



Guide to I/O System

2007-08-24 cs

Copyright SSAB Oxelösund AB 2007

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

Table of Contents

About this Guide	6
Introduction.....	7
Overview.....	8
Levels.....	8
Configuration.....	8
I/O System.....	9
PSS9000.....	10
Rack objekt.....	10
Rack_SSAB.....	10
Attributes.....	10
Driver.....	10
Ssab_RemoteRack.....	10
Attributes.....	10
Di card.....	11
Ssab_BaseDiCard.....	11
Ssab_DI32D.....	11
Do cards.....	11
Ssab_BaseDoCard.....	11
Ssab_DO32KTS.....	12
Ssab_DO32KTS_Stall.....	12
Ai cards.....	12
Ssab_BaseACard.....	12
Ssab_AI8uP.....	12
Ssab_AI16uP.....	13
Ssab_AI32uP.....	13
Ssab_AI16uP_Logger.....	13
Ao cards.....	13
Ssab_AO16uP.....	13
Ssab_AO8uP.....	13
Ssab_AO8uPL.....	13
Co kort.....	13
Ssab_CO4uP.....	13
Profibus.....	15
The profibus configurator.....	15
Address.....	17
SlaveGsdData.....	17
UserPrmData.....	17
Module.....	17
Specify the data area.....	19
Digital inputs.....	19
Analog inputs.....	19
Digital outputs.....	20
Analog outputs.....	20
Complex dataareas.....	20
Driver.....	20
Agent object.....	20

Pb_Profiboard.....	20
Slave objects.....	20
Pb_Dp_Slave.....	20
ABB_ACS_Pb_Slave.....	20
Siemens_ET200S_IM151.....	20
Siemens ET200M_IM153.....	20
Module objects.....	21
Pb_Module.....	21
ABB_ACS_PPO5.....	21
Siemens_ET200S_Ai2.....	21
Siemens_ET200S_Ao2.....	21
Siemens_ET200M_Di4.....	21
Siemens_ET200M_Di2.....	21
Siemens_ET200M_Do4.....	21
Siemens_ET200M_Do2.....	21
Adaption of I/O systems.....	22
Overview.....	22
Levels.....	22
Area objekt.....	23
I/O objects.....	23
Processes.....	23
Framework.....	23
Methods.....	24
Framework.....	24
Create I/O objects.....	26
Flags.....	28
Attributes.....	29
Description.....	29
Process.....	29
ThreadObject.....	29
Method objects.....	29
Agents.....	30
Racks.....	30
Cards.....	30
Connect-method for a ThreadObject.....	30
Methods.....	31
Local data structure.....	31
Agent-Methods.....	31
IoAgentInit.....	31
IoAgentClose.....	31
IoAgentRead.....	32
IoAgentWrite.....	32
IoAgentSwap.....	32
Rack-metoder.....	32
IoRackInit.....	32
IoRackClose.....	32
IoRackRead.....	32
IoRackWrite.....	32
IoRackSwap.....	32
Card-metoder.....	32

IoCardInit.....	32
IoCardClose.....	33
IoCardRead.....	33
IoCardWrite.....	33
IoCardSwap.....	33
Method registration.....	33
Class registration.....	33
Module in Proview base system.....	33
Project.....	33
Example of rack methods.....	34
Example of the methods of a digital input card.....	35
Example of the methods of a digital output card.....	38

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 signalobject 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 is installed in the rack. The cardobjects contains channelobjects for the channels on the cards. The channelobjects are connected to signalobjects 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 agentlevel consist of an object for the master card that is mounted on the computer. The racklevel consist of slave objects, that represent the profibus slaves that are connected to the Profibus circuit. The cardlevel consist of module objects that represent modules handled by the slaves. The Channel objects represent data sent on the bus from the mastercard to the modules or vice versa.

I/O System

This chapter contains descriptions of the I/O sytems 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 Carde 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 conneted 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 extention card.

The rackobjects 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

<i>Attributes</i>	<i>Description</i>
Address	ip-adress 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 plcprogram, 2 io_comm.
ThreadObject	Thread object for the plc thread that should handle the rack. Only used if Process

<i>Attributes</i>	<i>Description</i>
	is 1.
StallAction	No, ResetInputs or EmergencyBreak. Default EmergencyBreak.

