Guide to I/O System

2008-12-22

Table of Contents

About this Guide

The *Proview Guide to I/O System* is intended for persons who will connect different kinds of I/O systems to Proview, and for users that will gain a deeper understanding of how the I/O handling or Proview works. The first part is an overview of the I/O systems adapted to Proview, and the second part a description of how to adapt new I/O systems to Proview.

Introduction

The Proview I/O handling consists of a framework that is designed to

- be portable and runnable on different platforms.
- handle I/O devices on the local bus.
- handle distributed I/O systems and communicate with remote rack systems.
- make it possible to add new I/O-systems with ease.
- allow projects to implement local I/O systems.
- synchronize the I/O-system with the execution of the PLC program, or application processes.

Overview

The I/O devices of a process station is configured by creating objects in the Proview database. The objects are divided in two trees, the Plant hierarchy and the Node hierarchy.

The Plant hierarchy describes how the plant is structured in various process parts, motors, pumps, fans etc. Here you find signal objects that represents the values that are fetched from various sensors and switches, or values that are put out to motors, actuators etc. Signal objects are of the classes Di, Do, Ai, Ao, Ii, Io, Co or Po.

The node hierarchy describes the configuration of the process station, with server processes and I/O system. The I/O system is configured by a tree of agent, rack, card and channel objects. The channel objects represent an I/O signal attached to the computer at a channel of an I/O card (or via a distributed bus system). The channel objects are of the classes ChanDi, ChanDo, ChanAi, ChanAo, ChanIi, ChanIo and ChanCo. Each signal object in the plant hierarchy points to a channel object in the node hierarchy. The connection corresponds to the physical link between the sensor and the channel of a I/O unit.

Levels

The I/O objects in a process station are configured in a tree structure with four levels: Agent, Rack, Card and Channel. The Channel objects can be configured as individual objects, or reside as internal attributes in a Card object.

Configuration

When configuring an I/O system on the local bus, often the Rack and Card-levels are sufficient. A configuration can look like this. A Rack object is placed below the $Node object, and below this a Card object for each I/O card that i installed in the rack. The card objects contains channel objects for the channels on the cards. The channel objects are connected to signal objects in the plant hierarchy. The Channels for analog signals contains attributes for measurement ranges, and the card objects contains attributes for addresses.

The configuration of a distributed I/O system is a bit different. Still the levels Agent, Rack, Card and Channel are used, but the levels has another meaning. If we take Profibus as an example, the agent level consist of an object for the master card that is mounted on the computer. The rack level consist of slave objects, that represent the Profibus slaves that are connected to the Profibus circuit. The card level consist of module objects that represent modules handled by the slaves. The Channel objects represent data sent on the bus from the master card to the modules or vice versa.

I/O System

This chapter contains descriptions of the I/O systems that are implemented in Proview.

PSS9000

PSS9000 consist of a set of I/O cards for analog input, analog output, digital input and digital output. There are also cards for counters and PID controllers. The cards are placed in a rack with the bus QBUS, a bus originally designed for DEC's PDP-11 processor. The rack is connected via a PCI-QBUS converter to an x86 PC, or connected via Ethernet, so called Remoterack.

The system is configured with objects from the SsabOx volume. There are objects representing the Rack and Card levels. The agent level i represented by the $Node object.

Rack objekt

Rack_SSAB

The Rack_SSAB object represents a 19” PSS9000 rack with QBUS backplane. The number of card slots can vary.

The rack is connected to a x86 PC with a PCI-QBUS converter card, PCI-Q, that is installed into the PC and connected to the rack with a cable. Several racks can be connected via bus extension card.

The rack objects are placed below the $Node objects and named C1, C2 etc (in older systems the naming convention R1, R2 etc can be found).

Attributes

Rack_SSAB doesn't contain any attributes used by the system.

Driver

The PCI-QBUS converter, PCI-Q, requires installation of a driver.

Ssab_RemoteRack

The Ssab_RemoteRack object configures a PSS9000 rack connected via Ethernet. A BFBETH card is inserted into the rack and connected Ethernet.

The object is placed below the $Node object and named E1, E2 etc.

Attributes

Attributes	*Description*
Address	IP address for the BTBETH card.
LocalPort	Port in the process station.
RemotePort	Port for the BTBETH card. Default value 8000.
Process	Process that handles the rack. 1 the PLC program, 2 io_comm.
ThreadObject	Thread object for the PLC thread that should handle the rack. Only used if

Attributes	Description
	Process is 1.
StallAction	No, ResetInputs or EmergencyBreak. Default EmergencyBreak.

Di card

All digital input cards have a common baseclass, Ssab_BaseDiCard, that contains attributes common for all di cards. The objects for each card type are extended with channel objects for the channels of the card.

Ssab_BaseDiCard

Attributes	Description
RegAddress	QBUS adress.
ErrorHardLimit	Error limit that stops the system.
ErrorSoftLimit	Error limit that sends an alarm message.
Process	Process that handles the rack. 1 the PLC program, 2 io_comm.
ThreadObject	Thread object for the PLC thread that should handle the rack. Only used if Process is 1.
ConvMask1	The conversion mask states which channels will be converted to signal values. Handles channel 1 – 16.
ConvMask2	See ConvMask1. Handles channel 17 – 32.
InvMask1	The invert mask states which channels are inverted. Handles channel 1-16.
InvMask2	See InvMask1. Handles channel 17 – 32.

Ssab_DI32D

The object configures a digital input card of type DI32D. The card has 32 channels, which channel objects reside as internal attributes in the object. The object is placed as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseDiCard.

Do cards

All digital output cards have a common baseclass, Ssab_BaseDoCard, that contains attributes that are common for all do cards. The objects for each card type are extended with channel objects for the channels of the card.

Ssab_BaseDoCard

Attributes	Description
RegAddress	QBUS address.
ErrorHardLimit	Error limit that stops the system.
ErrorSoftLimit	Error limit that sends an alarm message.

Attributes	*Description*
Process	Process that handles the rack. 1 the PLC program, 2 io_comm.
ThreadObject	Thread object for the PLC thread that should handle the rack. Only used if Process is 1.
InvMask1	The invert mask states which channels are inverted. Handles channel 1-16.
InvMask2	See InvMask1. Handles channel 17 – 32.
FixedOutValue1	Bitmask for channel 1 to 16 when the I/O handling is emergency stopped. Should normally be zero.
FixedOutValue2	See FixedOutValue1. FixedOutValue2 is a bitmask for channel 17 – 32.
ConvMask1	The conversion mask states which channels will be converted to signal values. Handles channel 1 – 16.
ConvMask2	See ConvMask1. Handles channel 17 – 32.

Ssab_DO32KTS

The object configures a digital output card of type DO32KTS. The card has 32 output channels, whose DoChan objects are internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseDoCard.

Ssab_DO32KTS_Stall

The object configures a digital output card of type DO32KTS Stall. The card is similar to DO32KTS, but also contains a stall function, that resets the bus, i.e. all outputs are zeroed on all cards, if no write or read i done on the card in 1.5 seconds.

Ai cards

All analog cards have a common baseclass, Ssab_BaseACard, that contains attributes that are common for all analog cards. The objects for each card type are extended with channel objects for the channels of the card.

Ssab_BaseACard

Attributes	*Description*
RegAddress	QBUS address.
ErrorHardLimit	Error limit that stops the system.
ErrorSoftLimit	Error limit that sends an alarm message.
Process	Process that handles the rack. 1 the PLC program, 2 io_comm.
ThreadObject	Thread object for the PLC thread that should handle the rack. Only used if Process is 1.

Ssab_AI8uP

The object configures an analog input card of type Ai8uP. The card has 8 channels, whose AiChan objects are internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseACard.

Ssab_AI16uP

The object configures an analog input card of type Ai16uP. The card has 16 channels, whose AiChan objects is internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseACard.

Ssab_AI32uP

The object configures an analog input card of type Ai32uP. The card has 32 channels, whose AiChan objects are internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseACard.

Ssab_AI16uP_Logger

The object configures an analog input card of type Ai16uP_Logger. The card has 16 channels, whose AiChan objects are internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseACard.

Ao cards

Ssab_AO16uP

The object configures an analog input card of type AO16uP. The card has 16 channels, whose AoChan objects are internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseACard.

Ssab_AO8uP

The object configures an analog input card of type AO8uP. The card has 8 channels, whose AoChan objects are internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseACard.

Ssab_AO8uPL

The object configures an analog input card of type AO8uP. The card has 8 channels, whose AoChan objects are internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseACard.

Co kort

Ssab_CO4uP

The object configures a counter card of type CO4uP. The card has 4 channels, whose CoChan objects are internal attributes in the card object. The object is positioned as a child to a Rack_SSAB or Ssab_RemoteRack object. Attributes, see BaseACard.

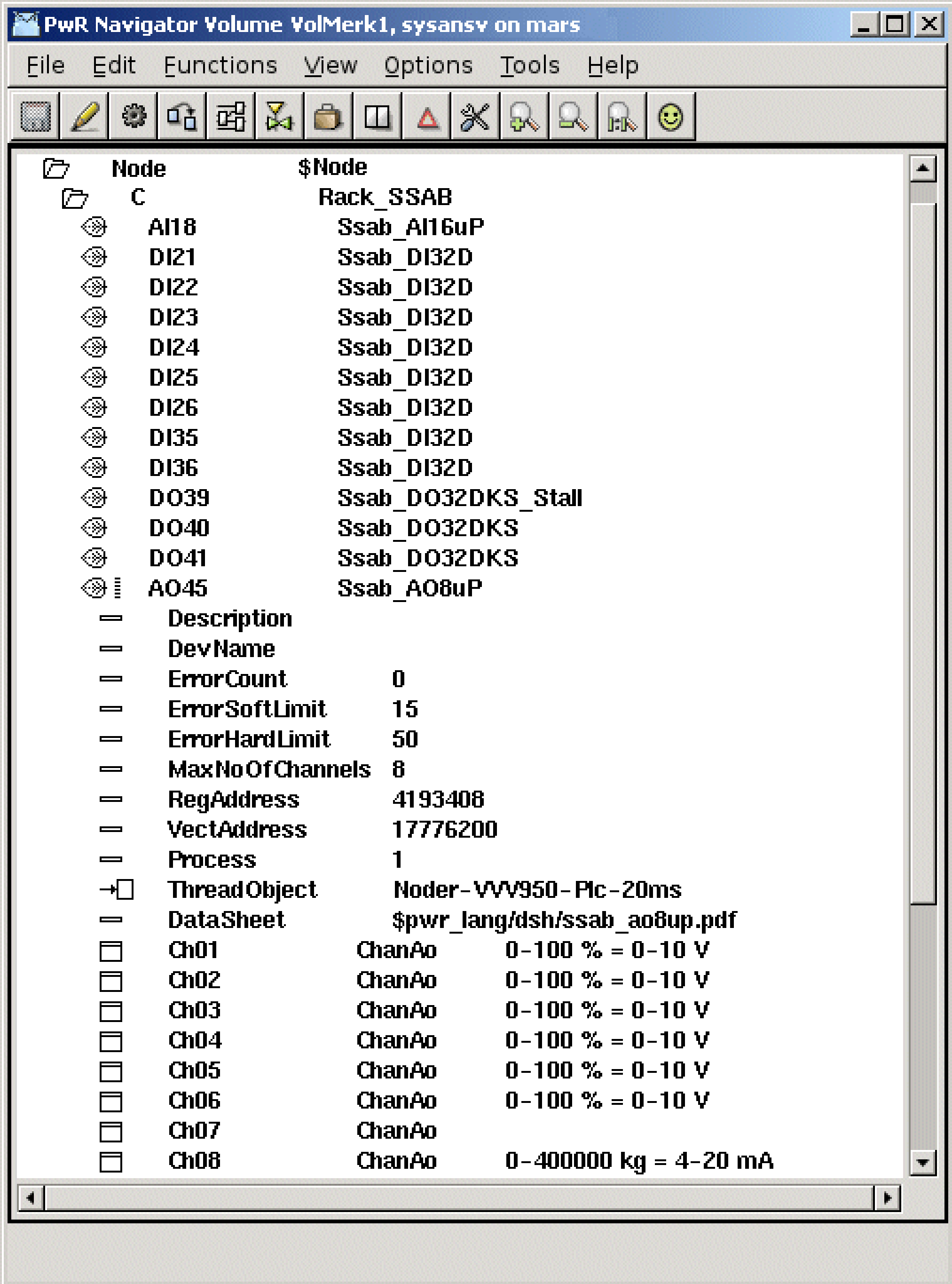


Fig PSS9000 configuration example

Profibus

Profibus is a fieldbus with nodes of type master and slave. The usual configuration is a single master system with one master and up to 125 slaves. Each slave can handle one or several modules.

In the Proview I/O handling the master represents the agent level, the slaves the rack level, and the module the card level.

Proview currently supports two different Profibus DP master boards, *Softing PROFIboard PCI* (see www.softing.com) and *Hilscher CIF 50-PB* (see www.hilscher.com). The master board is installed in a PCI-slot in the process station. To configure the board an object of class Profibus:Pb_Profiboard or Profibus:Pb_Hilscher is placed below the $Node object.

An important thing to keep in mind when configuring the Profibus master object is that the BusNumber is independent for different types of boards, i.e. boards using different agent classes. So for example if the system would have two Profibus cards installed, one Softing Profiboard and one Hilscher CIF 50-PB, both agent objects would have the BusNumber property set to 1. However, if the system would have two Softing Profiboard cards installed, one of them would have its BusNumber property set to 1 and the other to 2.

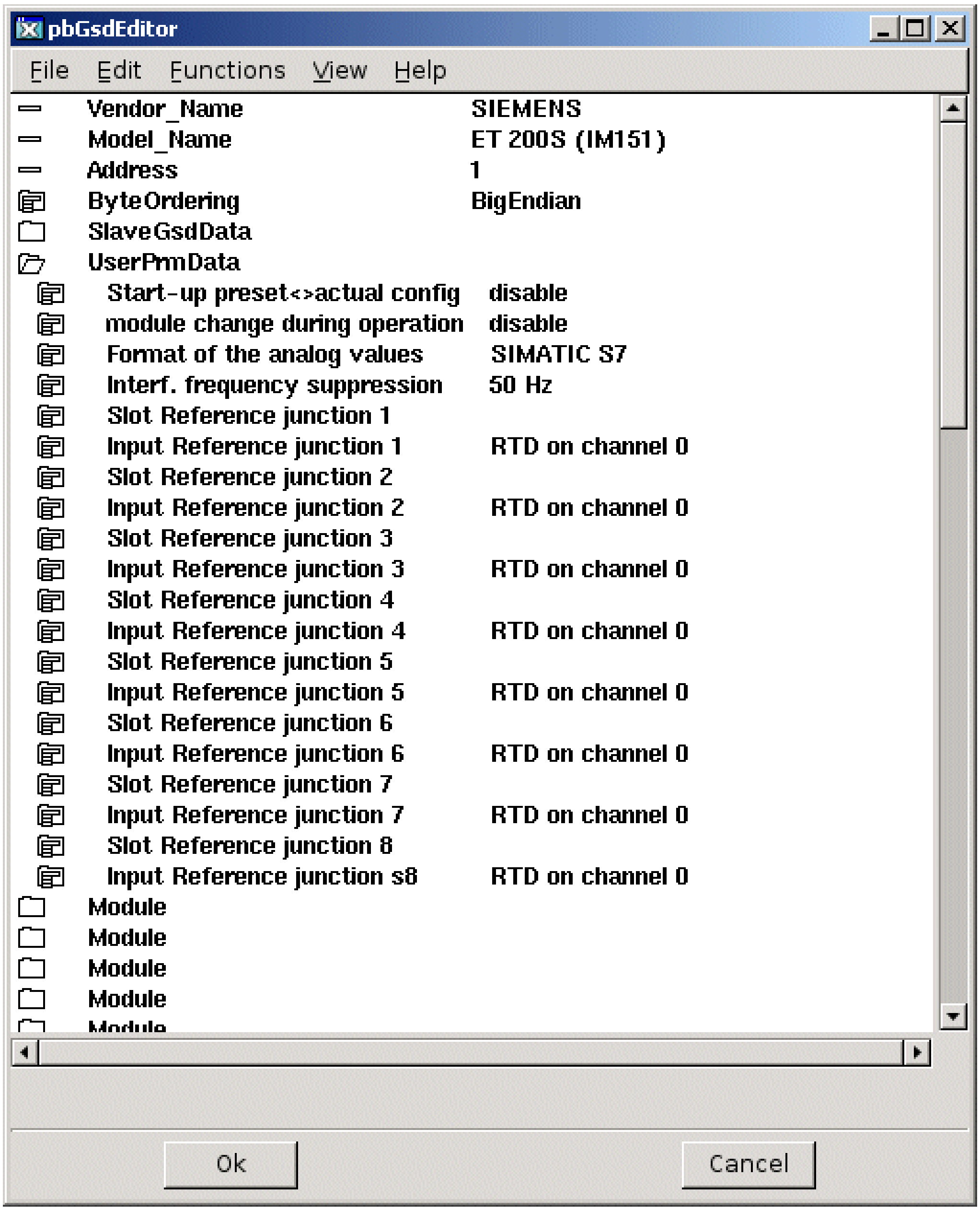
Each slave connected to the Profibus circuit is configured with an object of class Pb_DP_Slave, or a subclass to this class. The slave objects are placed as children to the master object. For the slave objects, a Profibus configurator can be opened, that configures the slave object, and creates module object for the modules that is handled by the slave. The Profibus configurator uses the gsd-file for the slave. The gsd-file is a text file supplied by the vendor, that describes the various configurations available for the actual slave. Before opening the Profibus configurator you has to specify the name of the gsd-file. Copy the file to $pwrp_exe and insert the filename into the attribute GSDfile in the slave object.

If there is a subclass present for the slave you are about to configure, e.g. Siemens_ET200S_IM151, the gsd-file is already stated in the slave object, and the gsd-file is included in the Proview distribution.

When this operation is preformed, the Profibus configurator is opened by right clicking on the object and activating 'Configure Slave' from the popup menu.

The Profibus configurator

The Profibus configurator is opened for a slave object, i.e. an object of class Pb_DP_Slave or a subclass of this class. There has to be a readable gsd-file stated in the GSDfile attribute in the slave object.



Address

The address of the slave is stated in the Address attribute. The address has a value in the interval 0-125 that is usually configured with switches on the slave unit.

SlaveGsdData

The map *SlaveGsdData* contains informational data.

UserPrmData

The map *UserPrmData* contains the parameter that can be configured for the current slave.

Module

A slave can handle one or several modules. There are modular slaves with one single module, where the slave and the module constitutes on unit. and there are slaves of rack type, into which a large number of modules can be inserted. The Profibus configurator displays on map for each module that can be configured for the current slave.

Each slave is given an object name, e.g. M1 M2 etc. Modules on the same slave has to have different object names.

Also the module type is stated. This is chosen from a list of module types supported by the current slave. The list is found below *Type*.

