



# **I/O System handbok**

2007-05-04 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

Om den här handledningen .....	6
Introduktion.....	7
Översikt.....	8
Nivåer.....	8
I/O System.....	9
PSS9000.....	10
Rack objekt.....	10
Rack_SSAB.....	10
Attribut.....	10
Drivrutin.....	10
Ssab_RemoteRack.....	10
Attribut.....	10
Di kort.....	11
Ssab_BaseDiCard.....	11
Ssab_DI32D.....	11
Do kort.....	11
Ssab_BaseDoCard.....	11
Ssab_DO32KTS.....	12
Ssab_DO32KTS_Stall.....	12
Ai kort.....	12
Ssab_BaseACard.....	12
Ssab_AI8uP.....	12
Ssab_AI16uP.....	12
Ssab_AI32uP.....	13
Ssab_AI16uP_Logger.....	13
Ao kort.....	13
Ssab_AO16uP.....	13
Ssab_AO8uP.....	13
Ssab_AO8uPL.....	13
Co kort.....	13
Ssab_CO4uP.....	13
Attribut.....	13
Profibus.....	14
Profibus konfiguratorn.....	14
Address.....	15
SlaveGsdData.....	15
UserPrmData.....	15
Module.....	15
Specificera dataarean.....	17
Digitala ingångar.....	17
Analoga ingångar.....	18
Digitala utgångar.....	18
Analoga utgångar.....	18
Komplexa dataareor.....	18
Drivrutin.....	18
Agent objekt.....	18

Pb_Profiboard.....	18
Slavobjekt.....	18
Pb_Dp_Slave.....	18
ABB_ACS_Pb_Slave.....	18
Siemens_ET200S_IM151.....	19
Siemens ET200M_IM153.....	19
Modulobjekt.....	19
Pb_Module.....	19
ABB_ACS_PPO5.....	19
Siemens_ET200S_Ai2.....	19
Siemens_ET200S_Ao2.....	19
Siemens_ET200M_Di4.....	19
Siemens_ET200M_Di2.....	19
Siemens_ET200M_Do4.....	19
Siemens_ET200M_Do2.....	19
Adaption av I/O system.....	20
Översikt.....	20
Nivåer.....	20
Area objekt.....	21
I/O objekt.....	21
Processer.....	21
Ramverk.....	21
Metoder.....	21
Ramverk.....	22
Skapa I/O-objekt.....	24
Flags.....	26
Attribut.....	27
Description.....	27
Process.....	27
ThreadObject.....	27
Metod objekt.....	27
Agenter.....	28
Rack.....	28
Kort.....	28
Connect-metod för ThreadObject.....	28
Metoder.....	29
Lokal datastruktur.....	29
Agent-Metoder.....	29
IoAgentInit.....	29
IoAgentClose.....	29
IoAgentRead.....	29
IoAgentWrite.....	30
IoAgentSwap.....	30
Rack-metoder.....	30
IoRackInit.....	30
IoRackClose.....	30
IoRackRead.....	30
IoRackWrite.....	30
IoRackSwap.....	30
Card-metoder.....	30

IoCardInit.....	30
IoCardClose.....	30
IoCardRead.....	31
IoCardWrite.....	31
IoCardSwap.....	31
Registrering av metoder.....	31
Registrering av klassen.....	31
Modul i Proview's bassystem.....	31
Projekt.....	31
Exempel på rack metoder.....	32
Exempel på metoder digitalt ingångskort.....	33
Exempel på metoder för digitalt utgångskort.....	36

# Om den här handledningen

Proview's *I/O System handbok* är avsedd för personer som vill knyta olika typer av I/O system till Proview, och för användare som vill ha en djupare förståelse för hur I/O hanteringen i Proview fungerar.

# Introduktion

IO-hanteringen i Proview består av ett ramverk som är designat för att

- vara portabelt och körbart på olika plattformar.
- enkelt kunna lägga till nya I/O-system.
- hantera I/O-kort på den lokala bussen.
- hantera distribuerade I/O-system och kommunicera med remota rack-system.
- tillåta projekt att implementera lokala I/O system.
- synkronisera I/O-system med exekveringen av plc-program, eller med applikationsprocesser.

# Översikt

I/O på en processnod konfigureras genom att skapa objekt i Proview databasen. Objekten är uppdelade i två träd, anläggningshierarkin och nodhierakin. Anläggningshierarkin beskriver hur anläggningen är uppbyggd med olika processavsnitt, motorer, pumpar, fläktar mm. Här återfinns signalobjekt som representerar värden som läses in från olika givare, eller värden som ställs ut till motorer, ställdon mm. Signalobjekten är av klasserna Di, Do, Ai, Ao, Ii, Io, Co eller Po. Nodhierarkin beskriver processdatorns uppbyggnad, med server processer och I/O system. I/O systemet konfigureras med ett träd av agent, rack, kort och kanal-objekt. Kanalobjekten representerar en I/O signal som kommer in till datorn via en kanal på ett I/O kort (eller via ett distribuerat bussystem). Kanalobjekten är av klasserna ChanDi, ChanDo, ChanAi, ChanAo, ChanIi, ChanIo och ChanCo. Varje signalobjekt i anläggningshierarkin pekar på ett kanalobjekt i nodhierarkin. Kopplingen motsvarar den fysika anslutningen mellan en givare och kanalen på ett I/O kort.

## Nivåer

I/O objekten för en processnod konfigureras i en trädstruktur med tre nivåer: Agent, Rack och Kort. I vissa fall kan även en fjärde nivå närvara, Kanaler. Kanal objekten kan konfigureras som individuella objekt eller ligga som interna attribut i Kort objektet.

## Konfigurering

För ett I/O-system på den lokala bussen används ofta endast rack och kort-nivån. En konfigurering kan gå till så här. Ett rackobjekt läggs under \$Node-objektet, och under detta ett kortobjekt för varje I/O kort som finns i racken. Kortobjektet innehåller kanalobjekt för de kanaler som finns på respektive kort. Kanalobjekten kopplas till signal-objekt i anläggningshierarkin. Kanalerna för analoga signaler innehåller attribut för att ange mätområden, och kortobjekten innehåller attribut för adresser.

Konfigureringen av ett distribuerat I/O-system kan se lite annorlunda ut. Fortfarande används nivåerna Agent, Rack, Kort och Kanal, men nivåerna får en annan innebörd. Om vi tar Profibus som exempel, utgörs agentnivån av ett objekt för masterkort som är monterat på datorn. Racknivån utgörs av slavobjekt, som representerar profibus-slavar som sitter inkopplade på profibus-slingan. Kortnivån utgörs av modulobjekt som representerar moduler som hanteras av slavar. Kanalobjekten representerar data som skickas på bussen från masterkortet ut till modulerna eller vv.



# I/O System

Här följer en beskrivning på de I/O system som är implementerade i Proview.

# PSS9000

PSS9000 består av I/O kort för analoga in, analoga ut, digitala ut och digitala in. Det finns även kort för pulsräkning och PID reglering. Korten sätts i ett rack med bussen QBUS, en buss ursprungligen utvecklad för Digitals PDP-11 processor. Racken kopplas via en PCI-QBUS konverterare till en x86-pc, eller kopplas via Ethernet, s k remoterack.

Systemet konfigureras med objekt som ligger i SsabOx volymen. Här finns objekt som representerar rack- och kort-nivån. Agent-nivån representeras av \$Node objektet.

## Rack objekt

### ***Rack\_SSAB***

Rack\_SSAB objektet representerar ett 19" PSS9000 ramverk med QBUS som bakplan. Antalet kortplatser kan variera.

Ramverket kopplas till en x86 pc med en PCI-QBUS konverteringskort, PCI-Q, som sätts i pc'n och ansluts till ramverket med kabel. Flera rack kan anslutas via bussförlängarkort.

