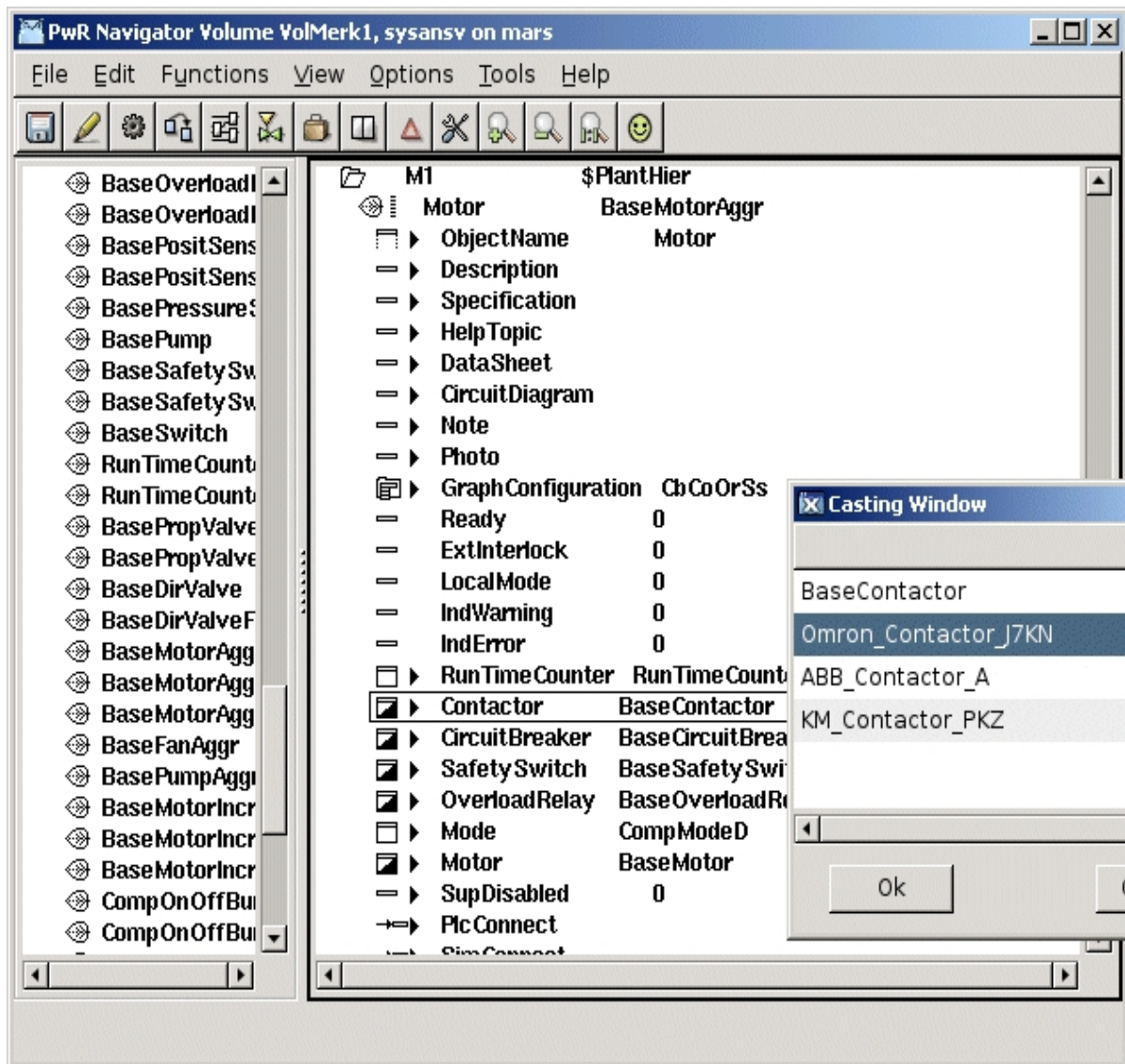
Casting of an instance

Methods
Method ConfigureComponent
Often there are several variants of a component. In the example with the solenoid valve above,
four different variants were found dependent on the configuration of switches. To simplify
the users configuration of the component, you can define the method 'ConfigureComponent'.

The ConfigureComponent method makes it possible to set Disable to one or a number of attributes
from a menu alternative in the popupmenu, and to select an object graph that is adapted
to the current configuration.

Meny
The menu alternatives for ConfigureComponent are defined by menu objects. Under the $ClassDef
object, a $Menu object with name 'ConfiguratorPosos' is placed, which makes the menu visible
in edit mode when the object is pointed at and selected. Below this, yet another $Menu object
is placed with the name 'Pointed', and below this a $MenuCascade object with the name
'ConfigureComponent'. The attribute ButtonName is set to ConfigureComponent for this object.
Below this, finally one $MenuButton object is placed for each configuration alternative.
The name is preferably set to the name of the configuration alternative, and is also put into

the attribute ButtonName. In the attribute MethodName '$Object-ConfigureComponent' is inserted and in the attribute FilterName '$Object-ConfigureComponentFilter'. You should also fill in arguments to the method in MethodArguments. MethodArguments[0] contains a bitmask, that decide which attributes that will be disabled in the current menu alternative. Each attribute, that is possible to disable is represented by a bit, and bit order corresponds to the attribute order in the object. MethodArguments[1] contains the graphic representation, see below.

If we look at the solenoid valve, we have two attributes that can be disabled, SwitchClosed and SwitchOpen. In the bitmask in MethodArguments[0] SwitchClosed corresponds to the first bit and SwitchOpen to the second, i.e. if the first bit is set, SwitchClosed will be disabled, and if the second bit is set, SwitchOpen is disabled. The four configuration alternatives TwoSwitches, SwitchClosed, SwitchOpen and NoSwitches correspond to the following masks

```
TwoSwitches    0    (both SwitchOpen and SwitchClosed are present)
SwitchClosed   2    (SwitchOpen is disabled)
SwitchOpen     1    (SwitchClosed is disabled)
NoSwitches     3    (both SwitchOpen and SwitchClosed are disabled)
```
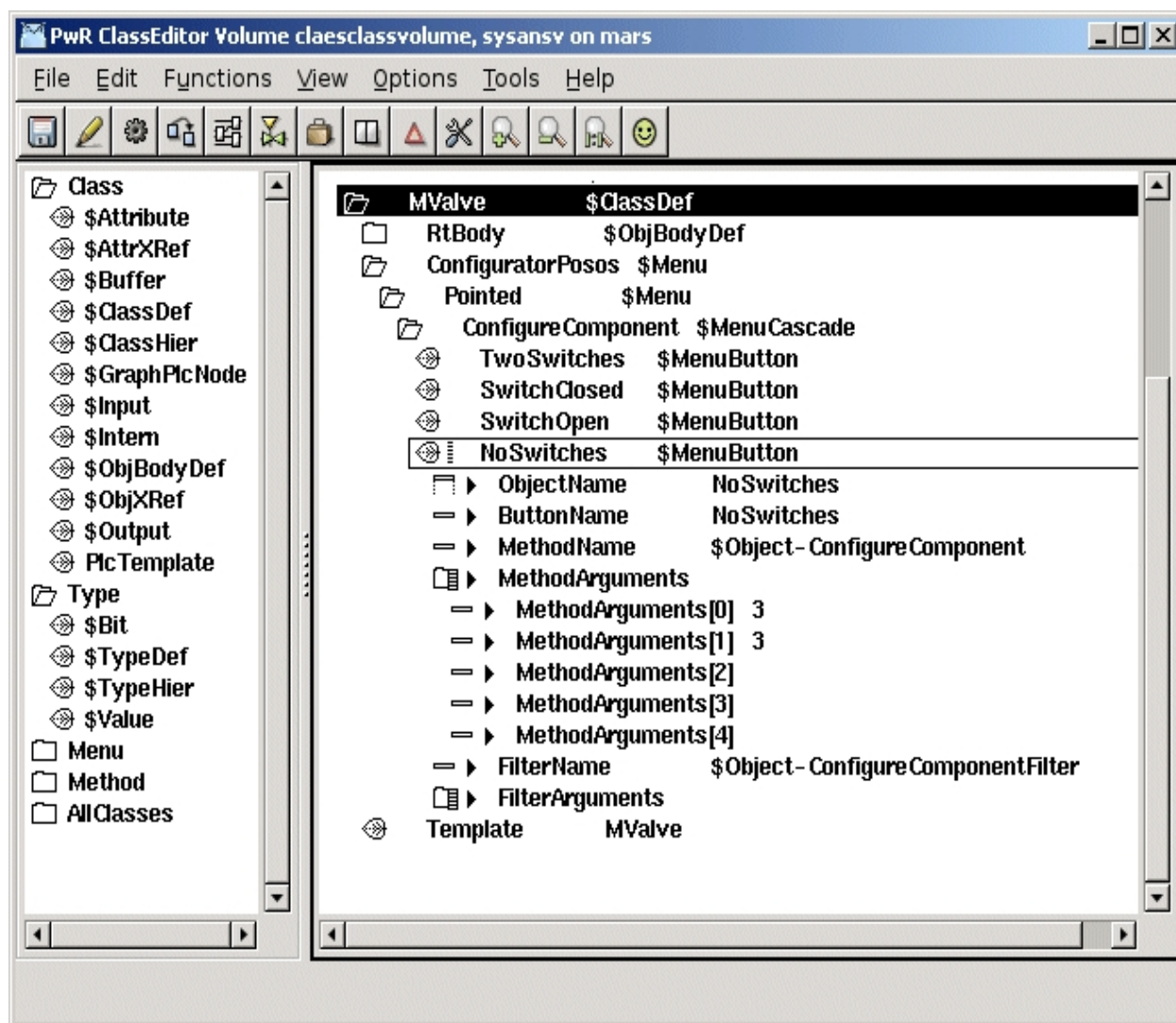

Configuration of component attributes

If you disable an attribute that is a component that contains signals, the signals in the component also have to be disabled. The I/O handling only looks at if the individual signal is disabled, and is not looking upwards on higher levels. To disable a signal in a component attribute, you add a comma and the name of the component followed by the disable mask that

is valid for the component to MethodArguments[0]. For example in an object where the
components Contactor and CircuitBreaker are disabled MethodArguments[0] can contain

```
3, Contactor 1, CircuitBreaker 1
```

where '3' is the Disable mask of the object (that disables the attributes Contactor and
CircuitBreaker), and 'Contactor 1' results in disabling a signal attribute in Contactor, and
'CircuitBreaker 1' disables a signal in CircuitBreaker.

There is also another syntax with paranthesis that allows more than two levels. In this example
the object above, Motor, is a part of a larger aggregate.

```
(5 (Motor 3 (Contactor 1, CircuitBreaker 1), Temp 1))
```

Component attributes with individual configuration

When the ConfigureComponent method is activated, Disable is removed from all component
attributes, to reset any previous configuration. Sometimes there are component attributes
that are not a part of the object configuration, but have to be configured individually. These
components should not be reset by the ConfigureComponent method, and have to be stated in
MethodArguments[2] with comma as delimiter. In the following example, the component attributes
Motor and Contactor should be configured by their own ConfigureComponent methods, and not
affected by the ConfigureComponent method of the object. In MethodArguments[2] is stated

```
Motor, Contactor
```


Object graph
When drawing the object graph for the component, you have to consider the different
configurations. If the differences between the configurations are small, you can use the
Invisible dynamic. If the differences are greater, it might be more convenient to draw
separate graphs for the configurations. Then you insert an attribute in the main object with
the name GraphConfiguration of type Enum. It is common to create a specific enumeration type
with the configuration alternatives. If GraphConfiguration is 0 the standard graph is used,
else the value in GraphConfiguration is set as suffix to the graphname.

In the example with the solenoid valve, MValve, we create an enum type, MValveGCEnum, and
define the values

```
TwoSwitches   0
SwitchClosed  1
SwitchOpen    2
NoSwitches    3
```

For the TwoSwitches configuration, with value 0, we draw an object graph with name mvalve.pwg.
For SwitchClosed, with value 1, we name the graph mvalve1.pwg, for SwichOpen mvalve2.pwg and
for NoSwitches mvalve3.pwg.

We also state the enumeration value in MethodArguments[1] in the $MenuButton object for the
current configuration. This will imply that GraphConfiguration will be set to this value
when the current menu alternative is activated.
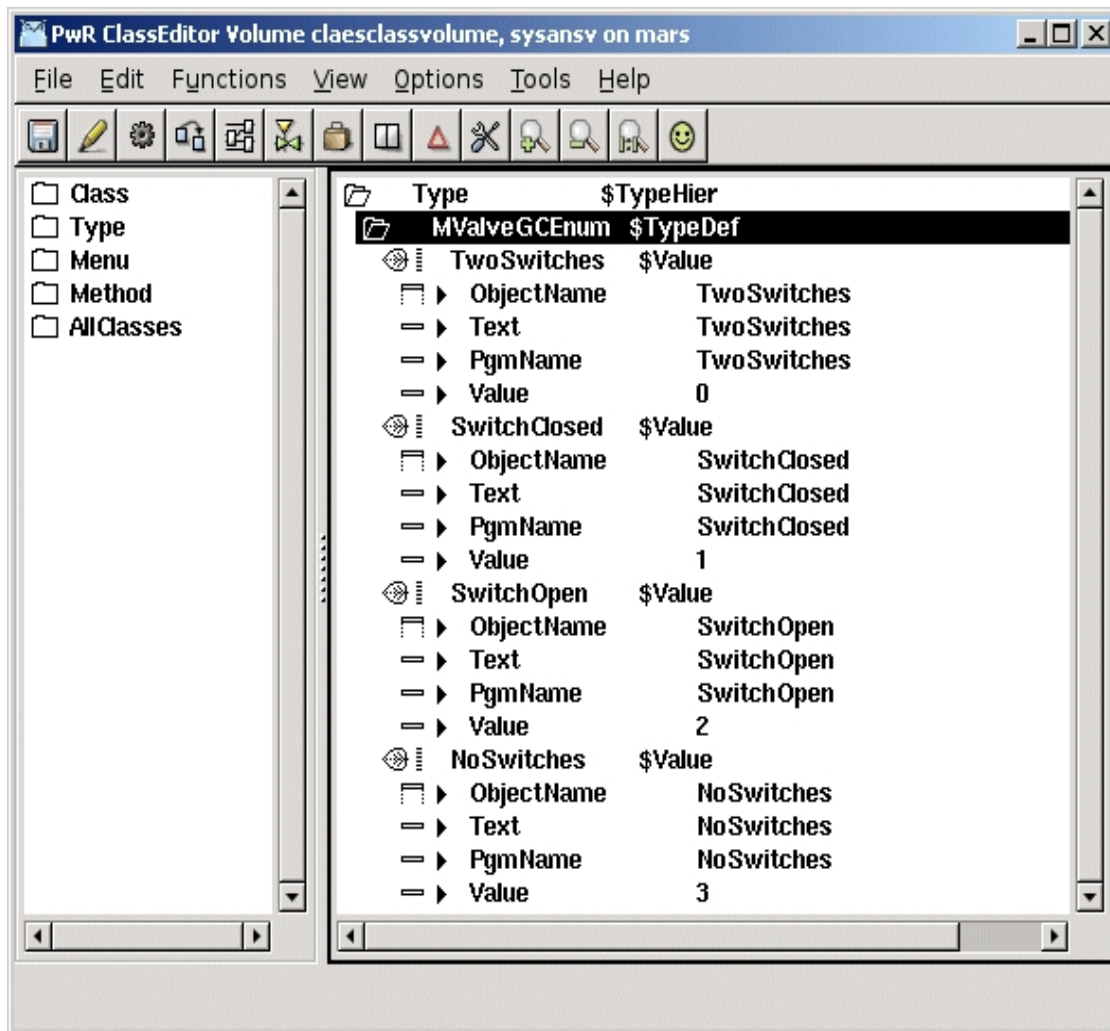
Enumeration type for GraphConfiguration

24.4.5.2 Functionobject

The functionobject is the interface of the component in the plc program. It defines inputs and outputs that can be connected to other functionobjects in the plc editor. Unlike an ordinary functionobject the code is also working with data in the main object.

The code can be written in plc-code or c-code.

plc-code
If you want to keep the code of the function object visible, and there is need of running PlcTrace in the code, it is suitable to use a functionobject with plc-code.

Create a $ClassDef object and name the object. Preferably use the same name as for the main object followed by the suffix 'Fo', e.g. MyComponentFo. Then activate Configure-ConnectedFo in the popupmenu.

Under RtBody a PlcConnect attribute of type AttrRef is created, that will contain a link to the main object, when an instance is connected in the plc-editor.

Configure inputs and outputs with $Input and $Output objects below the RtBody object. You

can also create $Intern objects, but this type of data is usually stored in the main object.
Note that the order of attribute objects should be $Input, $Intern, $Output.