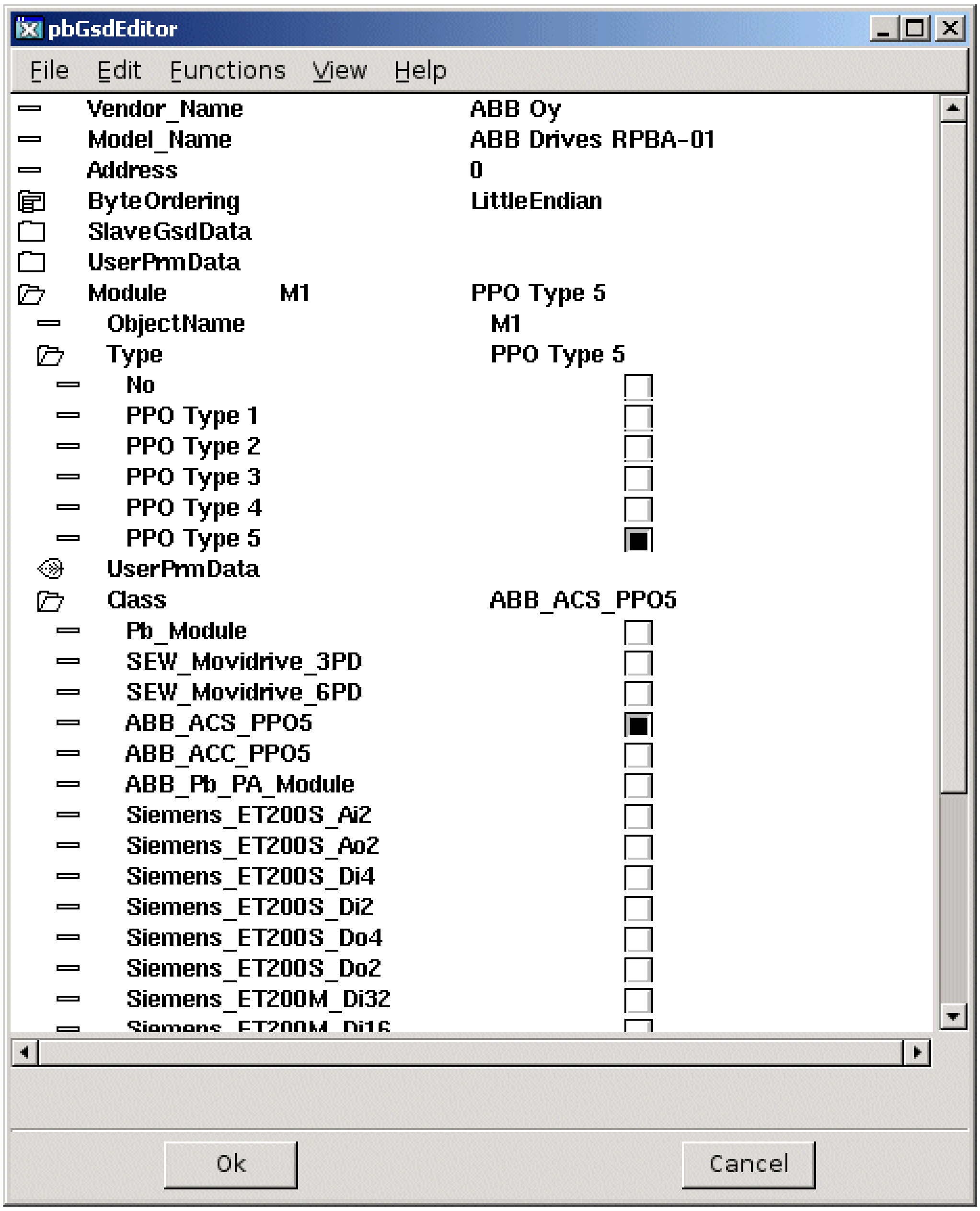


Fig Module Type and Class selected

When the type is chosen, the parameter of the selected module type is configured under *UserPrmData*.

You also have to state a class for the module object. At the configuration, a module object is created for each configured module. The object is of class Pb_Module or a subclass of that class. Under

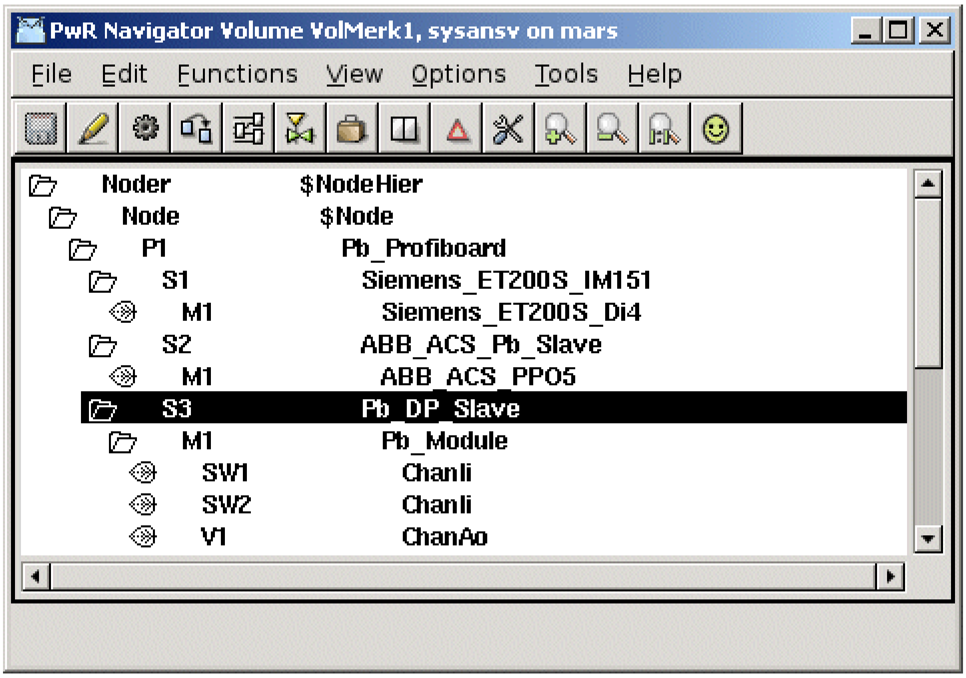
Class all the subclasses to Pb_Module are listed. If you find a class corresponding to the current module type, you select this class, otherwise you select the baseclass Pb_Module. The difference between the subclasses and the baseclass is that in the subclasses, the data area is specified with channel objects (see section Specify the data area).

When all the modules are configured you save by clicking on 'Ok' and leave by clicking 'Cancel'. The module objects with specified object names and classes are now created below the slave object.

If you are lucky, you will find a module object that corresponds to the current module. The criteria for the correspondence is whether the specified data area matches the current module or not. If you don't find a suitable module class there are two options: to create at new class with Pb_Module as baseclass, extended with channel objects to specify the data area, or to configure the channel as separate objects below a Pb_Module object. The second alternative is more convenient if there are one or a few instances. If there are several modules you should consider creating a class for the module.

Specify the data area

The next step is to specify the data area for a module. Input modules read data that are sent to the process station over the bus, and output modules receives data from the process station. There are also modules with both input and output data, e.g. frequency converters. The data areas that are sent and received via the bus has to configured, and this is done with channel objects. The input area is specified with ChanDi, ChanAi and ChanIi objects, the output area with ChanDo, ChanAo and ChanIo objects. The channel objects are placed as children to the module object, or, if you choose do make a specific class for the module, as internal attributes in the module object. In the channel object you should set *Representation*, that specifies the format of a parameter, and in some cases also *Number* (for Bit representation). In the slave object you might have to set the *ByteOrdering* (LittleEndian or BigEndian) and *FloatRepresentation* (Intel or IEEE).



Digital inputs

Digital input modules send the value of the inputs as bits in a word. Each input is specified with a

ChanDi object. Representation is set to Bit8, Bit16, Bit32 or Bit64 dependent on the size of the word, and in Number the bit number that contains the channel value is stated (first bit has number 0).

Analog inputs

An analog input is usually transferred as a integer value and specified with a ChanAi object. Representation matches the integer format in the transfer. In some cases the value is sent as a float, and the float format has to be stated in *FloatRepresentation* (FloatIntel or FloatIEEE) in the slave object. Ranges for conversion to engineering value are specified in *RawValRange*, *ChannelSigValRange*, *SensorSigValRange* and *ActValRange* (as the signal value is not used *ChannelSigValRange* and *SensorSigValRange* can have the same value as *RawValRange*).

Digital outputs

Digital outputs are specified with ChanDo objects. *Representation* should be set to Bit8, Bit16, Bit32 or Bit64 dependent on the transfer format.

Analog outputs

Analog outputs are specified with ChanAo objects. Set *Representation* and specify ranges for conversion from engineering unit to transfer value (set *ChannelSigValRange* and *SensorSigValRange* equal to *RawValRange*).

Complex data areas

Many modules sends a mixture of integer, float, bitmasks etc. You then have to combine channel objects of different type. The channel objects should be placed in the same order as the data they represent is organized in the data area. For modules with both in and out area, the channels of the input area i are usually placed first and thereafter the channels of the output area.

Drivers

Both Softing PROFIboard and Hilscher CIF 50-PB requires a driver to be installed. In both cases the drivers may be obtained from the respective manufacturer: www.softing.com for Softing and www.hilscher.com for Hilscher.

Agent objects

Pb_Profiboard

Agent object for a Profibus master of type Softing PROFIboard. The object is placed in the node hierarchy below the $Node object.

Pb_Hilscher

Agent object for a Profibus master of type Hilscher CIF 50-PB. The object is placed in the node hierarchy below the $Node object.

Slave objects

Pb_Dp_Slave

Base object for a Profibus slave. Resides below a Profibus agent object. In the attribute *GSDfile* the gsd-file for the current slave is stated. When the gsd-file is supplied the slave can be configured by the Profibus configurator.

ABB_ACS_Pb_Slave

Slave object for a frequency converter ABB ACS800 with protocol PPO5.

Siemens_ET200S_IM151

Slave object for a Siemens ET200S IM151.

Siemens ET200M_IM153

Slave object for a Siemens ET200M IM153.

Module objects

Pb_Module

Baseclass for a Profibus module. The object is created by the Profibus configurator. Placed as child to a slave object.

ABB_ACS_PPO5

Module object for a frequency converter ABB ACS800 with protocol PPO5.

Siemens_ET200S_Ai2

Module object for a Siemens ET200S module with 2 analog inputs.

Siemens_ET200S_Ao2

Module object for a Siemens ET200S module with 2 analog outputs.

Siemens_ET200M_Di4

Module object for a Siemens ET200M module with 4 digital inputs.

Siemens_ET200M_Di2

Module object for a Siemens ET200M module with 2 digital inputs.

Siemens_ET200M_Do4

Module object for a Siemens ET200M module with 4 digital outputs.

Siemens_ET200M_Do2

Module object for a Siemens ET200M module with 2 digital outputs.

Hilscher board configuration

The CIF 50-PB is factory set to be configured by Hilscher's proprietary configuration tool, SyCon, in offline mode. However, Hilscher provides instructions for removing the database to make the boards online configurable. Code for flashing the board is included in the Pb_Hilscher agent, it is however disabled by default. By defining the macro FLASH_WRITE_ENABLE at the beginning of the source file rt_io_m_pb_hilscher.c and recompiling, the code is enabled and will automatically try to remove the SyCon database if present when the agent initializes. Please note that this is not thoroughly tested.

There is also a tool named cif50_rmdb that will perform the steps to remove the database from the flash. After the database is removed, the board may be used with Proview.

MODBUS TCP

MODBUS is an application layer messaging protocol that provides client/server communication between devices. Proview implements the MODBUS messaging service over TCP/IP.

MODBUS is a request/reply protocol and offers services specified by function codes. For more information on the MODBUS protocol see the documents:

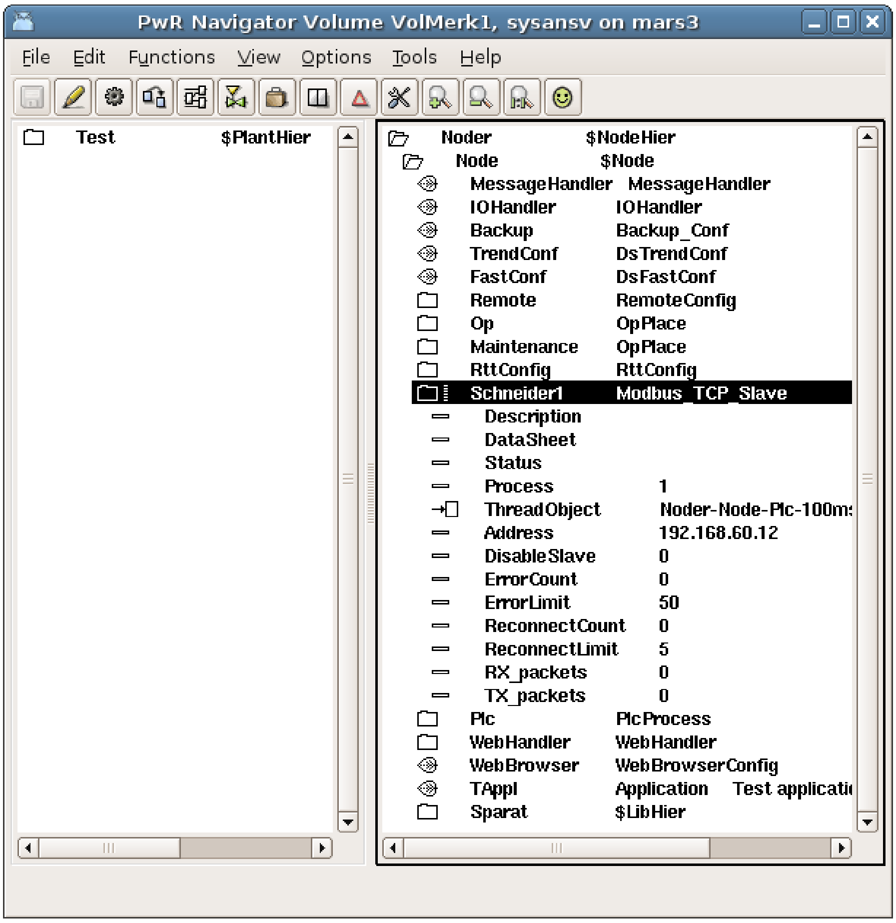
MODBUS Application Protocol Specification V1.1b

MODBUS Messaging on TCP/IP Implementation Guide V1.0b

Each device that is to be communicated with is configured with an object of class Modbus_TCP_Slave, or a subclass to this class. The interface to the device is defined by instances of the class Modbus_Module. Each instance of the Modbus_Module represents a function code for the service that is requested. The corresponding data area is defined with channels.

Configuration of a device

Insert a Modbus_TCP_Slave-object in the node-hierarchy. Specify the IP address of the MODBUS device. By default the device will be handled by a PLC thread. Connect the slave to a PLC thread. An example is given below. ReconnectLimit specifies how many times the slave will try to reconnect to the device if connection is lost.



Modules

With help of Modbus_Module's you define what type of actions you want to perform on the slave and at which address. The action is defined by a function code which means either reading or writing data to the Modbus slave. You also specify the address at which to read or write. The number of data to be read or written is defined by how you define the data area (see below).

The supported function codes are:

ReadCoils (FC 1)

This function code is used to read from 1 to 2000 contiguous status of coils in a remote device. Typically the input data area is defined by a number of ChanDi's which represent the number of coils you want to read. The representation on the ChanDi should be set to Bit8.

ReadDiscreteInputs (FC 2)

This function code is used to read from 1 to 2000 contiguous status of discrete inputs in a remote device. Typically the input data area is defined by a number of ChanDi's which represent the number of coils you want to read. The representation on the ChanDi should be set to Bit8.

ReadHoldingRegisters (FC 3)

This function code is used to read the contents of a contiguous block of holding registers in a remote device. A register is 2 bytes long. Typically the input data area is defined by a number of ChanIi's which represent the number of registers you want to read. The representation on the ChanIi should be set to UInt16 or Int16. ChanAi and ChanDi is also applicable. In case of ChanDi the representation should be set to Bit16.

ReadInputRegisters (FC 4)

This function code is used to read from 1 to 125 contiguous input registers in a remote device. Typically the input data area is defined by a number of ChanIi's which represent the number of registers you want to read. The representation on the ChanIi should be set to UInt16 or Int16. ChanAi and ChanDi is also applicable. In case of ChanDi the representation should be set to Bit16.

WriteMultipleCoils (FC 15)

This function code is used to force each coil in a sequence of coils to either ON or OFF in a remote Device. Typically the output data area is defined by a number of ChanDo's which represent the number of coils you want to write. The representation on the ChanDo should be set to Bit8.

WriteMultipleRegisters (FC 16)

This function code is used to write a block of contiguous registers (1 to 123 registers) in a remote device. Typically the output data area is defined by a number of ChanIo's which represent the number of registers you want to write. The representation on the ChanIo should be set to UInt16 or Int16. ChanAo and ChanDo is also applicable. In case of ChanDo the representation should be set to Bit16.

Specify the data area

To specify the data area a number of channel objects are placed as children to the module object. In the channel object you should set *Representation*, that specifies the format of a parameter, and in some cases also *Number* (for Bit representation). The data area is configured in much the same way as the Profibus I/O except for that you never have to think about the byte ordering which is specified by the MODBUS standard to be Big Endian.

To clarify how the data area is specified an example is given below.

Example

In this example we have a device which is a modular station of type *Schneider*. That means a station to which a number of different I/O-modules could be connected. We will use the function codes for reading and writing holding registers (FC3 and FC16). Our station consist of

1 Di 6 module

1 Do 6 module

1 Ai 2 module

1 Ao 2 module

According to the specification of this modular station the digital input module uses 2 registers, one to report data and one to report status. The digital output module uses one register to echo output data and reports one register as status. The analog input module uses 2 registers, one for each channel. The analog output module uses two registers for output data and reports 2 registers as status for each channel. Thus the data area looks like:

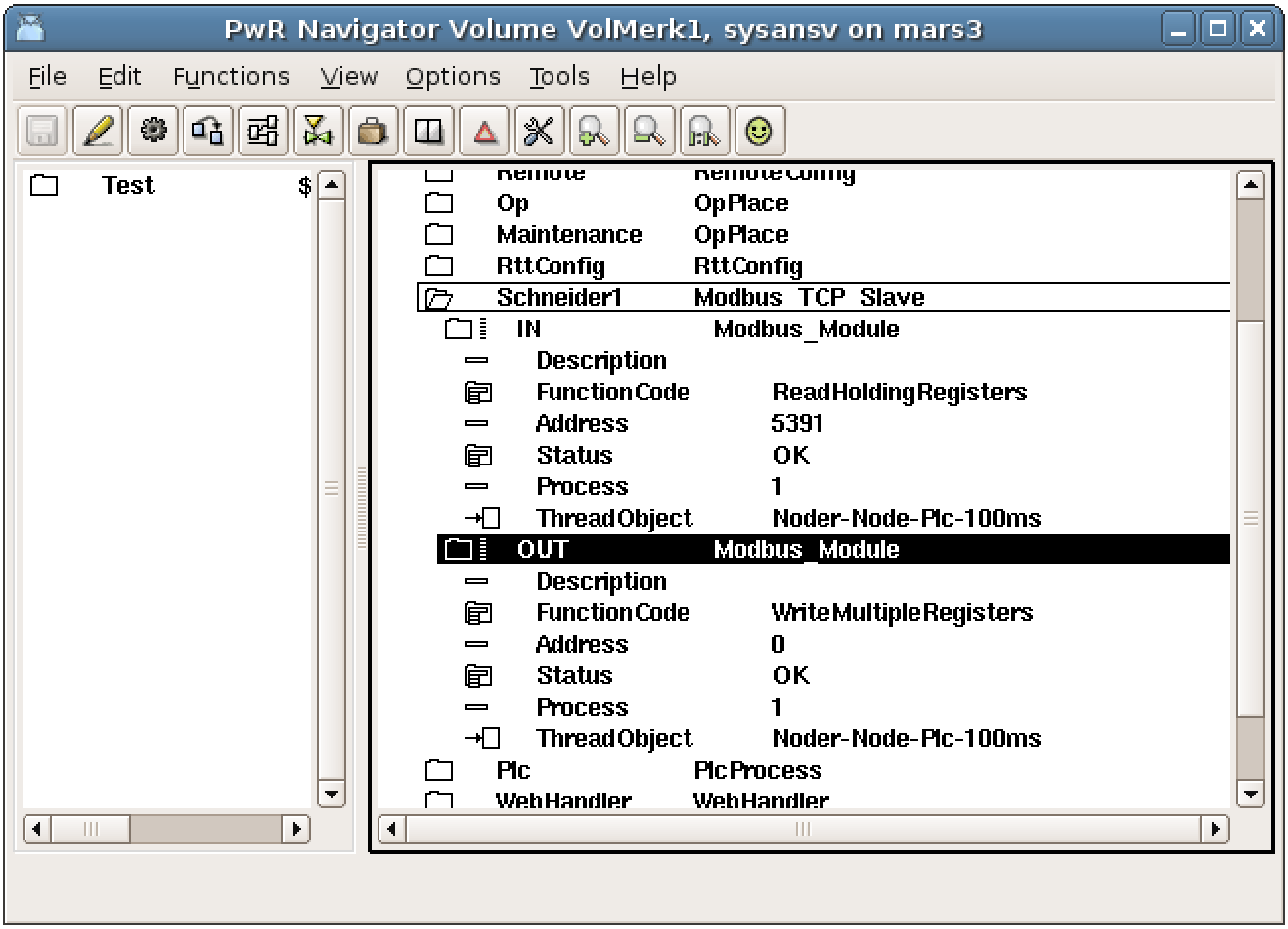
Input area

1 register (2 bytes) digital input data, 6 lowest bits represent the inputs
1 register (2 bytes) digital input status
1 register (2 bytes) echo digital output
1 register, digital output status
1 register, analog input channel 1
1 register, analog input channel 2
1 register, echo analog output channel 1
1 register, echo analog output channel 2