Rackobjekten placeras under \$Node objektet och namnges C1, C2, osv (i äldre system förekommer namnstandarden R1, R1 osv).

### **Attribut**

Rack\_SSAB innehåller inte några attribut används av systemet.

### **Drivrutin**

PCI-QBUS konverteringskortet, PCI-Q, kräver att en drivrutin installeras.

### ***Ssab\_RemoteRack***

Ssab\_RemoteRack objektet konfigurerar ett PSS9000 ramverk som ansluts till pc'n via Ethernet. Anslutningen av ramverket till Ethernet sker genom ett BFBETH kort sätts i ramverket.

Objektet placeras under \$Node objektet och namnges E1, E2 osv.

### **Attribut**

<i>Attribut</i>	<i>Beskrivning</i>
Address	ip-adress för BTBETH kortet.
LocalPort	Port i processtationen.
RemotePort	Port för BTBETH kortet. Default 8000.
Process	Process som ska hantera racken. 1 plcprogrammet, 2 io_comm.
ThreadObject	Om process är 1, anges här den tråd i plcprogrammet som ska hantera racken.
StallAction	No, ResetInputs eller EmergencyBreak. Default EmergencyBreak.

## Di kort

Samtliga digitala ingångskort har en gemensam basklass, Ssab\_BaseDiCard, som innehåller attribut som är gemensamma för alla di-kort. Objektet för respektive korttyp är utökade med kanalobjekt för de kanaler som kortet innehåller.

### **Ssab\_BaseDiCard**

<i>Attribut</i>	<i>Beskrivning</i>
RegAddress	QBUS adress för kortet.
ErrorHardLimit	Felgräns som stoppar systemet.
ErrorSoftLimit	Felgräns som larmar.
Process	Process som ska hantera racken. 1 plcprogrammet, 2 io_comm.
ThreadObject	Om process är 1, anges här den tråd i plcprogrammet som ska hantera racken.
ConvMask1	Konverteringsmasken anger vilka kanaler som ska omvandlas till signalvärden. Hanterar kanal 1 – 16.
ConvMask2	Se ConvMask1. Hanterar kanal 17 – 32.
InvMask1	Inverteringsmasken anger vilka kanaler som är inverterade. Hanterar kanal 1-16.
InvMask2	Se InvMask1. Hanterar kanal 17 – 32.

### **Ssab\_DI32D**

Objekt som konfigurerar ett digitalt ingångskort av typen DI32D. Kortet innehåller 32 kanaler, vars DiChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseDiCard ovan.

## Do kort

Samtliga digitala utgångskort har en gemensam basklass, Ssab\_BaseDoCard, som innehåller attribut som är gemensamma för alla do-kort. Objektet för respektive korttyp är utökade med kanalobjekt för de kanaler som kortet innehåller.

### **Ssab\_BaseDoCard**

<i>Attribut</i>	<i>Beskrivning</i>
RegAddress	QBUS adress för kortet.
ErrorHardLimit	Felgräns som stoppar systemet.
ErrorSoftLimit	Felgräns som larmar.
Process	Process som ska hantera racken. 1 plcprogrammet, 2 io_comm.
ThreadObject	Om process är 1, anges här den tråd i plcprogrammet som ska hantera kortet.
InvMask1	Inverteringsmasken anger vilka kanaler som är inverterade. Hanterar kanal 1-16.
InvMask2	Se InvMask1. Hanterar kanal 17 – 32.

<i>Attribut</i>	<i>Beskrivning</i>
FixedOutValue1	Bitmask för kanal 1 to 16 vid nödstopp av I/O hanteringen. FixedOutValue ska normalt vara 0, eftersom detta är värdet vi spänningsbortfall.
FixedOutValue2	Se FixedOutValue1. FixedOutValue2 är bitmask för kanal 17 – 32.
ConvMask1	Konverteringsmasken anger vilka kanaler som ska omvandlas till signalvärden. Hanterar kanal 1 – 16.
ConvMask2	Se ConvMask1. Hanterar kanal 17 – 32.

### ***Ssab\_DO32KTS***

Objekt som konfigurerar ett digitalt utgångskort av typen DO32KTS. Kortet innehåller 32 kanaler, vars DoChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseDoCard ovan.

### ***Ssab\_DO32KTS\_Stall***

Objekt som konfigurerar ett digitalt utgångskort av typen DO32KTS Stall. Kortet liknar DO32KTS men innehåller även en stall-funktion. som gör reset på bussen, dvs alla utgångar nollställs på samtliga kort, om ingen skrivning eller läsning har gjort på kortet inom ca 1.5 sekunder.

## **Ai kort**

Samtliga analoga kort har en gemensam basklass, Ssab\_BaseACard, som innehåller attribut som är gemensamma för alla analoga kort. Objekten för respektive korttyp är utökade med kanalobjekt för de kanaler som kortet innehåller.

### ***Ssab\_BaseACard***

<i>Attribut</i>	<i>Beskrivning</i>
RegAddress	QBUS adress för kortet.
ErrorHardLimit	Felgräns som stoppar systemet.
ErrorSoftLimit	Felgräns som larmar.
Process	Process som ska hantera racken. 1 plcprogrammet, 2 io_comm.
ThreadObject	Om process är 1, anges här den tråd i plcprogrammet som ska hantera kortet.

### ***Ssab\_AI8uP***

Objekt som konfigurerar ett analogt ingångskort av typen Ai8uP. Kortet innehåller 8 kanaler, vars AiChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseACard ovan.

### ***Ssab\_AI16uP***

Objekt som konfigurerar ett analogt ingångskort av typen Ai16uP. Kortet innehåller 16 kanaler, vars AiChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseACard ovan.

### ***Ssab\_AI32uP***

Objekt som konfigurerar ett analogt ingångkort av typen Ai32uP. Kortet innehåller 32 kanaler, vars AiChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseACard ovan.

### ***Ssab\_AI16uP\_Logger***

Objekt som konfigurerar ett analogt ingångkort av typen Ai16uP\_Logger. Kortet innehåller 16 kanaler, vars AiChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseACard ovan.

## **Ao kort**

### ***Ssab\_AO16uP***

Objekt som konfigurerar ett analogt ingångkort av typen AO16uP. Kortet innehåller 16 kanaler, vars AiChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseACard ovan.

### ***Ssab\_AO8uP***

Objekt som konfigurerar ett analogt ingångkort av typen AO8uP. Kortet innehåller 8 kanaler, vars AiChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseACard ovan.

### ***Ssab\_AO8uPL***

Objekt som konfigurerar ett analogt ingångkort av typen AO8uPL. Kortet innehåller 8 kanaler, vars AiChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt. Attribut: se BaseACard ovan.

## **Co kort**

### ***Ssab\_CO4uP***

Objekt som konfigurerar ett pulsräknarkort av typen CO4uP. Kortet innehåller 4 kanaler, vars CoChan objekt ligger internt i objektet. Objektet läggs som barn till ett Rack\_SSAB eller Ssab\_RemoteRack objekt.

## **Attribut**

# Profibus

Profibus är en fältbuss med noder av typen master eller slav. Det vanligaste är monomastersystem med en master och upp till 125 slavar. Varje slav kan innehålla en eller flera moduler.

I Proview's I/O hantering representerar mastern agent-nivån, slaverna rack-nivån och modulerna kort-nivån.

Proview har stöd för masterkortet Softing PROFiBoard PCI (se [www.softing.com](http://www.softing.com)) som ansluts till processnodens PCI-buss. Kortet konfigureras med ett objektet av klassen Profibus:Pb\_ProfiBoard som läggs under \$Node objektet.