The code is created by opening the plc editor for the Code object. In the code, you fetch
values from an input, by selecting the input attribute in the template object for the
functionobject in the navigator, and activate the connect function. Output is stored in a
similar way. When data should be fetched or stored in the main object, you select the attribute
in the template object of the main object. References to the function object are viewed in the
plc-code with the symbol $PlcFo, and references to the main object with the symbol $PlcMain.

If the object contains components, the function object of these components are put in the
plc-code.

If you have DisableAttr on signals or other attributes, this has to be handled with conditional
execution in the code. A signal that is disabled must not be read or written to in the code.
You use the object Disabled under the map Other, to evaluate if an attribute is disabled or not.
This can then be connected to a CArea object that handles the conditional execution.

Condition execution with Disabled and CArea

c-code
A function object with c-code is configured with a $ClassDef object. Name the object and then
activate Configure-ConnectedCCodeFo in the popupmenu.

Below RtBody, two attributes are created, PlcConnect of type AttrRef and PlcConnectP that is a
pointer. In PlcConnect, the reference to the main object is stored, when an instance is
connected in the plc editor. When the plc program is initialized in runtime, you fetch, with
help of the reference, a pointer to the main object. The pointer is stored in PlcConnectP.
This is done in the c code, that is separated in an init function that is executed at initialization
of the plc program, and an exec function that is executed at every scan. For the function object
MyComponentFo with the input In1 and In2, and the output Out2, the code is

```
void MyComponentFo_init( pwr_sClass_MyComponentFo  *o)
{
  pwr_tDlid dlid;
  pwr_tStatus sts;

  sts = gdh_DLRefObjectInfoAttrref( &o->PlcConnect, (void **)&o->PlcConnectP, &dlid);
  if ( EVEN(sts))
    o->PlcConnectP = 0;
}

void MyComponentFo_exec( plc_sThread      *tp,
                         pwr_sClass_MyComponentFo *o)
{
```

```
        pwr_sClass_MyComponent *co = (pwr_sClass_MyComponent *) o->PlcConnectP;

    if ( !co)
        return;

    o->Out = co->Value = co->Factor * (*o->In1P + *o->In2P);
}
```

24.4.5.3   Simulation object

A simulation object is used to simulate the process, both at normal conditions and when different errors occurs. The simulation object reads the output signals (Do, Ao, Io) in the main object, and sets values to the input signals (Di, Ai, Ii, Co). The object is a functionobject that can contain input and output attributes, but these are usually missing, and the object is working with data in the main object, and with internal attributes that configures the simulation and triggers various error conditions. The simulation object often has an object graph that is opened by the Simulate method of the main object.

A simulation object is connected to the main object by a connect method, in the same way as an ordinary function object. But the simulation class has another connect method than the Fo class. The main object should contain the attribute 'SimConnect' of type pwrs:Type-$AttrRef, into which the connection method will store the identity for the simulation object when a main instance object and a simulation instance object are connected.

A simulation class is created in the same way as a functionobject class, and can be written in c of plc-code. The class is preferably named with the same name as the main class, followed by the suffix 'Sim'.

Create a $ClassDef object and name the object. Then activate the Configure-ConnectedFo or Configure-ConnectedCCodeFo. Add any $Input, $Intern and $Output attributes, and write the code in plc or c-code. Change the connect method in GraphPlcNode to 26.

The object graph for simulation objects are often drawn with darkblue background and white text to easily be parted from other object graphs. Note that attributes in the main object can be referenced by the '&' notation, e.g. &($object.PlcConnect).IndError##Boolean.
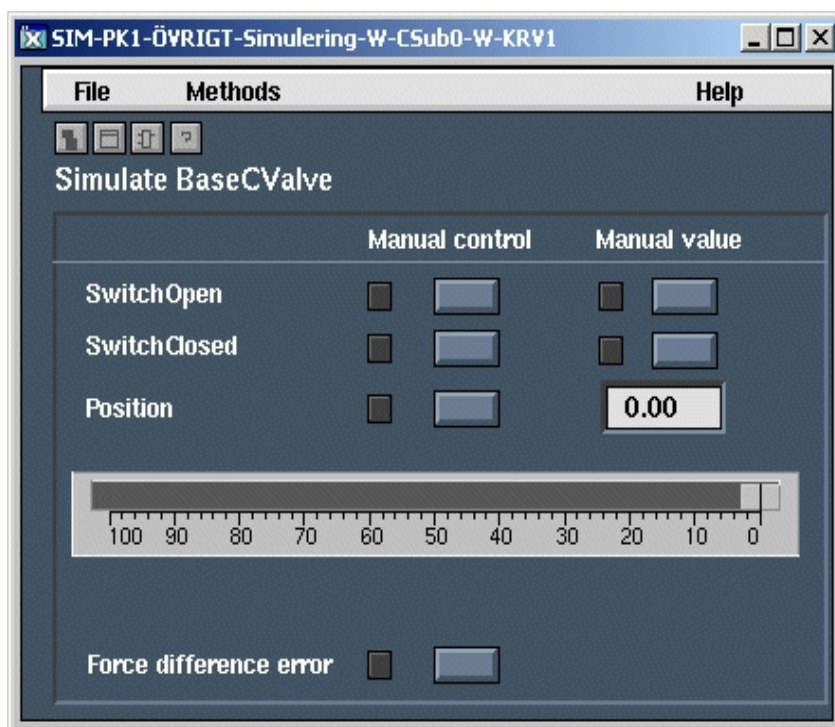
Object graph for a simulation object

24.4.5.4   I/O-module object

A main object for a component can contain signal objects of type Ai, Ai, Ii, Ao, Do, Io and
Co. The signals of an instance have to be connected to channel objects in the node hierarchy.
In, for example, Profibus you can create module objects, where the channels are adapted to the
dataarea that is received and sent on the bus. If the signals in a component are handled by
a module object, you can store symbolic references to the channel objects in the signal
objects. Then you only have to do one IoConnection between the component and the module object,
you don't have to connect each signal separately. The symbolic references are stored in the
template object of the template object of component, by connecting the signals in the template
object to the channels in the template object of the I/O module. The symbolic references are
of type $IoConnect, and are converted to real references at initialization of the I/O handling
in runtime.
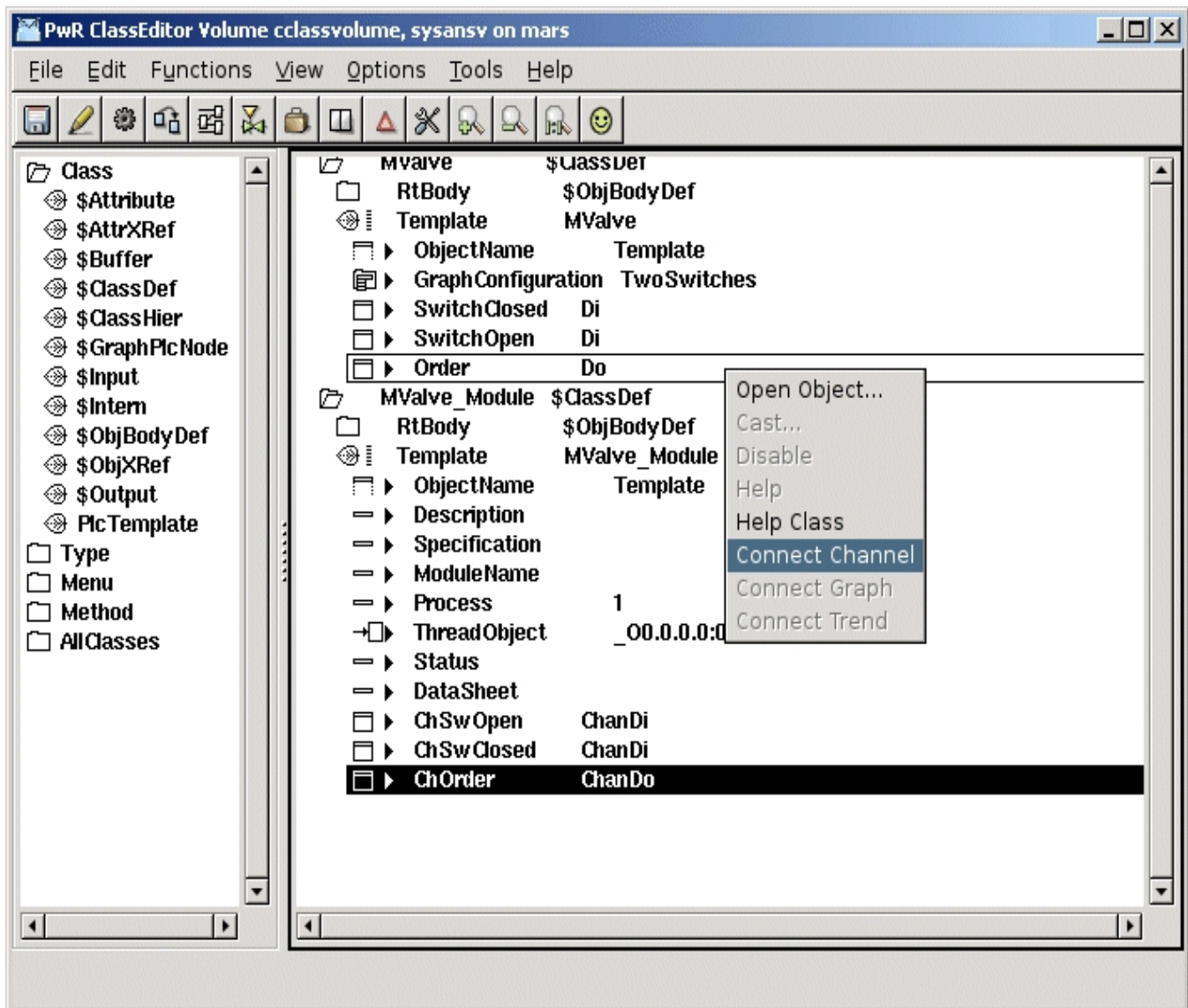
A signal is connected to a channel in an I/O module.

```
PwR ClassEditor Volume cclassvolume, sysansv on mars                        _ □ ×

File  Edit  Functions  View  Options  Tools  Help

[toolbar icons]

📂 Class                    ▲    📂  MValve          $ClassDef              ▲
  ◈ $Attribute                   📁  RtBody          $ObjBodyDef
  ◈ $AttrXRef                    ◈▤ Template          MValve
  ◈ $Buffer                      ▥▶ ObjectName         Template
  ◈ $ClassDef                    📖▶ GraphConfiguration  TwoSwitches
  ◈ $ClassHier                   ▥▤ SwitchClosed    Di
  ◈ $GraphPicNode                 ▭▶ Description
  ◈ $Input                       ▬▶ SigChanCon        $IoConnect:CClassVolume-Class-MValve_Module.ChSwOpen
  ◈ $Intern                       ▭▶ InitialValue      0
  ◈ $ObjBodyDef                   ▭  SigValue         0
  ◈ $ObjXRef                      ▬▶ DefGraph
  ◈ $Output                       ▬▶ DefTrend
  ◈ PicTemplate                   ▭▶ HelpTopic
  📁 Type                         ▭▶ DataSheet
  📁 Menu                         ▭▶ CircuitDiagram
  📁 Method                       ▭▶ Note
  📁 AllClasses               ▼    📖▶ FilterType        No
                                   ▭▶ Filter Attribute
◀          ▶                     ◀                                           ▶
```

Symbolic reference to channel object

24.4.5.5   Object graph

Object graph has the same name as the component, but with lower case. For classes in the ProviewR
base system you add the prefix 'pwr_c_'. The graphs are edited as normal in Ge. In the dynamic
you exchange the object name with '$object'. Object graph for objects in the base system are
drawn with the following guidelines.

Menu

There should be a menu with the pulldown menus File, Methods, Signals and Help.

File should have the entries

```
Print    Command      print graph/class/inst=$object
Close    CloseGraph
```

Methods should have the entries

```
Help            Invisible   $cmd(check method/method="Help"/object=$object)
                Command     call method/method="Help"/object=$object
Note            Invisible   $cmd(check method/method="Note"/object=$object)
                Command     call method/method="Note"/object=$object
Trend           Invisible   $cmd(check method/method="Trend"/object=$object)
                Command     call method/method="Trend"/object=$object
Fast            Invisible   $cmd(check method/method="Fast"/object=$object)
                Command     call method/method="Fast"/object=$object
Help            Invisible   $cmd(check method/method="Photo"/object=$object)
                Command     call method/method="Photo"/object=$object
DataSheet       Invisible   $cmd(check method/method="DataSheet"/object=$object)
                Command     call method/method="DataSheet"/object=$object
```

```
Hist Event...     Invisible    $cmd(check method/method="Hist Event..."/object=$object)
                  Command      call method/method="Hist Event..."/object=$object
Block Events...   Invisible    $cmd(check method/method="Block Events..."/object=$object
                  Command      call method/method="Block Events..."/object=$object
RtNavigator       Invisible    $cmd(check method/method="RtNavigator"/object=$object)
                  Command      call method/method="RtNavigator"/object=$object
Open Object       Invisible    $cmd(check method/method="Open Object"/object=$object)
                  Command      call method/method="Open Object"/object=$object
Open Plc          Invisible    $cmd(check method/method="Open Plc"/object=$object)
                  Command      call method/method="Open Plc"/object=$object
Circuit Diagram   Invisible    $cmd(check method/method="Circuit Diagram"/object=$object
                  Command      call method/method="Circuit Diagram"/object=$object
HelpClass         Invisible    $cmd(check method/method="HelpClass"/object=$object)
                  Command      call method/method="HelpClass"/object=$object
```

Signals should contain all signals in the component and open the object graph for each signal.

Example

```
SwitchOpen Di    Command      open graph/class /inst=$object.SwitchOpen
SwitchClosed Di  Command      open graph/class /inst=$object.SwitchClosed
Order Do         Command      open graph/class /inst=$object.Order
```

The Help menu should contain Help and HelpClass

```
Help             Command      call method/method="Help"/object=$object
HelpClass        Command      call method/method="HelpClass"/object=$object
```

 Toolbar
The toolbar contains buttons for the methods. The dynamics are the same as in the methods menu
above. To the right, there is also a button for the object graph of the simulate object. This
has the dynamic

```
Invisible    $cmd(check method/method="Simulate"/object=$object)
Command      call method/method="Simulate"/object=$object
```

Below the method buttons there are two textfields that display the Description and Specification
attributes in the component with the dynamic

```
Description      Value.Attribute $object.Description##String80
                 Value.Format    %s
Specification    Value.Attribute $object.Specification##String80
                 Value.Format    %s
```

On the lowest row in the graph, any Notes message are viewed, with a button to change or remove
the message.

```
Notes button     Invisible       $object.Note##String80
                 Command         call method/method="Note"/object=$object
Notes text       Value.Attribute $object.Note##String80
                 Value.Format    %s
```
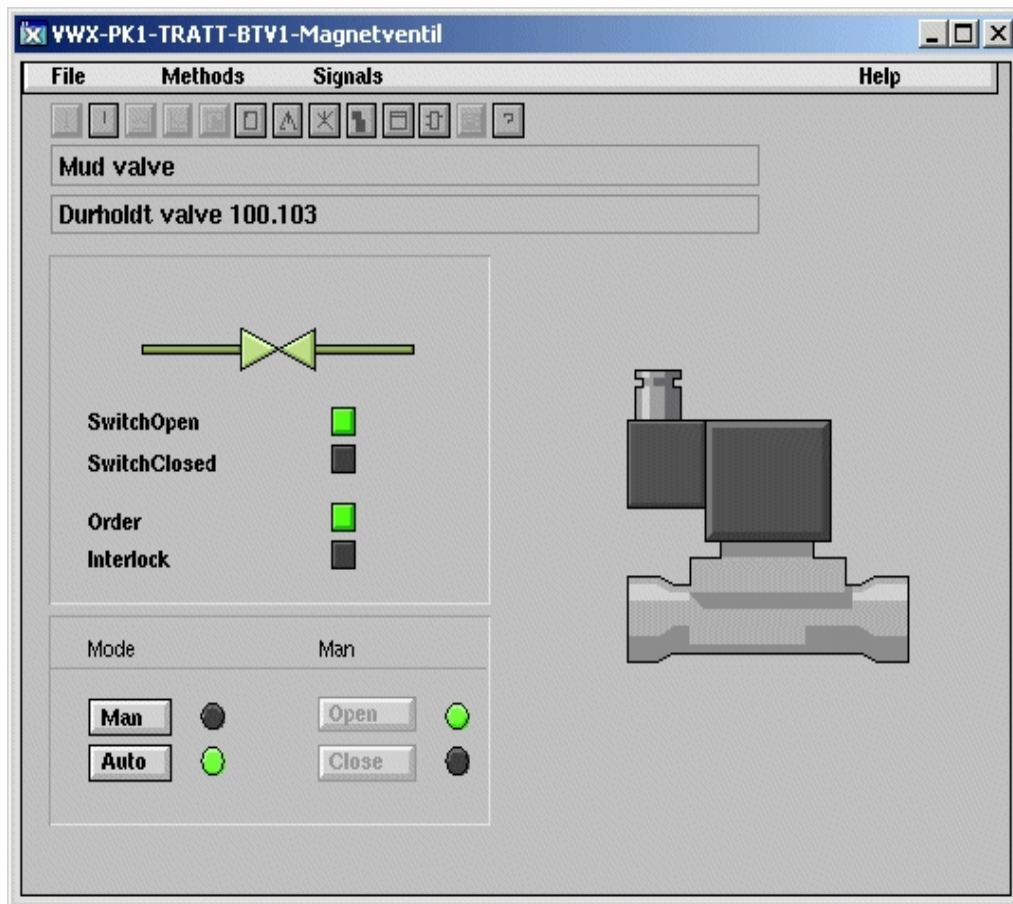
Object graph

24.4.5.6 Graphic symbol

The graphic symbol is drawn in Ge and given the default dynamic HostObject. In the host object dynamic, differenty types of dynamic are connected to attributes in the object. The objectname is exchanged to '$hostobject' in the dynamics. Often you create the attributes IndWarning and IndError in the main object, and color the symbol yellow or red, or flashing red, when these are set.