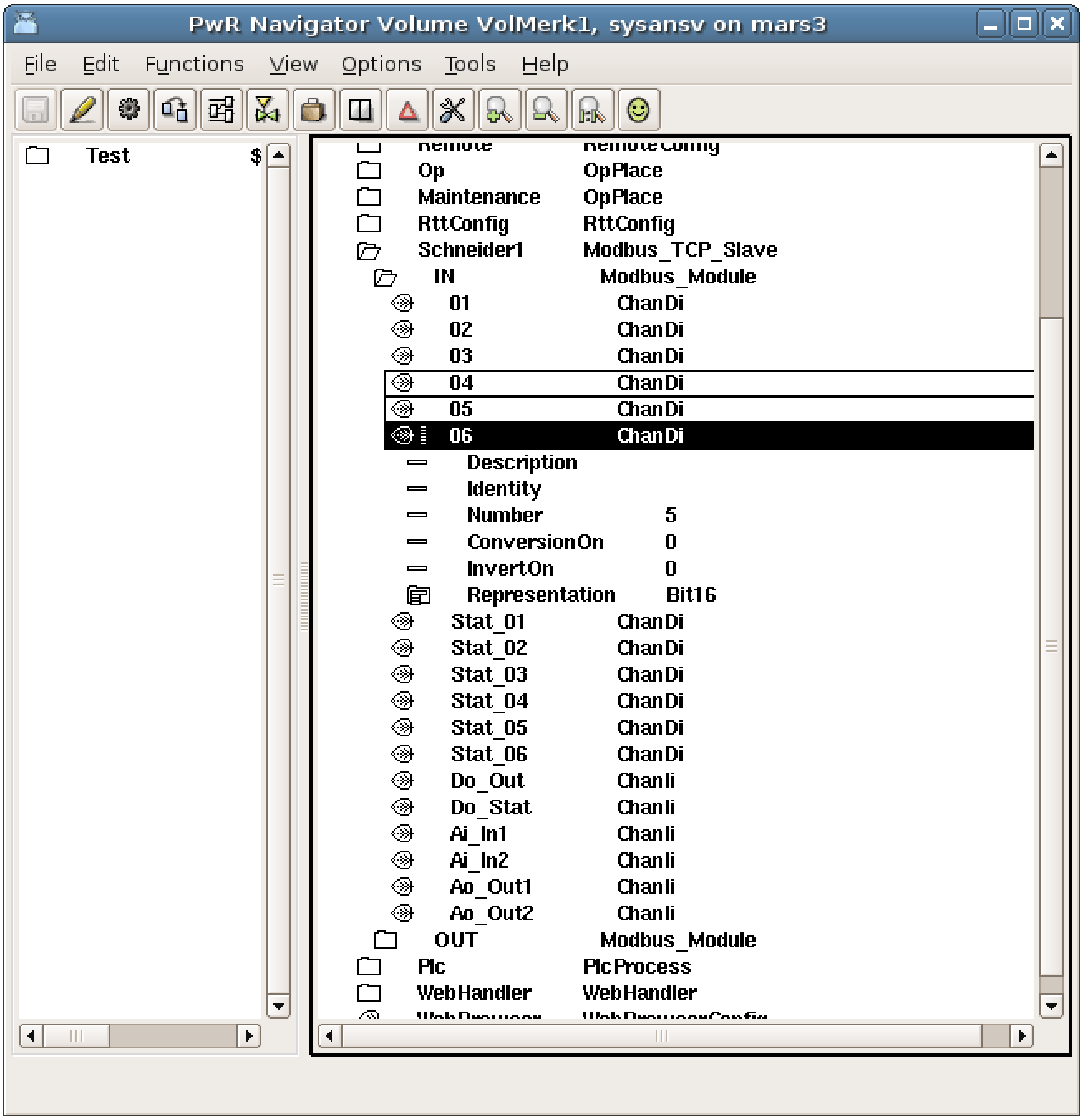
Output area

1 register digital output data, 6 lowest bits represent the outputs
1 register, analog output channel 1
1 register, analog output channel 2

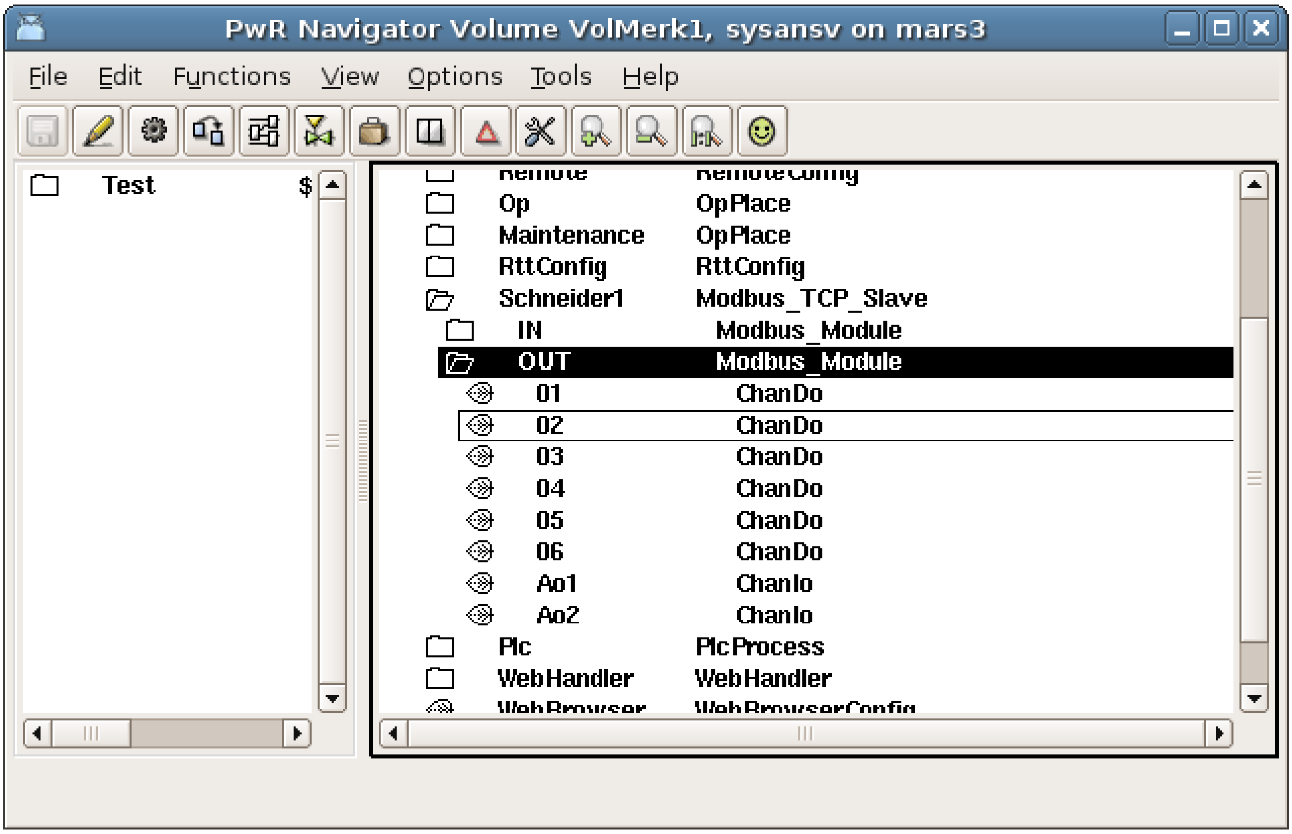
The configuration is made with 2 Modbus modules. One will read holding registers starting at address 5391 and one will write registers starting at address 0.



The input area is configured with channels as shown below. It consists of 6 ChanDi with representation Bit16 and numbered 0-5 meaning that in the first register (2 bytes, 16 bits) we will read the 6 lowest bits to the channels. The same is done for the statuses of the digital input channels. The rest of the input registers is read with ChanIi's with representation UInt16.



The output area is configured as shown below. The digital outputs are configured with ChanDo's with representation Bit16 and numbered 0 to 5. The analog outputs are specified with 2 ChanIi's, one for the respective channel.



Slave objects

Modbus_TCP_Slave

Base object for a Modbus slave. Reside in the node hierarchy. TCP/IP address of the device is configured.

Module objects

Modbus_Module

Baseclass for a Modbus module. Placed as child to a slave object. Wanted function code is chosen.

MotionControl USB I/O

Motion Control USB I/O is a device manufactured by Motion Control, www.motioncontrol.se. The device is connected to the USB port on the PC. The unit contains 21 channels of different types, divided into 3 ports, A, B and C. The first four channels (A1 – A4) are Digital outputs of relay type for voltage up to 230 V. The next four channels (A5 – A8) are Digital inputs with optocouplers. Next eight channels (B1 – B8) can either be configured as digital outputs, digital inputs or analog inputs. The last 5 channels (C1 – C5) can be digital outputs or inputs, where C4 and C5 also can be configured as analog outputs.

In Proview USB I/O is configured with a rack object, OtherIO:MotionControl_USB, that is positioned in the node hierarchy under the $Node object, and a card object, OtherIO:MotionControl_USBIO. Below the card object, channel objects of the type that corresponds to the configuration of the card, are placed.

The card has a watchdog the resets the outputs of the card, if the card is not written to within a certain time.

For the moment, the driver can only handle one device.

Driver

Download and unpack the tar-file for the driver.

```
> tar -xvzf usbio.tar.tz
```

Build the driver with make

```
> cd usbio/driver/linux-2.6
> make
```

Install the driver `usbio.ko` as root

```
> insmod usbio.ko
```

Allow all to read and write to the driver

```
> chmod a+rw /dev/usbio0
```

There is also an API to the driver with an archive, usbio/test/libusbio.a. Copy the archive to /usr/lib or $pwrp_lib on the development station.

Rack object

MotonControl_USB

The rack object is placed under the $Node object in the node hierarchy. Process should be 1. Connect the object to a PLC thread by selecting a PlcThread object and activate *Connect PlcThread* in the popup menu for the rack object.

Kortobjekt

MotionControl_USBIO

The card object is positioned under the rack object. Process should also here be 1 and the object should be connected to a PLC thread. State the card identity, that is found on the circuit card, in the Address attribute. The watchdog is activated if a value is set in WatchdogTime, that states the timeout time in seconds.

Channels

The channels of the card are configured under the card object with channels objects. The Number attribute of the channel object states which channel the object configures (0 – 20), and the class of the object states if the channel is used as a Di, Do, Ai or Ao. The table below displays how the channels can be configured.

Channel	*Type*	*Number*
A1	ChanDo	0
A2	ChanDo	1
A3	ChanDo	2
A4	ChanDo	3
A5	ChanDi	4
A6	ChanDi	5
A7	ChanDi	6
A8	ChanDi	7
B1	ChanDi, ChanDo or ChanAi	8
B2	ChanDi, ChanDo or ChanAi	9
B3	ChanDi, ChanDo or ChanAi	10
B4	ChanDi, ChanDo or ChanAi	11
B5	ChanDi, ChanDo or ChanAi	12
B6	ChanDi, ChanDo or ChanAi	13
B7	ChanDi, ChanDo or ChanAi	14
B8	ChanDi, ChanDo or ChanAi	15
C1	ChanDi or ChanDo	16
C2	ChanDi or ChanDo	17
C3	ChanDi or ChanDo	18
C4	ChanDi, ChanDo or ChanAo	19
C5	ChanDi, ChanDo or ChanAo	20

Ai configuration

The Ai channels has raw value range 0 – 1023 and signal value range 0 – 5 V, i.e. RawValRange and ChannelSigValRange should be set to

RawValRangeLow	0
RawValRangeHigh	1023
ChannelSigValRangeLow	0
ChannelSigValRangeHigh	5

For example, to configure ActualValue range to 0 – 100, set SensorSigValRange 0 - 5 and ActValRange 0 – 100.

Ao configuration

The Ao channels has raw value range 0 – 5 and signal value range 0 – 5, i.e. RawValRange and ChannelSigValRange should be set to

RawValRangeLow	0
RawValRangeHigh	5
ChannelSigValRangeLow	0
ChannelSigValRangeHigh	5

For example, to configure ActualValue range to 0 – 100, set SensorSigValRange 0 – 5 and ActValRange 0 – 100.

Link file

The archive with the driver API has to be linked to the PLC program. This is done by creating the file $pwrp_exe/ plc_'nodename'_'busnumber'.opt, e.g. `$pwrp_exe/plc_mynode_0517.opt` with the content

```
$pwr_obj/rt_io_user.o -lusbio
```

Adaption of I/O systems

This section will describe how to add new I/O systems to Proview.

Adding a new I/O system requires knowledge of how to create classes in Proview, and basic knowledge of C programming.

An I/O system can be added for a single project, for a number of projects, or for the Proview base system. In the latter case you have to install and build from the Proview source code.

Overview

The I/O handling in Proview consist of a framework that identifies the I/O objects on a process node, and calls the methods of the I/O objects to fetch or transmit data.

Levels

The I/O objects in a process node are configured in four levels: agent, rack, cards and channels. The channel objects can be configured as individual objects or as internal objects in a card object.

To the agent, rack and card objects methods can be registered. The methods can be of type Init, Close, Read, Write or Swap, and is called by the I/O framework in a specific order. The functionality of an I/O object consists of the attributes of the object, and the registered methods of the object. Everything the framework does is to identify the objects, select the objects that are valid for the current process, and call the methods for these objects in a specific order.

Look at a centralized I/O system with digital input cards and digital output cards mounted on the local bus of the process node. In this case the agent level is superfluous and represented by the $Node object. Below the $Node object is placed a rack object with an open and a close method. The open method attaches to the driver of the I/O system. Below the rack object, card objects for the Di and Do cards are configured. The Di card has an Open and a Close method that initiates and closes down the card, and a Read method the fetches the values of the inputs of the card. The Do card also has Open and Close methods, and a Write method that transfers suitable values to the outputs of the card.

If we study another I/O system, Profibus, the levels are not as easy to identify as in the previous example. Profibus is a distributed system, with a master card mounted on the local PCI-bus, that communicates via a serial connection to slaves positioned in the plant. Each slave can contain modules of different type, e.g. one module with 4 Di channels, and one with 2 Ao channels. In this case the master card represents the agent level, the slaves the rack level and the modules the card level.

The Agent, rack and card levels are very flexible, and mainly defined by the attributes and the methods of the classes of the I/O system. This does not apply to the channel level that consists of the object ChanDi, ChanDo, ChanAi, ChanAo, ChanIi, ChanIo and ChanCo. The task for the channel object is to represent an input or output value on an I/O unit, and transfer this value to the signal object that is connected to the channel object. The signal object reside in the plant hierarchy and represents for example a sensor or an order to an actuator in the plant. As there is a physical connection between the sensor in the plant and the channel on the I/O card, also the signal objects are connected to the channel object. PLC programs, HMI and applications refer to the signal object that represents the component in the plant, not the channel object, representing a channel on an I/O unit.

Area object

Values that are fetched from input units and values that are put out to output units are stored in special area objects. The area objects are created dynamically in runtime and reside in the system volume under the hierarchy pwrNode-active-io. There are one area object for each signal type. Normally you refer to the value of a signal through the ActualValue attribute of the signal. This attribute actually contains a pointer that points to the area object, and the attribute ValueIndex states in which index in the area object the signal value can be found. The reason to this construction with area objects is that during the execution of a logical net, you don't want any changes of signal values. Each PLC thread therefore takes a copy of the area objects before the start of the execution, and reads signal values from the copy, calculated output signal values though, are written in the area object.

I/O objects

The configuration of the I/O is done in the node hierarchy below the $Node object. To each type of component in the I/O hierarchy you create a class that contains attributes and methods. The methods are of type Open, Close, Read, Write and Swap, and is called by the I/O framework. The methods connects to the bus and read data that are transferred to the area objects, or fetches data from the area objects that are put out on the bus.

Processes

There are two system processes in Proview that calls the I/O framework,the PLC process and rt_io_comm. In the PLC process each thread makes an initialization of the I/O framework, which makes it possible to read and write I/O units synchronized with the execution of the PLC code for the threads.

Framework

The main task for the I/O framework is to identify I/O objects and call the methods that are registered for the objects.

A first initialization is made at start of the runtime environment, when the area objects are created, and each signal is allocated a place in the area object. The connections between signals and channels are also checked. When signals and channels are connected in the development environment, the identity for the channel is stored in the signals *SigChanCon* attribute. Now the identity of the signal object is put into the channels *SigChanCon* attribute, thus making it easy to find the signal from the channel.

The next initialization is made by every process that wants to connect to the I/O handling. The PLC process and rt_io_comm does this initialization, but also applications that need to read or write directly to I/O units can connect. At the initialization a data structure i allocated with all agents, racks, cards and channels that is to be handled by the current process, and the init methods for them are called. The process then makes a cyclic call of a read and write function, that calls the read and write methods for the I/O objects in the data structure.

Methods

The task of the methods are to initiate the I/O system, perform reading and writing to the I/O units, and finally disconnect the I/O system. How these tasks are divided, depend on the construction of the I/O system. In a centralized I/O on the local bus, methods for the different card objects can attach the bus and read and write data themselves to their unit, and the methods for the agent and rack object doesn't have much to do. In a distributed I/O the information for the units are often gathered in a package, and it is the methods of the agent or rack object that receives the package and distribute its content on different card objects. The card object methods identifies data for its channels, performs any conversion and writes or reads data in the area object.

Framework

A process can initiate the I/O framework by calling io_ini(). As argument you send a bitmask that indicates which process you are, and the threads of the PLC process also states the current thread. io_init() performs the following

- creates a context.
- allocates a hierarchic data structure of I/O objects with the levels agent, rack, card and channel. For agents a struct of type io_sAgent is allocated, for racks a struct of type io_sRack, for cards a struct of type io_sCard, and finally for channels a struct of type io_sChannel.
- searches for all I/O objects and checks their Process attributes. If the Process attribute matches the process sent as an argument to io_init(), the object is inserted into the data structure. If the object has a descendant that matches the process it is also inserted into the data structure. For the PLC process, also the thread argument of io_init() is checked against the ThreadObject attribute in the I/O object. The result is a linked tree structure with the agents, racks, card and channel objects that is to be handled by the current process.
- for every I/O objects that is inserted, the methods are identified, and pointers to the method's functions are fetched. Also pointers to the object and the objects name, is inserted in the data structure.
- the init methods for the I/O objects in the data structure is called. The methods of the first agent is called first, and then the first rack of the agent, the first card of the rack etc.

When the initialization is done, the process can call io_read() to read from the I/O units that are present i the data structure, and io_write() to put out values. A thread in the PLC process calls io_read() every scan to fetch new values from the process. Then the PLC code is executed and io_write() is called to put out new values. The read methods are called in the same order as the init methods, and the write methods in reverse order.

When the process terminates, io_close() is called, which calls the close methods of the objects in the data structure. The close methods are called in reverse order compared to the init methods.