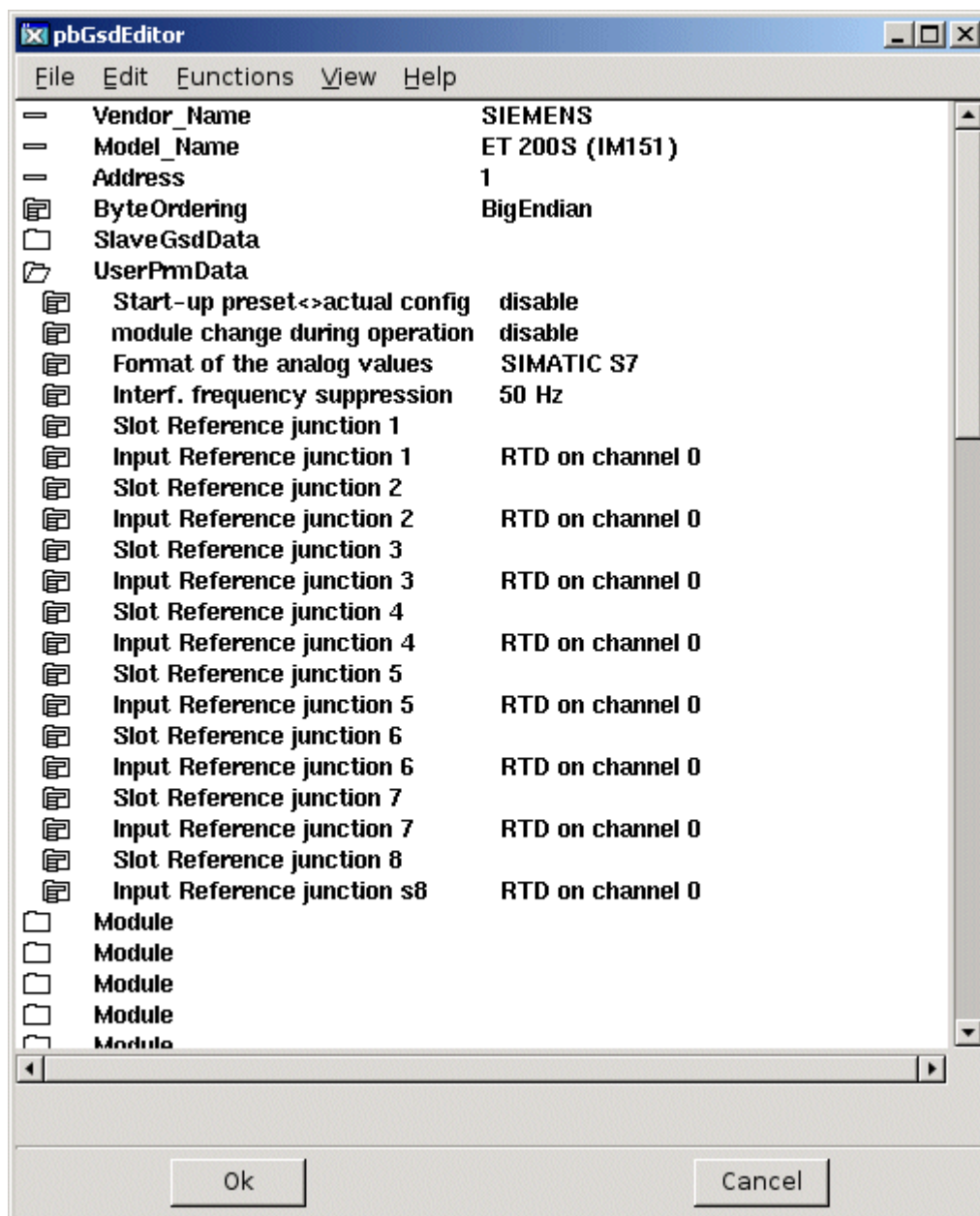
Varje slav som är kopplad till profibus-slingan konfigureras med objekt av klassen Pb\_DP\_Slave eller subclasser till denna klass. Slavobjekten läggs som barn till master-objektet. För slavobjektet kan man öppna en speciell profibus-konfigurator, som konfigurerar slavobjektet, och skapar modulobjekt för de moduler som är kopplade till slaven. Profibus-konfiguratorn utgår från gsd-filen för slaven. gsd-filen är en textfil som tillhandahålls av leverantören, och beskriver de olika konfigurationer som kan göras för den aktuella slaven. Innan man öppnar profibus-konfiguratorn måste man ange vilken gsd-fil som den ska utgå ifrån. Det här gör man genom att lägga gsd-filen för den aktuella slaven på \$pwrp\_exe, och lägga in filnamnet i attributet GSDfile i slavobjektet.

Om det finns en subclass för den slav man ska konfigurera, t ex, Siemens\_ET200S\_IM151, finns normalt gsd-filen angiven i slavobjektet och gsd-filen följer med Proview distributionen.

När den här operationen är utförd, kan man öppna profibus konfiguratorn genom att högerklicka på objektet och aktivera 'ConfigureSlave' i popupmenyn.

## Profibus konfiguratorn

Profibus konfiguratorn kan öppnas för ett slavobjekt, dvs ett objekt av klassen Pb\_DP\_Slave eller av en subclass av denna. Förutsättningen är att det finns en läsbar gsd-fil angiven i GSDfile attributet.



### **Address**

Slavens adress anges i Address attributet. Adressen har ett värde i intervallet 0-125 som vanligen anges med omkopplare på slaven.

### **SlaveGsdData**

Mappen SlaveGsdData innehåller diverse data av informativ karaktär.

### **UserPnmData**

Mappen SlaveGsdData innehåller de parametrar som kan konfigureras för den aktuella slaven.