Di card

All digital inputcards 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

<i>Attributes</i>	<i>Description</i>
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 plcprogram, 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 inputcard 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 outputcards 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

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

<i>Attributes</i>	<i>Description</i>
Process	Process that handles the rack. 1 the plcprogram, 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 outputcard 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 outputcard 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 is 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

<i>Attribut</i>	<i>Beskrivning</i>
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 plcprogram, 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 inputcard 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 inputcard 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 inputcard 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 inputcard 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 inputcard 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 inputcard 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 inputcard 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.

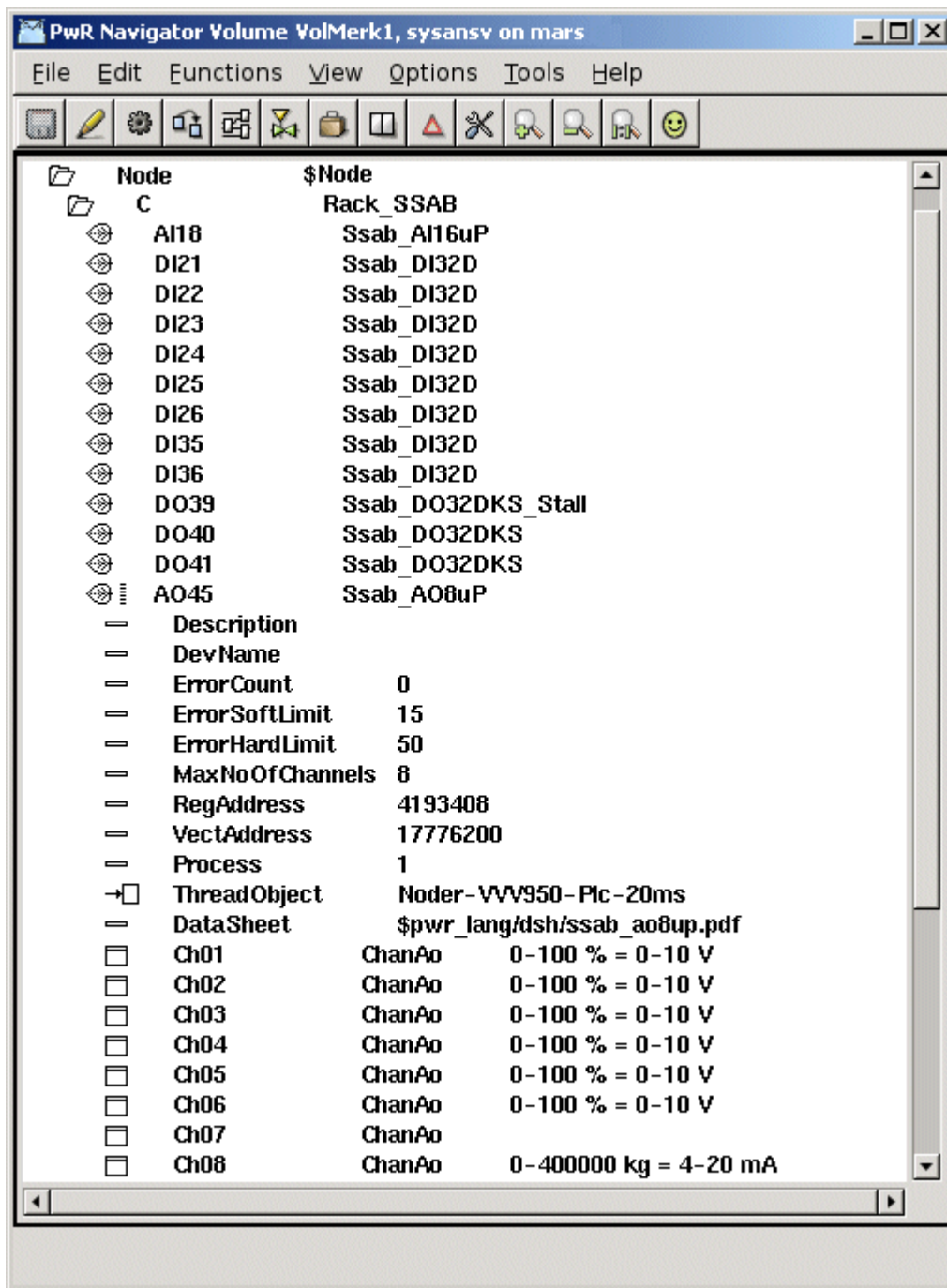


Fig PSS9000 configuration example

Profibus

Profibus is a fieldbus with nodes of type master and slave. The usual configuration is a monomastersystem 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 has support for the mastercard *Softing PROFiboard PCI* (see www.softing.com) that is installed in the PCI-bus of the process station. The card is configured by an object of class Profibus:Pb_Profiboard that is placed below the \$Node object.