When a soft restart is performed, a restart of the I/O handling is also performed. First the close methods are called, and then, during the time the restart lasts, the swap methods are called, and then the init methods. The call to the swap methods are done by rt_io_comm.

io_init, function to initiate the framework

```
pwr_tStatus io_init(
  io_mProcess   process,
```

```
  pwr_tObjid    thread,
  io_tCtx       *ctx,
  int           relativ_vector,
  float         scan_time
);
```

io_sCtx, the context of the framework

```
struct io_sCtx {
   io_sAgent       *agentlist;      /* List of agent structures */
   io_mProcess     Process;         /* Callers process number */
   pwr_tObjid      Thread;          /* Callers thread objid */
   int             RelativVector;   /* Used by plc */
   pwr_sNode       *Node;           /* Pointer to node object */
   pwr_sClass_IOHandler    *IOHandler; /* Pointer to IO Handler object */
   float           ScanTime;        /* Scantime supplied by caller */
   io_tSupCtx      SupCtx;          /* Context for supervise object lists */
};
```

Data structure for an agent

```
typedef struct s_Agent {
   pwr_tClassId    Class;           /* Class of agent object */
   pwr_tObjid      Objid;           /* Objid of agent object */
   pwr_tOName      Name;            /* Full name of agent object */
   io_mAction      Action;          /* Type of method defined (Read/Write)*/
   io_mProcess     Process;         /* Process number */
   pwr_tStatus     (* Init) ();     /* Init method */
   pwr_tStatus     (* Close) ();    /* Close method */
   pwr_tStatus     (* Read) ();     /* Read method */
   pwr_tStatus     (* Write) ();    /* Write method */
   pwr_tStatus     (* Swap) ();     /* Write method */
   void            *op;             /* Pointer to agent object */
   pwr_tDlid       Dlid;            /* Dlid for agent object pointer */
   int             scan_interval;   /* Interval between scans */
   int             scan_interval_cnt;/* Counter to detect next time to scan */
   io_sRack        *racklist;       /* List of rack structures */
   void            *Local;          /* Pointer to method defined data structure*/
   struct s_Agent  *next;           /* Next agent */
} io_sAgent;
```

Data structure for a rack

```
typedef struct s_Rack {
   pwr_tClassId    Class;           /* Class of rack object */
   pwr_tObjid      Objid;           /* Objid of rack object */
   pwr_tOName      Name;            /* Full name of rack object */
   io_mAction      Action;          /* Type of method defined (Read/Write)*/
   io_mProcess     Process;         /* Process number */
   pwr_tStatus     (* Init) ();     /* Init method */
   pwr_tStatus     (* Close) ();    /* Close method */
   pwr_tStatus     (* Read) ();     /* Read method */
   pwr_tStatus     (* Write) ();    /* Write method */
   pwr_tStatus     (* Swap) ();     /* Swap method */
   void            *op;             /* Pointer to rack object */
   pwr_tDlid       Dlid;            /* Dlid för rack object pointer */
   pwr_tUInt32     size;            /* Size of rack data area in byte */
   pwr_tUInt32     offset;          /* Offset to rack data area in agent */
   int             scan_interval;   /* Interval between scans */
   int             scan_interval_cnt;/* Counter to detect next time to scan */
```

```
   int             AgentControlled;/* TRUE if kontrolled by agent */
   io_sCard        *cardlist;      /* List of card structures */
   void            *Local;         /* Pointer to method defined data structure*/
   struct s_Rack   *next;          /* Next rack */
} io_sRack;
```

Data structure for a card

```
typedef struct s_Card {
   pwr_tClassId    Class;          /* Class of card object */
   pwr_tObjid      Objid;          /* Objid of card object */
   pwr_tOName      Name;           /* Full name of card object */
   io_mAction      Action;         /* Type of method defined (Read/Write)*/
   io_mProcess     Process;        /* Process number */
   pwr_tStatus     (* Init) ();    /* Init method */
   pwr_tStatus     (* Close) ();   /* Close method */
   pwr_tStatus     (* Read) ();    /* Read method */
   pwr_tStatus     (* Write) ();   /* Write method */
   pwr_tStatus     (* Swap) ();    /* Write method */
   pwr_tAddress    *op;            /* Pointer to card object */
   pwr_tDlid       Dlid;           /* Dlid for card object pointer */
   pwr_tUInt32     size;           /* Size of card data area in byte */
   pwr_tUInt32     offset;         /* Offset to card data area in rack */
   int             scan_interval;  /* Interval between scans */
   int             scan_interval_cnt;/* Counter to detect next time to scan */
   int             AgentControlled;/* TRUE if kontrolled by agent */
   int             ChanListSize;   /* Size of chanlist */
   io_sChannel     *chanlist;      /* Array of channel structures */
   void            *Local;         /* Pointer to method defined data structure*/
   struct s_Card   *next;          /* Next card */
} io_sCard;
```

Data structure for a channel

```
typedef struct {
   void            *cop;           /* Pointer to channel object */
   pwr_tDlid       ChanDlid;       /* Dlid for pointer to channel */
   pwr_sAttrRef    ChanAref;       /* AttrRef for channel */
   void            *sop;           /* Pointer to signal object */
   pwr_tDlid       SigDlid;        /* Dlid for pointer to signal */
   pwr_sAttrRef    SigAref;        /* AttrRef for signal */
   void            *vbp;           /* Pointer to valuebase for signal */
   void            *abs_vbp;       /* Pointer to absvaluebase (Co only) */
   pwr_tClassId    ChanClass;      /* Class of channel object */
   pwr_tClassId    SigClass;       /* Class of signal object */
   pwr_tUInt32     size;           /* Size of channel in byte */
   pwr_tUInt32     offset;         /* Offset to channel in card */
   pwr_tUInt32     mask;           /* Mask for bit oriented channels */
} io_sChannel;
```

Create I/O objects

For a process node the I/O system is configured in the node hierarchy with objects of type agent, rack and card. The classes for these objects are created in the class editor. The classes are defined with a $ClassDef object, a $ObjBodyDef object (RtBody), and below this one $Attribute object for each attribute of the class. The attributes are determined by the functionality of the methods of the class, but there are some common attributes (*Process*, *ThreadObject* and *Description*). In the $ClassDef objects, the *Flag* word should be stated if it is an agent, rack or card object, and the methods are defined with specific Method objects.

It is quite common that several classes in an I/O system share attributes and maybe even methods. An input card that is available with different number of inputs, can often use the same methods. What differs is the number of channel objects. The other attributes can be stored in a baseclass, that also contains the methods-objects. The subclasses inherits both the attributes and the methods. The y are extended with channel objects, that can be put as individual attributes, or, if they are of the same type, as a vector of channel objecs. If the channels are put as a vector or as individual attributes, depend on the how the reference in the PLC documents should look. With an array you get an index starting from zero, with individual objects you can control the naming of the attributes yourself.

In the example below a baseclass is viewed in Fig *Example of a baseclass* and a subclass in Fig *Example of a card class with a superclass and 32 channel objects*. The baseclass Ssab_BaseDiCard contains all the attributes used by the I/O methods and the I/O framework. The subclass Ssab_DI32D contains the Super attribute with TypeRef Sasb_BaseDiCard, and 32 channel attributes of type ChanDi. As the index for this card type by tradition starts from 1, the channels are put as individual attributes, but they could also be an array of type ChanDi.

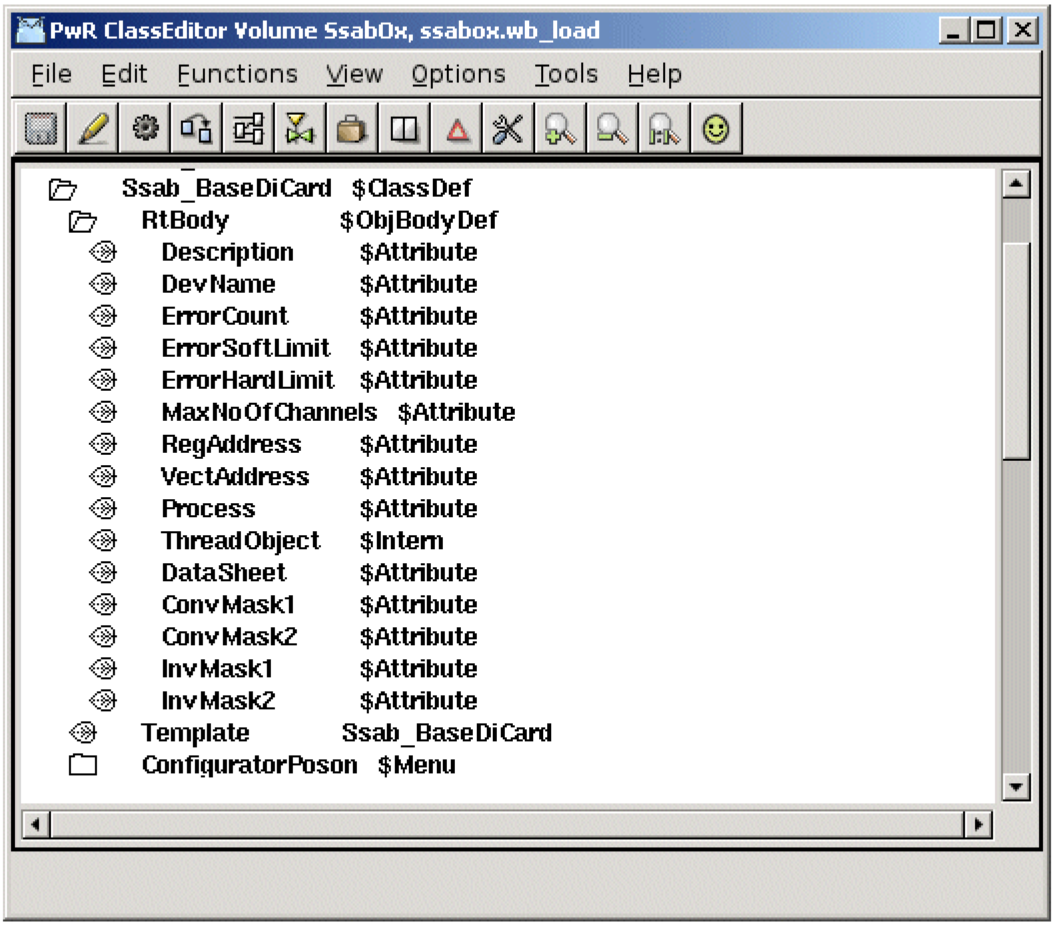


Fig Example of a baseclass for a card

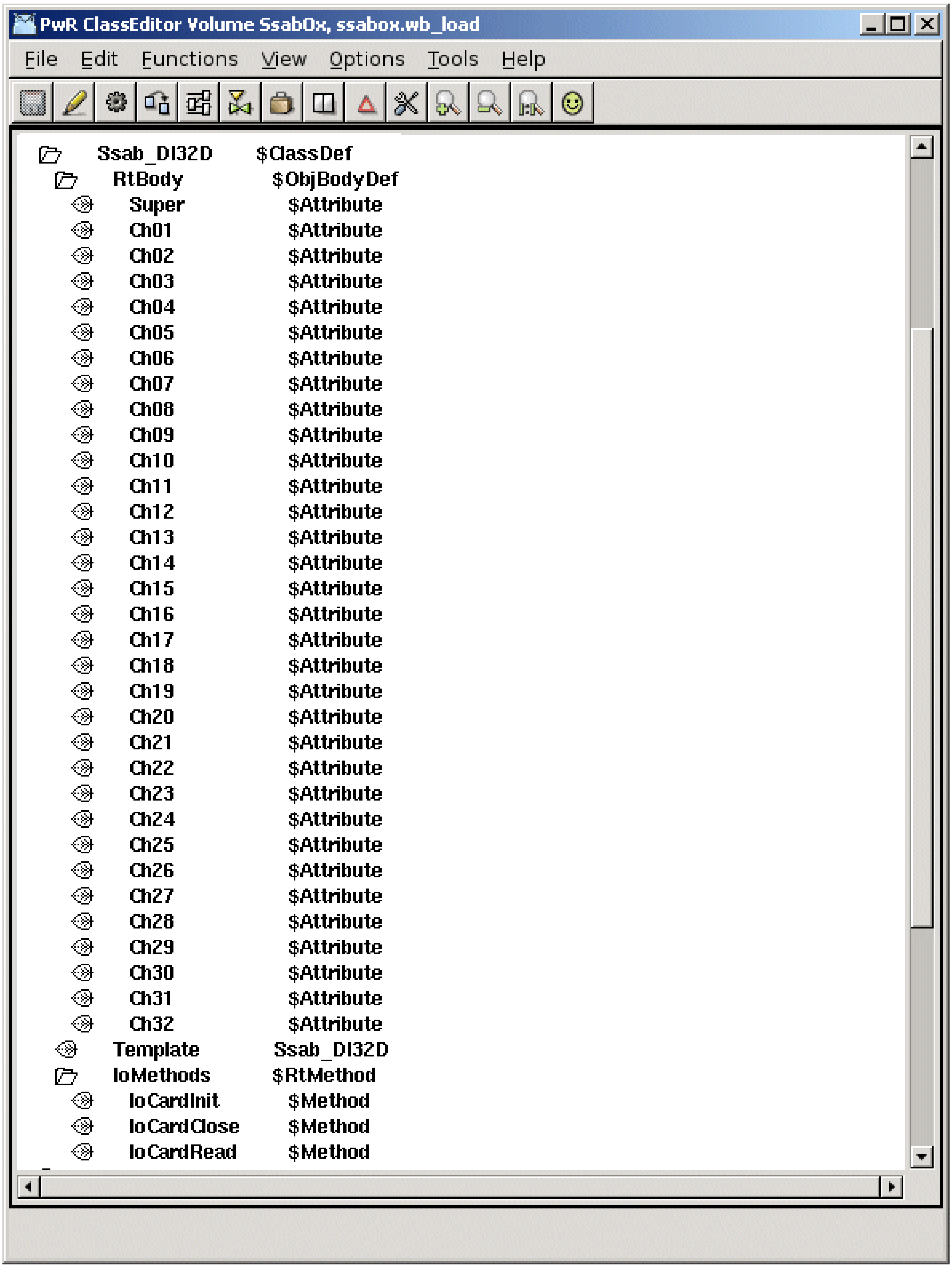


Fig Example of a cardclass with a superclass an 32 channel objecs

Flags

In the *Flag* attribute of the $ClassDef object, the *IOAgent* bit should be set for agent classes, the *IORack* bit for rack classes and the *IOCard* bit for card classes.

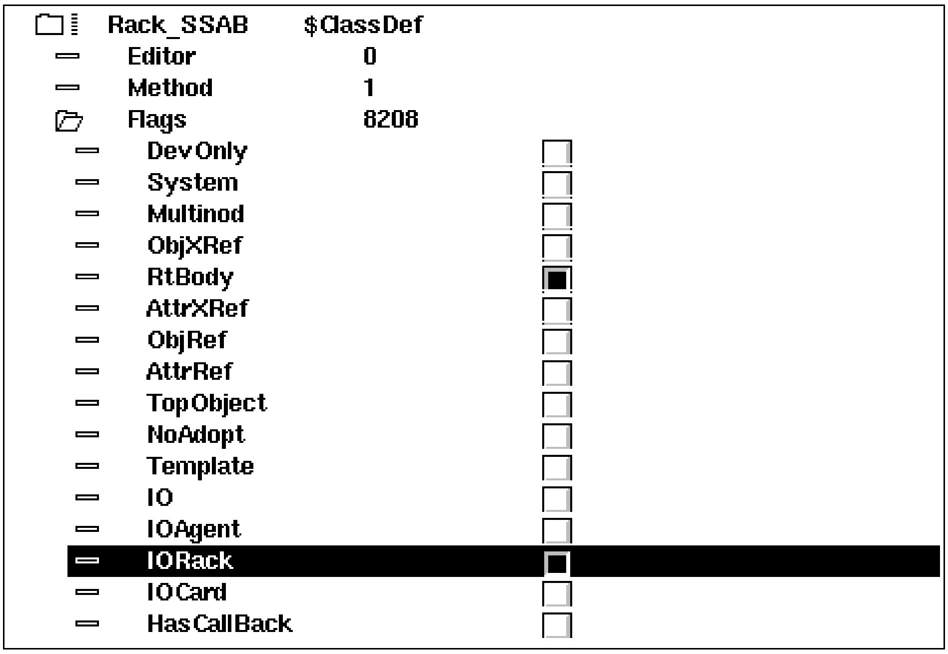


Fig IORack bit set for a rack class

Attributes

Description

Attribute of type pwrs:Type-$String80. The content is displayed as description in the navigator.

Process

Attribute of type pwrs:Type-$Uint32. States which process should handle the unit.

ThreadObject

Attribute of type pwrs:Type-$Objid. States which thread in the PLC process should handle the unit.

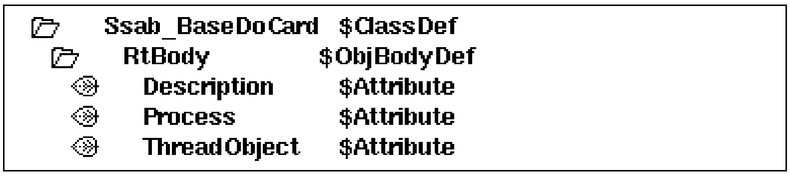


Fig Standard attributes

Method objects

The method objects are used to identify the methods of the class. The methods consist of C functions that are registered in the C code with a name, a string that consists of class name and method name, e.g. “Ssab_AIuP-IoCardInit”. The name is also stored in a method object in the class description, and makes is possible for the I/O framework to find the correct C function for the class.

Below the $ClassDef object, a $RtMethod object is placed with the name IoMethods. Below this one $Method object is placed for each method that is to be defined for the class. In the attribute MethodName the name of the method is stated.

Agents

For agents, $Method objects with the name IoAgentInit, IoAgentClose, IoAgentRead and IoAgentWrite are created.

Racks

For racks, $Method objects with the names IoRackInit, IoRackClose, IoRackRead och IoRackWrite are created.

Cards

For cards $Method objects with the names IoCardInit, IoCardClose, IoCardRead och IoCardWrite are created.

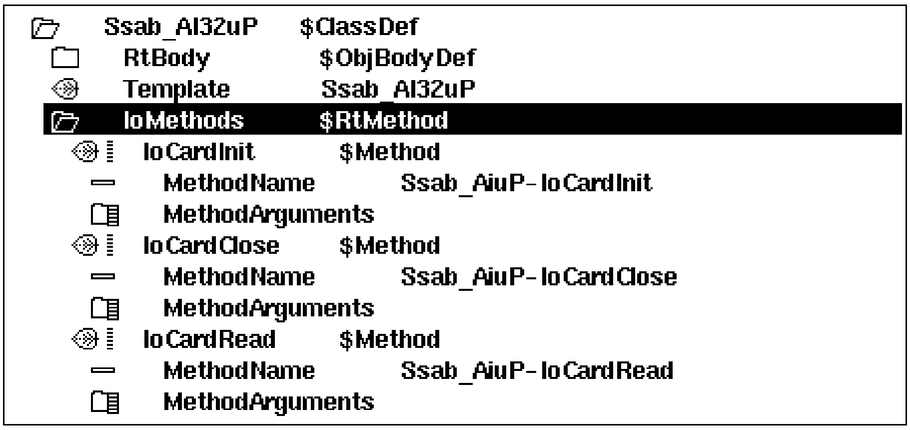


Fig Method objects

Connect-method for a ThreadObject

When the thread object in attribute *ThreadObject* should be stated for an I/O object, it can be typed manually, but one can also specify a menu method that inserts the selected thread object into the attribute. The method is activated from the popup menu of the I/O object in the configurator.

The method is defined in the class description with a $Menu and a $MenuButton object, see *Fig Connect Metod*. Below the $ClassDef object a $Menu object with the name *ConfiguratorPoson* is placed. Below this, another $Menu object named *Pointed*, and below this a $MenuButton object named *Connect.* State *ButtonName* (text in the popup menu for the method), MethodName and FilterName. The method and the filter used is defined in the $Objid class. *MethodName* should be $Objid-Connect and *FilterName* $Objid-IsOkConnected.

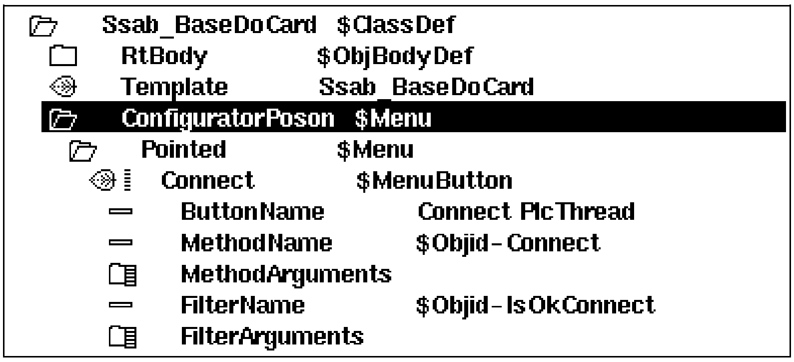


Fig Connect method

Methods

For the agent, rack and card classes you write methods in the programming language c. A method is a c function that is common for a class (or several classes) and that is called by the I/O framework for all instances of the class. To keep the I/O handling as flexible as possible, the methods are doing most of the I/O handling work. The task for the framework is to identify the various I/O objects and to call the methods for these, and to supply the methods with proper data structures.

There are five types of methods: Init, Close, Read, Write and Swap.

- Init-method is called at initialization of the I/O handling, i.e. at startup of the runtime environment and at soft restart.
- Close-method is called when the I/O handling is terminated, i.e. when the runtime environment is stopped or at a soft restart.
- Read-method is called cyclic when its time to read the input cards.
- Write-method is called cyclic when its time to put out values to the output cards.
- Swap-method is called during a soft restart.

Local data structure

In the data structures io_sAgent, io_sRack and io_sCard there is an element, *Local,* where the method can store a pointer to local data for an I/O unit. Local data is allocated by the init method and then available at each method call.