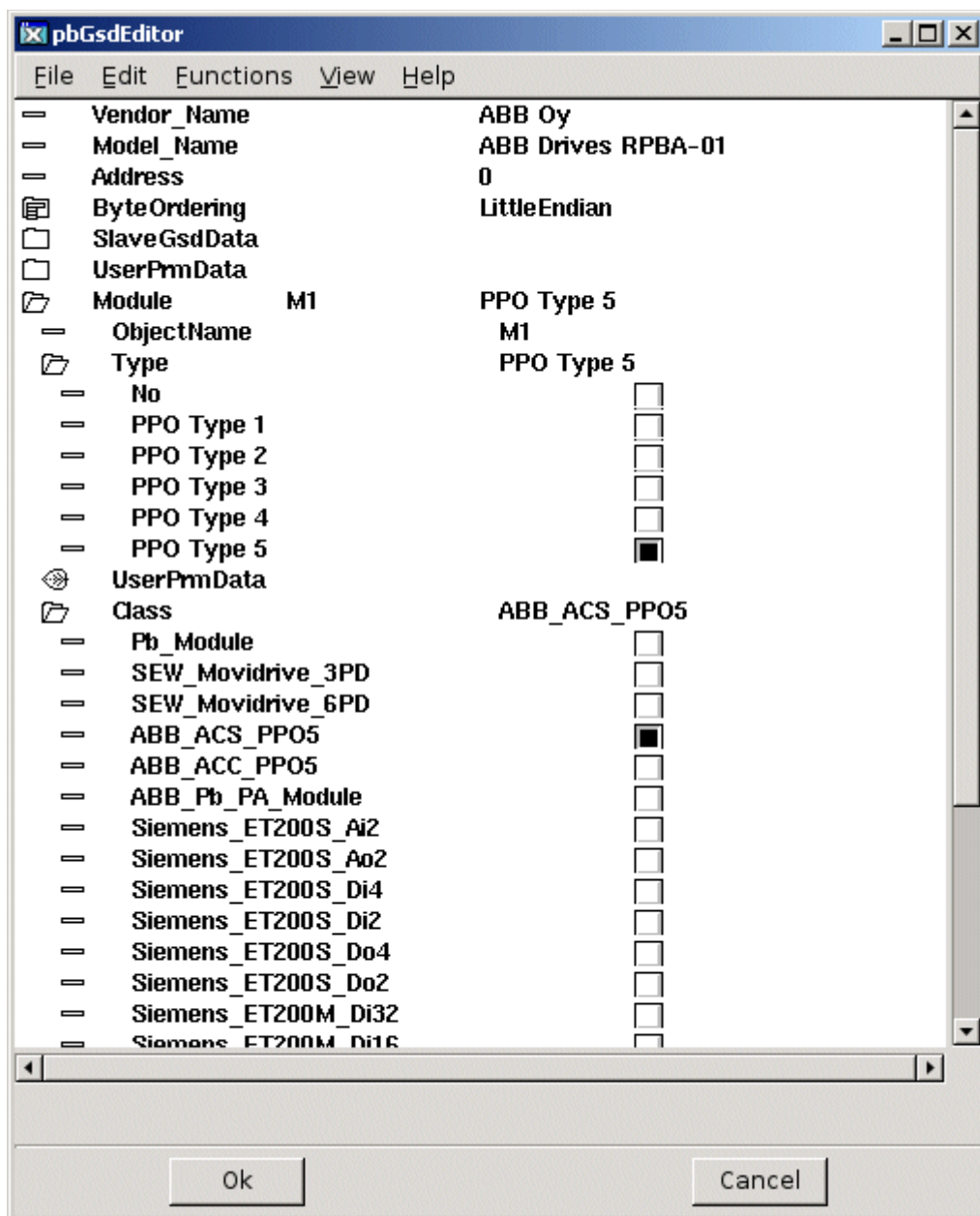
### **Module**

En slav har plats för en eller flera moduler. Det finns modulära slavar med enbart en modul, där

slaven och moduler utgör en enhet, och det finns slavar av rack typ som man kan hänga på ett stort antal moduler. Profibuskonfiguratorn visar en mapp för varje modul som kan konfigureras för slaven.

För varje modul anges ett objektsnamn, t ex M1, M2 etc. Moduler på samma slav måste ha olika objektsnamn.

Vidare anges även modultypen. Denna väljs från en lista av modultyper som supportas av den aktuella slaven. Listan ligger under 'Type'.



När man har valt typ konfigureras parametrar för den valda modultypen under UserPmData.

Man måste även ange en klass för modul-objektet. Vid konfigureringen skapas ett modulobjekt för varje konfigurerad modul. Objektet är av klassen Pb\_Module eller en subklass av denna, och under 'Class' listas alla subklasser till Pb\_Module. Hittar man någon som motsvarar aktuell modultyp anger man denna, annars väljer man basklassen Pb\_Module. Skillnaden mellan subklasserna och basklassen är att det i subklasserna finns en definierad dataarea i form av kanalobjekt (se avsnitt Specifiera Dataarean nedan).

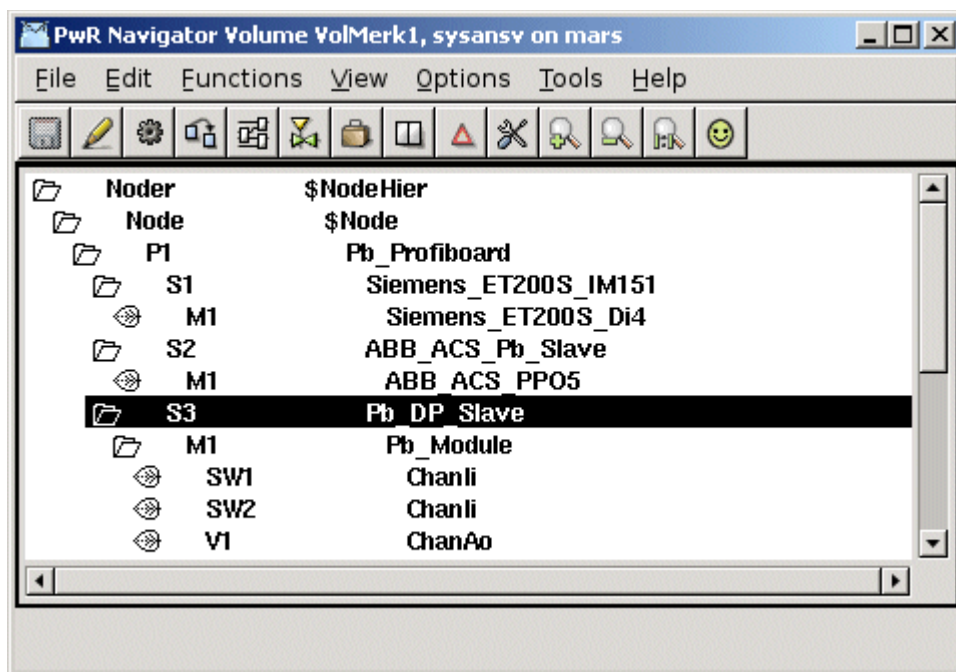


När alla moduler är konfigurerade sparar man genom att klicka på 'Ok' och går ur genom att klicka på 'Cancel'. Modulobjekt med angivet objektsnamn och angiven klass har nu skapats under slavobjektet.

Förhoppningsvis hittar man ett modulobjekt som motsvarar den aktuella modulen. Kriteriet för om ett modulobjekt är användbar eller inte, är om specifikationen av dataarean matchar den aktuella modulen. Om man inte hittar ett lämpligt modulobjekt finns två möjligheter: att i en skapa en ny klass med Pb\_Module som basclass, och i denna lägga in lämpliga kanalobjekt, eller att konfigurera kanalobjekten under ett Pb\_Module objekt. Det senaste alternativet är enklast om det är frågan om enstaka modulobjekt, men har man många moduler av samma typ bör man överväga att skapa en klass för denna.

## Specificera dataarean

Nästa steg är att specificera dataarean för en modul. Ingångsmoduler läser in data som skickas till processnoden på bussen, och utgångsmoduler tar emot data som ställs ut. Det finns även moduler som båda tar emot och ställer ut data, t ex frekvensomformare. Hur dataarean som tas emot resp skickas på bussen måste konfigureras, och det görs med kanalobjekt. Inarean specificeras med ChanDi, ChanAi och ChanIi objekt, och utarean med ChanDo, ChanAo och ChanIo objekt. Kanalobjekten läggs som barn till modulobjektet, eller, om man väljer att göra en speciell klass för modulen, som interna attribut i modulobjektet. I kanalobjektet måste man ange Representation, som specificerar formatet på en variabel, och i vissa fall även Number (för Bit-representation). I slavobjektet kan man även behöva ange ByteOrdering (LittleEndian eller BigEndian) och FloatRepresentation (Intel eller IEEE).



## Digitala ingångar

Digitala ingångsmoduler skickar ingångarna som bitar i ett ord. Varje ingång specificeras med ett ChanDi objekt. Representation sätts till Bit8, Bit16, Bit32 eller Bit64, beroende på ordets storlek, och Number anger det bit-nummer i ordet som innehåller kanalvärdet (första biten har nummer 0).

## **Analoga ingångar**

En analog ingång överförs vanligtvis som ett heltal och specificeras med ett ChanAi objekt. Representation matchas mot heltalsformatet i överföringen. I vissa fall skickas värdet som ett flyttal, och då måste flyttalsformatet anges i slavobjektets FloatRepresentation (FloatIntel eller FloatIEEE). Område för konvertering till ingenjörstorhet specificeras i RawValueRange, ChannelSigValueRange, SensorSigValueRange och ActValRange (eftersom signalvärde inte används kan ChannelSigValRange och SensorSigValRange sättas till samma värde som RawValRange).

## **Digitala utgångar**

Digitala utgångsmoduler specificeras med ChanDo objekt. Representation sätts till Bit8, Bit16, Bit32 eller Bit64 beroende på formatet vid överföringen.

## **Analoga utgångar**

För analoga utgångar används ChanAo objekt. Sätt Representation och ange områden för konvertering från ingenjörstorhet till överförings område (sätt ChannelSigValRange och SensorSigValRange till samma värde som RawValRange).