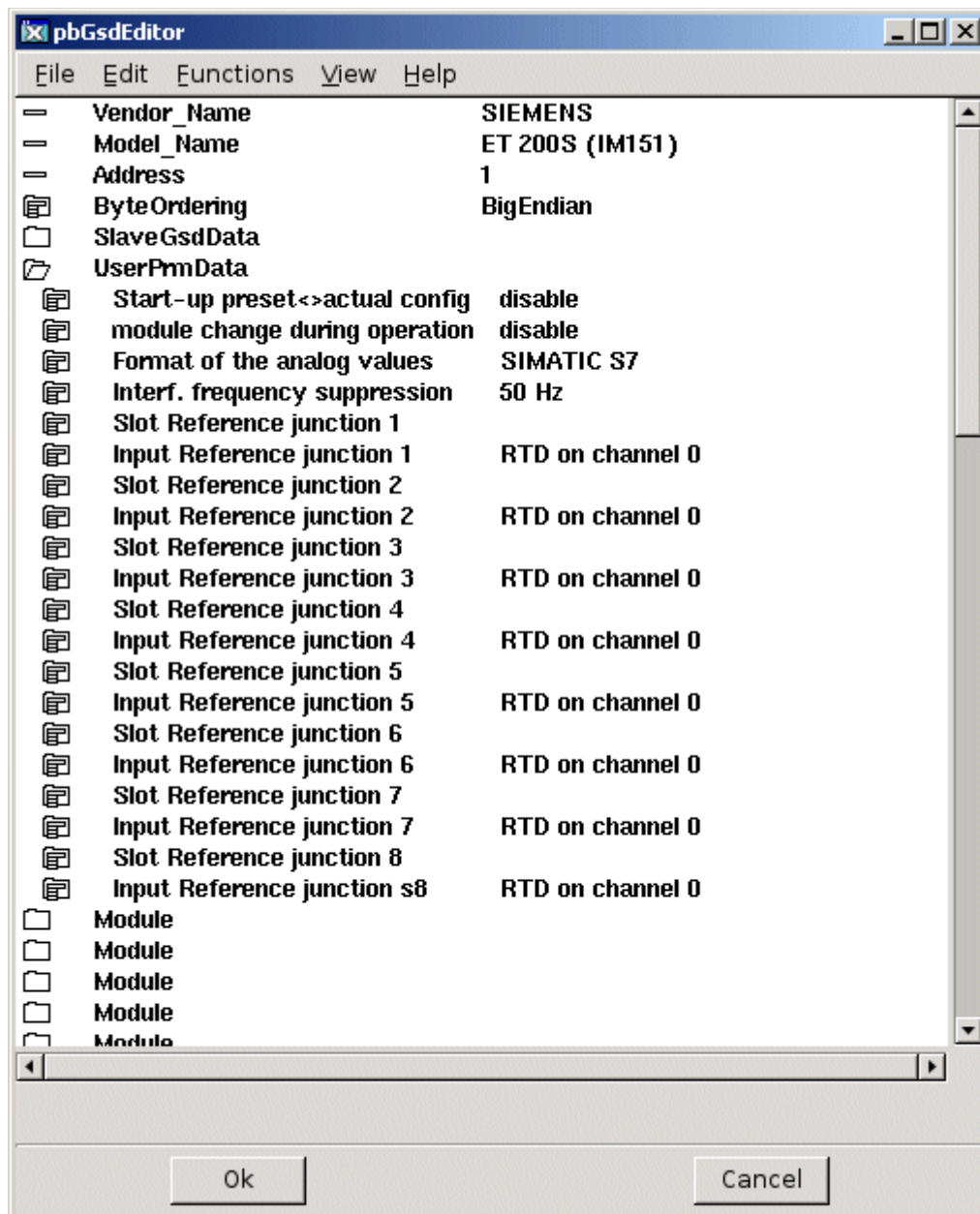
Each slave connected to the profibus circuit is configured with an object of class Pb_DP_Slave, or a subclass to this class. The slave objects are placed as children to the master object. For the slave objects, a profibus configurator can be opened, that configures the slave object, and creates module 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 textfile 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 your 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 rightclicking 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 one unit. and there are slaves of rack type, into which a large number of modules can be inserted. The Profibus configurator displays one 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 objectnames.

