SIEMENS

PROFINET IO Development Kit V 3.2.0

Schnittstellenbeschreibung

Ausgabe (10/2010)

Vorwort

Zweck des Handbuchs

Die Anwenderbeschreibung beschreibt die Software-Funktionalität des Development Kits für ein PROFINET IO-Device:

- Einleitung
- Detaillierte Beschreibung der einzelnen SW-Funktionen
- Beispiel für den Anwender

Zielgruppe des Handbuchs

Dieses Handbuch ist für Softwareentwickler und für Applikationentwickler gedacht, die das Devkit für neue Produkte auf einer beliebigen Echtzeitplattform einsetzen wollen. Der Entwickler erhält eine CD, auf der der komplette Sourcecode des IO-Stacks, Dokumentation, eine Beispielapplikation sowie eine Beispiel-Plattformportierung enthalten ist.

Dieses Handbuch gilt sowohl für ERTEC-basierte Plattformen (ERTE200, ERTEC400) als auch für Standard-Ethernetcontroller. Bestimmte im Handbuch beschriebene Funktionalitäten wie IRT können hingegen nur mit spezieller Hardwareunterstützung wie z.B. ERTEC400/200 realisiert werden.

Aufbau des Handbuchs

Das vorliegende Handbuch beschreibt das PROFINET IO-Device Developers Kit. Es ist wie folgt aufgebaut:

- Kapitel 1 Einleitung
- Kapitel 2 PROFINET IO-Device Software Übersicht
- Kapitel 3 Allgemeine Hinweise zur Softwareerstellung von PROFINET IO Devices
- Kapitel 4 Schnittstellenbeschreibung
- Kapitel 5 Begriffs- und Literaturverzeichnis

Dieses Handbuch enthält die Beschreibung des PROFINET IO-Stacks für die unterstützten Development Kits, welche zum Zeitpunkt der Herausgabe der Anwenderbeschreibung gültig sind. Wir behalten uns vor, bei neueren Erzeugnisständen die Anwenderbeschreibung zu aktualisieren.

Wegweiser

Um Ihnen den schnellen Zugriff auf spezielle Informationen zu erleichtern, enthält das Handbuch folgende Zugriffshilfen:

- Am Anfang des Handbuchs finden Sie ein vollständiges Inhaltsverzeichnis und jeweils eine Liste aller im gesamten Handbuch enthaltenen Abbildungen und Tabellen.
- Im Anschluss an die Anhänge finden Sie ein Glossar, in welchem wichtige Fachbegriffe definiert sind, die in diesem Handbuch verwendet werden.
- Hinweise auf weitere Dokumente sind mit Hilfe von Literaturnummern in Schrägstrichen /Nr./ angegeben. Damit können Sie dem Literaturverzeichnis am Ende des Handbuchs den genauen Titel der Dokumente entnehmen.

Weitere Unterstützung

Bei Fragen zur Nutzung des beschriebenen Entwicklungs-Kits, die Sie nicht in der Dokumentation beantwortet finden, wenden Sie sich bitte an Ihre Siemens Ansprechpartner in den für Sie zuständigen Vertretungen oder Geschäftsstellen.

Fragen, Anmerkungen und Verbesserungen zum vorliegenden Handbuch bitte schriftlich an die angegebene E-Mail-Adresse schicken.