## **Komplexa dataareor**

Många moduler skickar en blandning av heltal, flyttal, bitmaskar etc. Man får då kombinera kanalobjekt av olika typ. Kanalobjekten ska ligga i samma ordning som det data de representerar är organiserade i dataarean. För moduler med både in- out utarea lägger man normalt inareans kanalerna först och därefter utareans kanalerna.

## **Drivrutin**

Softing PROFiBoard kräver installatin av en drivrutin som kan laddas hem från [www.softing.com](http://www.softing.com).

## **Agent objekt**

### ***Pb\_ProfiBoard***

Agent objekt för en profibus master av typen Softing PROFiBoard. Objektet placeras i nodhierarkin under nodobjektet.

## **Slavobjekt**

### ***Pb\_Dp\_Slave***

Basobjekt för en profibus slav. Placeras under ett profibus agentobjekt. I GSDfile attributet anges en gsd-fil för den aktuella slaven. När gsd-filen är angiven kan slaven konfigurera med Profibuskonfiguratorn.

### ***ABB\_ACS\_Pb\_Slave***

Slavobjekt för en frekvensomformare ABB ACS800 med protokoll PPO5.

### ***Siemens\_ET200S\_IM151***

Slavobjekt för en Siemens ET200S IM151.

### ***Siemens\_ET200M\_IM153***

Slavobjekt för en Siemens ET200M IM153.

## **Modulobjekt**

### ***Pb\_Module***

Basklass för en profibus modul. Skapas av profibuskonfiguratorn. Placeras som barn till ett slavobjekt.

### ***ABB\_ACS\_PPO5***

Modulobjekt för en frekvensomformare ABB ACS800 med protokoll PPO5.

### ***Siemens\_ET200S\_Ai2***

Modulobjekt för en Siemens ET200S modul med 2 analoga ingångar.

### ***Siemens\_ET200S\_Ao2***

Modulobjekt för en Siemens ET200S modul med 2 analoga utgångar.

### ***Siemens\_ET200M\_Di4***

Modulobjekt för en Siemens ET200M modul med 4 digitala ingångar.

### ***Siemens\_ET200M\_Di2***

Modulobjekt för en Siemens ET200M modul med 2 digitala ingångar.

### ***Siemens\_ET200M\_Do4***

Modulobjekt för en Siemens ET200M modul med 4 digitalt utgångar.

### ***Siemens\_ET200M\_Do2***

Modulobjekt för en Siemens ET200M modul med 2 digitala utgångar.

# Adaption av I/O system

I detta avsnitt beskrivs hur man inför nya I/O system i Proview.

Att lägga in ett nytt I/O system kräver kunskap i hur man skapar klasser i Proview, samt baskunskap i c programmering.

Ett I/O system kan läggas in för ett enskilt projekt eller ett antal projekt, eller i Proview's bassystem. I det första fallet räcker det med att installera Proview's utvecklingsmiljö. I det senare fallet måste man installera och bygga från Proview's källkod.

## Översikt

I/O hanteringen i Proview består av ett ramverk som identifierar I/O objekt på en processnod, och anropar I/O objektens metoder för att hämta och ställa ut data.

## Nivåer

I/O objekten i en processnod konfigureras i tre nivåer: agent, rack och kort. Ibland finns även en fjärde nivå närvarande: kanal. Kanalobjekten kan konfigureras som individuella objekt, eller existerar som interna objekt i ett kort-objekt.

Till agent-, rack- och kortobjekten kan man registrera metoder. Metoderna kan vara av typen Init, Close, Read, Write och Swap, och anropas av I/O ramverket i en specifik ordning. Funktionaliteten hos ett I/O objekt utgörs av objektets attribut, och de registrerade metoderna för objektet. Allt I/O ramverket gör är att identifiera objekten, välja ut de objekt som är giltiga för den aktuella processen, och anropa metoderna för dessa objekt i en specifik ordning.

Betrakta ett centraliserat I/O system med digitala ingångskort (Di) och digitala utgångskort (Do) monterade på processnodens lokala bus. I det här fallet är agent nivån överflödigt och \$Node objektet rycker in som ställföreträdande agentobjekt. Under \$Node-objektet läggs ett rack-objekt med en open och en close metod. Open metoden knyter upp sig mot drivrutinen för korten. Under rackobjektet konfigureras kortobjekt för Di och Do korten. Di korten har en Open och en Close metod som initierar resp stänger ner kortet, och en Read metod som hämtar värdet på kortets ingångar. Do-kort objekten har också Open och Close metoder, samt en Write metod som ställer ut lämpliga värden på kortens utgångar.

Om vi tittar på ett annat I/O system, Profibus, är nivåerna inte lika lätta att identifiera som i föregående exempel. Profibus är ett distribuerat system, med ett masterkort monterat på den lokala PCI-bussen, som kommunicerar via en seriell förbindelse med slavar placerade ute i anläggningen. Varje slav kan innehålla moduler av olika typ, t ex en modul med 4 Di kanaler, och en med 2 Ao kanaler. I det här fallet representerar masterkortet agentnivån, slavarackerna racknivån och modulerna kortnivån.

Agent, rack och kort nivåerna är mycket flexibla, och definieras huvudsakligen av attributen och metoderna för I/O systemet klasser. Det gäller inte på kanalnivån, som består av objekt av klasserna ChanDi, ChanDo, ChanAi, ChanAo, ChanLi, ChanIo och ChanCo. Uppgiften för ett kanalobjekt är att representera ett ut eller ingångs värde på I/O enheten och överföra detta värde till det

signalobjekt som är kopplat till kanalobjektet. Signalobjekt ligger i anläggningshierarkin och representerar en t ex en givare eller en order till ett ställdon i anläggningen. Liksom det finns en fysisk förbindelse mellan givaren i anläggningen och kanalen på I/O kortet, kopplas även signalobjektet ihop med kanalobjektet. Plcprogram, HMI och applikationer refererar alla signalobjektet som representerar komponenten i anläggningen, inte kanalobjektet som representerar en kanal på en I/O enhet.

## Area objekt

Värden som läses in från ingångsenheter och värden som ställs ut till utgångsenheter lagras i speciella areaobjekt. Areaobjekten skapas dynamiskt i runtime och ligger i systemvolymen under hierarkin *pwrNode-active-io*. Det finns ett area objekt för varje signaltyp. Normalt refererar man värdet på en signal genom signalens ActualValue attribut, men detta attribut innehåller i själva verket en pekare som pekar in i areaobjektet. Attributet ValueIndex anger vilket index i in areaobjekt som signalvärdet återfinns på. Orsaken till konstruktionen med areaobjekt är att man under exekveringen av ett logiknät inte vill ha förändringar i signalvärden. Varje plc-tråd tar därför en kopia av areaobjekten innan exekveringen startar och läser signalvärden från kopian, däremot skrivs beräknade signalvärden i areaobjektet.

## I/O objekt

Konfigureringen av I/O görs i nodehierarkin under \$Node-objektet. Till varje typ av komponent i I/O hierarkin skapar man en klass som innehåller attribut och metoder. Metoderna är av typen Open, Close, Read, Write och Swap och anropas av I/O-ramverket. Metoderna knyter upp sig mot bussen och läser in data som överförs till areaobjekten, eller hämtar data från areaobjekten som ställs ut på bussen.

## Processer

Det finns två systemprocesser i Proview som anropar I/O ramverket: plc processen och rt\_io\_comm. I plc processen gör varje tråd en initiering av I/O ramverket, vilket medför att I/O enheter kan läsas och skrivas synkront med exekveringen av plc-koden för respektive tråd.

## Ramverk

I/O-ramverket huvudsakliga uppgift är att identifiera I/O-objekt och anropa de metoder som finns registrerade för objekten.