Agent-Methods

IoAgentInit

Initialization method for an agent.

```
static pwr_tStatus IoAgentInit( io_tCtx       ctx,
                                io_sAgent     *ap)
```

IoAgentClose

Close method for an agent.

```
static pwr_tStatus IoAgentClose( io_tCtx       ctx,
                                 io_sAgent     *ap)
```

IoAgentRead

Read method for an agent.

```
static pwr_tStatus IoAgentRead( io_tCtx       ctx,
                                io_sAgent     *ap)
```

IoAgentWrite

Write method for an agent.

```
static pwr_tStatus IoAgentWrite( io_tCtx       ctx,
                                 io_sAgent     *ap)
```

IoAgentSwap

Swap method for an agent.

```
static pwr_tStatus IoAgentSwap( io_tCtx       ctx,
                                io_sAgent     *ap)
```

Rack-metoder

IoRackInit

```
static pwr_tStatus IoRackInit( io_tCtx       ctx,
                               io_sAgent     *ap,
                               io_sRack      *rp)
```

IoRackClose

```
static pwr_tStatus IoRackClose( io_tCtx       ctx,
                                io_sAgent     *ap,
                                io_sRack      *rp)
```

IoRackRead

```
static pwr_tStatus IoRackRead( io_tCtx       ctx,
                               io_sAgent     *ap,
                               io_sRack      *rp)
```

IoRackWrite

```
static pwr_tStatus IoRackWrite( io_tCtx       ctx,
                                io_sAgent     *ap,
                                io_sRack      *rp)
```

IoRackSwap

```
static pwr_tStatus IoRackSwap( io_tCtx       ctx,
                               io_sAgent     *ap,
                               io_sRack      *rp)
```

Card-metoder

IoCardInit

```
static pwr_tStatus IoCardInit( io_tCtx       ctx,
                               io_sAgent     *ap,
```

```
                                io_sRack      *rp,
                                io_sCard      *cp)
```

IoCardClose

```
static pwr_tStatus IoCardClose( io_tCtx       ctx,
                                io_sAgent     *ap,
                                io_sRack      *rp,
                                io_sCard      *cp)
```

IoCardRead

```
static pwr_tStatus IoCardRead( io_tCtx       ctx,
                               io_sAgent     *ap,
                               io_sRack      *rp,
                               io_sCard      *cp)
```

IoCardWrite

```
static pwr_tStatus IoCardWrite( io_tCtx       ctx,
                                io_sAgent     *ap,
                                io_sRack      *rp,
                                io_sCard      *cp)
```

IoCardSwap

```
static pwr_tStatus IoCardSwap( io_tCtx       ctx,
                               io_sAgent     *ap,
                               io_sRack      *rp,
                               io_sCard      *cp)
```

Method registration

The methods for a class have to be registered, so that you from the the method object in the class description can find the correct functions for a class. Below is an example of how the methods IoCardInit, IoCardClose and IoCardRead are registered for the class Ssab_AiuP.

```
pwr_dExport pwr_BindIoMethods(Ssab_AiuP) = {
  pwr_BindIoMethod(IoCardInit),
  pwr_BindIoMethod(IoCardClose),
  pwr_BindIoMethod(IoCardRead),
  pwr_NullMethod
};
```

Class registration

Also the class has to be registered. This is done in different ways dependent on whether the I/O system is implemented as a module in the Proview base system, or as a part of a project.

Module in Proview base system

If the I/O system are implemented as a module in the Proview base system, you create a file `lib/rt/src/rt_io_'modulename'.meth`, and list all the classes that have registered methods in this file.

Project

If the I/O system is a part of a project, the registration is made in a c module that is linked with the PLC program. In the example below, the classes Ssab_Rack and Ssab_AiuP are registered in the file ra_plc_user.c

```
#include "pwr.h"
#include "rt_io_base.h"

pwr_dImport pwr_BindIoUserMethods(Ssab_Rack);
pwr_dImport pwr_BindIoUserMethods(Ssab_Aiup);

pwr_BindIoUserClasses(User) = {
  pwr_BindIoUserClass(Ssab_Rack),
  pwr_BindIoUserClass(Ssab_Aiup),
  pwr_NullClass
};
```

The file is compiled and linked with the PLC program by creating a link file on $pwrp_exe. The file should be named `plc_'nodename'_'busnumber'.opt`, e.g. `plc_mynode_0517.opt`. The content of the file is sent as input data to the linker, ld, and you must also add the module with the methods of the class. In the example below these modules are supposed to be found in the archive `$pwrp_lib/libpwrp.a.`

```
$pwr_obj/rt_io_user.o -lpwrp
```

Example of rack methods

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>

#include "pwr.h"
#include "pwr_baseclasses.h"
#include "pwr_ssaboxclasses.h"
#include "rt_io_base.h"
#include "rt_errh.h"
#include "rt_io_rack_init.h"
#include "rt_io_m_ssab_locals.h"
#include "rt_io_msg.h"

/* Init method */
static pwr_tStatus IoRackInit( io_tCtx ctx,
                               io_sAgent *ap,
                               io_sRack   *rp)
{
  io_sRackLocal   *local;

  /* Open Qbus driver */
  local = calloc( 1, sizeof(*local));
  rp->Local = local;

  local->Qbus_fp = open("/dev/qbus", O_RDWR);
  if ( local->Qbus_fp == -1) {
    errh_Error( "Qbus initialization error, IO rack %s", rp->Name);
    ctx->Node->EmergBreakTrue = 1;
    return IO__ERRDEVICE;
  }

  errh_Info( "Init of IO rack %s", rp->Name);
  return 1;
}

/* Close method */
static pwr_tStatus IoRackClose( io_tCtx  ctx,
                                io_sAgent *ap,
```

```
                                io_sRack *rp)
{
  io_sRackLocal   *local;

  /* Close Qbus driver */
  local = rp->Local;

  close( local->Qbus_fp);
  free( (char *)local);

  return 1;
}

/* Every method to be exported to the workbench should be registred here. */

pwr_dExport pwr_BindIoMethods(Rack_SSAB) = {
  pwr_BindIoMethod(IoRackInit),
  pwr_BindIoMethod(IoRackClose),
  pwr_NullMethod
};
```

Example of the methods of a digital input card

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

#include "pwr.h"
#include "rt_errh.h"
#include "pwr_baseclasses.h"
#include "pwr_ssaboxclasses.h"
#include "rt_io_base.h"
#include "rt_io_msg.h"
#include "rt_io_filter_di.h"
#include "rt_io_ssab.h"
#include "rt_io_card_init.h"
#include "rt_io_card_close.h"
#include "rt_io_card_read.h"
#include "qbus_io.h"
#include "rt_io_m_ssab_locals.h"

/* Local data */
typedef struct {
      unsigned int      Address[2];
      int         Qbus_fp;
      struct {
        pwr_sClass_Di *sop[16];
        void      *Data[16];
        pwr_tBoolean Found;
      } Filter[2];
      pwr_tTime      ErrTime;
} io_sLocal;

/* Init method */
static pwr_tStatus IoCardInit( io_tCtx   ctx,
                               io_sAgent *ap,
                               io_sRack   *rp,
                               io_sCard   *cp)
```

```
{
  pwr_sClass_Ssab_BaseDiCard  *op;
  io_sLocal                   *local;
  int                   i, j;

  op = (pwr_sClass_Ssab_BaseDiCard *) cp->op;
  local = calloc( 1, sizeof(*local));
  cp->Local = local;

  errh_Info( "Init of di card '%s'", cp->Name);

  local->Address[0] = op->RegAddress;
  local->Address[1] = op->RegAddress + 2;
  local->Qbus_fp = ((io_sRackLocal *)(rp->Local))->Qbus_fp;

  /* Init filter */
  for ( i = 0; i < 2; i++) {
    /* The filter handles one 16-bit word */
    for ( j = 0; j < 16; j++)
      local->Filter[i].sop[j] = cp->chanlist[i*16+j].sop;
    io_InitDiFilter( local->Filter[i].sop, &local->Filter[i].Found,
            local->Filter[i].Data, ctx->ScanTime);
  }

  return 1;
}

/* Close method */
static pwr_tStatus IoCardClose( io_tCtx  ctx,
                                io_sAgent *ap,
                                io_sRack *rp,
                                io_sCard *cp)
{
  io_sLocal             *local;
  int             i;

  local = (io_sLocal *) cp->Local;

  errh_Info( "IO closing di card '%s'", cp->Name);

  /* Free filter data */
  for ( i = 0; i < 2; i++) {
    if ( local->Filter[i].Found)
      io_CloseDiFilter( local->Filter[i].Data);
  }
  free( (char *) local);

  return 1;
}

/* Read method */
static pwr_tStatus IoCardRead( io_tCtx   ctx,
                               io_sAgent *ap,
                               io_sRack   *rp,
                               io_sCard   *cp)
{
  io_sLocal             *local;
  io_sRackLocal         *r_local = (io_sRackLocal *)(rp->Local);
  pwr_tUInt16           data = 0;
  pwr_sClass_Ssab_BaseDiCard  *op;
  pwr_tUInt16           invmask;
  pwr_tUInt16           convmask;
```

```
  int             i;
  int             sts;
  qbus_io_read          rb;
  pwr_tTime             now;

  local = (io_sLocal *) cp->Local;
  op = (pwr_sClass_Ssab_BaseDiCard *) cp->op;

  for ( i = 0; i < 2; i++) {
    if ( i == 0) {
      convmask = op->ConvMask1;
      invmask = op->InvMask1;
    }
    else {
      convmask = op->ConvMask2;
      invmask = op->InvMask2;
      if ( !convmask)
        break;
      if ( op->MaxNoOfChannels == 16)
        break;
    }

    /* Read from local Q-bus */
    rb.Address = local->Address[i];
    sts = read( local->Qbus_fp, &rb, sizeof(rb));
    data = (unsigned short) rb.Data;

    if ( sts == -1) {
      /* Increase error count and check error limits */
      clock_gettime(CLOCK_REALTIME, &now);

      if (op->ErrorCount > op->ErrorSoftLimit) {
        /* Ignore if some time has expired */
        if (now.tv_sec - local->ErrTime.tv_sec < 600)
          op->ErrorCount++;
      }
      else
        op->ErrorCount++;
      local->ErrTime = now;

      if ( op->ErrorCount == op->ErrorSoftLimit)
        errh_Error( "IO Error soft limit reached on card '%s'", cp->Name);
      if ( op->ErrorCount >= op->ErrorHardLimit)
      {
        errh_Error( "IO Error hard limit reached on card '%s', IO stopped", cp-
>Name);
        ctx->Node->EmergBreakTrue = 1;
        return IO__ERRDEVICE;
      }
      continue;
    }

    /* Invert */
    data = data ^ invmask;

    /* Filter */
    if ( local->Filter[i].Found)
      io_DiFilter( local->Filter[i].sop, &data, local->Filter[i].Data);

    /* Move data to valuebase */
    io_DiUnpackWord( cp, data, convmask, i);
  }
```

```
  return 1;
}

/* Every method to be exported to the workbench should be registred here. */

pwr_dExport pwr_BindIoMethods(Ssab_Di) = {
  pwr_BindIoMethod(IoCardInit),
  pwr_BindIoMethod(IoCardClose),
  pwr_BindIoMethod(IoCardRead),
  pwr_NullMethod
};
```

Example of the methods of a digital output card

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

#include "pwr.h"
#include "rt_errh.h"
#include "pwr_baseclasses.h"
#include "pwr_ssaboxclasses.h"
#include "rt_io_base.h"
#include "rt_io_msg.h"
#include "rt_io_filter_po.h"
#include "rt_io_ssab.h"
#include "rt_io_card_init.h"
#include "rt_io_card_close.h"
#include "rt_io_card_write.h"
#include "qbus_io.h"
#include "rt_io_m_ssab_locals.h"

/* Local data */
typedef struct {
      unsigned int      Address[2];
      int         Qbus_fp;
      struct {
        pwr_sClass_Po *sop[16];
        void      *Data[16];
        pwr_tBoolean Found;
      } Filter[2];
      pwr_tTime       ErrTime;
} io_sLocal;

/* Init method */
static pwr_tStatus IoCardInit( io_tCtx   ctx,
                               io_sAgent *ap,
                               io_sRack   *rp,
                               io_sCard   *cp)
{
  pwr_sClass_Ssab_BaseDoCard *op;
  io_sLocal            *local;
  int             i, j;

  op = (pwr_sClass_Ssab_BaseDoCard *) cp->op;
  local = calloc( 1, sizeof(*local));
  cp->Local = local;

  errh_Info( "Init of do card '%s'", cp->Name);
```

```
  local->Address[0] = op->RegAddress;
  local->Address[1] = op->RegAddress + 2;
  local->Qbus_fp = ((io_sRackLocal *)(rp->Local))->Qbus_fp;

  /* Init filter for Po signals */
  for ( i = 0; i < 2; i++) {
    /* The filter handles one 16-bit word */
    for ( j = 0; j < 16; j++) {
      if ( cp->chanlist[i*16+j].SigClass == pwr_cClass_Po)
        local->Filter[i].sop[j] = cp->chanlist[i*16+j].sop;
    }
    io_InitPoFilter( local->Filter[i].sop, &local->Filter[i].Found,
            local->Filter[i].Data, ctx->ScanTime);
  }

  return 1;
}

/* Close method */
static pwr_tStatus IoCardClose( io_tCtx  ctx,
                                io_sAgent *ap,
                                io_sRack *rp,
                                io_sCard *cp)
{
  io_sLocal             *local;
  int             i;

  local = (io_sLocal *) cp->Local;

  errh_Info( "IO closing do card '%s'", cp->Name);

  /* Free filter data */
  for ( i = 0; i < 2; i++) {
    if ( local->Filter[i].Found)
      io_ClosePoFilter( local->Filter[i].Data);
  }
  free( (char *) local);

  return 1;
}

/* Write method */
static pwr_tStatus IoCardWrite( io_tCtx  ctx,
                                io_sAgent *ap,
                                io_sRack *rp,
                                io_sCard *cp)
{
  io_sLocal             *local;
  io_sRackLocal         *r_local = (io_sRackLocal *)(rp->Local);
  pwr_tUInt16           data = 0;
  pwr_sClass_Ssab_BaseDoCard *op;
  pwr_tUInt16           invmask;
  pwr_tUInt16           testmask;
  pwr_tUInt16           testvalue;
  int             i;
  qbus_io_write         wb;
  int             sts;
  pwr_tTime             now;

  local = (io_sLocal *) cp->Local;
  op = (pwr_sClass_Ssab_BaseDoCard *) cp->op;
```

```
for ( i = 0; i < 2; i++) {
  if ( ctx->Node->EmergBreakTrue && ctx->Node->EmergBreakSelect == FIXOUT) {
    if ( i == 0)
      data = op->FixedOutValue1;
    else
      data = op->FixedOutValue2;
  }
  else
    io_DoPackWord( cp, &data, i);

  if ( i == 0) {
    testmask = op->TestMask1;
    invmask = op->InvMask1;
  }
  else {
    testmask = op->TestMask2;
    invmask = op->InvMask2;
    if ( op->MaxNoOfChannels == 16)
      break;
  }

  /* Invert */
  data = data ^ invmask;

  /* Filter Po signals */
  if ( local->Filter[i].Found)
    io_PoFilter( local->Filter[i].sop, &data, local->Filter[i].Data);

  /* Testvalues */
  if ( testmask) {
    if ( i == 0)
      testvalue = op->TestValue1;
    else
      testvalue = op->TestValue2;
    data = (data & ~ testmask) | (testmask & testvalue);
  }

  /* Write to local Q-bus */
  wb.Data = data;
  wb.Address = local->Address[i];
  sts = write( local->Qbus_fp, &wb, sizeof(wb));

  if ( sts == -1) {
    /* Increase error count and check error limits */
    clock_gettime(CLOCK_REALTIME, &now);

    if (op->ErrorCount > op->ErrorSoftLimit) {
      /* Ignore if some time has expired */
      if (now.tv_sec - local->ErrTime.tv_sec < 600)
        op->ErrorCount++;
    }
    else
      op->ErrorCount++;
    local->ErrTime = now;


    if ( op->ErrorCount == op->ErrorSoftLimit)
      errh_Error( "IO Error soft limit reached on card '%s'", cp->Name);
    if ( op->ErrorCount >= op->ErrorHardLimit)
    {
      errh_Error( "IO Error hard limit reached on card '%s', IO stopped", cp-
```

```
>Name);
        ctx->Node->EmergBreakTrue = 1;
        return IO__ERRDEVICE;
      }
      continue;
    }
  }
  return 1;
}

/* Every method to be exported to the workbench should be registred here. */
pwr_dExport pwr_BindIoMethods(Ssab_Do) = {
  pwr_BindIoMethod(IoCardInit),
  pwr_BindIoMethod(IoCardClose),
  pwr_BindIoMethod(IoCardWrite),
  pwr_NullMethod
};
```

Step by step description

This sections contains an example of how to attach an I/O system to Proview.

The I/O system in the example is USB I/O manufactured by Motion Control. It consist of a card with 21 channels of different type. The first four channels (A1 – A4) are Digital outputs of relay type for voltage up to 230 V. The next four channels (A5 – A8) are Digital inputs with optocouplers. Next eight channels (B1 – B8) can either be configured as digital outputs, digital inputs or analog inputs. The last 5 channels (C1 – C5) can be digital outputs or inputs, where C4 and C5 also can be configured as analog outputs. In our example, not wanting the code to be too complex, we lock the configuration to: channel 0-3 Do, 4-7 Di, 8-15 Ai, 16-18 Di and 19-20 Ao.

Attach to a project

In the first example we attach the I/O system to a project. We will create a class volume, and insert rack and card classes into it. We will write I/O methods for the classes and link them to the PLC program. We create I/O objects i the node hierarchy in the root volume, and install the driver for USB I/O, and start the I/O handling on the process station.

Create classes

Create a class volume

The first step is to create classes for the I/O objects. The classes are defined in class volumes, and first we have to create a class volume in the project. The class volume first has to registered in the GlobalVolumeList. We start the administrator with

```
> pwra
```

and opens the GlobalVolumeList by activating *File/Open/GlobalVolumeList* in the menu. We enter edit mode and create a VolumeReg object with the name *CVolMerk1*. The volume identity for user class volumes should chosen in the interval 0.0.2-249.1-254 and we choose 0.0.99.20 as the identity for our class volume. In the attribute Project the name of our project is stated, *mars2*.

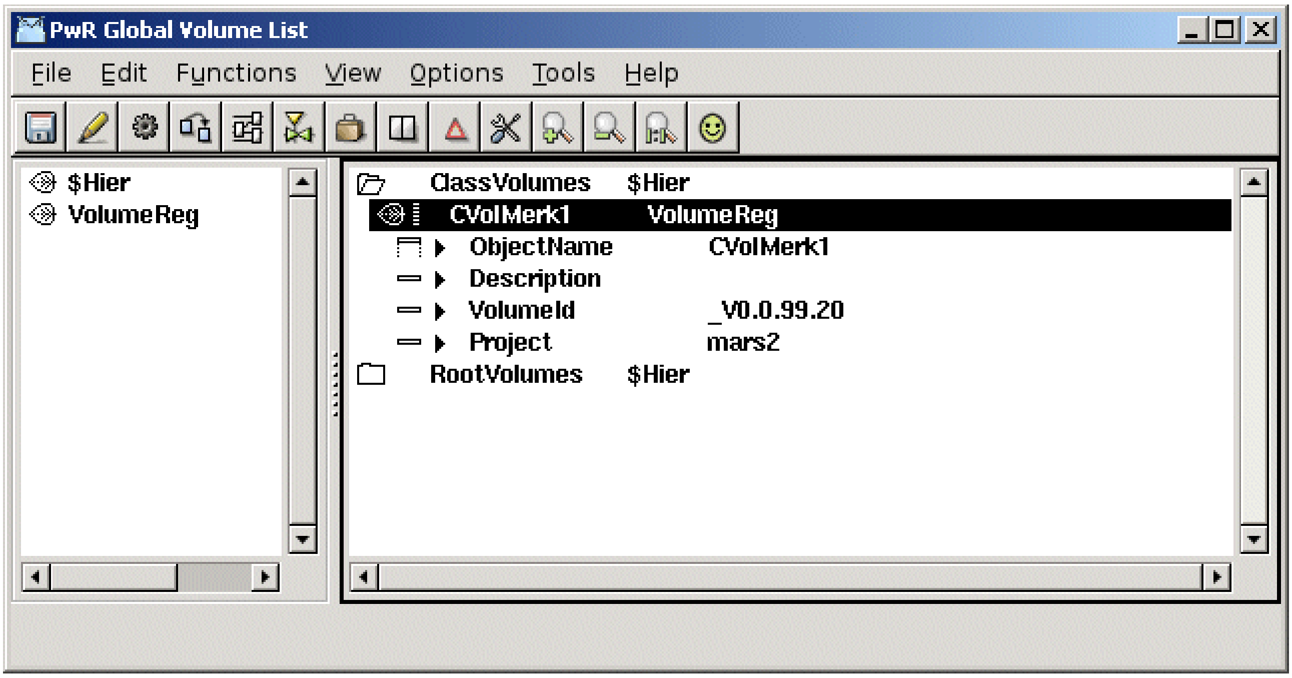


Fig Classvolume registration

Open the class volume

Next step is to configure and create the class volume in the project. This is done in the directory volume.

We enter the directory volume