The grahic symbol is drawn in Ge witch HostObject default dynamic.

24.5 Build the classvolume

When building the classvolume, a loadfile and two structfiles are created.

Loadfile
The loadfile is stored in $pwrp_load and has the same name as the volume, with lower case.
The filetype is .dbs, e.g the loadfile for the volume CVolMerk1 is $pwrp_load/cvolmerk1.dbs.

The time when the loadfile is created is stored in the file. Furthermore the version of other
classvolumes that the loadfile is dependent on are stored. At runtime startup, there is a check
that the current versions in the system coincide with the versions registered in the loadfile.
If any version doesn't coincide, you get the message 'Version mismatch' and the startup is
aborted.

You can display the version of a loadfile, and the versions of the dependent volumes with
wb_ldlist.

```
$ wb_ldlist $pwrp_load/cvolmerk1.dbs
Volume     CVolMerk1      21-DEC-2007 13:52:05.22 (1198241525,227130443) 25364
VolRef     CVolMerk1      21-DEC-2007 13:52:05.22 (1198241525,227130443) 25364
VolRef     pwrs           12-DEC-2007 08:35:06.98 (1197444906,983976467) 1
VolRef     pwrb           12-DEC-2007 08:35:09.93 (1197444909,930182284) 2
VolRef     BaseComponent  12-DEC-2007 08:35:26.92 (1197444926,926679004) 10
```

Structfiles
When building a classvolume, two includefiles are generated, a .h-file and a .hpp-file.

If the classvolume contains functions objects, or classes that are used in CArithm or
DataArithm objects, you have to include the .h file in $pwrp_inc/ra_plc_user.h.

Update Classes
When the classvolume is built, you have to update the classes in the root or sub volumes in
the project. The update is activated in the configurator for the root or sub volume, from the
menu with 'Function/Update Classes'. If a class is changed, instance objects of the class are
adapted to the changed class. All references to instances of the class will also be updated.

24.6   Documentation of classes

For $ClassDef and $Attribute objects, there is a documentation block, that is filled in from
the object editor. The documentations block, together with the class description, is used
when class documentation is generated to xtthelp or html format when the class volume is
built.

The documentation block for the $ClassDef object should contain a description of the class,
and the documentation block for the $Attribute object a description of the attribute.

24.6.1   Generate Xtt helpfiles

Helpfiles for xtt is generated with the command

```
co_convert -xv -d $pwrp_exe/ $pwrp_db/userclasses.wb_load
```

The command generates a helpfile $pwrp_exe/'volumename'_xtthelp.dat, and it is proper to put
a link to the file in the xtt helpfile for the project $pwrp_exe/xtt_help.dat:

Example for the classvolume cvoltank

<topic> index

...

User classes<link>cvoltank,"",$pwrp_exe/cvoltank_xtthelp.dat

</topic>

...

<include> $pwrp_exe/cvoltank_xtthelp.dat

24.6.2 Generate html documentation

html files are generated by the command

```
co_convert -wv -d $pwrp_web/ $pwrp_db/userclasses.wb_load
```

The command generates, among others, the file $pwrp_web/'volumename'_index.html that contains the start page for the class documentation. This, together with the other files ($pwrp_web/'volumename'_*.html) should be copied to a proper directory of the web server.

A link to the documentation is made with a WebLink object pointing at the URL 'volumename'_index.html.

If you want to be able to show the c struct for the classes, you convert the h-file with co_convert

```
co_convert -cv $pwrp_web/ $pwrp_inc/'volymename'classes.h
```

If you also want to be able to display the code of plc-objects, you have to add aref tags in the c or h file of the code, and convert it with

```
co_convert -sv -d $pwrp_web/ 'filename'
```

24.6.3 ClassDef

Example

```
@Author Homer Simpson
@Version 1.0
@Code ra_plc_user.c
@Summary Brief description of this class
Description of
this class.

See also
@link Example plat.html
@classlink AnotherPlate cvolvhxn2r_anotherplate.html
```

Tags

Author   Author or the class description

Version	Version of the class
Code	File that contains the code for the class
Summary	Summary
Link	Arbitrary link
Classlink	Link to another class
wb_load syntax	

24.6.3.1 @Author

Author. Optional.

Syntax
```
@Author 'name of author'
```

24.6.3.2 @Version

Version. Optional.

Syntax
```
@Version 'version number'
```

24.6.3.3 @Code

For classes with plc-code you can state the name of the c-file. Optional.
Also the c-file has to be converted by the command: co_convert -c -d $pwrp_web/'filename'

Syntax
```
@Code 'filename'
```

24.6.3.4 @Summary

Short description in one line. Optional.
This is shown in the indexfile in the xtt helpfile. Not used in html.

Syntax
```
@Summary 'text'
```

24.6.3.5 @Link

A link to an arbitrary URL. Is only displayed in the html documentation, not in Xtt.
The link should be placed after the description of the class.

Syntax
```
@Link 'URL'
```

24.6.3.6 @Classlink

A link to another class. This link works in both html and xtt.
The link should be placed after the description of the class.

Syntax
```
@Classlink 'html-filename'
```

24.6.3.7 wb_load syntax

The documentation of a class is written above the $ClassDef row.

```
!
!/**
!  @Author Homer Simpson
!  @Version 1.0
!  @Code ra_plc_user.c
!  @Summary Brief description of this class
!  Description of
!  this class.
!
!  See also
!  @link Example plat.html
!  @classlink AnotherPlate cvolvhxn2r_anotherplate.html
!*/
!
Object      Plat    $ClassDef 1
```

!/**

Start of a documentation block.
All the text between !/** and !*/ will be written as a description of the class.

!*/

End of a documentation block.

24.6.4   Attribute

Example

```
@Summary Plate length
A more detailed description
of the attribute Length...
```

@Summary

Short description in one line. If there is a @Summary, this text is put into the table of
attributes in the html file. If there is no summary, the whole description is written instead.
Not used in xtt.

wb_load syntax

24.6.4.1   wb_load syntax

Documentation of an attribute is written above the $Attribute, $Input, $Output or $Intern line.

```
!/**
!  @Summary Plate length
!  A more detailed description
!  of the attribute Length...
!*/
Object   Length $Attribute 3
  Body          SysBody
    Attr        TypeRef = "pwrs:Type-$Float32"
  EndBody
```

```
        EndObject
```

!/

Start of a documentation block.
All the text between !/** and !*/ will be written as a description of the attribute.

!*/

End of a documentation block.

24.6.5   Syntax for c- and h-files

If you want to use the links to c- and h-files, these also have to be converted to html.
There is also a function to add bookmarks.

The structfile for the classes is automatically generated with bookmarks.

```
/**
    MyPlcClass

    Description for the class that is not displayed anywhere but in the code.

    @aref MyPlcClass MyPlcClass
*/
void MyPlcClass_exec(...)
```

@aref

@aref has to be placed inside a /*_* ... */ block. Inside the block, there can also be comments
that are not handled by the converter.

Syntax
```
@aref 'bookmark' 'text'
```

25 Administration

25.1 Users

To gain access to the ProviewR development and runtime environment you need to login with username and password. Users are kept in the user database and are granted privileges which state the users authority to make changes in the system.

Systems that share the same users are grouped into a system group, and the users for this group are defined. You can also build a hierarchy of system groups where child groups inherit the users of their parent, and additional users can be defined for each child.

A system is connected to a system group by the SystemGroup attribute in the $System object. The notation for a system group in a hierarchy is the names of the group separated by a period, for example 'Main.West.F1'.

In the example below Eric is responsible for all the systems in the plant, and is defined on the highest level in the hierarchy. Nils is working with the west side of the plant and is defined on the 'West' system group. Finally, Carl working with the systems in the F1 part of the plant. All system groups have the attribute UserInherit, which states that a child group inherits all the users of the parent.
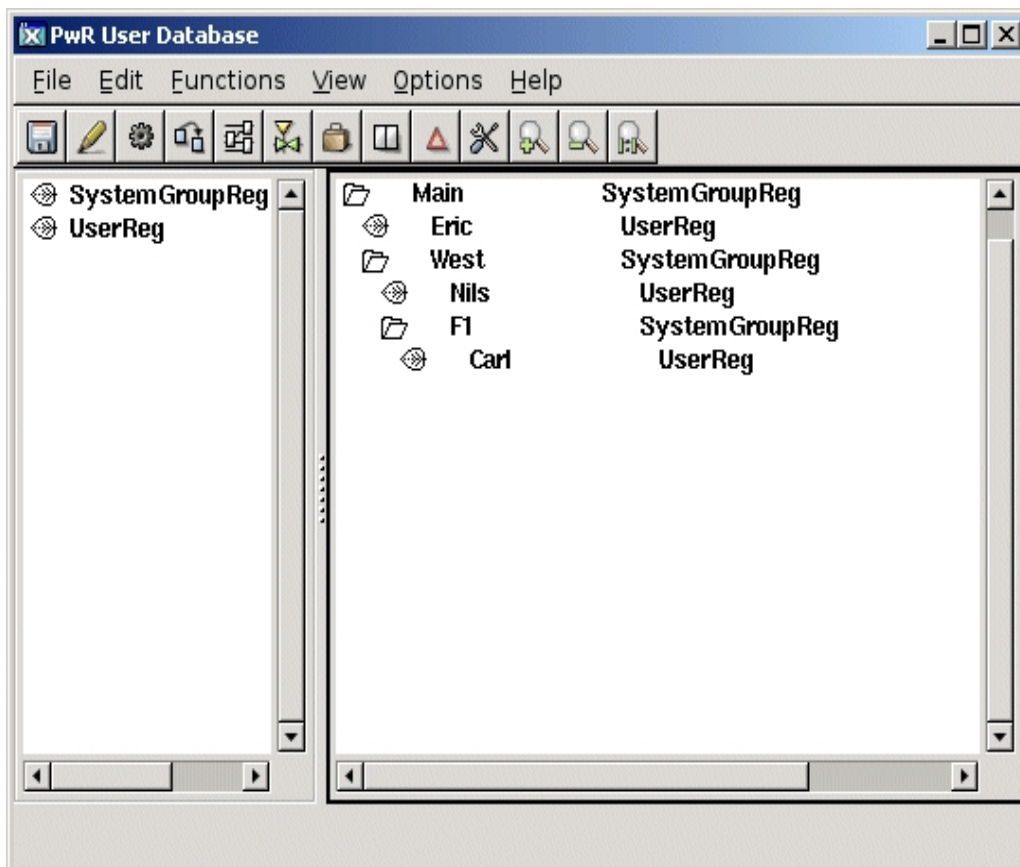
Users and systemgroups are created in the Administrator:

- Start the administrator with the command 'pwra'
- Enter the UserDatabase from the menu 'File/Open/UserDatabase'.
- Login by entering the login command. Open the login prompt from the menu 'Functions/Command'
  and enter 'login /adm' on the command line. If the systemgroup 'administrator' is present
  you also have to add username and password to a user defined in the administrator systemgroup.
- Enter edit mode from the menu 'Edit/Edit mode'.
  Systemgroups and users are represented by object of the classes SystemGroupReg and
  UserReg, that are displayed in the palette to the left. An object is created by selecting
  a class in the palette. After that, you click with the middle mousebutton on the future
  sibling or parent to the new object. If you click on the map/leaf in the destination object,
  the new object is placed as the first child, if you click to the right of the map/leaf,
  it is placed as a sibling.
- Create a systemgroup by selecting 'SystemGroupReg' in the palette, and click with MB2
  (the middle mousebutton) in the right window. Open the SystemGroupReg object and enter
  name ant attribute for the system group. Enter the complete hierarchy name, e.g.
  'Main.West'.
- Create a user by selecting 'UserReg' in the palette and click with MB2 on the map/leaf
  of the SystemGroupReg object that the UserReg should be a child of. Open the object and
  enter username, password and privileges for the user.
- Save.
- Logout with the command 'logout'.

The user database resides in the directory $pwra_db.

25.2   Register Volumes

All volumes in a network must have a unique volumname and volume identity. To assure this,
all volumes are registered in a global volume.

The registration is done by the administrator:

- Start the administrator with the command 'pwra'
- Enter volume mode from the menu 'File/Open/GlobalVolumeList'.
- Login as administrator.
- Enter edit mode from the menu 'Edit/Edit mode'.
  Volumes are registered by objects of class VolumeReg, that are displayed in the palette to
  the left. In the palette, there is also the $Hier class, that can be used to order the
  VolumeReg objects in a tree structure.
- Create a VolumeReg object, open the object and enter volumename (equals objectname),
  volumeidentity and project.
- Save.
- Logout with the command 'logout'.

Volume name

The name of the volume, a unique name with max 31 characters.

Volume Identity

The volume identity is a 32 bit word specified in the form v1.v2.v3.v4 where v1, v2, v3 and v4
are numbers in the interval 0-255. Dependent on the class of the volume, the numbers can be

chosen in separate intervals.

RootVolumes   0. 1-254. 1-254. 1-254
User ClassVolumes   0. 0. 2-254. 1-254

The DirectoryVolume always has the identity 254.254.254.253

25.3   Create project

A project is a number of nodes and volumes that share the same development environment. Usually it consists of some process stations and a couple of operators stations that control a part of the plant, but there are no restrictions in the size of a project. You can choose to have each node in its own project or all the nodes in the same project.

- All the nodes in a project (on the same bus) have a nethandler link between each other.
- All the volumes and nodes share the same directory tree.
- All nodes have to be upgraded to new ProviewR versions at the same time.

A common size is 1-10 nodes in a project. Too many nodes will increase the communication overhead and make it harder to upgrade the project.

Create the project in the administrator:

- Start the administrator with the command 'pwra'.
- The projectlist is shown as default when starting the administrator. It can also be opened
  from the menu (File/Open/ProjectList).
- Login as administrator.
- Enter edit mode from the menu 'Edit/Edit mode'.
  Projects are represented by objects of class ProjectReg, that are displayed in the palette
  to the left. $Hier objects can be used to order the ProjectReg objects in a tree structure.
- Create a ProjectReg object and enter project name, base version, path and description.
- The project is created when saving. First you have to confirm the changes.
- Save and logout.

Project name

A project has a project name that identifies the project in the development environment. It is similar the system name that identifies the project in the runtime environment, but it doesn't have to be the same. Actually a system can have several projects in the development environment. When upgrading or making a major modification of the system, it is advisable to take a copy of the project and keep the currently running version of the system available for minor corrections. The copy is then created under a new project name, though it has the same system name.

Base

ProviewR is a multiversion system, i.e. different versions of ProviewR can be installed in the same development environment and projects of different ProviewR versions can coexist in the same development environment. A project points at a ProviewR base, e.g. V3.4, V4.0, and when creating a project you have to choose which base the project should point at.

Path

The project consists of a directory tree where databases, source files, archives etc are stored. The path is the root directory of this tree.

26 Revisions

A revision is a state in the development environment that is stored in a version control system. The revision should include all source files to make it possible to restore a revision. A revision can be used to review the state when the revision was created, and also to build runable systems from the restored source to test or run a restored revision in production.

The currently supported version control system is Git. A new repository will be created on $pwrp_root/src when the first revision is created.

Revision window

Revisions are handled from the Revision window that is opened from the File menu in the configurator.