En första initiering av I/O sker vid uppstart av runtimemiljön, när areaobjekten skapas och varje signal blir tilldelad en plats i areaobjektet. Dessutom kontrolleras kopplingen mellan signal och kanal. Signaler och kanaler har kopplats i utvecklingsmiljön på så sätt att identiteten för kopplad kanal har lagts i signalens SigChanCon attribut. Nu läggs signalens identitet in i kanalen SigChanCon så att man enkelt kan gå från kanal till signal.

Nästa initiering sker av varje process som vill knyta upp sig mot I/O-hanteringen. Plc-processen och rt\_io\_comm gör den här initieringen, men det är öppet även för applikationer som vill läsa eller skriva direkt mot I/O enheter att knyta upp sig. Vid initieringen läggs upp en datastruktur med alla agenter, rack, kort och kanaler som ska hanteras av just den här processen, och init-metoderna för dem anropas. Processen anropar sedan cykliskt en read- och en write-funktion, som anropar read- resp write-metoderna för I/O-objekten i datastrukturen.

## Metoder

Metoderna har som uppgift att initiera I/O-systemet, utföra läsning och utställning till I/O-enheterna, och slutligen att koppla ner. Hur de har uppgifterna fördelas beror på I/O systemet uppbyggnad. I ett centraliserat I/O på den lokala bussen, kan metoderna för olika kortobjekt själva gå ut och läsa resp skriva data till sin enhet, och metoderna för agent- och rack-objekten för en ganska lugn tillvaro. I ett distribuerat I/O kommer informationen för enheterna ofta samlade i ett paket, och det blir metoden för agent- eller rack-objektet som tar emot paketet och fördelar innehållet på olika kort-objekt. I kort-objektets metod lägger man lämpligen uppgiften att identifiera data för enskilda kanaler, och utföra eventuell konvertering och läsa resp skriva data i areaobjekten.

## Ramverk

En process kan initiera I/O ramverket genom att anropa `io_init()`. Som argument skickar man en bitmask som anger vilken process man är, och trådarna i `plcprocessen` anger även aktuell tråd. `io_init()` utför följande

- skapar en kontext.
- lägger upp en hierarkisk datastruktur av I/O objekt med nivåerna agent, rack, kort och kanal. För agenter allokeras en struct av typen `io_sAgent`, för rack en struct av typ `io_sRack`, för kort en struct av typ `io_sCard`, och slutligen för kanal en struct av typ `io_sChannel`.
- letar upp alla I/O objekt och kontrollerar Process attributet. Om Process attributet matchar den process som skickats med som argument till `io_init()`, läggs objektet in i datastrukturen. Om objektet har ett underliggande I/O objekt som matchar processen läggs det också in i datastrukturen. För `plcprocessen` kontrollerar man dessutom att tråd-argumentet i `io_init()` matchar ThreadObject-attributet i I/O-objektet. Resultatet blir en länkad trädstruktur med de agent, rack, kort och kanal objekt som ska hanteras av den aktuella processen.
- För varje I/O-objekt som läggs in, identifieras metoderna, och pekare till metod-funktionerna hämtas upp. Dessutom hämtas en pekare till objektet upp och objektsnamnet läggs in i datastrukturen.
- `init`-metoden för I/O objekten i datastrukturen anropas. Metoden för första agenten anropas först, därefter agentens första rack, rackets första kort, andra kort osv.

När initieringen är gjord kan processen anropa `io_read()` för att läsa från de I/O enheter som finns med i datastrukturen, och `io_write()` för att ställa ut värden. En tråd i `plcprocessen` anropar `io_read()` varje scan för att hämta in nya värden från processen. Därefter exekveras plc-koden och slutligen anropas `io_write()` för att ställa ut nya värden. Read-metoderna anropas i samma ordning som init-metoderna, och write-metoderna i omvänd ordning.

När processen terminerar, anropar den `io_close()` som i sin tur anropar close-metoderna för objekten i datastrukturen. Close-metoderna anropas i omvänd ordning jämfört med init-metoderna.

Vid en mjuk omstart gör även en omstart av I/O hanteringen. Först anropas close-metoderna, därefter anropas Swap-metoderna under den tiden omstarten pågår, och därefter init-metoderna. Anropet av swap-metoderna görs av processen `rt_io_comm`.

### `io_init`, funktion för initiering av ramverket

```
pwr_tStatus io_init(  
    io_mProcess    process,  
    pwr_tObjid     thread,
```

```

io_tCtx      *ctx,
int          relativ_vector,
float        scan_time
);

```

### io\_sCtx, ramverkets kontext

```

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 */
};

```

### Datastruktur för en 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;

```

### Datastruktur för ett 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 kontrollad 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;

```

## Datastruktur för ett kort

```

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 kontrollad 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;

```

## Datastruktur för en kanal

```

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;

```

## Skapa I/O-objekt

I en processnod konfigureras I/O-systemet i nodhierarkin med objekt av typen agent, rack och kort. Klasserna för de här objekten skapas man i klasseditorn. Klasserna definieras med ett \$ClassDef objekt, ett \$ObjBodyDef objekt (RtBody), och under detta med ett \$Attribute objekt för varje attribut i klassen. Attributen bestäms av funktionaliteten i metoderna för klassen, men det finns några generella attribut (Process, ThreadObject och Description). I \$ClassDef-objektets Flag-ord ska anges om det är ett agent, rack eller kort-objekt, och metoderna definieras med speciella *Method* objekt.

Det är ganska vanligt att flera klasser i ett I/O-system delar attribut och kanske även metoder. Ett ingångskort som finns med olika antal ingångar, kan ofta använda samma metod. Det som skiljer är



antalet kanalobjekt. De övriga attributen kan då läggas i en basklass, som även innehåller metod-objekten. Subklasser ärver både attributen och metoderna, det som tillkommer är kanalobjekten, som kan läggas som enskilda attribut, eller, om de är av samma typ, som en vektor av kanalobjekt. Om kanalerna läggs som vektor eller som enskilda attribut påverkas av hur man vill att referensen i plcdokumentet ska se ut. Men en array får man indexering från 0, med enskilda attribut kan man styra namngivningen själv.

I exemplet nedan visas en basklass i Fig x och en subklass i Fig x. Basklassen Ssab\_BaseDiCard innehåller alla attribut som används av metoderna och I/O-ramverket. Subklassen Ssab\_DI32D innehåller super-attributet med TypeRef Ssab\_BaseDiCard, och 32 kanalattribut av typen ChanDi. Eftersom indexeringen av den här korttypen av tradition går från 1 har man valt att lägga kanalerna som enskilda attribut, men de kan också läggas som en vektor av typen ChanDi.