```
> pwrs
```

in edit mode and create an object of type ClassVolumeConfig in the volume hierarchy. The object is named with the volume name CVolMerk1. After saving and leaving edit mode we can open the class volume by activating *OpenClassEditor...* in the popup menu of the object.

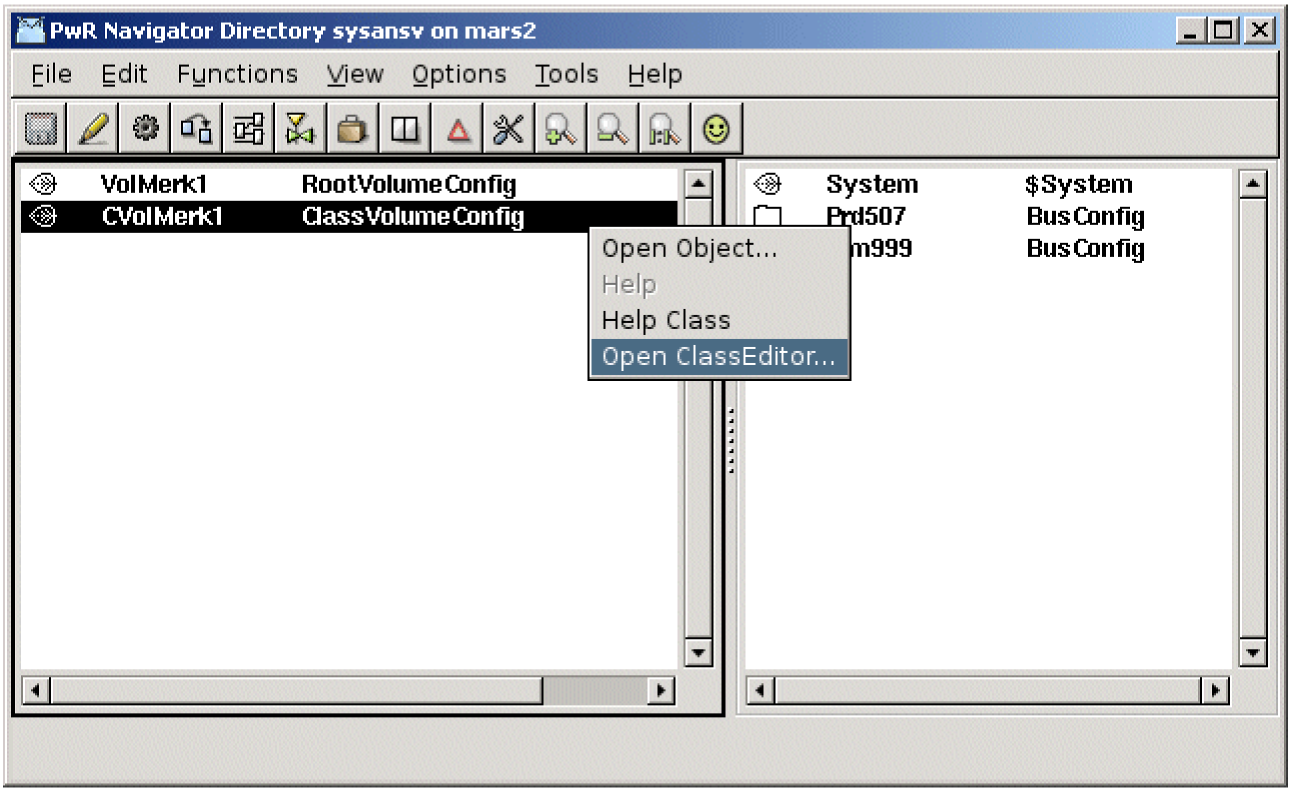


Fig Configuration of the class volume in the directory volume and start of the class editor

In the class editor classes are defined with specific class definition objects. We are going to create two classes, a rack class, *MotionControl_USB* and a card class *MotionControl_USBIO*.

Note! The class names has to be unique, which actually is not true for these names any more as they exist in the volume OtherIO.

Create a rack class

In our case the card class will do all the work and contain all the methods. The rack class is there only to inhabit the rack level, and doesn't have any methods or attributes. We will only put a description attribute in the class. We create a $ClassHier object, and under this a $ClassDef object for the class rack. The object is named MotionControl_USB and we set the IORack and IO bits in the Flag attribute.

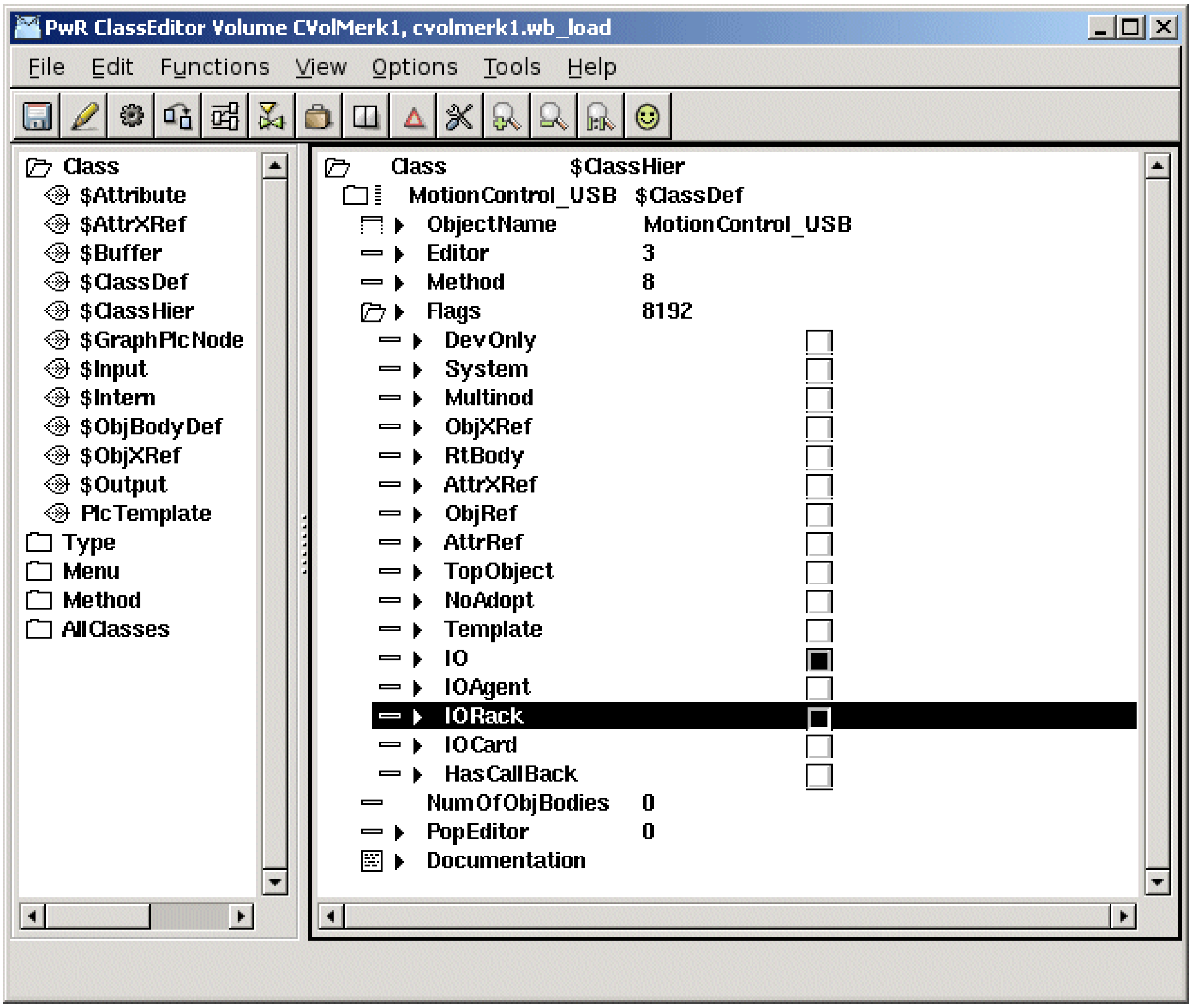


Fig IO and IORack bits in Flags

Below the $ClassDef object the attributes of the class are defined. We create a $ObjBodyDef object and below this a $Attribute object with the name Description and with type (TypeRef) pwrs:Type-$String80.

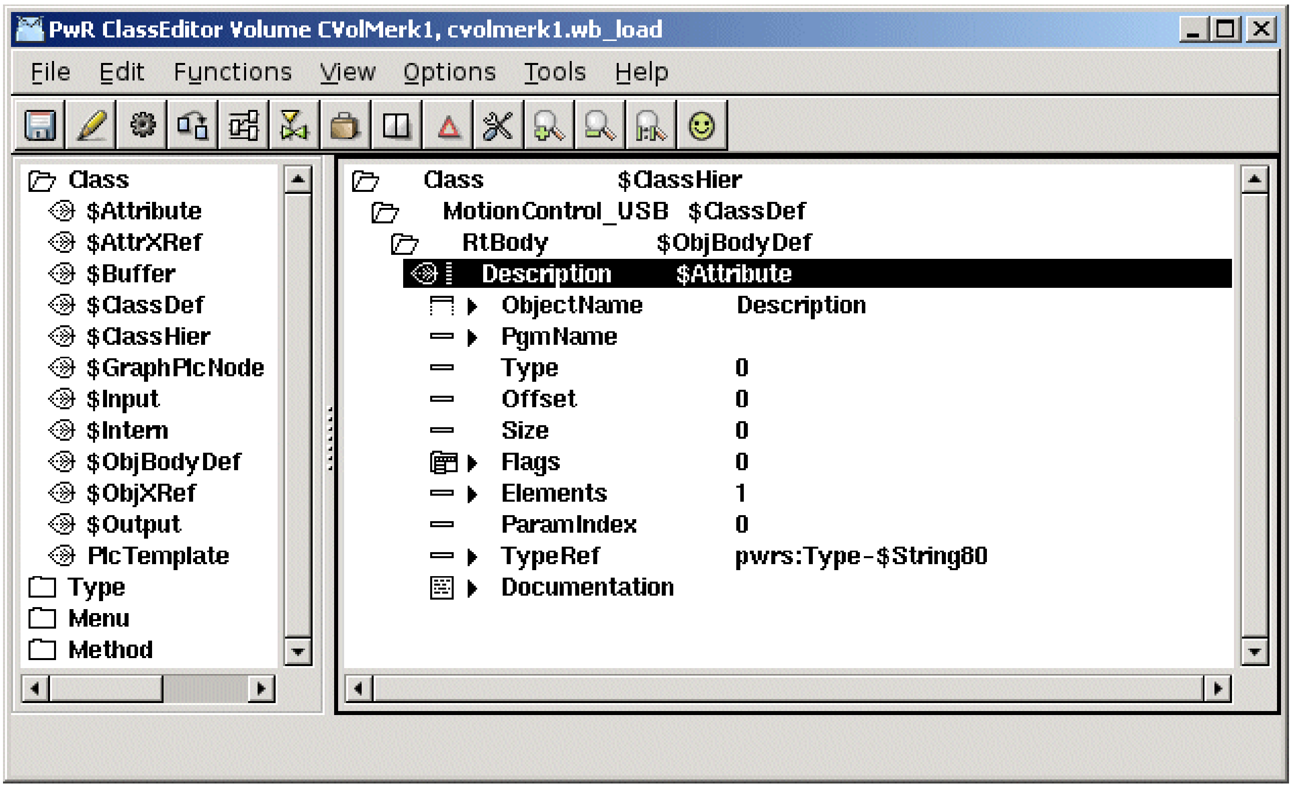


Fig Attribute object

Create a card class

There is a baseclass for card objects, *Basecomponent:BaseIOCard,* that we can use, and that contains the most common attributes in a card object. We create another $ClassDef object with the name MotionControl_USBIO, and set the IOCard and IO bits in the Flags attribute.

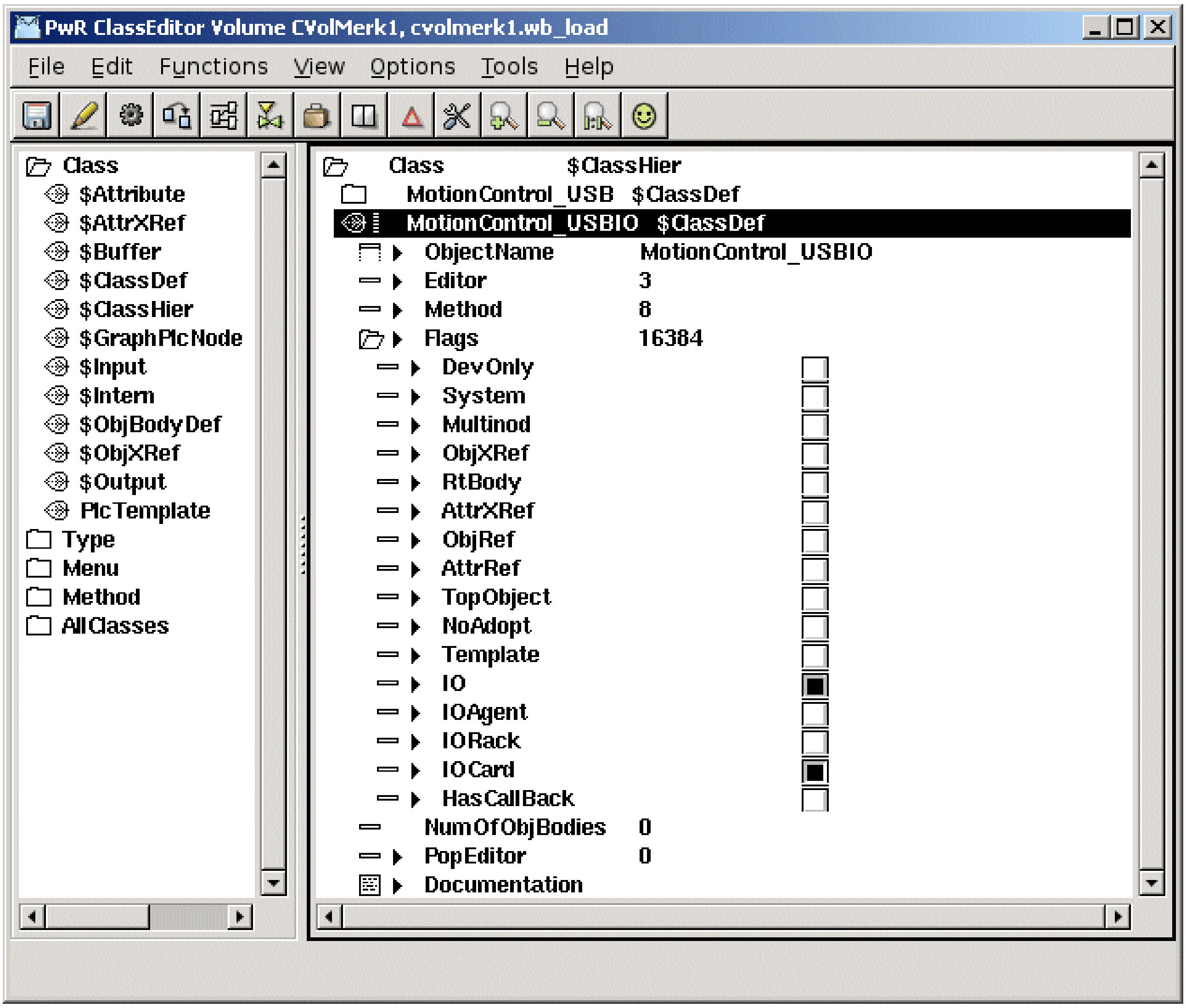


Fig IO and IOCard bits in Flags

We create an $ObjBodyDef object and an $Attribute object to state BaseIOCard as a superclass. The attribute is named Super and in TypeRef Basecomponent:Class-BaseIOCard is set. We will now inherit all attributes and methods defined in the class BaseIOCard.

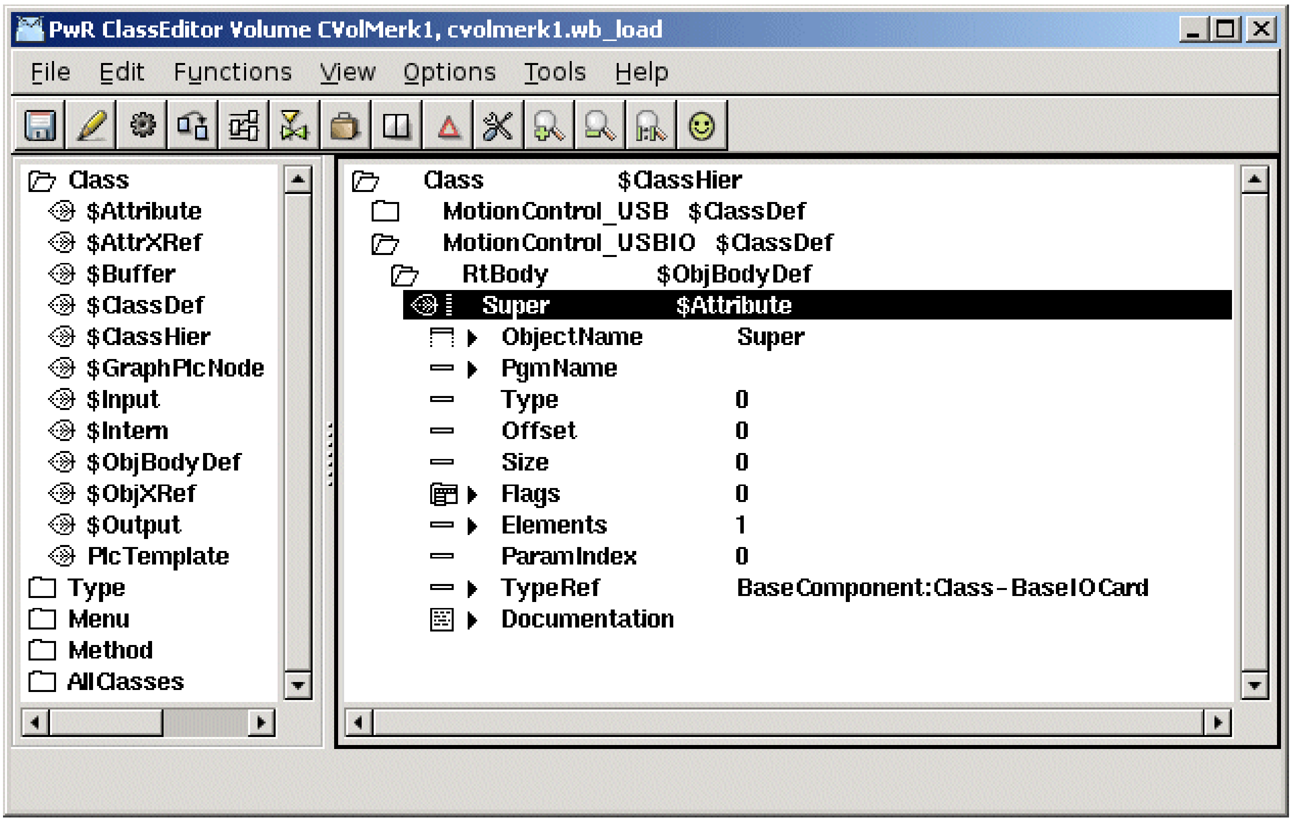


Fig Configration of the superclass BaseIOCard

We add another attribute for the card status, and for the status we create an enumeration type, MotionControl_StatusEnum, that contains the various status codes that the status attribute can contain.

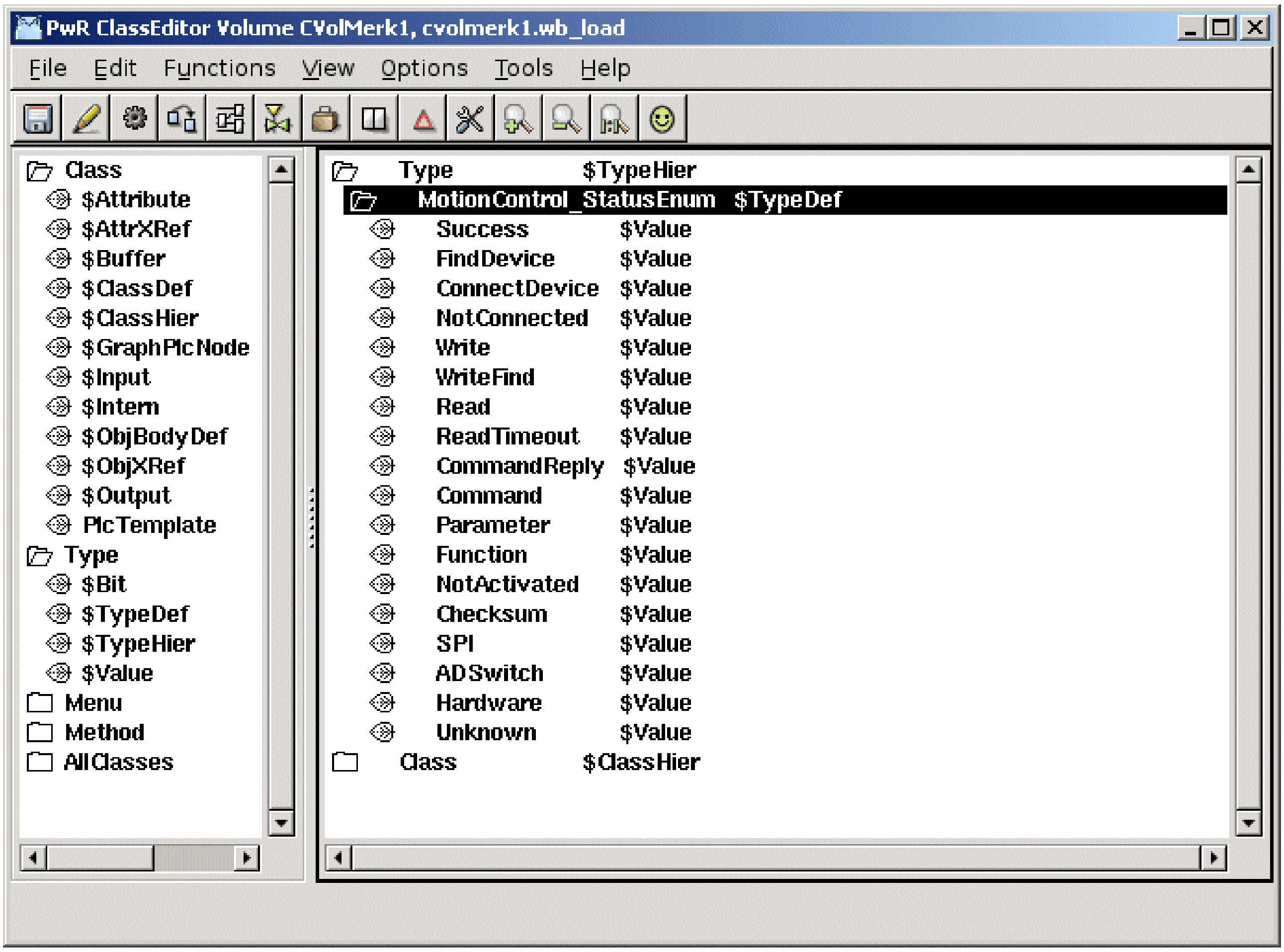


Fig Definition of an enumeration type

The type of the status attribute is set to the created status type, and in Flags, the bits *State* and *NoEdit* is set, as this attribute is not to be set in the configurator, but will be given a value in the runtime environment.

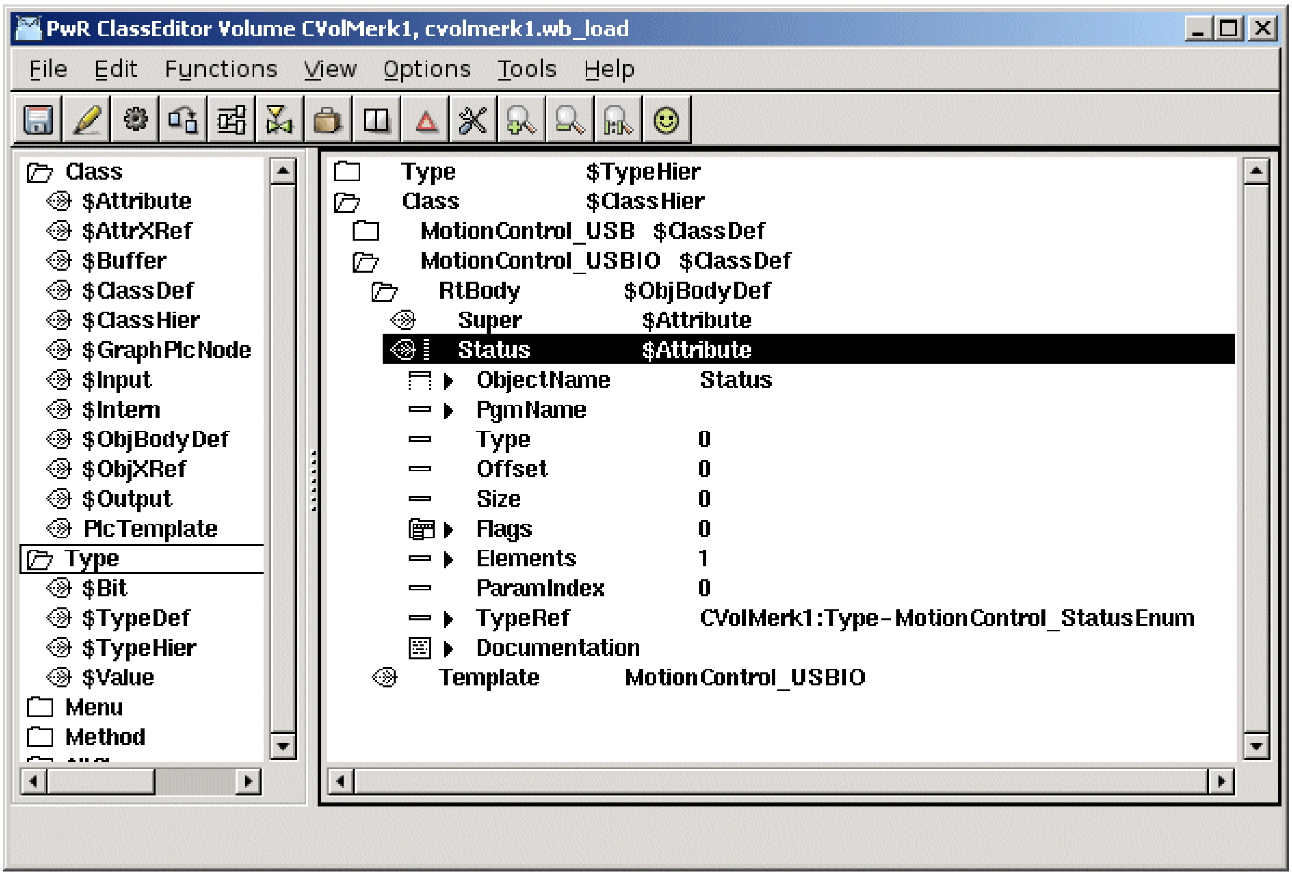


Fig Status attribute of enumeration type

Normally you also add attributes for the channel objects in the card class, but as the USB I/O device is so flexible, the same channel can be configured as a Di, Do or Ai channel, we choose not to place the channel objects as attributes. They will be configured as individual objects and placed as children to the card object in the root volume.

USB I/O contains a watchdog that will reset the unit if it is not written to within a certain time. We also add the attribute WatchdogTime to configure the timeout time.

When a class is saved for the first time, a Template object is created under the $ClassDef object. This object is an instance of the actual class where you can set default values of the attributes. We state Specification, insert an URL to the data sheet, and set Process to 1. We also set MaxNoOfChannels to 21, as this card has 21 channels.

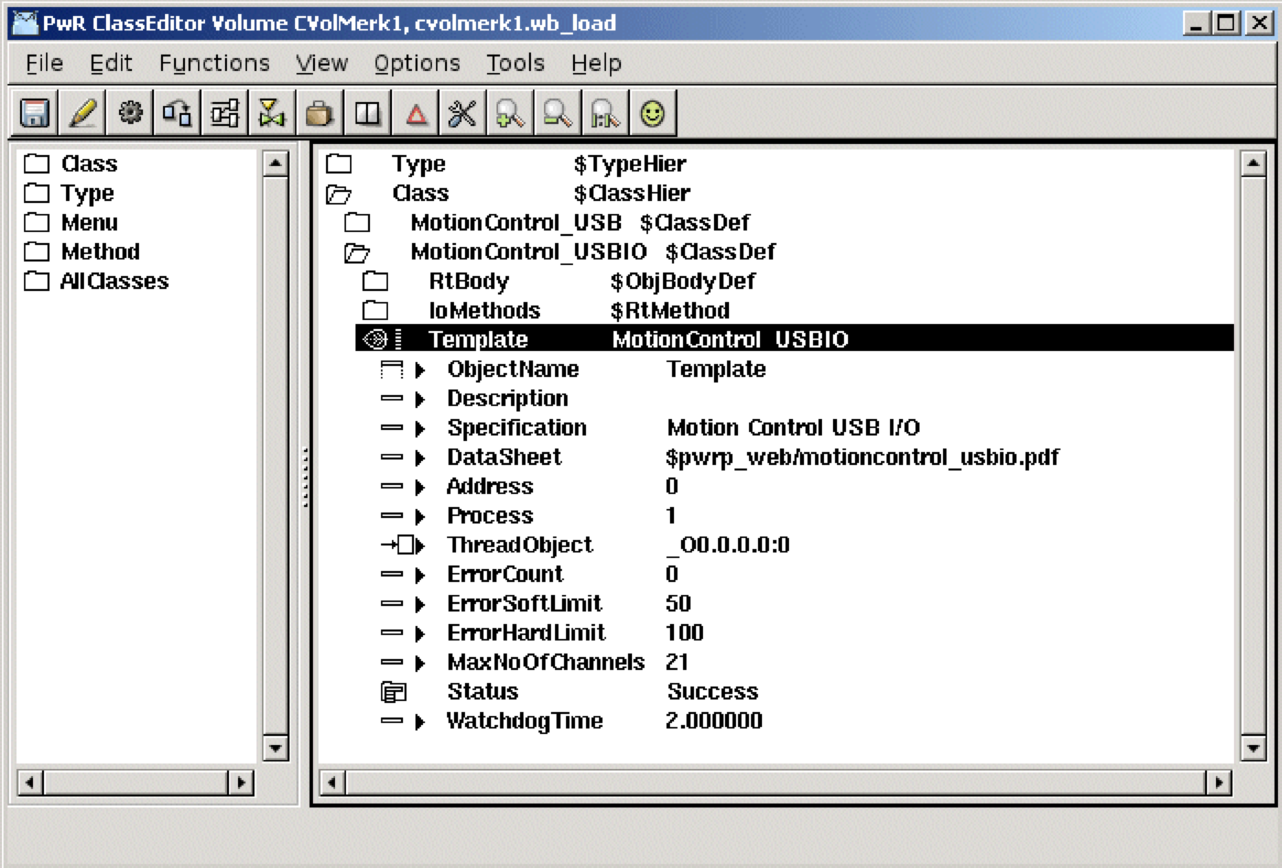


Fig Template object

Next step is to add the methods to the class description. While the card contains both inputs and outputs, we need to create Init, Close, Read and Write methods. These will be configured with method objects of type $Method. First we put a $RtMethod object, named IoMethods, under the $ClassDef object. Below this, we create one $RtMethod object for each method. The objects are named IoCardInit, IoCardClose, IoCardRead and IoCardWrite. In the attribute MethodName we state the string with witch the methods will be registered in the C code, i.e. “MotionControl_USBIO-IoCardInit”, etc.

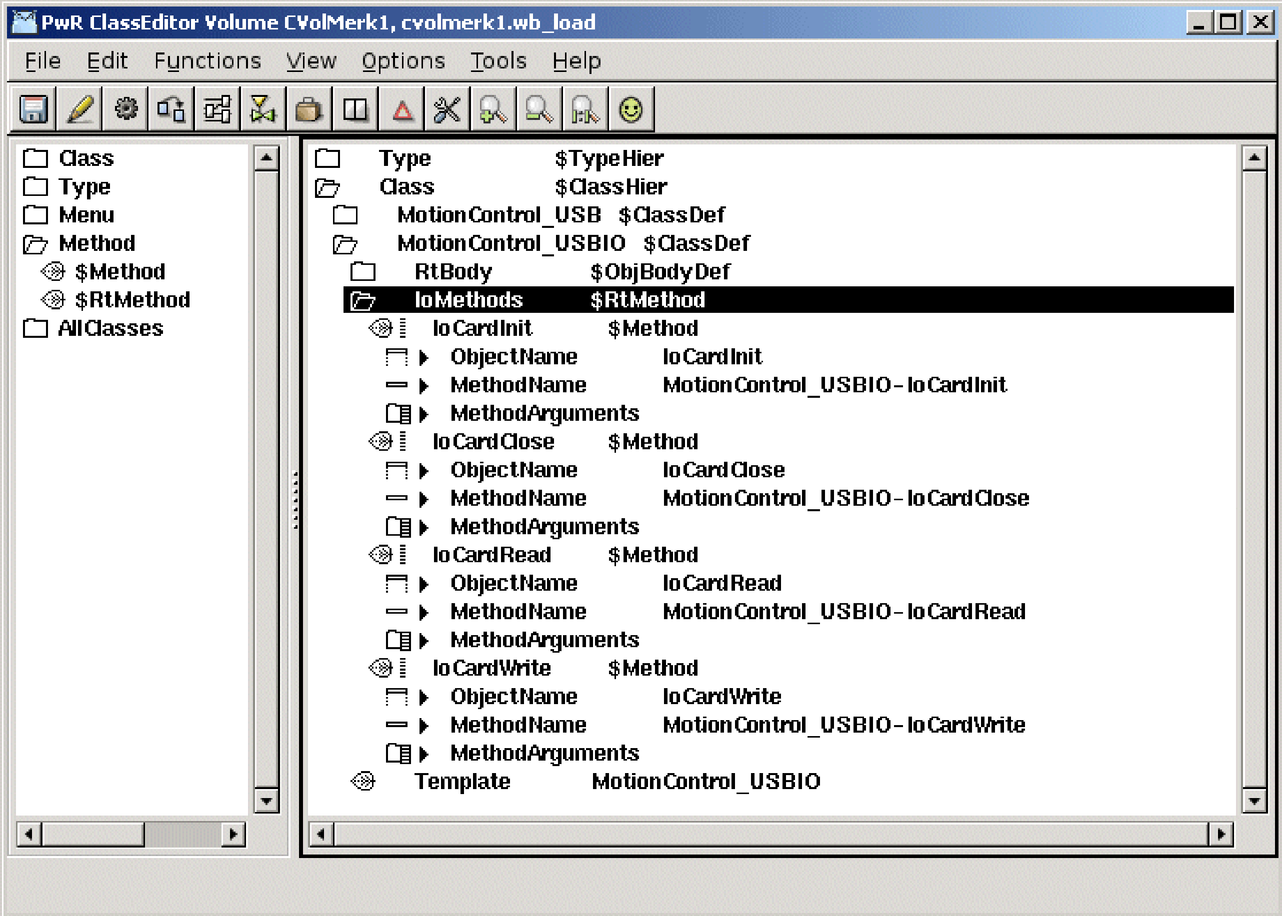


Fig I/O method configuration

From the superclass BaseIOCard we inherit a method to connect the object to a PLC thread in the configurator.

Build the class volume

Now the classes are created, and we save, leave edit mode, and create load files for the class volume by activating *Functions/Build Volume* in the menu. An include-file containing C structs for the classes, $pwrp_inc/pwr_cvolmerk1classes.h, is also created when building the volume.

Install the driver

Download the driver and unpack the tar-file for the driver