Create a revision

A new revision is created from Functions/New Revison in the menu. New revisions can only be created if the currently checked out revision is the end point of a branch.
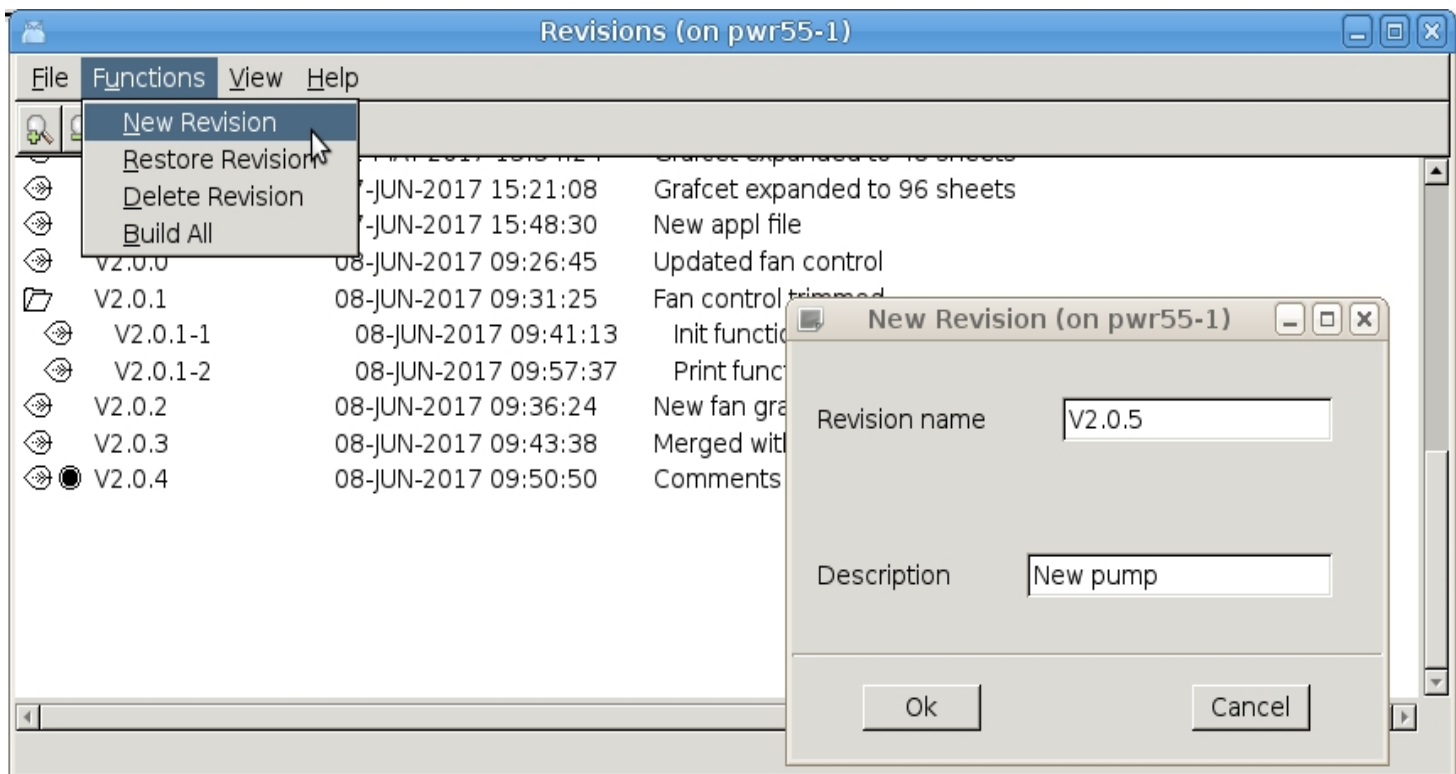
Fig Create a new revision

Restore a revision

A revision is restored from Functions/Restore in the menu. Select the revision to restore first.
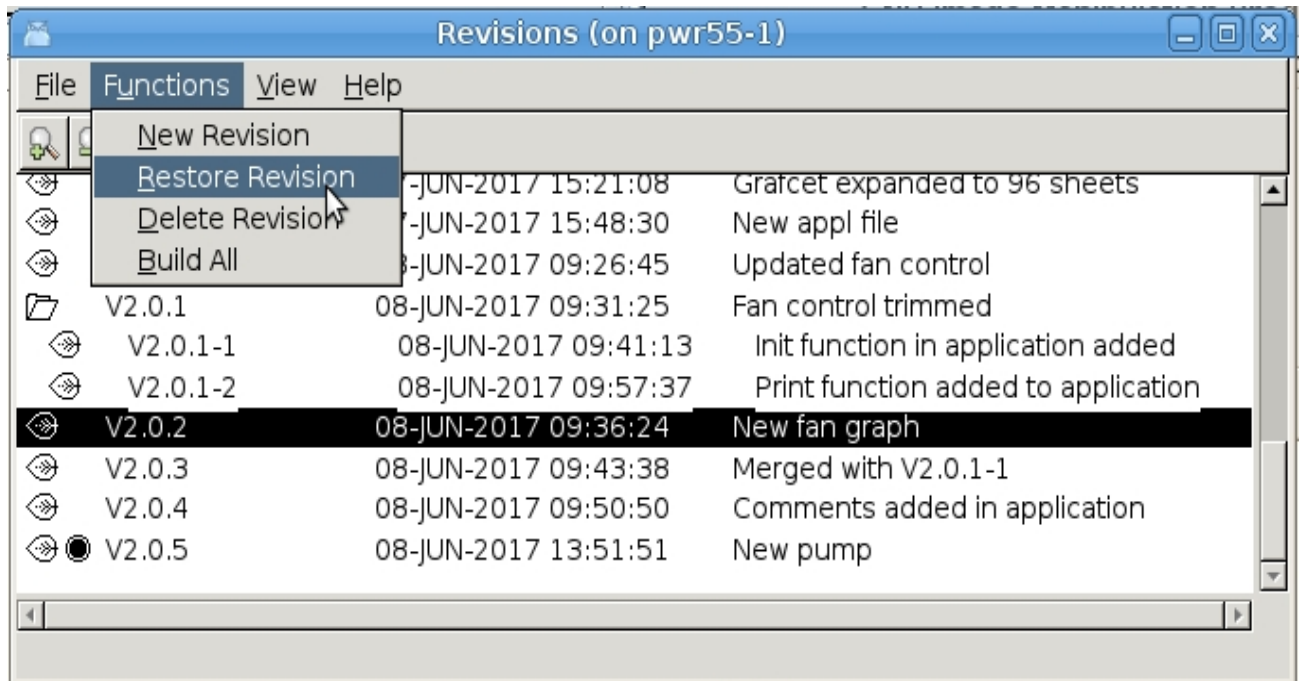
Fig Restore a revision

When a revision is restored the checked out revision is marked with a radio button in the list.
Also in the configurator title bar, the current revision is written.
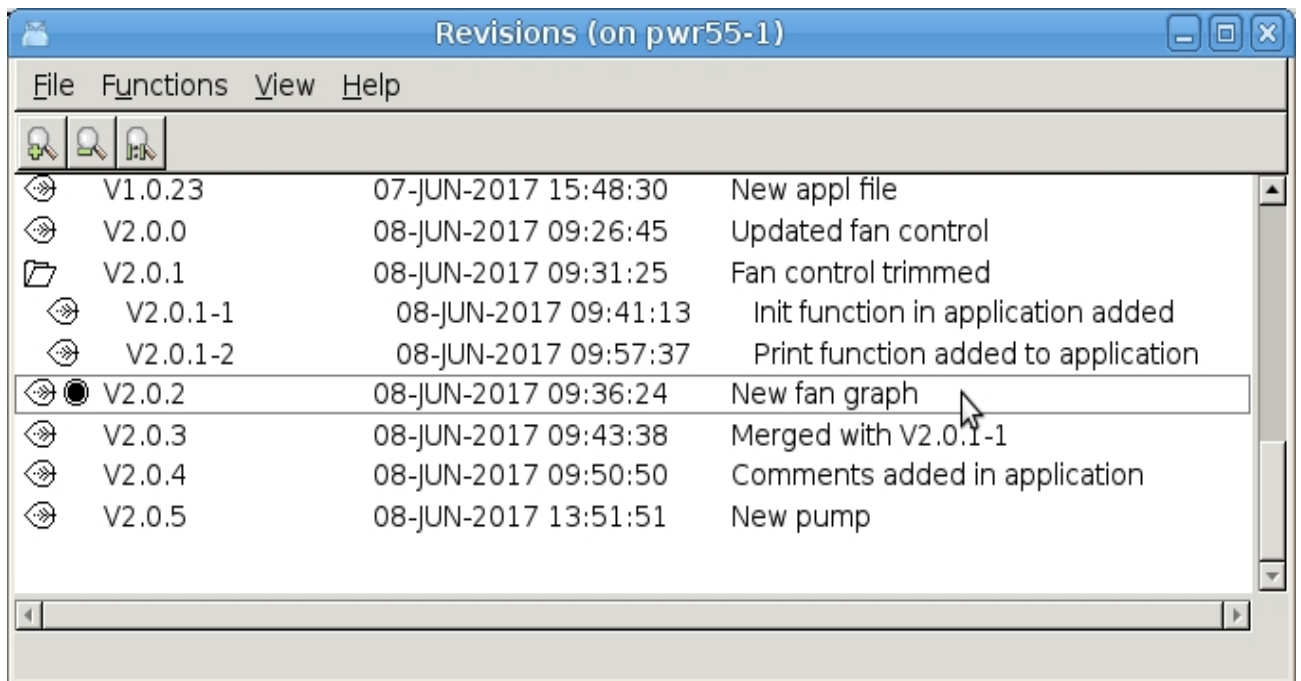
Fig An old revision is restored

Create a branch

If an old revision in the main line is restored, a branch can be created by creating a new revision from this point. In the figure above, a branch has be created from revision V2.0.1. New revisions can also be created from the end point of the branch. In the figure above, a new revision can be created from V2.0.1-2 that is an end point, but not from V2.0.1-1.

New files

New files, e.g. c-applications, are not automatically added to the git repository.
This has to be done manually. New files are displayed by Git as untracked files in the 'git status' command. Untracked files can be added with the 'git add' command and will then be included in the next revision.

Note! It's important to add new files to be able to restore a complete revision.

Example
```
> git status
On branch B_V2.0.5
Untracked files:
  (use "git add <file>..." to include in what will be committed)

      appl/ra_appl.c

> git add appl/ra_appl.c
```

27 Tools

pwrc	Command interface to development database.
wb_ge	Ge editor
co_help	Help window.
pwr_user	Command line interface to user database.
wb_ldlist	Check loadfile versions.

27.1 pwrc

Command interface to development database.

```
pwrc        ProviewR workbench commands

Arguments:
  -v 'volume'   Load volume 'volume'.
  -h            Print usage.

  Other arguments are treated as a command and passed to the command parser
  If a command is given as an argument, the command will be executed and the
  program is then terminated.
  If no command is given, pwrc will prompt for a command.

Examples:
  $ pwrc -v MyVolume
  pwrc>

  $ pwrc -a show volume
  directory      Attached Db  $DirectoryVolume 254.254.254.253
  MyVolume                Db  $RootVolume         0.1.99.20
```

27.2 co_help Help window

Open a ProviewR help file. By default the project help file is opened.

```
>co_help

Usage:

  co_help [-t 'topic'] [-s 'sourcefile'] [-b 'bookmark']

Arguments:

  -t    Help topic, default 'index'
  -s    Source helpfile
  -b    Bookmark
```

```
-l    Language, e.g sv_se
-c    Open Configuration help
-d    Open Designer's Guide
-g    Open Ge Reference Manual
-o    Open Operator Help
```

27.3   wb_ge Ge editor

Start the Ge editor without opening a database.
Useful when you fast want to open a Ge graph, but will not make any database
connections.

```
> wb_ge ['pwg-file']
```

27.4   pwr_user

You use pwr_user to create systemgroups and users in the user database.
The configuration is performed with commands.

pwr_user is started from the command prompt.

Below is a description of the different commands available to create, modify and list
systemgroups and users.

add group	Add a system group
add user	Add a user
get	Get a user
list	List systemgroups and users
load	Load the latest saved database
modify group	Modify a system group
modify user	Modify a user
remove group	Remove a system group
remove user	Remove a user
save	Save
su	Login as super user

27.4.1   add

add group
add user

27.4.1.1   add group

Create a systemgroup

```
pwr_user> add group 'name' [/nouserinherit]
```

/nouserinherit   The attribute UserInherit is not set for the systemgroup.
                        As default UserInherit is set.

27.4.1.2   add user

Create a user.

```
pwr_user> add user 'name' /group= /password= [/privilege=]
 [/rtread][/rtwrite][/system][/maintenance][/process]
[/instrument][/operator1][/operator2]...[oper10][/devread]
[/devplc][/devconfig][/devclass]
```

/group	Systemgroup of the user
/password	Password of the user
/privilege	Privileges if this is supplied as a mask, i.e an integer value
/rtread	The user is granted RtRead
/rtwrite	The user is granted RtWrite
/system	The user is granted System
/maintenance	The user is granted Maintenance
/process	The user is granted Process
/operator1	The user is granted Operator1
...	
/operator9	The user is granted Operator9
/operator10	The user is granted Operator10
/devread	The user is granted DevRead
/devplc	The user is granted DevPlc
/devconfig	The user is granted DevConfig
/devclass	The user is granted DevClass

27.4.2 get

Fetches a user with an algorithm used in runtime.

```
pwr_user> get 'username' /group= /password=
```

27.4.3 list

Lists systemgroups and users.

```
pwr_user> list
```

27.4.4 load

Loads the latest saved database and reverts the current session.

27.4.5 modify

```
modify group
modify user
```

27.4.5.1 modify group

Modify a systemgroup.

```
pwr_user> modify group 'name' /[no]userinherit
```

/userinherit	Sets the attribute UserInherit that states that the systemgroup inherits users form its parent in the systemgroup hierarchy. Negated with /nouserinherit

27.4.5.2 modify user

Modify a user.

```
pwr_user> modify user 'name' /group= [/password=][/privilege=]
 [/rtread][/rtwrite][/system][/maintenance][/process]
     [/instrument][/operator1][/operator2]...[oper10][/devread]
     [/devplc][/devconfig][/devclass]
```

/group	Systemgroup of the user
/password	Password of the user
/privilege	Privileges if this is supplied as a mask, i.e an integer value
/rtread	The user is granted RtRead
/rtwrite	The user is granted RtWrite
/system	The user is granted System
/maintenance	The user is granted Maintenance
/process	The user is granted Process
/operator1	The user is granted Operator1
...	
/operator9	The user is granted Operator9
/operator10	The user is granted Operator10
/devread	The user is granted DevRead
/devplc	The user is granted DevPlc
/devconfig	The user is granted DevConfig
/devclass	The user is granted DevClass

27.4.6   remove

remove group
remove user

27.4.6.1   remove group

Remove a system group.

```
pwr_user> remove group 'name'
```

27.4.6.2   remove user

Remove a user.

```
pwr_user> remove user 'name' /group=
```

27.4.7   save

Save the current session.

```
pwr_user> save
```

27.4.8   su

Login as super user. As super user you can see passwords for users when listing the database.
su requires password.

```
pwr_user> su 'password'
```

27.5   wb_ldlist

Display version of volumes in dbs-files.
Used to investigate version mismatch.

> wb_ldlist <dbs-file>

Example
```
wb_ldlist $pwrp_load/volpwrdemo.dbs
Volume      VolPwrDemo       27-MAR-2014 11:06:48.67   0.254.254.200    RootVolume
VolRef      VolPwrDemo       27-MAR-2014 11:06:48.67   0.254.254.200    RootVolume
VolRef      pwrs             14-FEB-2014 16:57:21.19   0.0.0.1          ClassVolume
VolRef      pwrb             14-FEB-2014 16:57:24.82   0.0.0.2          ClassVolume
VolRef      BaseComponent    14-FEB-2014 16:57:52.68   0.0.0.10         ClassVolume
VolRef      OtherManufacturer 14-FEB-2014 16:58:22.73  0.0.250.1        ClassVolume
VolRef      ABB              14-FEB-2014 16:58:10.44   0.0.250.2        ClassVolume
VolRef      Profibus         14-FEB-2014 16:57:34.40   0.0.250.7        ClassVolume
VolRef      Inor             14-FEB-2014 16:58:16.18   0.0.250.8        ClassVolume
VolRef      OtherIO          14-FEB-2014 16:57:55.31   0.0.250.10       ClassVolume
```

28 OPC

ProviewR has implemented the OPC XML/DA protocol for data exchange with other automation software. For more information about OPC see www.opcfoundation.org.

28.1 OPC XML/DA Server

An OPC XML/DA Server is a web service from which an OPC XML/DA Client can get information of a ProviewR system. A opc client can, for example, browse the object hierachy, read and write attribute value, and subscribe to attributes.