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

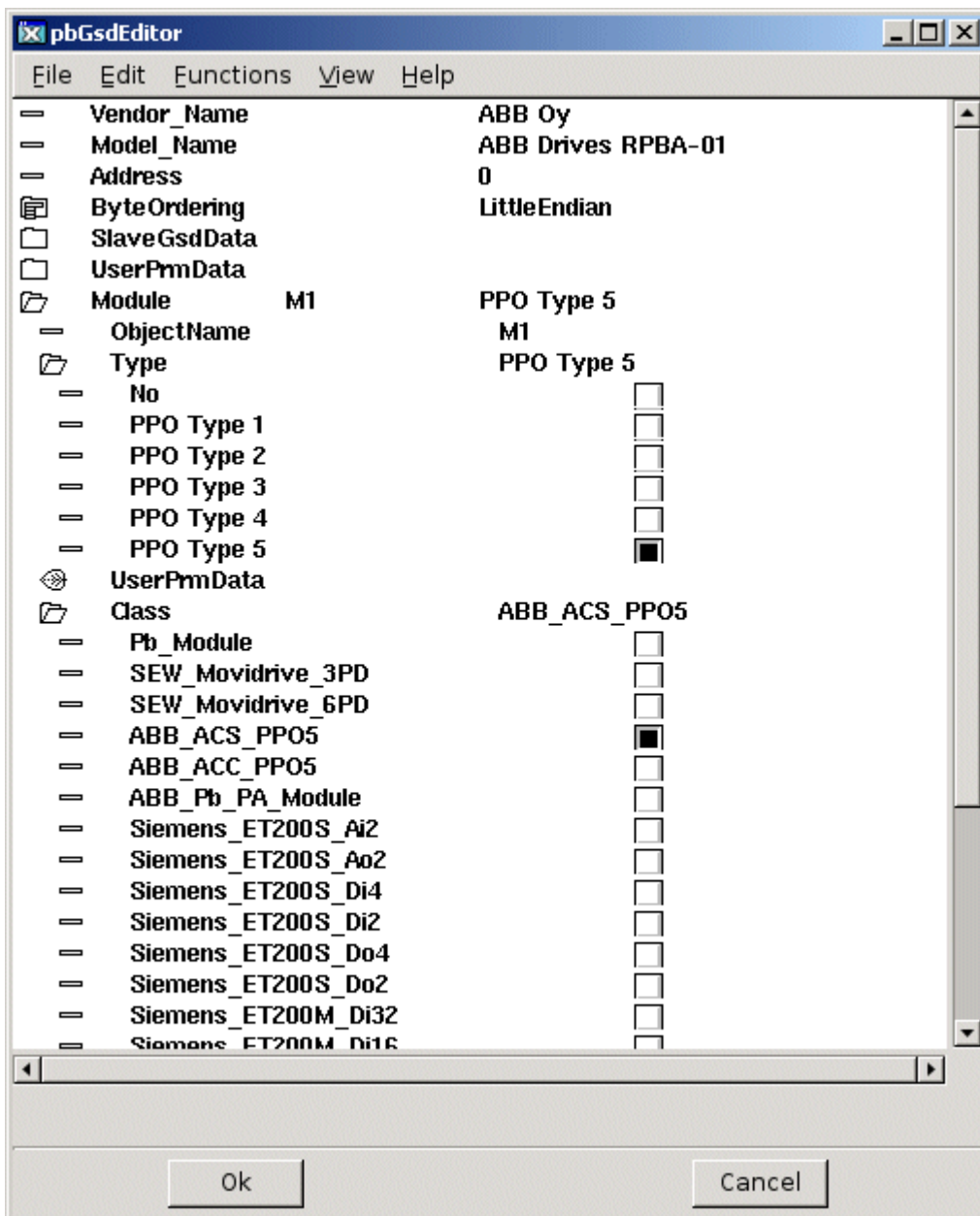


Fig Module Type and Class selected

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).