```
> tar -xvzf usbio.tar.tz
```

Build the driver with make

```
> cd usbio/driver/linux-2.6
> make
```

Install the driver `usbio.ko` as root

```
> insmod usbio.ko
```

Allow all users to read and write to the driver

```
> chmod a+rw /dev/usbio0
```

Write methods

The next step is to write the C code for the methods.

The C file ra_io_m_motioncontrol_usbio.c are created on $pwrp_src.

As Proview has a GPL license, also the code for the methods has to be GPL licensed if the program is distributed to other parties. We therefore put a GPL header in the beginning of the file.

To simplify the code we limit our use of USB I/O to a configuration where channel 0-3 are digital outputs, 4-7 digital inputs, 8-15 analog inputs, 16-18 digital inputs and 19-20 analog outputs.

ra_io_m_motioncontrol_usbio.c

```
/*
 * Proview   $Id$
 * Copyright (C) 2005 SSAB Oxelösund.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation, either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with the program, if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include "pwr.h"
#include "pwr_basecomponentclasses.h"
#include "pwr_cvolmerk1classes.h"
#include "rt_io_base.h"
#include "rt_io_card_init.h"
#include "rt_io_card_close.h"
#include "rt_io_card_read.h"
#include "rt_io_card_write.h"
#include "rt_io_msg.h"
#include "libusbio.h"


typedef struct {
  int USB_Handle;
} io_sLocal;


// Init method
static pwr_tStatus IoCardInit( io_tCtx ctx,
                               io_sAgent *ap,
                               io_sRack *rp,
                               io_sCard *cp)
{
  int i;
  int timeout;
  io_sLocal *local;
  pwr_sClass_MotionControl_USBIO *op = (pwr_sClass_MotionControl_USBIO *)cp->op;

  local = (io_sLocal *) calloc( 1, sizeof(io_sLocal));
  cp->Local = local;
```

```
  // Configure 4 Do and 4 Di on Port A
  op->Status = USBIO_ConfigDIO( &local->USB_Handle, 1, 240);
  if ( op->Status)
    errh_Error( "IO Init Card '%s', Status %d", cp->Name, op->Status);


  // Configure 8 Ai on Port B
  op->Status = USBIO_ConfigAI( &local->USB_Handle, 8);
  if ( op->Status)
    errh_Error( "IO Init Card '%s', Status %d", cp->Name, op->Status);


  // Configure 3 Di and 2 Ao on Port C
  op->Status = USBIO_ConfigDIO( &local->USB_Handle, 3, 7);
  if ( op->Status)
    errh_Error( "IO Init Card '%s', Status %d", cp->Name, op->Status);
  op->Status = USBIO_ConfigAO( &local->USB_Handle, 3);
  if ( op->Status)
    errh_Error( "IO Init Card '%s', Status %d", cp->Name, op->Status);


  // Calculate conversion coefficients for Ai
  for ( i = 8; i < 16; i++) {
    if ( cp->chanlist[i].cop &&
         cp->chanlist[i].sop &&
         cp->chanlist[i].ChanClass == pwr_cClass_ChanAi)
      io_AiRangeToCoef( &cp->chanlist[i]);
  }

  // Calculate conversion coefficients for Ao
  for ( i = 19; i < 21; i++) {
    if ( cp->chanlist[i].cop &&
         cp->chanlist[i].sop &&
         cp->chanlist[i].ChanClass == pwr_cClass_ChanAo)
      io_AoRangeToCoef( &cp->chanlist[i]);
  }

  // Configure Watchdog
  timeout = 1000 * op->WatchdogTime;
  op->Status = USBIO_ConfigWatchdog( &local->USB_Handle, 1, timeout, 1,
                                     port_mask, port, 3);

  errh_Info( "Init of USBIO card '%s'", cp->Name);

  return IO__SUCCESS;
}

// Close method
static pwr_tStatus IoCardClose( io_tCtx ctx,
                                io_sAgent *ap,
                                io_sRack *rp,
                                io_sCard *cp)
{
  free( cp->Local);
  return IO__SUCCESS;
}


// Read Method
static pwr_tStatus IoCardRead( io_tCtx ctx,
                               io_sAgent *ap,
                               io_sRack   *rp,
```

```
                              io_sCard   *cp)
{
  io_sLocal *local = cp->Local;
  pwr_sClass_MotionControl_USBIO *op = (pwr_sClass_MotionControl_USBIO *)cp->op;
  int value = 0;
  int i;
  unsigned int m;
  pwr_tUInt32 error_count = op->Super.ErrorCount;

  // Read Di on channel 4 - 8
  op->Status = USBIO_ReadDI( &local->USB_Handle, 1, &value);
  if ( op->Status)
    op->Super.ErrorCount++;
  else {
    // Set Di value in area object
    m = 1 << 4;
    for ( i = 4; i < 8; i++) {
      *(pwr_tBoolean *)cp->chanlist[i].vbp = ((value & m) != 0);
      m = m << 1;
    }
  }