The opc server implements the http protocol as well and is not connected to a web server. The port number of the opc_server is set to 80, the URI for the web service is on node 'mynode' is

```
http://mynode
```

If a web server is present, this normally has allocated the port 80, and another port has to be chosen for the opc_sever. If port 8080 is chosen, the URI will be

```
http://mynode:8080
```

Browsing

The OPC XML/DA browsing supports branches and items. The item contains a value, while the branch is an hierarchical component without a value. There is no support for objects, so an object has to be implemented as a branch, while an attribute is implemented as an item. Also arrays are implemented as branches, while array elements (that is not a class) is implemented as an item.

Threads

If the opc client uses the HoldTime and WaitTime attributes in the SubscriptionPolledRefresh request, the opc server has to be multi threaded, that is, for every request, a new thread is created. If the HoldTime and WaitTime is not used (as in the ProviewR opc client), all requests can be handled in a single thread, which is less time consuming. Multithreads or not are configured in the configuration object for the opc server. The default value is 'IfNeeded' which turns on the multithreading for a client if HoldTime or WaitTime are detected.

Client access

To gain access to a ProviewR opc server, the ip address of the client has to be configured in the configuration object for the opc server. Here you can also choose if the client has ReadWrite or ReadOnly access, where ReadOnly allows the client to read and subscribe to attribute values, while ReadWrite also is allowed to write attribute values.

Buffering of subscriptions

The server does not support buffering of subscriptions.

Configuration

The opc server is configured with a Opc_ServerConfig object that is placed in the Node hierarchy. The configuration object will cause a server process (opc_server) to start at ProviewR startup.

28.1.1 OPC XML/DA Client

The ProviewR opc client is implemented as an extern volume, which is mounted somewhere in the object tree of the root volume. Under the mount object, the branches and items of the server are displayed with special opc objects. An Opc_Hier object represents a branch and Opc_Int an item with an integer value, Opc_Boolean an item with a boolean value etc. If an item object is opened, the item values are displayed in a Value attribute, and some other properties as description, lowEU, highEU, engineeringUnit, lowIR and highIR are displayed. When the object is opened a subscription is started, and the value is continuously updated. For integer and float items there is also an object graph that displays a trend of the value.

With the opc client you can

- browse the branches and items in Xtt, and also display item values and set item values.
- subscribe item values and display them in a Ge graph.
- fetch item values into the plc logic and also write values to items.

The opc client requires that name browsing is implemented in the opc server.

Ge

An item value can be displayed in a Ge graph by using the name in the extern volume. For example, if the mount object for the extern volume is 'Ext-P1', and the local name of the item is

```
/P1/Signals/Ai22
```

the signal name in Ge will probably be (this is dependent of the browsing function of the server)

```
Ext-P1-P1-Signals-Ai22.Value##Float32
```

presuming that it is a float datatype.

Plc

Item values can also be handled in the plc program, using the GetExt... and CStoExt... objects. The objects normally used for getting and storing attributes GetDp, GetAp, StoDp, StoAp etc. can not be used, as they require that the referenced name is known in the development environment, which is not the case for most extern volumes. In the Ext objects, the reference is made with a name string, making it possible to enter the item name. To get the value of the item in the previous example, you should use a GetExtFloat32 object, and the object name should be

```
Ext-P1-P1-Signals-Ai22.Value
```

To store a value in an item, lets say /P1/Signals/Ao5, you use a CStoExtFoat32. This object makes a conditional storage, and only on a positive edge of the condition. Compare with the CStoAp, where the value is stored, as long as the condition is true. The reference name in the CStoExtFloat32 object in this case should be

```
Ext-P1-P1-Signals-Ao5.Value
```

Client process

For each opc client-server connection a client process has to be started. The executable for this process is opc_provider that has the arguments

1. Opc server URL
2. Extern volume id
3. Extern volume name
4. Server identity (optional, default 200)

Configuration

Register ExternVolume

Register the externvolume in the GlobalVolumeList with a volume name and identity.

Application file

Add a line in the application file to start the opc_provider. Here is an example for an opc client connecting to the opc server 'http://servernode:8080'. The registered externvolume has the name MyOpcVolume with volume id 0.1.99.55

```
opc_provider, opc_provider, noload, run, opc_provider, 9, nodebug,
   http://servernode:8080 0.1.99.55 MyOpcVolume
```

If item values are fetched into the plc, the priority should be set to 4 (sixth argument).

Mount object

Create a mount object in the plant hierarchy of the rootvolume, and insert the objid of the volumeobject in the externvolume into the Object attribute. In the example above this objid is _O0.1.99.55:0.

Hint

The application file resides on $pwrp_load and has the name

```
$pwrp_load/ld_appl_'nodename'_'busnumber'.txt
```

where nodename is the name of the node, and busnumber the qcom bus number. If the node is 'mynode' and the busnumber is 507, the filename will be

```
 $pwrp_load/ld_appl_mynode_507.txt
```

29 Commands

build	Build node, volume or object
check classes	Check if any classes need update
close graph	Close a Ge graph
compile	Compile plcpgm
configure card	Configure a card object
connect	Connect signal and channel
copy	Copy selected object trees
copy object	Copy an object
create bootfiles	Create bootfiles
create crossreferencefiles	Create crossreferencefiles
create flowfiles	Create flow files for plc trace
create loadfiles	Create loadfiles
create object	Create an object
create structfiles	Create structfiles
create volume	Create a volume
cut	Cut objects
define	Define a symbol
delete object	Delete an object
delete tree	Delete an object tree
delete volume	Delete a volume
disconnect	Disconnect signal and channel
display	Display a window
distribute	Distribute to operator or process station
edit	set edit mode
exit	close wtt
help	Display help
generate web	Generate webpages
list channels	List channels
list descriptor	List from listdescriptor
list hierarchy	List hierarchy
list plcpgm	List plcprogram
list signals	List signals
login	User login
logout	User logout
move object	Move an object
new buffer	Create a new buffer
one	One window
open buffer	Open buffer selection window
open graph	Open a Ge graph
paste	Paste buffer
print	Print plcpgm
redraw	Redraw plcpgm
release subwindow	Continue execution with graph in window object.
revert	Revert session
save	Save session
search	Search
set advanceduser	Set advanced user

set alltoplevel	**Display all toplevel objects**
set attribute	**Set object attributes**
set db	**Set database**
set inputfocus	**Set input focus to window**
set showalias	**Display alias name**
set showattrref	**Display attribute references**
set showattrxref	**Display attribute x-references**
set showclass	**Display object class**
set showdescription	**Display description**
set showobjref	**Display object references**
set showobjxref	**Display object x-references**
set subwindow	**Open a graph in a window object**
set template	**Set template values for objects**
set verify	**Script verification**
set window	**Set window width and height**
setup	**Wtt setup**
show children	**Show the children of an object**
show license	**Show license terms**
show object	**Show an object**
show objid	**Show object identity**
show script	**Show scriptfiles**
show symbol	**Show a symbol**
show user	**Show current user**
show version	**Show wtt version**
show volumes	**Show all volumes in the database**
sort	**Sort the children of an object**
two	**Two windows**
update classes	**Update classes**
wb dump	**Dump objects to textfile**
wb load	**Load objects from textfile**

Symbols

related subjects

script

29.1 Command build

Call the build method for a node, a volume or an object.

wtt> build node /name= [/force][/debug][/manual][/crossreference]
wtt> build volume /name= [/force][/debug][/manual][/crossreference]
wtt> build classvolume /name= [/force][/debug][/manual][/crossreference]
wtt> build object /name= [/force][/debug][/manual][/crossreference]

/name	Node name, volume name or object name.
/force	Don't check any dependencies, build everything.
/debug	Build with debug, i.e. compile and link with debug.
/manual	Just build the specified item.
/crossreferences	Create crossreference files. Valid for building volumes.

29.2 Command   check classes

Check if any classes need update.

wtt> check classes

29.3 Command   close graph

Close a Ge graph.

wtt> close graph /file=

/file   Name of the Ge graph.

29.4 Command   compile

Compile plcprograms.

If no hierarchy, plcpgm or window is specified, the selected plcpgm
will be compiled.

wtt> compile [/debug]
wtt> compile /plcpgm= [/debug]
wtt> compile /window= [/debug]
wtt> compile /hierarchy= [/debug][/modified][/from_plcpgm=]
wtt> compile /volume= [/debug][/modified][/from_plcpgm=]
wtt> compile /allvolumes [/debug][/modified][/from_plcpgm=]

/plcpgm	Name of plcpgm object that will be compiled.
/window	Name of plcwindow object that will be compiled.
/hierarchy	All plcpgm's in the hierarchy will be compiled.
/volume	Volume name. All plcpgm's in the volume will be compiled.
/allvolumes	All plcpgm's in all volumes in the database will be compiled.
/debug	Compile with debug.
/modified	Only modified plcwindows will be compiled.

29.5 Command   configure card

Create a card with channels.

wtt> configure card /rack= /cardname= /channelname= /chanidentity=
**   /chandescription= /table=**

/rack   Name of rack object that the card will belong to.

/cardname   Name of card. Last segment of name.

/channelname   Name of channel. Last segment of name.
                               A '#' will be replaced with the channel number.
                               For example /chan=di33## will give the channelnames
                               di3301, di3302... If there is more than one channel
                               channelname has to contain a '#' sign.

/chanidentity   Identity of the channel. Will be inserted into the Identity-
                               attribute of the channel.

/chandescripton   Channel description. Will be inserted into the Description-
                               attribute of the channel.

29.6 Command connect

Connect a signal and a channel.

wtt> connect /source= /destination= [/reconnect]

/source	A signal or channel object.
/destination	A signal or channel object.
/reconnect	If the source or destination already is connected it will first be disconnected.

29.7 Command copy

Copy selected object trees to paste buffer.

wtt> copy [/keepreferences] [/ignore_errors]

/keepreferences   Keep references to objects outside the copied trees. By default these references will be zeroed.

/ignore_errors   Try to complete the copy despite detected errors.

29.8 Command   copy object

Copy an object or an object tree.

wtt> copy object /source= /destination= /name= [/hierarchy]
[/first] [/last] [/after] [/before]

/source	The object that will be copied.
/destination	The parent or sibling to the created object.
/name	The name of the created object. Last segment.
/hierarchy	If the source-object has children, the child tree will also be copied.
/first	The object will be inserted as first child to the destination object.
/last	The object will be inserted as last child to the destination object.
/after	The object will be inserted as sibling after the destination object.
/before	The object will be inserted as sibling before the destination object.

29.9 Command   create bootfiles

Create new bootfiles.

wtt> create bootfiles /nodeconfig= [/debug]
wtt> create bootfiles /allnodes [/debug]

/nodeconfig   The name of the NodeConfig-object of the
                       node for which nodefile will be created.

/all   Create bootfiles for all nodes in the project.

/debug   Link plcprogram with debug.

29.10 Command   create crossreferencefiles

Create files for displaying crossreferences in xtt and rtt for
the current volume.

wtt> create crossreferencefiles [/graph] [/simulation]

/graph   Search Ge graphs for cross references.
/simulation   Add cross references in simulation objects.

29.11 Command   create flowfiles

Create flowfiles for plc trace.
The layout of the plc windows are stored in flow files and used
in plc trace.

wtt> create flowfiles /plcpgm=
wtt> create flowfiles /hier=
wtt> create flowfiles /all

Command to create flowfiles from template plcpgm's in a class volume
wtt> create flowfiles /template/plcpgm=
wtt> create flowfiles /template/hier=Class

/all	Create flowfiles for all plc programs in the volume (can not be used in class volumes, use /hier=Class instead).
/plcpgm	Create flowfiles for the specified PlcPgm object.
/hierarchy	Create flowfiles for all PlcPgm object under the specified hierarchy.
/template	Create flowfiles for PlcTemplate programs in a class volume.

29.12 Command   create loadfiles

Create loadfiles for a volume.

wtt> create loadfile /volume=
wtt> create loadfile [/class] [/all]

/volume   Create loadfiles for a specific volume.

/all   Create loadfiles for all root volumes
                            in the database.

/class   Create loadfiles for all classvolumes in the database.

29.13 Command   create object

Create an object.

wtt> create object /destination= /name= /class=
[/first] [/last] [/after] [/before]

/destination   The destination of the new object. The position
                      of the new object will be child or sibling relative
                      to the destination object.

/name   Name of the new object. Last segment.

/class   Class of new object.

/first   The object will be inserted as first child to
                      the destination object.

/last   The object will be inserted as last child to
                      the destination object.

/after   The object will be inserted as sibling after
                      the destination object.

/before   The object will be inserted as sibling before
                      the destination object.

29.14 Command   create structfiles

Create c include-files for classes in a classvolume.

wtt> create structfiles [/files=]

/files   Name of wb_load-file.
                                Default name $pwrp_db/userclasses.wb_load

29.15 Command cut

Copy selected object trees to paste buffer, and remove the objects
in the current volume.

wtt> cut [/keepreferences]

/keepreferences   Keep references to objects outside the copied trees. By default
                         these references will be zeroed.

29.16 Command define

Define a symbol.

wtt> define 'symbolname' 'text'

related subjects

symbol
show symbol
symbolfile

29.17 Command   delete object

Delete an object.

wtt> delete object /name= [/noconfirm] [/nolog]

/name	Name of object.
/noconfirm	Delete without confirm.
/nolog	The operation will not be logged on the output device.

29.18 Command   delete tree

Delete an object tree.

wtt> delete tree /name= [/noconfirm] [/nolog]

/name	The root object of the tree.
/noconfirm	Delete without confirm.
/nolog	The operation will not be logged on the output device.

29.19 Command disconnect

Disconnect a signal or a channel.

wtt> disconnect /source=

/source   A signal or channel object.

29.20 Command display

Display plant or node hierarchy window (w1 or w2).

wtt> display w1
wtt> display w2

29.21 Command distribute

Distribute files to operator or process station.
Creates a distribution package, copies the package to
the station and unpacks the package.

wtt> distribute /node= [/package] [/file=]

/node	Node to distribute to.
/package	Only create package. The package is created but not copied.
/file	Package file name. Copies an existing package without creating a new package.

29.22 Command edit

Enter or leave edit mode.

wtt> edit
wtt> noedit

29.23 Command   exit

Close wtt.

wtt> exit

29.24 Command   help

Display help information for a subject.
The help information will be searched for in a help file. The file can be the base helpfile,
the project helpfile or another help file.

If no helpfile is supplied the subject will be searched for in the base and project helpfiles.

wtt> help 'subject'
wtt> help 'subject' /helpfile=

/helpfile   A help file that contains information of the help subject.

related subjects

helpfile

29.25 Command   generate web

Generate html-files for webpages configured by Web-objects in the node hierarchy of the current volume.

wtt> generate web

29.26 Command   list

Print a list of objects and attributes.

The lists will be sent to a printer queue specified by the
symbol PWR_FOE_PRINT.

wtt> list descriptor /descriptor=
wtt> list channels [/node=]
wtt> list signals [/hierarchy=]
wtt> list plcpgm [/plcpgm=] [/hierarchy=]
wtt> list hierarchy [/hierarchy=]

29.27 Command   list channels

List cards and channels.

wtt> list channels [/node=] [/volume=] [/allvolumes] [output=]

/node	$Node object.
/volume	List objects in this volume.
/allvolume	List objects in all volumes.
/output	Output file. If output file is supplied, the list will not be sent to the printer.

29.28 Command   list descriptor

Print a list described by a ListDescriptor object.

wtt> list descriptor /descriptor=

/descriptor   ListDescriptor object.

29.29 Command   list hierarchy

List of PlantHier and NodeHier objects.

wtt> list hierarchy [/hierarchy=] [/volume=] [/allvolumes] [output=]

/hierarchy   Hierarchy object.
/volume   List objects in this volume.
/allvolume   List objects in all volumes.
/output   Output file. If output file is supplied, the list
                    will not be sent to the printer.

29.30 Command   list plcpgm

List of PlcPgm objects.

wtt> list plcpgm [/hierarchy=] [plcpgm=] [/volume=] [/allvolumes] [output=]

/plcpgm	Plcpgm object.
/hierarchy	Hierarchy object.
/volume	List objects in this volume.
/allvolume	List objects in all volumes.
/output	Output file. If output file is supplied, the list will not be sent to the printer.

29.31 Command   list signals

List of signals and crossreferences to the signals.

wtt> list signals [/hierarchy=] [/volume=] [/allvolumes] [output=]

/hierarchy	Hierarchy object.
/volume	List objects in this volume.
/allvolume	List objects in all volumes.
/output	Output file. If output file is supplied, the list will not be sent to the printer.