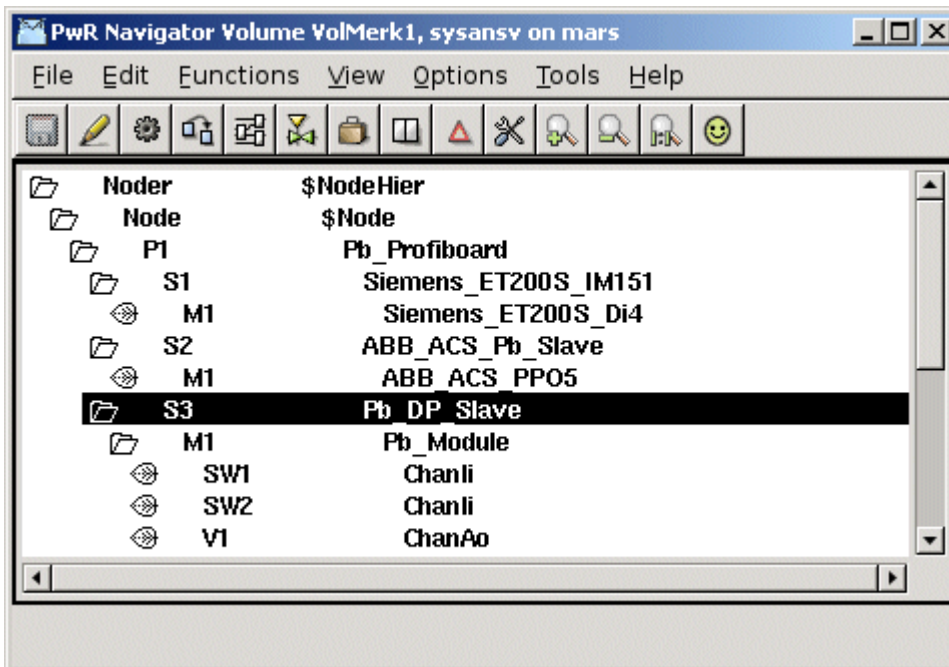
When all the modules are configured you save by clicking on 'Ok' and leave by clicking 'Cancel'. The module objects with specified object names and classes are now created below the slave object.

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 a 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 dataarea for a module. Input modules read data that are sent to the process station over the bus, and output modules receive 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 have to be configured, and this is done with channel objects. The inarea is specified with ChanDi, ChanAi and ChanIi objects, the outarea 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 an 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 signalvalue 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 dataareas

Many modules send 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 inarea are usually placed first and thereafter the channels of the outarea.

Driver

Softing PROFIBoard requires a driver to be installed. Download the driver from www.softing.com.

Agent object

Pb_Profiboard

Agent object for a profibus master of type Softing PROFIBoard. The object is placed in the nodehierarchy below the \$Node object.

Slave objects

Pb_Dp_Slave

Baseobject for a profibus slave. Reside 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 frequencyconverter ABB ACS800 with protocol PPO5.

Siemens_ET200S_IM151

Slaveobject 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

Moduleobject for a frequencyconverter ABB ACS800 with protocol PPO5.

Siemens_ET200S_Ai2

Moduleobject for a Siemens ET200S module with 2 analog inputs.

Siemens_ET200S_Ao2

Moduleobject for a Siemens ET200S module with 2 analog outputs.

Siemens_ET200M_Di4

Moduleobject for a Siemens ET200M module with 4 digital inputs.

Siemens_ET200M_Di2

Moduleobject for a Siemens ET200M module with 2 digital inputs.

Siemens_ET200M_Do4

Moduleobject for a Siemens ET200M modul with 4 digital outputs.

Siemens_ET200M_Do2

Moduleobject for a Siemens ET200M modul with 2 digital outputs.

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 baseknowledge 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 channelobjects can be configured as individual objects or as internal objects in a card object.

To the agent, rack and card objects methods can be registred. 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 inutcards and digital outputcards 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, cardobjects 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 transferes 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 distributes sytem, with a mastercard 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 mastercard represents the agentlevel, the slaves the racklevel and the modules the cardlevel.