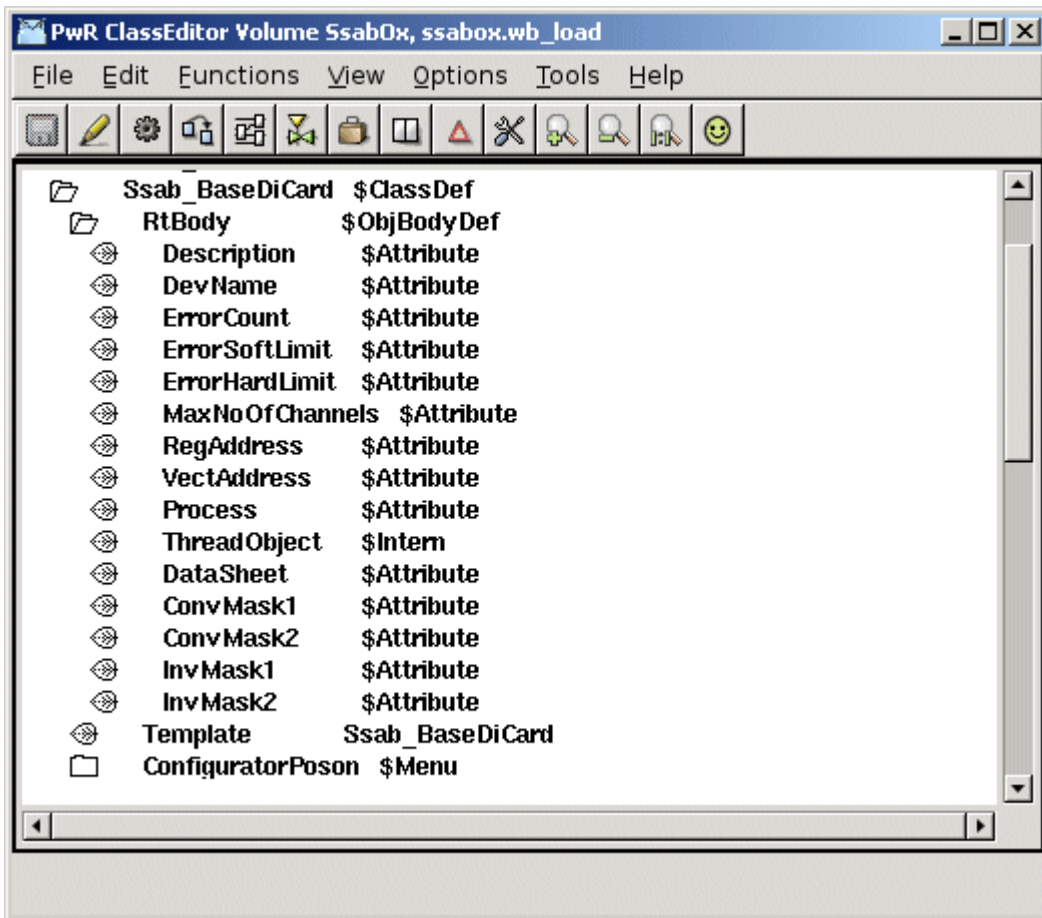


Fig Exempel på en basklass för ett kort

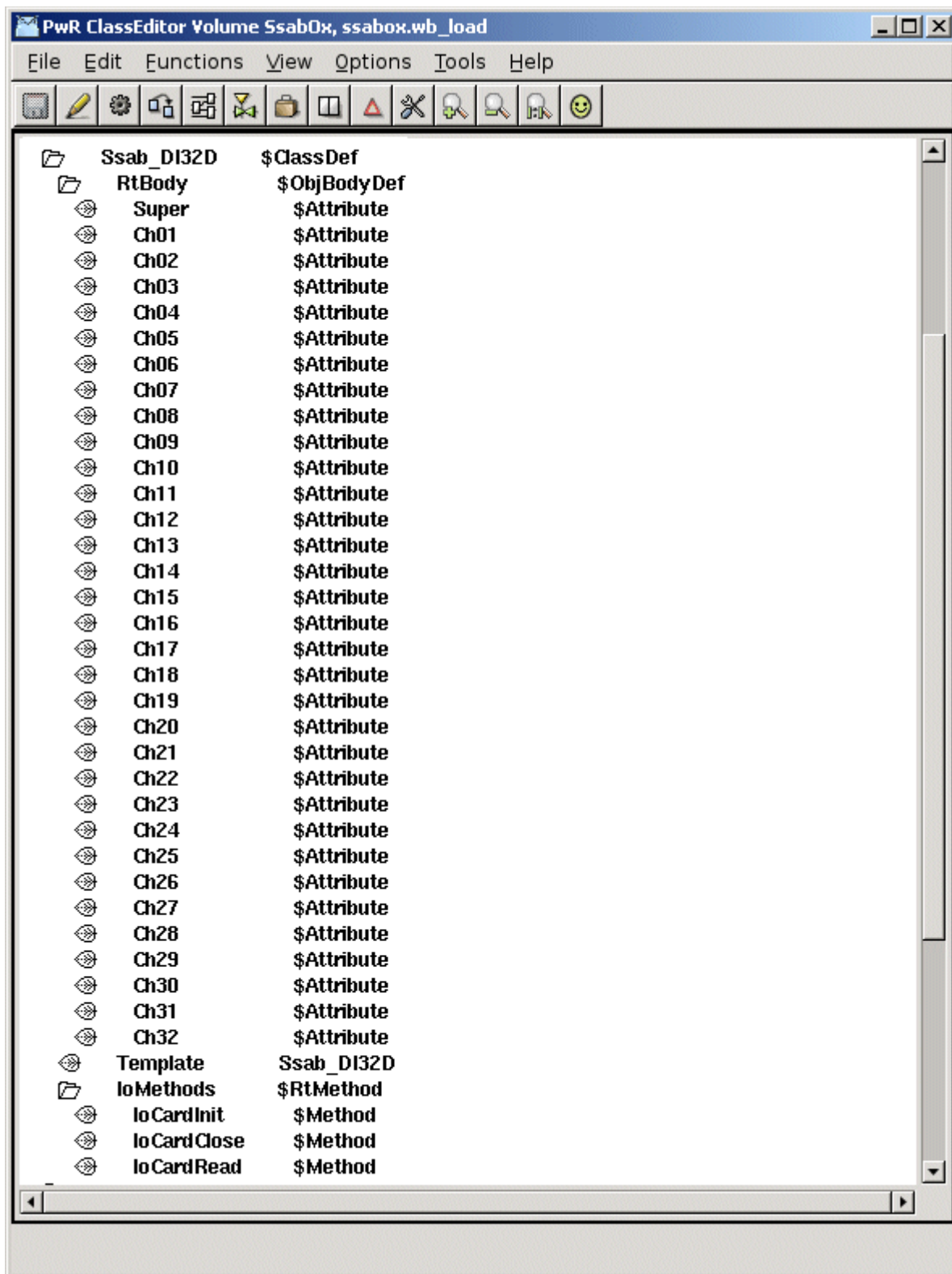


Fig Exempel på en kortklass med superklass och 32 kanalobjekt

## Flags

I \$ClassDef objektets Flags attribut ska IOAgent biten sätts för agent-klasser, IORack biten för rack-klasser och IOCard biten för kort-klasser.

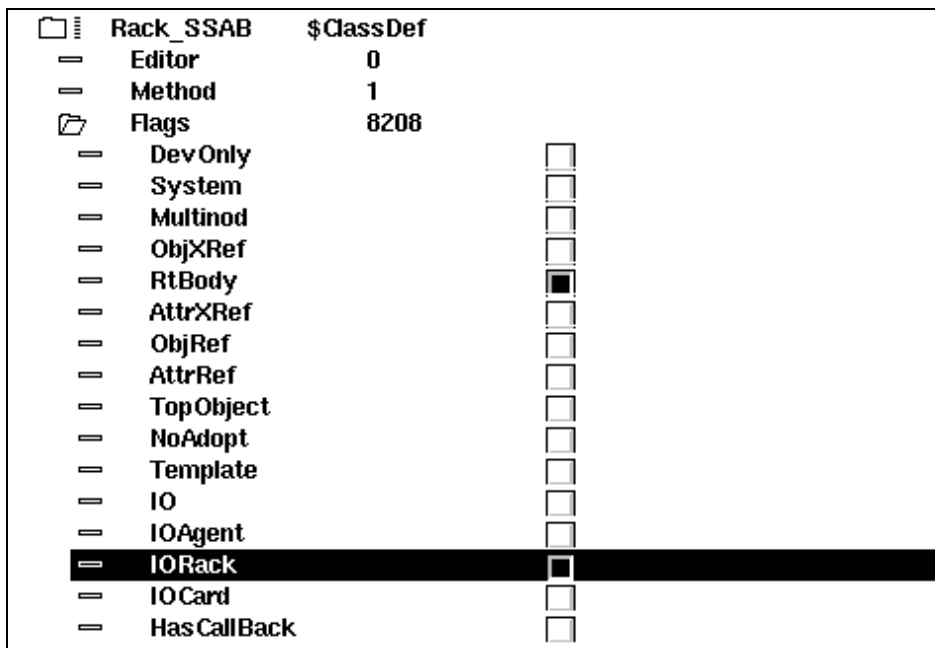


Fig IORack biten satt för en rack klass

## Attribut

### Description

Attribut av typ pwr:Type-\$String80. Innehållet visas som beskrivning i navigatören.