29.32 Command login

Login with username an password. The privileges of the user will be fetched from the user database, and affect the access to the system.

wtt> login 'username' 'password'

If you want to create or modify a project, user or register a volume, you login as administrator with the qualifier /administrator. You must specify a user in the systemgroup 'administrator'. If this systemgroup doesn't exist, username and password are not required.

wtt> login /administrator 'username' 'password'

related subjects

logout
show user

29.33 Command logout

Logout a user, and return to the original user.

wtt> logout

related subjects

login

29.34 Command   move object

Move an object.

wtt> move object /source= /destination= [/rename=] [/first] [/last] [/after] [/before]
wtt> move object /source= /rename=

/source	Name of object to move.
/destination	The parent or sibling to the object after the move.
/rename	New object name, if the object name should be changed. Last segment. If no destination is supplied, the object will only be renamed, not moved.
/first	The object will be inserted as first child to the destination object.
/last	The object will be inserted as last child to the destination object.
/after	The object will be inserted as sibling after the destination object.
/before	The object will be inserted as sibling before the destination object.

29.35 Command   new buffer

Create a new empty buffer.

wtt> new buffer /name=

/name   Name of the buffer

29.36 Command one

Display one window. The window which currently owns the input focus is kept.

wtt> one

29.37 Command   open buffer

Open the buffer selection window.

wtt> open buffer

29.38 Command   open graph

Open a Ge graph.
If modal is selected, the execution of the script is continued
when the graph is closed.

wtt> open graph /file= /modal

/file	Name of the Ge graph.
/modal	Modal.

29.39 Command paste

Paste object from the last copy or cut operation into the current volume.
With the buffer option, an older paste buffer can be pasted.

wtt> paste [/keepoid] [/buffer=]

/keepoid	Keep the object identities if possible.
/buffer	Name of the buffer that should be pasted. By default the last buffer is used.
/into	Copy the root objects of the paste buffer as child to the selected object.
/toplevel	Copy the root objects of the paste buffer to the toplevel. Has to be used when copying to an empty volume.

29.40 Command print

Print plc documents.

wtt> print /plcpgm= [/nodocument] [/nooverview]
wtt> print /hierarchy= [/nodocument] [/nooverview]

/plcpgm	Print documents in a plcpgm.
/hierarchy	Hierarchy object. All plc in the hierarchy will be printed.
/nodocument	The plc-documents will not be printed.
/nooverview	The overview of the plc-window will not be printed.
/pdf	Print to pdf file.
/all	Print all plcpgms.

29.41 Command redraw

Redraw the plc code.

wtt> redraw /all
wtt> redraw /hierarchy=
wtt> redraw /plcpgm=

/plcpgm	Redraw a plcpgm.
/hierarchy	Hierarchy object. All plc in the hierarchy will be redrawn.
/all	Redraw all plcpgms.

29.42 Command   release subwindow

Continue the execution of a script that has opened a graph
in a window object by the command 'set subwindow' or the
function 'SetSubwindow' with modal selected.
The release command should be executed from a pushbutton
in the graph with actiontype command.

wtt> release subwindow 'graph'

graph   Name of the main graph.

29.43 Command   revert

Revert session.

wtt> revert

29.44 Command save

Save session.

wtt> save

29.45 Command search

Search for an objectname or a string.

wtt> search 'object'
wtt> search /regularexpression 'expression'
wtt> search /next

29.46 Command   set advanceduser

Set or reset advanced user.

wtt> set advanceduser
wtt> set noadvanceduser

related subjects

advanced user

29.47 Command   set alltoplevel

Show all the root objects in the database, not only the
root objects defined for the plant hierarchy or the
node hierarchy.

wtt> set alltoplevel
wtt> set noalltoplevel

29.48 Command   set attribute

Set a value to an attribute.
Objects are selected by the name, class and hierarchy qualifiers.

wtt> set attribute /attribute= [/value=] [/name=] [/class=] [/hierarchy=]
                         [/noconfirm] [/nolog] [/output] [/noterminal]

/attribute	Name of attribute.
/name	Name of object.
/value	Value to insert in the attribute. If no value is given a question will be asked for each object.
/class	Select object of this class.
/hierarchy	Only successors to this object will be selected.
/noconfirm	No confirm request is issued.
/nolog	Operation is not logged to output device.
/output	Output file.
/noterminal	Operations will not be logged in terminal.

Example
```
wtt> set attribute /name=H1-Pump /attr=Description /value="Water pump" /noconf
```

29.49 Command set db

Connect to the database with the supplied id.
This has no affect if a database already is open.

wtt> set db /dbid=

/dbid   Database identity.

29.50 Command set inputfocus

Set input focus to the plant or the node hierarchy window (w1 or w2).

wtt> set inputfocus w1
wtt> set inputfocus w2

29.51 Command set showalias

Display the aliasname of the objects in the plant and node hierarchy.

wtt> set showalias
wtt> set noshowalias

29.52 Command set showattrref

Display the number of connected attribute references
of the objects in the plant and node hierarchy.

wtt> set showattrref
wtt> set noshowattrref

29.53 Command   set showattrxref

Display the number of connected attribute x-references of
the objects in the plant and node hierarchy.

wtt> set showattrxref
wtt> set noshowattrxref

29.54 Command   set showclass

Display the class of the object in the plant and node hierarchy.

wtt> set showclass
wtt> set noshowclass

29.55 Command   set showdescription

Display the description of the objects in the plant and node hierarchy.

wtt> set showdescription
wtt> set noshowdescription

29.56 Command   set showobjref

Display the number of connected object references of the objects in the
plant and node hierarchy.

wtt> set showobjref
wtt> set noshowobjref

29.57 Command set showobjxref

Display the number of connected object x-references of the objects in the plant and node hierarchy.

wtt> set showobjxref
wtt> set noshowobjxref

29.58 Command set subwindow

Open a graph in a window object in a previously opened graph, or
exchange a graph in a multiview cell.

wtt> set subwindow 'graph' /name= /source=
wtt> set subwindow 'multiview-object' /name= /source= [/continue]

/name	Name of the window object.
/source	Name of graph that is to be opened in the window object or in the multiview cell.
/continue	Can be used if the command is executed from a Ge button with other dynamics to execute. Should not be used if the graph with the button itself is exchanged.

29.59 Command   set template

Set template values for some attributes that affect the layout
in the plceditor.

wtt> set template [/signalobjectseg=] [/sigchanconseg=] [/shosigchancon=]
                      [/shodetecttext=]

/signalobjectseg   Number of segments of the signal name that will
                        be displayed in 'Get' and 'Set' objects in
                        the plc-editor.
/sigchanconseg   Number of segments of the channel name that will
                        be displayed in 'Get' and 'Set' objects in
                        the plc-editor.
/shosigchancon   Display the channel name in 'Get' and 'Set'
                        objects in the plc-editor.
/shodetecttext   Display the detect text in ASup and DSup
                        objects in the plc-editor.

29.60 Command   set verify

Display all executed lines when running a script.

wtt> set verify
wtt> set noverify

29.61 Command   set window

Set window width and height.

wtt> set window /width= /height=

/width   width in pixels.
/height   height in pixels.

29.62 Command set volume

set volume is obsolete.

29.63 Wtt setup

Setup of wtt properies

DefaultDirectory	Default directory for commandfiles.
SymbolFilename	Symbolfile.
Verify	Verify commandfile execution.
AdvancedUser	User is advanced.
AllToplevel	Display all toplevel objects.
Bypass	Bypass some edit restrictions.

29.64 Command   show children

Display en object and it's children

wtt> show children /name=

/name   Name of the parent object.

29.65 Command   show license

Show license terms.

wtt> show license

29.66 Command   show object

List objects.

wtt> show object [/name=] [/hierarchy=] [/class=] [/volume=] [/allvolumes]
**   [/parameter=] [/full] [/output=] [/noterminal]**
wtt> show object /objid=

/name	Object name. Wildcard is supported.
/hierarchy	Hierarchy object. Only object in the hierarchy will be selected.
/class	Only objects of this class will be selected.
/volume	Name of volume.
/allvolumes	Search of objects will be performed in all volumes.
/parameter	List the value of an attribut for the selected objects.
/full	Display the content of the objects. Attributes that differ from template value will be displayed.
/output	Output file.
/noterminal	Output will not be written to terminal.
/objid	Display object for a specified objid.

29.67 Command show objid

Show the objid of an object.
If name is ommitted, the objid of the current selected
object is shown.

wtt> show objid [/name=]

/name   Object name.

29.68 Command   show script

Provides a list of scriptfiles.
Wildcard with asterisk (*) can be used to look up files.

wtt> show script ['scriptspec']

29.69 Command   show symbol

Show one symbol, or all symbols

wtt> show symbol 'symbol' **Show symbol 'symbol'**
wtt> show symbol **Show all symbols**

related subjects

define
symbol

29.70 Command show version

Show the wtt version

wtt> show version

29.71 Command   show volumes

Show all volumes in the database.

wtt> show volumes

29.72 Command sort

Sort the children of an object in alphabetical order, or in class order.
If no parent is given, the children of the selected objects will be sorted.

wtt> sort /parent= [/class] [/signals]

/parent	Parent to the objects that will be sorted.
/class	Sort in class order.
/signals	Sort signal and plcpgm objects in class order, and other objects in alphabetical order.

29.73 Command two

Display two windows. Both the plant and the node hierarchy window are displayed.

wtt> two

29.74 Command update classes

Update classes in the attached volume.

wtt> update classes

29.75 Command   wb dump

Dump the volume or a part of the volume to text file.

wtt> wb dump /output= [/hierarchy=]

/hierarchy	Hierarchy object. The object and its child tree will be written to text file.
/output	Output file.
/nofocode	Don't write plc code for functionobjects with template code. This will reduce the size of the dumpfile. New code will be copied when the plc is compiled.
/keepname	Write extern references by name instead of identity string.
/noindex	Don't write object index in the dumpfile.

29.76 Command   wb load

Load the database or from wb_load-file or dbs-file.

wtt> wb load /loadfile=

/loadfile   Name of file. Can be of type .wb_load, .wb_dmp or .dbs.
/noindex   Ignore object indexes in the dumpfile and create new object identities.

29.77   Symbol

A wtt symbol can be used as a short command or as string replacement
in a command. If the symbol is used as string replacement the symbol-
name should be surrounded by quotes.

Symbols are created with the define command.
The define-commands can be executed by the symbolfile.

Example of symbol used as a short command.

```
wtt> define p1 "show child/name=hql-hvk-pumpar-pump1"
wtt> p1
```

Example of symbol used as string replacement

```
wtt> define p1 hql-hvk-pumpar-StartPump1
wtt> open trace 'p1'
```

related subjects

define
show symbol
symbolfile

30   Wtt script

Execute script

Datatypes and declarations

Datatypes
Datatype conversions
Variable declarations
Operators

Statements

main-endmain
function-endfunction
if-else-endif
while-endwhile
for-endfor
break
continue
goto
include

Input/output functions

ask()
printf()
say()
scanf()

File handling functions

fclose()
felement()
fgets()
file_search()
fopen()
fprintf()
fscanf()
translate_filename()

String functions

edit()
element()
extract()
sprintf()
strchr()
strrchr()
strlen()
strstr()
toupper()

tolower()

Database functions

GetAttribute()
GetChild()
GetParent()
GetNextSibling()
GetNextVolume()
GetClassList()
GetNextObject()
GetClassListAttrRef()
GetNextAttrRef()
GetTemplateObject()
GetNextTemplateAttrRef()
GetObjectClass()
GetNodeObject()
GetRootList()
GetVolumeClass()
GetVolumeList()
SetAttribute()
CreateObject()
RenameObject()
MoveObject()
InLib()
OpenPlcPgm()
CreatePlcObject()
ClosePlcObject()
CreatePlcConnection()
SetPlcObjectAttr()
PlcConnect()

System functions

exit()
get_namespace()
set_namespace()
system()
terminate()
time()
tzset()
verify()

Miscellaneous functions

GetProjectName()
CheckSystemGroup()
CutObjectName()
MessageError()
MessageInfo()
GetCurrentText()
GetCurrentObject()
GetCurrentVolume()
IsW1()
IsW2()
EditMode()
MessageDialog()
ConfirmDialog()

ContinueDialog()
PromptDialog()
OpenGraph()
CloseGraph()
SetSubwindow()
GetVersion()
get_pwr_config()
get_node_name()
getmsg()
EVEN()
ODD()

wtt-commands

wtt-commands

30.1 Execute a script

A script-file will be executed from the command-line with the command

wtt> @'filename'

30.2 Datatypes

The datatypes are float, int and string.

int	integer value.
float	32-bit float value.
string	80 character string (null terminated).

There are three different tables in which a variable can be declared: local, global and extern. A local variable is known inside a function, a global is known in all functions in a file (with include-files), an external is known for all files executed in a session.

30.3 Datatype conversions

If an expression consists of variables and functions of different datatypes the variables will be converted with the precedence string, float, int. If two operands in an expression is of type int and float, the result will be float If two operands is of type float and string, or int and string, the result will be string. In an assignment the value of an expression will be converted to the type of the assignment variable, even if the result is a string and the variable is of type float or int.

Example

```
string str;
int    i = 35;
str = "Luthor" + i;
```
The value in str will be "Luthor35".

```
float   f;
string  str = "3.14";
int     i = 159;
f = str + i;
```
The value in f will be 3.14159.

30.4   Variable declarations

A variable must be declared before it is used.
A declaration consists of
- the table (global or extern, if local the table is suppressed)
- the datatype (int, float or string)
- the variable name (case sensitive)
- if array, number of elements
- equal mark followed by an init value, if omitted the init value is zero or
  null-string
- semicolon

An extern variable should be deleted (by the delete statement).
Global variables can also be deletet with the deletegbl statement.

Example

```
int             i;
float           flow = 33.4;
string          str = "Hello";
float           width[5] = (1.20, 2.44, 4.81, 7.77, 9.20);
extern int      jakob[20];
global float    ferdinand = 1234;
...
delete jakob[20];
deletegbl ferdinand;
```

Namespace

Extern variables can be declared in different namespaces by setting the namespace with
set_namespace(). This can be used for running sets of scripts concurrently in the same
process. A set of script can handle common extern variables, and by using different
namespaces, the same set can be run in several instances without mix-up of the extern
variables.

30.5   Operators

The operators have the same function as i c, with some limitations. All
operators are not implemented. Some operators (+,=,==) can also operate on
string variables. Precedence of operators is similar to c.

Operator	Description	Datatypes
+	plus	int, float, string
-	minus	int, float
*	times	int, float
/	divide	int, float
++	increment, postfix only.	int, float
--	decrement, postfix only	int, float
>>	bits right-shifted	int
<<	bits left-shifted	int
<	less than	int, float
>	greater than	int, float
<=	less equal	int, float
>=	greater equal	int, float
==	equal	int, float, string
!=	not equal	int, float, string
&	bitwise and	int
|	bitwise or	int
&&	logical and	int
||	logical or	int
!	logical not	int
=	assign	int, float, string
+=	add and assign	int, float
-=	minus and assign	int, float
&=	logical and and assign	int
|=	logical or and assign	int

30.6 Script statements

main-endmain	Main function.
function-endfunction	Function declaration.
if-else-endif	Conditional execution.
while-endwhile	While loop.
for-endfor	For loop.
break	Terminate while or for loop.
continue	Continue while or for loop.
goto	Jump to label.
include	Include script file.

30.6.1 main-endmain

The main and endmain statements controls where the execution starts and stops
If no main and endmain statements will be found, the execution will start
att the beginning of the file and stop at the end.

Example

```
main()
  int a;

  a = p1 + 5;
  printf( "a = %d", a);
endmain
```

30.6.2 function-endfunction

A function declaration consists of
- the datatype of the return value for the function
- the name of the function
- an argumentlist delimited by comma and surrounded by parenthesis. The
  argumentlist must include a typedeclaration and a name for each argument.

The arguments supplied by the caller will be converted to the type of the
to the type declared in the argument list. If an argument is changed inside
the function, the new value will be transferred to the caller. In this way
it is possible to return other values then the return value of the function.
A function can contain one or several return statements. The return will hand
over the execution to the caller and return the supplied value.

Example

```
function float calculate_flow(float a, float b)
  float c;
  c = a + b;
  return c;
endfunction

...
flow = korr * calculate_flow( v, 35.2);
```

30.6.3 if-else-endif

The lines between a if-endif statement will be executed if the expression
in the if-statement is true. The expression should be surrounded by parentheses.
If an else statement is found between the if and endif the lines between else
and endif will be executed if the if-expression is false.

Example

```
if ( i < 10 && i > 5)
  a = b + c;
endif

if ( i < 10)
  a = b + c;
else
  a = b - c;
endif
```

30.6.4 while-endwhile

The lines between a while-endwhile statement will be executed as long as the expression in the while-statement is true. The expression should be surrounded by parentheses.

Example

```
while ( i < 10)
  i++;
endwhile
```

30.6.5 for-endfor

The lines between a for-endfor statement will be executed as long as the
middle expression in the for-statement is true. The for expression consists
of three expression, delimited by semicolon and surrounded by parentheses.
The first expression will be executed the before the first loop, the third
will be executed after every loop, the middle is executed before every loop
and if it is true, another loop is done, if false the loop is leaved.