  // Read Ai on channel 8 - 16
  for ( i = 0; i < 8; i++) {
    io_sChannel *chanp = &cp->chanlist[i + 8];
    pwr_sClass_ChanAi *cop = (pwr_sClass_ChanAi *)chanp->cop;
    pwr_sClass_Ai *sop = (pwr_sClass_Ai *)chanp->sop;

    if ( cop->CalculateNewCoef)
      // Request to calculate new coefficients
      io_AiRangeToCoef( chanp);

      op->Status = USBIO_ReadADVal( &local->USB_Handle, i + 1, &ivalue);
      if ( op->Status)
        op->Super.ErrorCount++;
      else {
        io_ConvertAi( cop, ivalue, &actvalue);

      // Filter the Ai value
      if ( sop->FilterType == 1 &&
           sop->FilterAttribute[0] > 0 &&
           sop->FilterAttribute[0] > ctx->ScanTime) {
        actvalue = *(pwr_tFloat32 *)chanp->vbp +
                   ctx->ScanTime / sop->FilterAttribute[0] *
                   (actvalue - *(pwr_tFloat32 *)chanp->vbp);
      }

      // Set value in area object
      *(pwr_tFloat32 *)chanp->vbp = actvalue;
      sop->SigValue = cop->SigValPolyCoef1 * ivalue + cop->SigValPolyCoef0;
      sop->RawValue = ivalue;
    }
  }

  // Check Error Soft and Hard Limit

  // Write warning message if soft limit is reached
  if ( op->Super.ErrorCount >= op->Super.ErrorSoftLimit &&
       error_count < op->Super.ErrorSoftLimit)
    errh_Warning( "IO Card ErrorSoftLimit reached, '%s'", cp->Name);

  // Stop I/O if hard limit is reached
```

```
  if ( op->Super.ErrorCount >= op->Super.ErrorHardLimit) {
    errh_Error( "IO Card ErrorHardLimit reached '%s', IO stopped", cp->Name);
    ctx->Node->EmergBreakTrue = 1;
    return IO__ERRDEVICE;
  }

  return IO__SUCCESS;
}


// Write method
static pwr_tStatus IoCardWrite( io_tCtx ctx,
                                io_sAgent *ap,
                                io_sRack  *rp,
                                io_sCard  *cp)
{
  io_sLocal *local = cp->Local;
  pwr_sClass_MotionControl_USBIO *op = (pwr_sClass_MotionControl_USBIO *)cp->op;
  int value = 0;
  float fvalue;
  int i;
  unsigned int m;
  pwr_tUInt32 error_count = op->Super.ErrorCount;
  pwr_sClass_ChanAo *cop;

  // Write Do on channel 1 - 4
  m = 1;
  value = 0;
  for ( i = 0; i < 4; i++) {
    if ( *(pwr_tBoolean *)cp->chanlist[i].vbp)
      value |= m;
    }
    m = m << 1;
  }
  op->Status = USBIO_WriteDO( &local->USB_Handle, 1, value);
  if ( op->Status) op->Super.ErrorCount++;

  // Write Ao on channel 19 and 20
  if ( cp->chanlist[19].cop &&
       cp->chanlist[19].sop &&
       cp->chanlist[19].ChanClass == pwr_cClass_ChanAo) {
    cop = (pwr_sClass_ChanAo *)cp->chanlist[19].cop;

    if ( cop->CalculateNewCoef)
      // Request to calculate new coefficients
      io_AoRangeToCoef( &cp->chanlist[19]);

    fvalue = *(pwr_tFloat32 *)cp->chanlist[19].vbp * cop->OutPolyCoef1 +
             cop->OutPolyCoef0;
    op->Status = USBIO_WriteAO( &local->USB_Handle, 1, fvalue);
    if ( op->Status) op->Super.ErrorCount++;
  }

  if ( cp->chanlist[20].cop &&
       cp->chanlist[20].sop &&
       cp->chanlist[20].ChanClass == pwr_cClass_ChanAo) {
    cop = (pwr_sClass_ChanAo *)cp->chanlist[20].cop;

    if ( cop->CalculateNewCoef)
      // Request to calculate new coefficients
      io_AoRangeToCoef( &cp->chanlist[20]);

    fvalue = *(pwr_tFloat32 *)cp->chanlist[20].vbp * cop->OutPolyCoef1 +
```

```
            cop->OutPolyCoef0;
    op->Status = USBIO_WriteAO( &local->USB_Handle, 2, fvalue);
    if ( op->Status) op->Super.ErrorCount++;
  }


  // Check Error Soft and Hard Limit

  // Write warning message if soft limit is reached
  if ( op->Super.ErrorCount >= op->Super.ErrorSoftLimit &&
       error_count < op->Super.ErrorSoftLimit)
    errh_Warning( "IO Card ErrorSoftLimit reached, '%s'", cp->Name);

  // Stop I/O if hard limit is reached
  if ( op->Super.ErrorCount >= op->Super.ErrorHardLimit) {
    errh_Error( "IO Card ErrorHardLimit reached '%s', IO stopped", cp->Name);
    ctx->Node->EmergBreakTrue = 1;
    return IO__ERRDEVICE;
  }

  return IO__SUCCESS;
}

// Every method should be registred here

pwr_dExport pwr_BindIoUserMethods(MotionControl_USBIO) = {
  pwr_BindIoUserMethod(IoCardInit),
  pwr_BindIoUserMethod(IoCardClose),
  pwr_BindIoUserMethod(IoCardRead),
  pwr_BindIoUserMethod(IoCardWrite),
  pwr_NullMethod
};
```

Class registration

To make it possible for the I/O framework to find the methods of the class, the class has to be registered. This is done by creating the file `$pwrp_src/rt_io_user.c`. You use the macros pwr_BindIoUserMethods and pwr_BindIoUserClass for each class that contains methods.

rt_io_user.c

```
#include "pwr.h"
#include "rt_io_base.h"

pwr_dImport pwr_BindIoUserMethods(MotionControl_USBIO);

pwr_BindIoUserClasses(User) = {
  pwr_BindIoUserClass(MotionControl_USBIO),
  pwr_NullClass
};
```

Makefile

To compile the C files we create a makefile on $pwrp_src, `$pwrp_src/makefile`. This will compile `ra_io_m_motioncontro_usbio.c` and `rt_io_user.c`, and place the object modules on the directory $pwrp_obj.

makefile

```
mars2_top : mars2

include $(pwr_exe)/pwrp_rules.mk
```

```
mars2_modules = $(pwrp_obj)/ra_io_m_motioncontrol_usbio.o \
                $(pwrp_obj)/rt_io_user.o

# Main rule
mars2 : $(mars2_modules)
        @ echo "****** Mars2 modules built ******"

# Modules

$(pwrp_obj)/ra_io_m_motioncontrol_usbio.o : \
      $(pwrp_src)/ra_io_m_motioncontrol_usbio.c \
      $(pwrp_inc)/pwr_cvolmerk1classes.h

$(pwrp_obj)/rt_io_user.o : $(pwrp_src)/rt_io_user.c
```

Link file

We choose to call the methods from the PLC process, and have to link the PLC program with the object modules of the methods. To do this, we create a link file on $pwrp_exe. We also have to add the archive with the USB I/O driver API, `libusbio.a`. The name of the link file contains node name and bus number.

plc_mars2_0507.opt

```
$pwrp_obj/rt_io_user.o $pwrp_obj/ra_io_m_motioncontrol_usbio.o -lusbio
```

Configure the node hierarchy

Now its time to configure the I/O objects in the node hierarchy with objects of the classes that we have created.

The root volume in our project is VolMerk1 and we open the configurator with

```
> pwrs volmerk1
```

In the palette to the left, under the map AllClasses we find the classvolme of the project, and under this the two classes for the USB I/O that we have created. Below the $Node object we place a rack-object of class MotionControl_USB, and below this a card object of class MotionControl_USBIO. As the channel objects are not internal objects in the card object, we have to create channel objects for the channels that we are going to use, below the card object. See the result in *Fig The Nodehierarchy*.

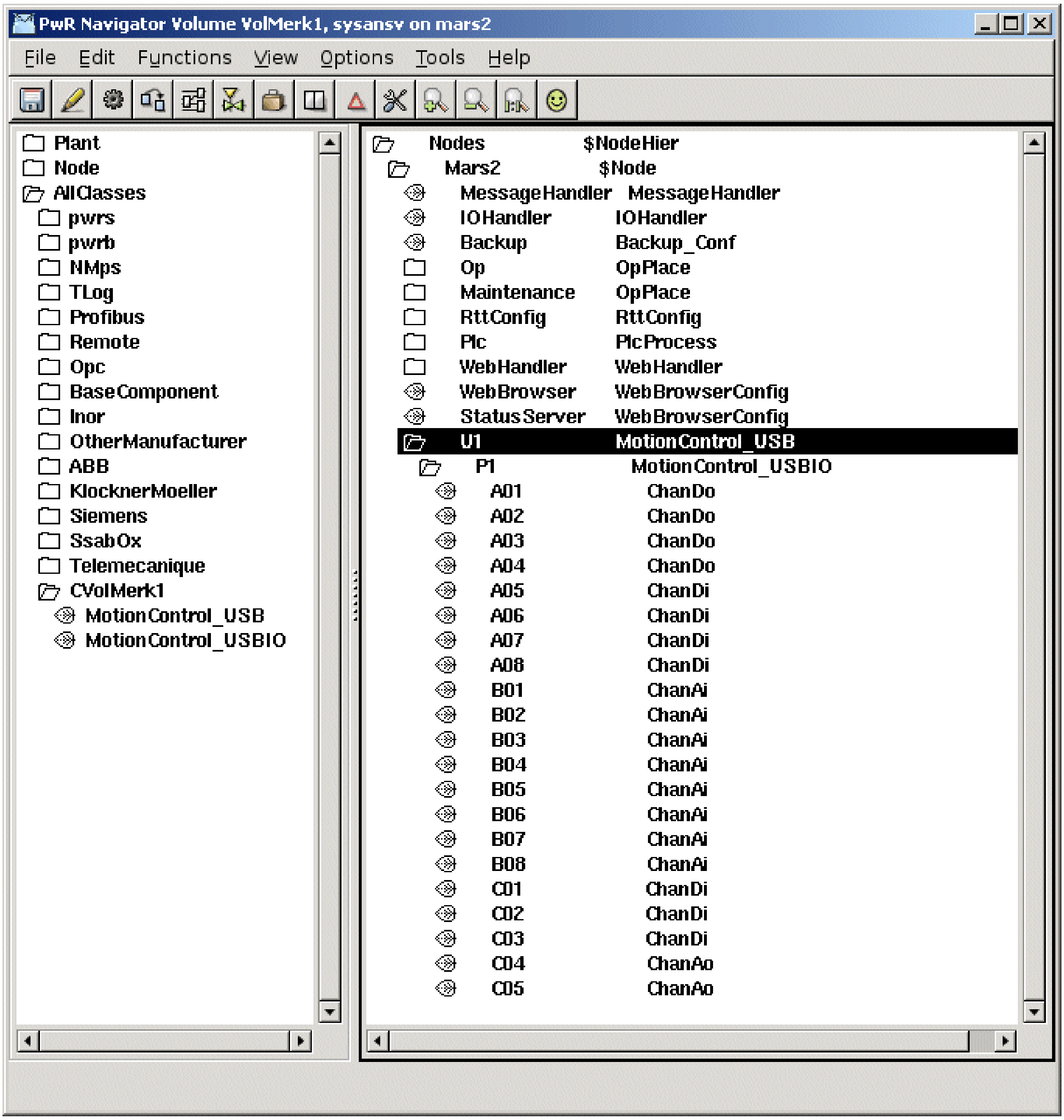


Fig The Node hierarchy

We set the attribute Number, which states the index in the channel list, to 0 for the first channel object, 1 for the second etc. We set Process to 1 in the rack and card objects, and connects these objects to a PLC thread by selecting a PlcThread object, and activating Connect PlcThread in the popup menu for the rack and card object.

For the analog channels, ranges for conversion to/from ActualValue unit, has to be stated. The RawValue range for Ai channels are 0-1023, and the signal range 0-5 V. We configure the channels with an ActualValue range 0-100 in *Fig Ai channel*.

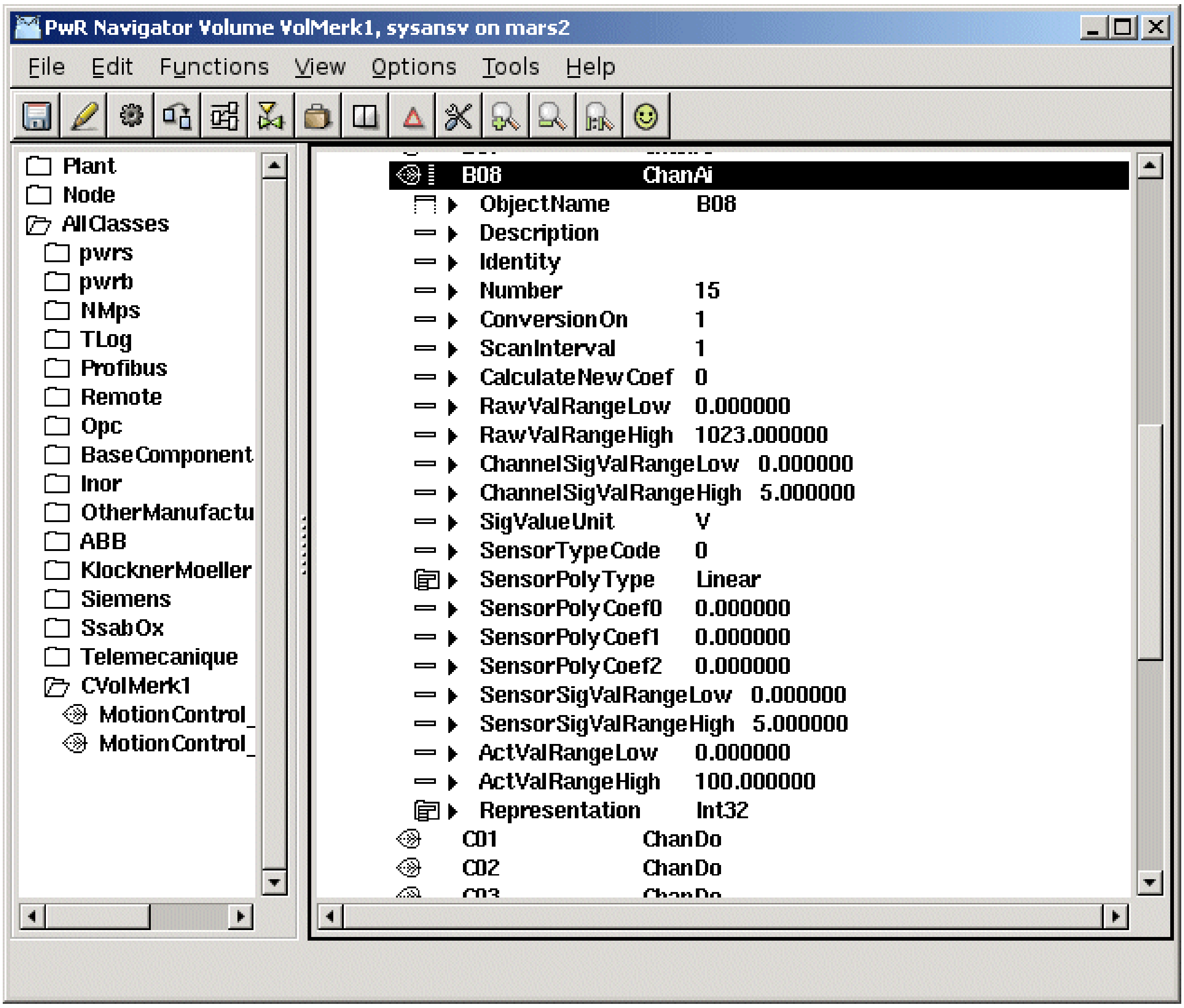


Fig Ai channel

When reading the Ao channels, you receive the signal value 0-5 V, and a configuration for ActualValue range 0-100 can be seen in *Fig Ao channel*.

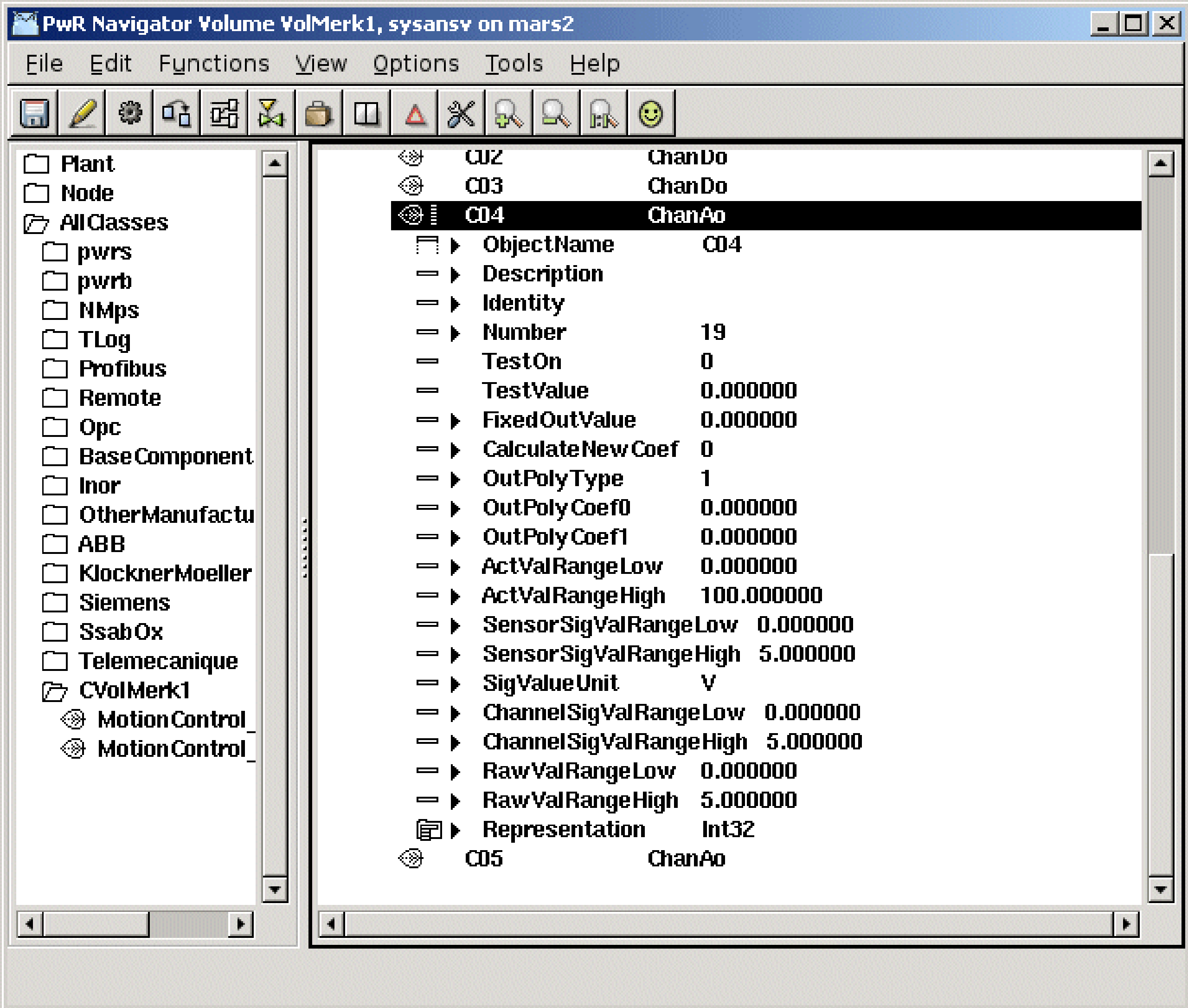


Fig Ao channel

We also have to create signal objects in the plant hierarchy of types Di, Do, Ai and Ao, and connect these to each channel respectively. There also has to be a PlcPgm on the selected thread, to really create a thread in runtime. The remaining activities now are to build the node, distribute, check the linkage of the USB I/O device, connect it to the USB port and start Proview.

Notes on the Hilscher Profibus agent implementation

This section aims to document some of the implementation details of the Pb_Hilscher agent. The agent links with the API provided by Hilscher. This API requires a kernel module which handles the interface to the board. For the API to communicate with the kernel module the character device /dev/cif is used.

When working with the Hilscher agent, there are two manuals that are of interest. One of these can be found in Hilscher's Linux driver tarball and is named CIFLinux_en.pdf, this describes many general aspects of the Linux version of the API. The other manual is dpm_pie.pdf, available on the driver disc that comes with the board. That manual covers how to use the Profibus DP functionality of the card. Also worth mentioning is the toolkit manual, TKE.PDF (also found on the driver disc), this explains some hardware details not covered in the other documents.

The agent module contains all board specific code. The first section provides some higher level functions on top of the API. The second section contains the agent methods exported to Proview.