The Agent, rack and card levels are very flexible, and mainly defined by the attributes and the methodes 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 signalobject reside in the plant hierarchy and represents for exampel 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 signalobjects

are connected to the channel object. Plcprograms, HMI and applications refer to the signalobject that represents the component in the plant, not the channelobject, representing a channel on an I/O unit.

Area objekt

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 systemvolume 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 areaobject 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 signalvalues from the copy, calculated output signalvalues 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 areaobjects 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 datastructure is 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_init()`. 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 methods/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 in 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;

```

Datastructure 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 kontrollled 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 kontrollled 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 I/O system 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. They are extended with channel objects, that can be put as individual attributes, or, if they are of the same type, as a vector of channelobjects. 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 cardclass 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 channelattributes of type ChanDi. As the index for this cardtype by tradition starts from 1, the channels are put as individual attributes, but they could also be an array of type ChanDi.

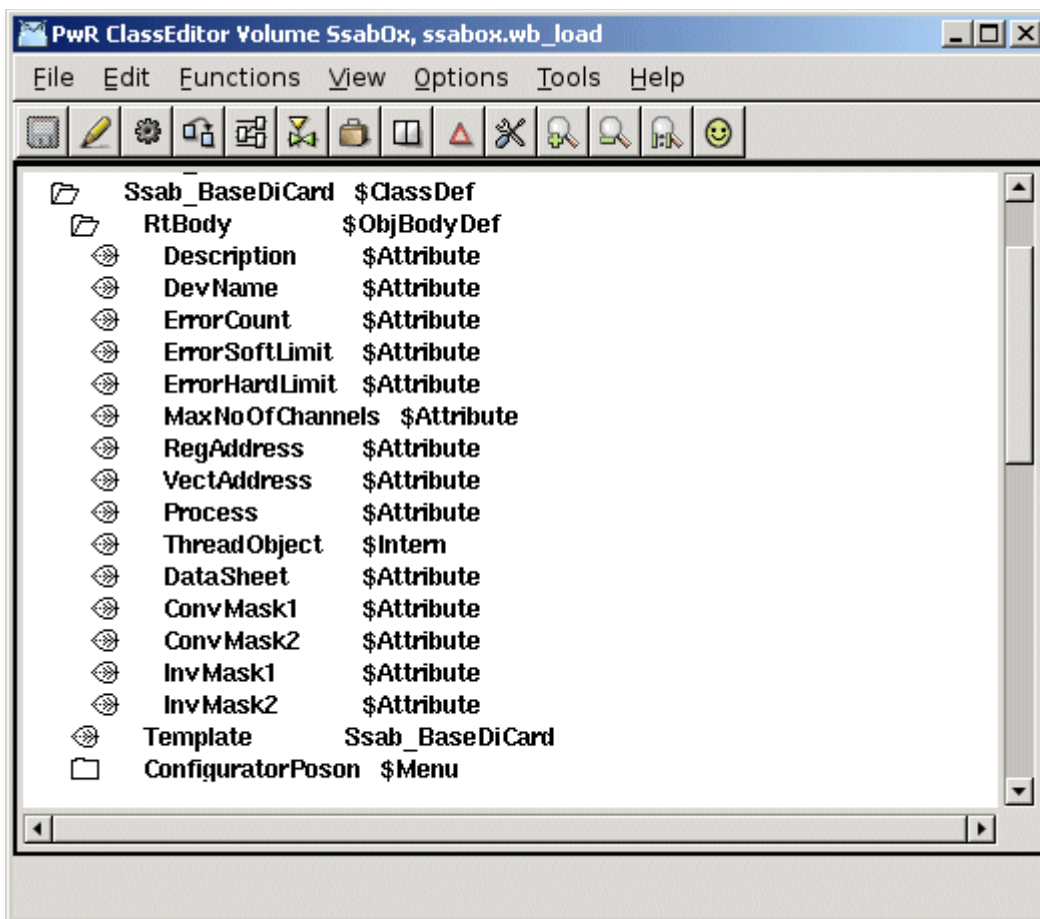


Fig Example of a baseclass for a card

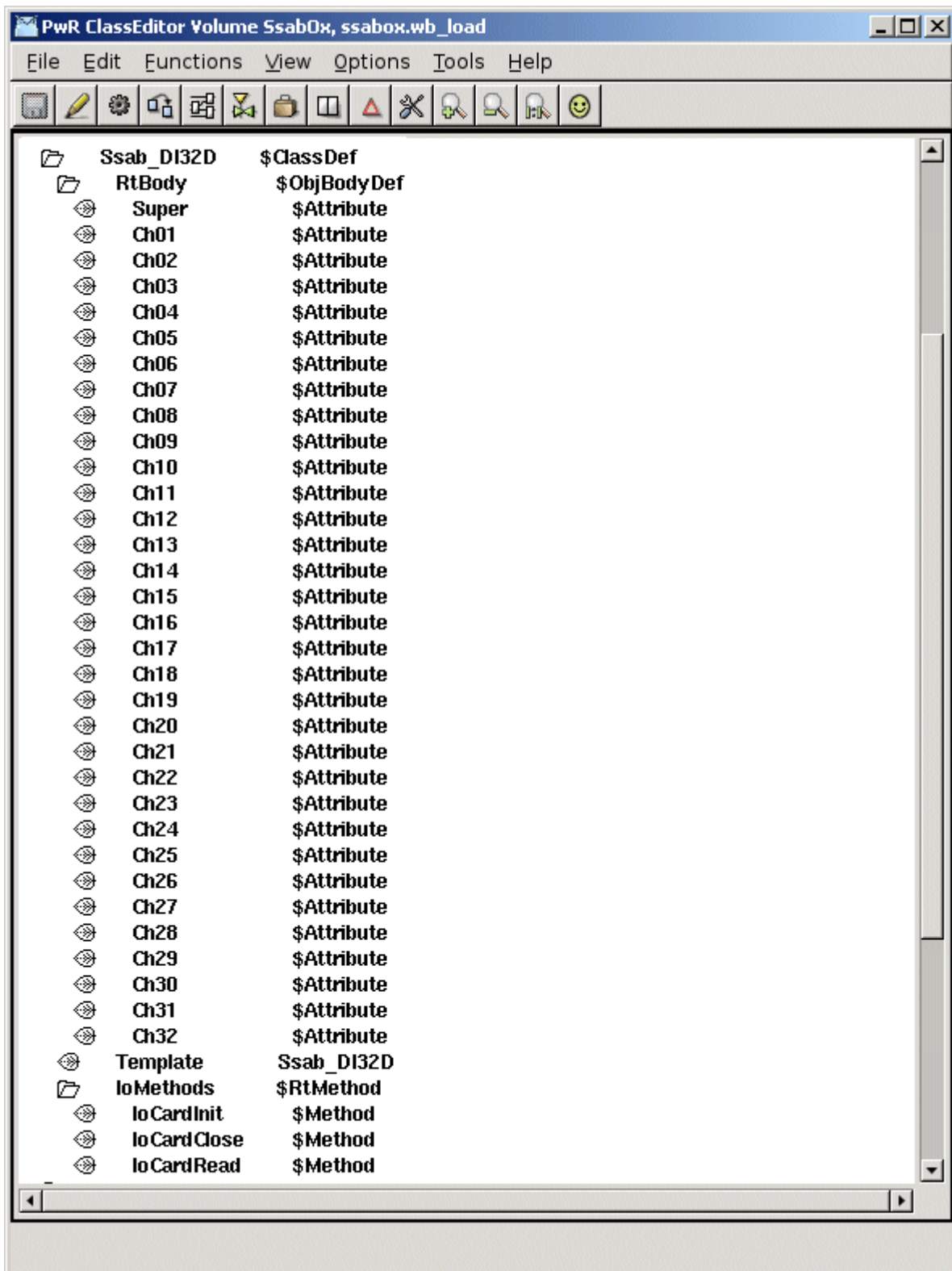


Fig Example of a cardclass with a superclass an 32 channel objects

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.

	Rack_SSAB	\$ClassDef	
	Editor	0	
	Method	1	
	Flags	8208	
	DevOnly		<input type="checkbox"/>
	System		<input type="checkbox"/>
	Multinod		<input type="checkbox"/>
	ObjXRef		<input type="checkbox"/>
	RtBody		<input checked="" type="checkbox"/>
	AttrXRef		<input type="checkbox"/>
	ObjRef		<input type="checkbox"/>
	AttrRef		<input type="checkbox"/>
	TopObject		<input type="checkbox"/>
	NoAdopt		<input type="checkbox"/>
	Template		<input type="checkbox"/>
	IO		<input type="checkbox"/>
	IOAgent		<input type="checkbox"/>
	IORack		<input checked="" type="checkbox"/>
	IOCard		<input type="checkbox"/>
	HasCallBack		<input type="checkbox"/>

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 plcprocess should handle the uing.

	Ssab_BaseDoCard	\$ClassDef
	RtBody	\$ObjBodyDef
	Description	\$Attribute
	Process	\$Attribute
	ThreadObject	\$Attribute

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 classname and methodname, e.g. "Ssab_AluP-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

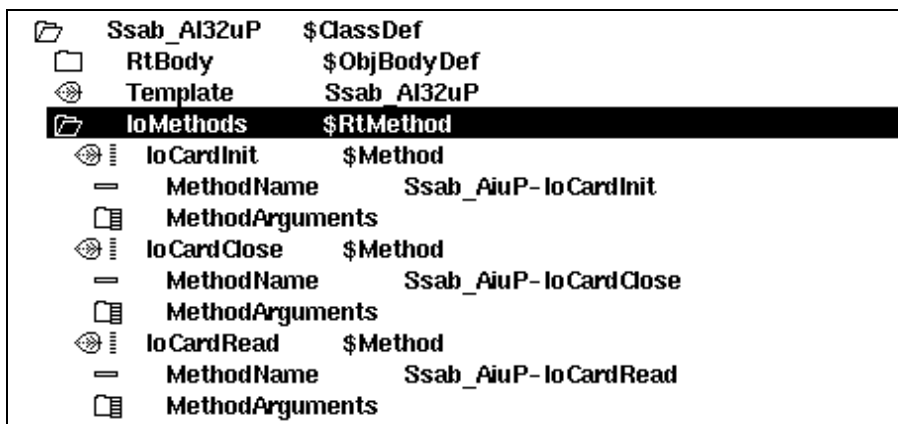
For agents, \$Method objects with the name IoAgentInit, IoAgentClose, IoAgentRead and IoAgentWrite are created.