Example

```
for ( i = 0; i < 10; i++)
  a += b;
endfor
```

30.6.6 break

A break statement will search for the next endwhile or endfor statement continue the execution at the line after.

Example

```
for ( i = 0; i < 10; i++)
  a += b;
  if ( a > 100)
    break;
endfor
```

30.6.7 continue

A continue statement will search for the previous while or for statement continue the loop execution.

Example

```
for ( i = 0; i < 10; i++)
  b = my_function(i);
  if ( b > 100)
    continue;
  a += b;
endfor
```

30.6.8 goto

A goto will cause the execution to jump to a row defined by label.
The label line is terminated with colon.

Example

```
  b = attribute("MOTOR-ON.ActualValue", sts);
  if (!sts)
    goto some_error;
  ...
some_error:
  say("Something went wrong!");
```

30.6.9 include

An script include-file containing functions can be included with the #include statement. The default file extention is '.pwr_com'

Example

```
#include <my_functions>
```

30.7 Input/Output functions

Function	Description
ask	Print a question an read an answer.
printf	Formatted print.
say	Print a text.
scanf	Formatted read.

30.7.1 ask()

int ask(string question, (arbitrary type) reply)

Description

Prompts for input with supplied string.
Returns number of read tokens, 1 or 0.

Arguments

string	question	Prompt.
arbitrary type	reply	Entered reply. Can be int, float or string.

Example
```
string reply;

ask( "Do you want to continue? [y/n] ", reply);
if ( reply != "y")
  exit();
endif
```

30.7.2 printf()

int printf(string format [, (arbitrary type) arg1, (arbitrary type) arg2])

Description

Formatted print. C-syntax. Format argument and non, one or two value arguments.
Returns number of printed characters.

Arguments

string	format	Format.
arbitrary type	arg1	Value argument. Optional. Can be int, float or string.
arbitrary type	arg2	Value argument. Optional. Can be int, float or string.

Example

```
printf( "Watch out!");
printf( "a = %d", a);
printf( "a = %d and str = %s", a, str);
```

30.7.3 say()

int say(string text)

Description

Prints a string.

Arguments

string   text   Text to print.

Example
```
say( "Three quarks for Muster Mark!");
```

30.7.4 scanf()

int scanf(string format , (arbitrary type) arg1)

Description

Formatted input. C-syntax.
Returns number of read characters.

Arguments

string	format	Format.
arbitrary type	arg1	Value argument. Returned. Can be int, float or string.

Example

```
scanf( "%d", i);
```

30.8 Input/Output functions

Function	Description
fclose	Close a file
felement	Extract one element from the last read line.
fgets	Read a line from a file.
file_search	Search for files.
fopen	Open a file.
fprintf	Formatted write to file.
fscanf	Formatted read from file.
translate_filename	Replace environment variables in a file name.

30.8.1 fclose()

int fclose(int file)

Description

Closes an opened file.

Arguments

int   file   file-id returned by fopen.

Example

```
int infile;
infile = fopen("some_file.txt","r");
...
fclose( infile);
```

30.8.2 felement()

string felement(int number, string delimiter)

Description

Extracts one element from a string of elements read from a file with the fgets() function. felement() can be used in favour of element() when the read string is larger than the string size 256. felement() can parse lines up to 1023 characters.

Arguments

| int | number | the number of the element. |
| string | delimiter | delimiter character. |

Returns

| string | | The extracted element. |

Example

```
string elem1;
int file;
string line;

file = fopen( "my_file.txt", "r");
while( fgets( line, file))
  elem1 = felement( 1, " ");
endwhile
```

30.8.3 fgets()

int fgets(string str, int file)

Description

Reads a line from a specified file.
Returns zero if end of file.

Arguments

string   str   Read line. Returned.
int   file   file returned by fopen.

Example

```
file = fopen("some_file.txt","r");
while( fgets( str, file))
  say( str);
endwhile
fclose( file);
```

30.8.4 file_search()

int file_search(string pattern, string found_file, int pass)

Description

Search for files.
A pattern with wildcard can be specified to search for several files.
The search sequence is divided in the passes init, next and end.
At the first call pass init (1) is specified. At search of more
files with the same patter the pass next (1) is specified. The
search is closed with the pass end (2).
Returns odd status if a file is found, else even status.

Arguments

string	pattern	Name of file to search for. Can contain wild card ('*').
string	found_file	Found file.
int	pass	Pass. Init (1), next (0) or end (2).

Example

```
string pattern = "*.txt";
string found_file;
int sts;

sts = file_search(pattern, found_file, 1);
while (sts & 1)
  printf("Processing %s\n", found_file);
  ...
  sts = file_search(pattern, found_file, 0);
endwhile
file_search(pattern, found_file, 2);
```

30.8.5 fopen()

int fopen(string filespec, string mode)

Description

Opens a file for read or write.
Returns a file identifier. If the file could not be opened, zero is returned.

Arguments

| string | filespec | Name of file. |
| string | mode | Access mode |

Returns

| int | File identifier, or zero on error. |

Example

```
int infile;
int outfile;

infile = fopen("some_file.txt","r");
outfile = fopen("another_file.txt","w");
...
fclose( infile);
fclose( outfile);
```

30.8.6 fprintf()

int fprintf(int file, string format [, (arbitrary type) arg1,
 (arbitrary type) arg2])

Description

Formatted print on file. C-syntax. Format argument and non, one or two value
arguments.
Returns number of printed characters.

Arguments

int	file	File id returned by fopen.
string	format	Format.
arbitrary type	arg1	Value argument. Optional. Can be int, float or string.
arbitrary type	arg2	Value argument. Optional. Can be int, float or string.

Example

```
int outfile;
outfile = fopen( "my_file.txt", "w");
if (!outfile)
  exit();
fprintf( outfile, "Some text");
fprintf( outfile, "a = %d", a);
fclose( outfile);
```

30.8.7 fscanf()

int fscanf(int file, string format , (arbitrary type) arg1)

Description

Formatted read from file. C-syntax.
Returns number of read characters.

Arguments

int	file	File id.
string	format	Format.
arbitrary type	arg1	Value argument. Returned. Can be int, float or string.

Example

```
int file;
int i;

file = fopen( "my_file.txt", "r");
if (file)
  fscanf( file, "%d", i);
  fclose( file);
endif
```

30.8.8 translate_filename()

string translate_filename(string fname)

Description

Replace environment variables in filename.

Arguments

string	fname	A filename.

Returns

string	String with expanded env variables.

Example

```
string fname1 = "$pwrp_db/a.wb_load";
string fname2;
fname2 = translate_filename( fname1);
```

30.9 String functions

Function	Description
edit	Remove superfluous spaces and tabs.
element	Extract one element from a string.
extract	Extract a substring from a string.
sprintf	Formatted print to a string.
strchr	Return the first occurence of a character in a string.
strrchr	Return the last occurence of a character in a string.
strlen	Calculate the length of a string.
strstr	Return the first occurence of a substring in a string.
tolower	Convert string to lower case.
toupper	Convert string to upper case.

30.9.1 edit()

string edit(string str)

Description

Removes leading and trailing spaces and tabs, and replaces multiple tabs and spaces with a single space.
Returns the edited string.

Arguments

string   str   string to be edited.

Example

```
collapsed_str = edit(str);
```

30.9.2 element()

string element(int number, string delimiter, string str)

Description

Extracts one element from a string of elements.
Returns the extracted element.

Arguments

int	number	the number of the element.
string	delimiter	delimiter character.
string	str	string of elements.

Example

```
string str = "mary, lisa, anna, john";
string elem1;
elem1 = element( 1, ",", str);
```

30.9.3 extract()

string extract(int start, int length, string str)

Description

Extracts the specified characters from the specified string.
Returns the extracted characters as a string.

Arguments

int	start	start position of the first character.
		First character has position 1.
int	length	number of characters to be extracted.
string	str	string from which characters should be extracted.

Example

```
extracted_str = extract( 5, 7, str);
```

30.9.4 sprintf()

int sprintf(string str, string format [, (arbitrary type) arg1, (arbitrary type) arg2])

Description

Formatted print to buffer. C-syntax. Format argument and non, one or two value arguments. Returns number of printed characters.

Arguments

string	str	String to print to.
string	format	Format.
arbitrary type	arg1	Value argument. Optional. Can be int, float or string.
arbitrary type	arg2	Value argument. Optional. Can be int, float or string.

Example

```
string str;
int items;

sprintf( str, "Number of items: %d", items);
```

30.9.5 strchr()

int strchr(string str, string c)

Description

Return the first occurence of a character in a string.

Arguments

| string | str | String to search in. |
| string | c | Character to search for. |

Returns

| int | | Index for first occurence of character. First character has index 1. Returns zero if the character is not found. |

Example

```
string str = "index.html";
int idx;

idx = strchr( str, "." );
```

30.9.6 strrchr()

int strrchr(string str, string c)

Description

Return the last occurence of a character in a string.

Arguments

string	str	String to search in.
string	c	Character to search for.

Returns

int	Index for last occurence of character. First character has index 1. Returns zero if the character is not found.

Example

```
string str = "/usr/local/pwrrt";
int idx;

idx = strrchr( str, "/");
```

30.9.7 strlen()

int strlen(string str, string c)

Description

Calculates the length of a string.

Arguments

string   str   String to calculate length for.

Returns
int   Length of string.

Example

```
string str = "/usr/local/pwrrt";
int len;

len = strlen( str);
```

30.9.8 strstr()

int strstr(string str, string substr)

Description

Return the first occurence of a substring in a string.

Arguments

| string | str | String to search in. |
| string | substr | Substring to search for. |

Returns

| int | | Index for first occurence of substring. First character has index 1. Returns zero if the substring is not found. |

Example

```
string str = "index.html";
int idx;

idx = strstr( str, ".html");
```

30.9.9 toupper()

string toupper(string str)

Description

Convert string to upper case.

Arguments

string   str   String to convert.

Returns
string   String in upper case.

Example

```
string str1 = "Buster Wilson";
string str2;
str2 = toupper( str);
```

30.9.10 tolower()

string tolower(string str)

Description

Convert string to lower case.

Arguments

string   str   string to convert.

Returns
string   string in lower case.

Example

```
string str1 = "Buster Wilson";
string str2;
str2 = tolower( str);
```

30.10 System functions

Function	Description
exit	Exit script.
get_namespace	Get current namespace.
set_namespace	Set namespace for extern variables.
system	Execute shell command.
terminate	Terminate the process.
time	Get system time.
tzset	Set time zone.
verify	Print executed lines.

30.10.1 exit()

int exit()

Description

Terminates executions of the file.

Example

```
exit();
```

30.10.2 get_namespace()

string get_namespace()

Description

Returns the current namespace.

Returns

string   Current namespace.

Example

```
string current_namespace;
current_namespace = get_namespace();
```

30.10.3 set_namespace()

set_namespace(string namespace)

Description

Set namespace for extern variables.
The maximum size of the namespace is 31 characters. If the length of the input string exceeds the maximum size, the last 31 characters of the string is used.

Arguments

string   namespace   New namespace.

Example

```
set_namespace(p1);
```

30.10.4 system()

int system(string cmd)

Description

Execute a shell command.

Arguments

string	cmd	Shell command to execute.

Returns

int	The return value is -1 on error and the return status of the command otherwise.

Example

```
string cmd;

cmd = "firefox http://www.proview.se";
system( cmd);
```

30.10.5 terminate()

int terminate()

Descriptions

Terminate the process.

30.10.6 time()

string time()

Description

Returns the current time in string format.

Example

```
string t;
t = time();
```

30.10.7 tzset()

string tzset(string timezone)

Description

Set time zone.

Example

```
tzset("Europe/Stockholm");
```

30.10.8 verify()

int verify([int mode])

Description

Sets or shows verification mode. If verification is on all executed lines will
be displayed on the screen.
Returns the current verification mode.

Arguments

int   mode   verification on (1) or off (0). Optional.

Example

```
verify(1);
```

30.11 Database functions

Function	Description
GetAttribute()	Get attribute value.
GetChild()	Get object child.
GetParent()	Get object parent.
GetNextSibling()	Get object sibling.
GetNextVolume()	Get next volume.
GetClassList()	Get first instance of a class.
GetNextObject()	Get next instance of a class.
GetClassListAttrRef()	Get first instance of a class, attribute objects included.
GetNextAttrRef()	Get next instance of a class, attribute objects included.
GetTemplateObject()	Get the template object for a class.
GetNextTemplateAttrRef()	Get the next instance in a template object .
GetObjectClass()	Get class of an object.
GetNodeObject()	Get node object.
GetRootList()	Get first object in root list.
GetVolumeClass()	Get class of a volume.
GetVolumeList()	Get fist volume.
SetAttribute()	Set attribute value.
CreateObject()	Create an object.
RenameObject()	Change name of an object.
MoveObject()	Move an object.
InLib()	Check if an object is in a $LibHier.
OpenPlcPgm()	Open a PlcPgm.
ClosePlcPgm()	Close a PlcPgm.
CreatePlcObject()	Create a plc object.
CreatePlcConnection()	Create a plc connection.
SetPlcObjectAttr()	Set attribute in a plc object.
PlcConnect()	Connect a plc object.

30.11.1 GetAttribute()

(variable type) GetAttribute(string name [, int status])

Description

Get the value of the specified attribute. The returned type is dependent
of the attribute type. The attribute will be converted to int, float or string.

Arguments

string	name	name of the attribute to be fetched.
int	status	status of operation. Returned. If zero, the attribute could not be fetched. Optional.

Example

```
int alarm;
int sts;

alarm = GetAttribute("Roller-Motor-Alarm.ActualValue");
on = GetAttribute("Roller-Motor-On.ActualValue", sts);
if ( !sts)
  say("Could not find motor on attribute!");
```

30.11.2 GetChild()

string GetChild(string name)

Description

Get the first child of an object. The next children can be fetched with
GetNextSibling().
Returns the name of the child. If no child exists a null-string is returned

Arguments

string   name   name of object.

Example

```
string child;

child = GetChild("Roller-Motor");
```

30.11.3 GetParent()

string GetParent(string name)

Description

Get the parent of an object.
Returns the name of the parent. If no parent exists a null-string is returned.

Arguments

string   name   name of object.

Example

```
string parent;

parent = GetParent("Roller-Motor");
```

30.11.4 GetNextSibling()

string GetNextSibling(string name)

Description

Get the next sibling of an object.
Returns the name of the sibling. If no next sibling exists a null-string is
returned.

Arguments

string   name   name of object.

Example

```
string name;
int not_first;

name = GetChild("Rt");
not_first = 0;
while ( name != "")
  if ( !not_first)
    create menu/title="The Rt objects"/text="'name'"/object="'name'"
  else
    add menu/text="'name'"/object="'name'"
  endif
  not_first = 1;
  name = GetNextSibling(nname);
endwhile
if ( !not_first )
  MessageError("No objects found");
```

30.11.5 GetClassList()

string GetClassList(string class)

Description

Get the first object of a specified class. The next object of the class
can be fetched whith GetNextObject().
Returns the name of the first object. If no instances of the class exists a
null-string is returned.

Arguments

string   name   name of class.

Example

```
string name;

name = GetClassList("Dv");
```

30.11.6 GetNextObject()

string GetNextObject(string name)

Description

Get the next object in a classlist.
Returns the name of the object. If no next object exist a null-string is
returned.

Arguments

string   name   name of object.

Example

```
string name;

name = GetClassList("Di");
while ( name != "")
  printf("Di object found: %s", name);
  name = GetNextObject(name);
endwhile
```

30.11.7 GetClassListAttrRef()

string GetClassListAttrRef(string class)

Description

Get the first object or attribute object of a specified class. The next object
or attribute object of the class can be fetched whith GetNextAttrRef().
Returns the name of the first object. If no instances or attribute object of the
class exists a null-string is returned.

Arguments

string   name   name of class.

Example

```
string name;

name = GetClassListAttrRef("Dv");
```

30.11.8 GetNextAttrRef()

string GetNextAttrRef(string class, string name)

Description

Get the next object or attribute object in a classlist.
Returns the name of the object or attribute object . If no next
object exist a null-string is returned.

Arguments

string	class	name of class.
string	name	name of object or attribute object.

Example

```
string name;

name = GetClassListAttrRef("Di");
while ( name != "")
  printf("Di object found: %s", name);
  name = GetNextAttrRef( "Di", name);
endwhile
```

30.11.9 GetTemplateObject()

string GetTemplateObject(string class)

Description

Get the template object of a specified class.

Arguments

string   name   name of class.