### Process

Attribut av typ pwr:Type-\$UInt32. Anger vilken process som ska hantera enheten.

### ThreadObject

Attribut av typ pwr:Type-\$Objid. Anger vilken tråd i plcprocessen som ska hantera enheten.

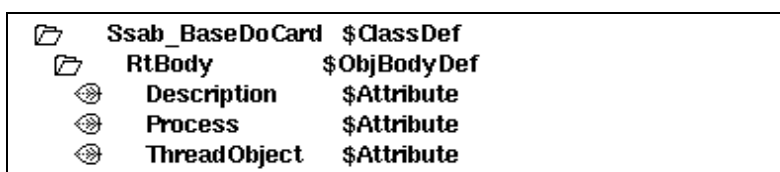


Fig Standard attribut

## Metod objekt

Metodobjekten används för att identifiera metoderna för klassen. Metoderna utgörs av c-funktioner som registreras med en i c-kodennamn, en sträng som består av klassnamn och metodnamn, t ex "Ssab-AIuP-IOCardInit". Namnet läggs även in i ett metodobjekt i klassbeskrivning och gör att I/O-ramverket kan hitta rätt c-funktion för klassen.

Under \$ClassDef objektet läggs ett \$RtMethod objekt men namnet IoMethods. Under detta läggs ett \$Method objekt för varje metod som ska definieras för klassen. I attributet MethodName anges namnet för metoden.

## Agenter

För agenter skapas \$Method objekt med namnen IoAgentInit, IoAgentClose, IoAgentRead och IoAgentWrite.

## Rack

För rack skapas \$Method objekt med namn IoRackInit, IoRackClose, IoRackRead och IoRackWrite.

## Kort

För kort skapas \$Method objekt med namn IoCardInit, IoCardClose, IoCardRead och IoCardWrite.

























	<b>Ssab_AI32uP</b>	<b>\$ClassDef</b>
	<b>RtBody</b>	<b>\$ObjBodyDef</b>
	<b>Template</b>	<b>Ssab_AI32uP</b>
	<b>IoMethods</b>	<b>\$RtMethod</b>
	<b>IoCardInit</b>	<b>\$Method</b>
	Method Name	Ssab_AiuP-IoCardInit
	Method Arguments	
	<b>IoCardClose</b>	<b>\$Method</b>
	Method Name	Ssab_AiuP-IoCardClose
	Method Arguments	
	<b>IoCardRead</b>	<b>\$Method</b>
	Method Name	Ssab_AiuP-IoCardRead
	Method Arguments	

Fig Metodobjekt

## Connect-metod för ThreadObject

När trådobjektet i attributet ThreadObject ska anges för en instans, kan det matas inför hand, men man kan även definiera en meny-metod som lägger in ett utvalt trådobjekt i attributet. Metoden aktiveras från popupmenyn för IO-objektet i konfiguratören.

Metoden definieras i klassbeskrivningen med \$Menu och \$MenuButton objekt, se fig x. Under \$ClassDef objektet läggs ett \$Menu objekt med namnet ConfiguratorPoson. Under detta ytterligare ett \$Menu objekt med namnet Pointed, och under detta ett \$MenuButton objekt med namnet Connect. Ange ButtonName (texten i popupmenyn för metoden) och MethodName och FilterName. Metoden och filtret som används finns definierade i \$Objid klassen. MethodName ska vara \$Objid-Connect och FilterName \$Objid-IsOkConnect.

	<b>Ssab_BaseDoCard</b>	<b>\$ClassDef</b>
	<b>RtBody</b>	<b>\$ObjBodyDef</b>
	<b>Template</b>	<b>Ssab_BaseDoCard</b>
	<b>ConfiguratorPoson</b>	<b>\$Menu</b>
	<b>Pointed</b>	<b>\$Menu</b>
	<b>Connect</b>	<b>\$MenuButton</b>
	ButtonName	Connect PlcThread
	MethodName	\$Objid-Connect
	Method Arguments	
	FilterName	\$Objid-IsOkConnect
	Filter Arguments	

## Fig Connect metod

## Metoder

För agent, rack och kort klasserna skriver man metoder i programmeringsspråket c. En metod är en c-funktion som är gemensam för en klass (eller flera klasser) och som anropas av I/O-ramverket för alla instanser av en klassen. För att I/O-hanteringen ska bli så flexibel som möjlig, utför metoderna det mesta av I/O-hanterings jobbet. Ramverkets uppgift är egenligen bara att identifiera de olika I/O-objekten och anropa metoderna för dessa, samt tillhandahålla lämpliga datastrukturer för metoderna.

Det finns fem typer av metoder: Init, Close, Read, Write och Swap.

- Init-metoden anropas vid initieringen av I/O-hanteringen, dvs vid uppstart av runtime miljön och vid en mjuk omstart.
- Close-metoden anropas när I/O-hanteringen avslutas, dvs när runtime-miljön stoppas och vid en mjuk omstart.
- Read-metoden anropas cykliskt när det är dags att läsa av ingångkort.
- Write-metoden anropas cyliskt när det är dags att ställa ut värden till utgångskorten.
- Swap-metoden anropas under en mjuk omstart.

### Lokal datastruktur

I datastrukturerna io\_sAgent, io\_sRack och io\_sCard finns ett element Local där metoden kan lagra en pekare till lokal data för en I/O-enhet. Lokala data allokeras i init-metoden och finns sedan tillgängligt vid varje metoanrop.

### Agent-Metoder

#### IoAgentInit

Initierings metod för en 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)
```

## Registrering av metoder

Metoderna för en klass måste registreras, så att man från metod-objektet i klassbeskrivningen kan hitta rätt funktioner för en specifik klass. Nedan visas ett exempel på hur metoderna IoCardInit, IoCardClose och IoCardRead registreras för klassen Ssab\_Aiup.

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

## Registrering av klassen

Dessutom måste klassen registreras. Det här sker på olika sätt beroende på om I/O systemet är implementerat som en modul i Proview's bassystem, eller som en del i ett projekt.

## Modul i Proview's bassystem

Är I/O systemet implementerat som en modul i Proview's bassystem, skapar man en fil, lib/rt/src/rt\_io\_'modulnamn'.meth, och listar alla klasser som har registrerade metoder i denna.

## Projekt

Om I/O systemet är en del av en projekt, sker registreringen i en c-modul som länkas med plc-programmet. I exemplet nedan registreras klasserna Ssab\_Rack och Ssab\_Aiup i filen rt\_io\_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
};
```

Filen kompileras och länkas med plc-programmet genom att en länk-fil skapas på \$pwrp\_exe. Filen ska namnges plc\_'nodnamn'\_busnr'.opt, tex plc\_mynode\_0517.opt. Innehållet i filen skickas med som indata till länkaren, ld, och man måste även ta med modulerna med metoderna för klassen. I exemplet nedan antas att dessa moduler ligger i arkivet \$pwrp\_lib/libpwrp.a.

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

### ***Exempel på rack metoder***

```
#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  
};
```

### ***Exempel på metoder digitalt ingångskort***

```
#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
};

```

### ***Exempel på metoder för digitalt utgångskort***

```
#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 registered here. */
pwr_dExport pwr_BindIoMethods(Ssab_Do) = {

```

```
pwr_BindIoMethod(IoCardInit),  
pwr_BindIoMethod(IoCardClose),  
pwr_BindIoMethod(IoCardWrite),  
pwr_NullMethod  
};
```