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.



Folder icon	Ssab_AI32uP	\$ClassDef
Folder icon	RtBody	\$ObjBodyDef
Folder icon	Template	Ssab_AI32uP
Folder icon	IoMethods	\$RtMethod
Method icon	IoCardInit	\$Method
Property icon	MethodName	Ssab_AiuP-IoCardInit
Property icon	MethodArguments	
Method icon	IoCardClose	\$Method
Property icon	MethodName	Ssab_AiuP-IoCardClose
Property icon	MethodArguments	
Method icon	IoCardRead	\$Method
Property icon	MethodName	Ssab_AiuP-IoCardRead
Property icon	MethodArguments	

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 threadobject 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, se *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.

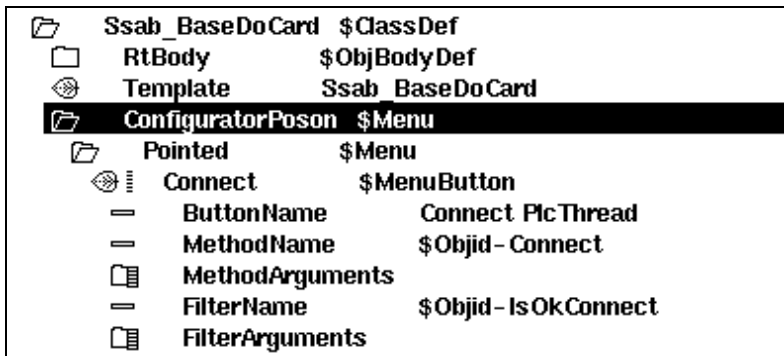


Fig Connect metod

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 flexibel 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 ouput cards.
- Swap-method is called during a soft restart.

Local data structure

In the datastructures 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 metod för en agent.

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

IoAgentRead

Read metod för en agent.

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

IoAgentWrite

Write metod för en agent.

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

IoAgentSwap

Swap metod för en 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 modul 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 registered 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 registered 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
};

```