Example

```
string name;

name = GetTemplateObject("Dv");
```

30.11.10 GetNextTemplateAttrRef()

string GetNextTemplateAttrRef(string class, string name)

Description

Get the next attribute object of the specified class in a template object. Returns
the name of the attribute object. If no next object exist a null-string is returned.
The first template object is fetched with GetTemplateObject().

Arguments

string   class   name of class.
string   name   name of template object or template attribute object.

Example

```
string name;

name = GetTemplateObject("Di");
while ( name != "")
  printf("Di object found: %s", name);
  name = GetNextTemplateAttrRef( "Di", name);
endwhile
```

30.11.11 GetObjectClass()

string GetObjectClass(string name)

Description

Get the class of an object.
Returns the name of the class.

Arguments

string   name   name of object.

Example

```
string class;

class = GetObjectClass("Motor-Enable");
```

30.11.12 GetNodeObject()

string GetNodeObject()

Description

Get the node object.
Returns the name of the node object.

Example

```
string node;
node = GetNodeObject();
```

30.11.13 GetRootList()

string GetRootList()

Description

Get the first object in the root list.
Returns the name of the root object. The next object in the root list can be
fetched with GetNextSibling().

Example

```
string name;

name = GetRootList();
while( name != "")
  printf( "Root object found: %s", name);
  name = GetNextSibling(name);
endwhile
```

30.11.14 GetNextVolume()

string GetNextVolume(string name)

Description

Get the next volume. The first volume is fetched widh GetVolumeList().
Returns the name of the volume. If there is no next volume, a null-string
is returned.

Argument

string   name   name of volume.

30.11.15 GetVolumeClass()

string GetVolumeClass(string name)

Description

Get the class of a volume.
Returns the classname.

Argument

string   name   volume name.

Example

```
string class;

class = GetVolumeClass("CVolVKVDKR");
```

30.11.16 GetVolumeList()

string GetVolumeList()

Description

Get the first volume in the volumelist.
Returns the name of the volume. The next volume will be fetched with
GetNextVolume().

Example

```
string name;

name = GetVolumeList();
while( name != "")
  printf( "Volume found: %s", name);
  name = GetNextVolume(name);
endwhile
```

30.11.17 SetAttribute()

int SetAttribute(string name, (arbitrary type) value)

Description

Set the value of an attribute.
The attribute is specified with full object and
attribute name.
Returns the status of the operation.

Argument

string   name   attribute name.
arbitrary type   value   attribute value.

Example

```
SetAttribute( "Pump-V1-Switch.Description", "Valve switch open");
```

30.11.18 CreateObject()

int CreateObject(string name, string class, string destination, int destcode)

Description

Create an object.
Returns the status of the operation.

Argument

string	name	object name. Without path.
string	class	object class.
string	destination	destination object. A father or sibling to the object.
int	destcode	destination code. 1 first child, 2 last child, 3 after, 4 before.

Example

```
CreateObject( "Temperature", "BaseTempSensor", "Pump-V1", 2);
```

30.11.19 RenameObject()

int RenameObject(string name, string newname)

Description

Change the name of an object.
Returns the status of the operation.

Argument

| string | name | old name with path. |
| string | newname | new name without path. |

Example

```
RenameObject( "H1-Zon1-Temp2", "PT2");
```

30.11.20 MoveObject()

int MoveObject(string name, string destination, int destcode)

Description

Move an object.
Returns the status of the operation.

Argument

string	name	name with path.
string	destination	destination object. A father or sibling to the object in the new position.
int	destcode	destination code. 1 first child, 2 last child, 3 after, 4 before.

Example

```
MoveObject( "H1-Zon1-Temp2", "H1-Zon2", 1);
```

30.11.21 InLib()

int InLib(string name)

Description

Check if an object is in a library hierarchy.
Returns 1 if it is in a library hierarchy, else 0.

Argument

string   name   object name with path.

Returns

int   1 if the object is in a library
                                                    hierarchy, else 0.

Example

```
if ( !InLib( "H1-Motor"))
  ...
endif
```

30.11.22 OpenPlcPgm()

int OpenPlcPgm(string name)

Description

Open a plc program and a plc editing session.
Returns the status of the operation.
There should not be any unsaved operations when this function is called.
Only one program can be opened concurrently.

The plc editing session should be closed with a call to ClosePlcPgm().
In the plc editing session only the plc functions CreatePlcObject(),
CreatePlcConnection(), SetPlcObjectAttr() and PlcConnect() should be
used to manipulate and create objects. Access for the previous session
is temporary set to readonly when the plc editing is active.

Argument

string   name   name of PlcPgm object.

Example

```
OpenPlcPgm( "Pump-V1-Control");
...
ClosePlcPgm();
```

30.11.23 ClosePlcPgm()

int ClosePlcPgm()

Description

Closes a plc editing session.

Example

```
OpenPlcPgm( "Pump-V1-Control");
...
ClosePlcPgm();
```

30.11.24 CreatePlcObject()

int CreatePlcObject(string name, string class, float x, float y [, string destination,
            int inputmask, int outputmask, int invertmask])

Description

Creates a plc function object in a plc editing session.
The function can only be used in a plc editing session started with a call to
OpenPlcPgm().

The object is positioned on the coordinates x and y. If destination, a document object,
is supplied, the coordinates are relative to this object, otherwise they are absolute.
If the masks are left out the default masks are used.

Argument

string	name	Object name. Without path.
string	class	Object class.
float	x	x coordinate.
float	y	y coordinate.
string	destination	Optional. A document object. If supplied the coordinates are relative to this object.
int	inputmask	Optional. Mask where the bits indicate visible input pins in the function block.
int	outputmask	Optional. Mask where the bits indicate visible output pins in the function block.
int	invmask	Optional. Mask where the bits indicate inverted input pins.

Example

```
OpenPlcPgm( "Pump-V1-Control");
CreatePlcObject( "Document0", "Document", 1.2, 0.0);
CreatePlcObject( "And0", "And", 0.3, 0.1, "Document0", 15, 1, 3);
CreatePlcObject( "V1", "BaseCValve", 0.3, 0.4, "Document0");
ClosePlcPgm();
```

30.11.25 CreatePlcConnection()

int CreatePlcConnection(string source, string sourceattr, string dest, string destattr [, int feedback])

Description

Creates a plc connection between the source function object and the destination function object.

The function can only be used in a plc editing session started with a call to OpenPlcPgm().

The pins to connect to are specified by the attributes names of the pins. If a dashed feedback connection is to be created, this can be stated with the optional feedback argument.

Argument

string	source	Source object name. Without path.
string	sourceattr	Attribute for pin on the source object.
string	dest	Destination object name. Without path.
string	destattr	Attribute for pin on the destination object.
int	feedback	Optional. If 1 a dashed feedback connection is created.

Example

```
OpenPlcPgm( "Pump-V1-Control");
...
CreatePlcObject( "And0", "And", 0.3, 0.1, "Document0", 15, 1, 3);
CreatePlcObject( "And1", "And", 0.6, 0.1, "Document0", 15, 1, 3);
CreatePlcConnection( "And0", "Status", "And1", "In1");
# Feedback connection
CreatePlcConnection( "And1", "Status", "And0", "In4", 1);
ClosePlcPgm();
```

30.11.26 PlcConnect()

int PlcConnect(string plcobject, string connectobject)

Description

Activates the plc connect function, for example to connect Get and Sto function objects to signals and attributes in the plant hierarchy.

The function can only be used in a plc editing session started with a call to OpenPlcPgm().

Argument

string	plcobject	Name, without path, of plc function object that should be connected.
string	connectobject	Name of object or attribute to connect to. Full name with path.

Example

```
OpenPlcPgm( "Pump-V1-Control");
...
CreatePlcObject( "GetDv0", "GetDv", 0.3, 0.1, "Document0");
PlcConnect( "GetDv0", "Pump-V1-Active");
...
ClosePlcPgm();
```

30.11.27 SetPlcObjectAttr()

int SetPlcObjectAttr(string attribute, (arbitrary type) value)

Description

Set a value in a plc object.

The function can only be used in a plc editing session started with a call to OpenPlcPgm().

Argument

string   name   attribute name. Without path.
arbitrary type   value   attribute value.

Example

```
OpenPlcPgm( "Pump-V1-Control");
CreatePlcObject( "Document0", "Document", 1.3, 0.0);
SetPlcObjectAttr( "Document0.DocumentOrientation", 1);
SetPlcObjectAttr( "Document0.DocumentSize", 3);
...
ClosePlcPgm();
```

30.12 Miscellaneous functions

Function	Description
GetProjectName()	Get project name.
CheckSystemGroup()	Check that system group exist.
CutObjectName()	Cut off an object name.
MessageError()	Print error message.
MessageInfo()	Print info message.
GetCurrentText()	Get the text of the current item.
GetCurrentObject()	Get the object associated with the current item.
GetCurrentVolume()	Get the attached volume.
IsW1()	Check if Plant hierarchy has focus.
IsW2()	Check if Node hierarchy has focus.
EditMode()	Check if in edit mode.
MessageDialog()	Open a message dialog.
ConfirmDialog()	Open a confirm dialog.
ContinueDialog()	Open a Continue/Quit dialog.
PromptDialog()	Open an input dialog.
OpenGraph()	Open a Ge graph.
CloseGraph()	Close a Ge graph.
SetSubwindow()	Open a graph in a window object.
GetVersion()	Get ProviewR version.
get_pwr_config()	Get configuration values.
get_node_name()	Get node name.
getmsg()	Get status text.
EVEN()	Check if value is even.
ODD()	Check if value is odd.

30.12.1 GetProjectName()

string GetProjectName()

Description

Get the project name.
Returns the name of the project.

Example

```
string name;

name = GetProjectName();
```

30.12.2 CheckSystemGroup()

int CheckSystemGroup()

Description

Check if a system group exists.
Returns 1 if the system group exist, else 0.

Example

```
if ( !CheckSystemGroup( "MyGroup"))
  return;
endif
```

30.12.3 CutObjectName()

string CutObjectName(string name, int segments)

Description

Cut the first segments of an object name.
Returns the last segments of an object name. The number of segments left is
specified by the second argument

Arguments

string   name   Path name of object.
int   segments   Number of segments that should be left.

Example

```
string path_name;
string object_name;

path_name = GetChild("Rt-Motor");
object_name = CutObjectName( path_name, 1);
```

30.12.4 MessageError()

string MessageError(string message)

Description

Print an error message on the screen.

Example

```
MessageError("Something went wrong");
```

30.12.5 MessageInfo()

string MessageInfo(string message)

Description

Print an rtt info message on the screen.

Example

```
MessageInfo("Everything is all right so far");
```

30.12.6 GetCurrentText()

string GetCurrentText()

Description

Get the text of the current menu item or update field.

Example

```
string text;

text = GetCurrentText();
```

30.12.7 GetCurrentObject()

string GetCurrentObject()

Description

Get the object associated with the current menu item.
If no object is associated, a null-string i returned.

Example

```
string object;

object = GetCurrentObject();
```

30.12.8 GetCurrentVolume()

string GetCurrentVolume()

Description

Get the attached volume.
If no volume is attached, a null-string i returned.

Example

```
string current_volume;

current_volume = GetCurrentVolume();
set volume/volume=SomeOtherVolume
...
set volume/volume='current_volume'
```

30.12.9 IsW1()

int IsW1()

Description

Returns 1 if the current focused window in wtt is the Plant hierarchy window.
Otherwise returns 0.

30.12.10 IsW2()

int IsW2()

Description

Returns 1 if the current focused window in wtt is the Node hierarchy window.
Otherwise returns 0.

30.12.11 EditMode()

int EditMode()

Description

Returns 1 if wtt is int edit mode.
Otherwise returns 0.

30.12.12 MessageDialog()

MessageDialog(string title, string text)

Description

Display a message dialog box.

Arguments

string	title	Title.
string	text	Message text.

Example

```
MessageDialog( "Message", "This is a message");
```

30.12.13 ConfirmDialog()

int ConfirmDialog(string title, string text [, int cancel])

Description

Display a confirm dialog box.
Returns 1 if the yes-button is pressed, 0 if the no-button i pressed.
If the third argument (cancel) is added, a cancel-button is displayed.
If the cancel-button i pressed or if the dialogbox is closed,
the cancel argument is set to 1.

Arguments

string	title	Title.
string	text	Confirm text.
int	cancel	Optional. A cancel button is displayed. Cancel is set to 1 if the cancel-button is pressed, or if the dialog-box is closed.

Example 1

```
if ( ! ConfirmDialog( "Confirm", "Do you really want to..."))
  printf( "Yes is pressed\n");
else
  printf( "No is pressed\n");
endif
```

Example 2

```
int cancel;
int sts;

sts =  ConfirmDialog( "Confirm", "Do you really want to...", cancel);
if ( cancel)
  printf("Cancel is pressed\n);
  exit();
endif

if ( sts)
  printf( "Yes is pressed\n");
else
  printf( "No is pressed\n");
endif
```

30.12.14 ContinueDialog()

ContinueDialog(string title, string text)

Description

Display a message dialog box with the buttons 'Continue' and 'Quit'.
Returns 1 if continue is pressed, 0 if quit is pressed.

Arguments

string   title   Title.
string   text   Message text.

Example

```
if ( ! ContinueDialog( "Message", "This script will...");
  exit();
endif
```

30.12.15 PromptDialog()

int PromptDialog(string title, string text, string value)

Description

Display a prompt dialog box which promps for a input value.
Returns 1 if the yes-button is pressed, 0 if the cancel-button i pressed,
or if the dialogbox is closed.

Arguments

string	title	Title.
string	text	Value text.
string	value	Contains the entered value.

Example

```
string name;

if ( PromptDialog( "Name", "Enter name", name))
  printf( "Name : '%s'\n", name);
else
  printf( "Cancel...\n");
endif
```

30.12.16 OpenGraph()

int OpenGraph(string name, int modal)

Description

Open a Ge graph.
If modal is selected, the execution of the script is continued when
the graph is closed.

Arguments

string	name	Graph name.
int	modal	Modal.

Example

```
OpenGraph( "pwr_wizard_frame", 0);
```

30.12.17 CloseGraph()

int CloseGraph(string name)

Description

Close a Ge graph.

Arguments

string   name   Graph name.

Example

```
CloseGraph( "pwr_wizard_frame");
```

30.12.18 SetSubwindow()

int SetSubwindow(string name, string windowname, string source, int modal)

Description

Open a Ge graph in a window object in a previously opened graph.
If modal is selected, the execution of the script is continued when
the command 'release subwindow' is executed by a pushbutton in the graph.

Argument

string	name	Name of the main graph.
string	windowname	Name of the window object in which the source graph is to be opened.
string	source	Name of the graph that is to be opened in the window object.
int	modal	Modal.

Example

```
SetSubwindow( "pwr_wizard_frame", "Window1", "MyGraph", 1);
```

30.12.19 GetVersion()

int GetVersion()

Description

Get the ProviewR version for the current release.
Returns an integer value that is 10000 * major + 100 * minor + release.
For example V5.3.1 returns 50301.

Example

```
if ( GetVersion() > 50300)
  # Only for versions larger than V5.3.0
  ...
endif
```

30.12.20 get_pwr_config()

string get_pwr_config(string name)

Description

Get the value of a configuration variable.
Configuration values are set in /etc/proview.cnf.
Returns the value of the configuration variable.

Example

```
group = get_pwr_config( "defaultSystemGroup");
```

30.12.21 get_node_name()

string get_node_name()

Description

Get the host name for the current node.
Returns the host name.

Example

```
name = get_node_name();
```

30.12.22 getmsg()

string getmsg(int status)

Description

Get the corresponding text for a status variable.
Returns the text.

Example

```
msg = getmsg(sts);
```

30.12.23 EVEN()

int EVEN(int sts)

Description

Check is an integer is even.
Returns 1 if even and 0 if odd.

Example

```
sts = SetAttribute( "Pump-V1-Switch.Description", "Valve switch open");
if ( EVEN(sts))
  printf("Couldn't set attribute\n");
endif
```

30.12.24 ODD()

int ODD(int sts)

Description

Check is an integer is odd.
Returns 1 if odd and 0 if even.

Example

```
sts = SetAttribute( "Pump-V1-Switch.Description", "Valve switch open");
if ( ODD(sts))
  printf("Set operation successful\n");
endif
```

30.13 Wtt commands

All the wtt-commands is available in the script code. An wtt-command line
should NOT be ended with a semicolon. Variables can be substituted in the
command line by surrounding them with apostrophes.

Example

```
string name = "PUMP-VALVE-Open";
string value = "The valve is open";
set attribute/name='name'/attr="Description"/value='value'
```

Example

```
string name;
string parname;
int j;
int i;
for ( i = 0; i < 3; i++)
  parname = "vkv-test-obj" + (i+1);
  create obj/name='parname'
  for ( j = 0; j < 3; j++)
    name = parname + "-obj" + (j+1);
    create obj/name='name'
  endfor
endfor
```