Zusätzlich erhalten Sie allgemeine Informationen, aktuelle Produkt-Informationen und Downloads, die beim Einsatz nützlich sein können, im Internet unter folgenden Link:

http://www.siemens.de/comdec

Technischer Ansprechpartner für Deutschland / weltweit

Siemens AG
Automation & Drive
ComDeC

Tel.: +49 911 750 4384
Tel.: +49 911 750 2080
Fax: +49 911 750 2100
E-Mail: ComDeC@siemens.com

Hausadresse:
Würzburgerstr.121
90766 Fürth

Briefadresse:
Postfach 2355
90713 Fürth

Technischer Ansprechpartner für USA

PROFI Interface Center:
One Internet Plaza
PO Box 4991
Johnson City, TN 37602-4991

Fax: (423)- 262- 2103
Tel: (423)- 262- 2576
E-Mail: profibus.sea@siemens.com

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Rahmen von PROFINET ist PROFINET IO ein Automatisierungskonzept für die Realisierung modularer, dezentraler Applikationen. Mit PROFINET IO erstellen Sie Automatisierungslösungen, wie sie Ihnen von PROFIBUS her bekannt und vertraut sind. Die Umsetzung von PROFINET IO wird einerseits durch den PROFINET Standard für Automatisierungsgeräte und andererseits durch das Engineering Tool STEP 7 realisiert. Das bedeutet, dass Sie in STEP 7 nahezu die gleiche Applikationssicht haben – unabhängig davon, ob Sie PROFINET- Geräte oder PROFIBUS-Geräte projektieren. Die Programmierung Ihres Anwenderprogramms ist damit für PROFINET und PROFIBUS nahezu identisch.

Für PROFINET IO wird ein Softwarestack angeboten. Auf dieser Basis können PROFINET IO-Devices erstellt werden. Der Stack entlastet den Anwender von der Erstellung der kompletten Kommunikationssoftware. Die Funktionalität beinhaltet

- zyklischer und azyklischer Datenaustausch mit einem oder mehreren PROFINET IO-Controller
- Senden und Empfangen von Diagnose- und Prozessalarmen, Plug- und Pull-Alarmen
- Vergabe von IP-Adressen und Gerätenamen über Ethernet

Der Stack wird im Quellcode ausgeliefert und kann damit auf jede beliebige Hardware- und Betriebssystemplattform portiert werden. Notwendige Anpassungen sind dabei in definierten Schnittstellen zu Hardware und Betriebssystem gekapselt, um eine Portierung möglichst einfach und kostengünstig durchführen zu können.

Gute PROFINET IO-Kenntnisse werden vorausgesetzt, um den Firmware Stack implementieren zu können.

1.1 Inhalt und Zielgruppe dieser Schnittstellenbeschreibung

Die vorliegende Dokumentation ist gedacht für Entwickler von PROFINET IO-Devices. Sie beinhaltet eine

- Übersicht über den Aufbau des Softwarestacks
- Beschreibung der Anwenderschnittstelle des PROFINET IO-Stacks
- Beschreibung der Netzwerk- und Betriebssystemanbindung des PROFINET IO-Stacks
- Beschreibung des Anwenderbeispiels

Diese Dokumentation beinhaltet nicht

- eine Übersicht über PROFINET IO
- eine Beschreibung der PROFINET IO-Busprotokolle
- eine detaillierte Beschreibung über Aufbau und Abläufe im PROFINET IO-Stack

1.2 Beispiel-Plattformen

Die Dokumentation ist weitgehend plattformunabhängig gestaltet, wobei an einigen Stellen zwischen zwei Beispiel-Plattformvarianten unterschieden werden muss:

- ERTEC basierte Plattformen (ERTEC200,ERTEC400), unterstütztes Betriebssystem eCos, Entwicklungskit enthält Software, Dokumentation, EB400/EB200 Evaluation-Board
- Plattformen basierend auf Standard Ethernetcontrollern, unterstütztes Betriebssystem NETOS 6.3 GHS (Betriebssystem ThreadX, Greenhill-Toolkette), Entwicklungskit enthält Software + Dokumentation, aber keine Hardware.

Mit Standard-Ethernetcontroller basierten Plattformen kann RT-Datenaustausch, mit Ertec Plattformen RT + IRT-Datenaustausch realisiert werden. Wird in dieser Dokumentation IRT erwähnt, so bezieht sich dieses lediglich auf ERTEC Plattformen, nicht aber auf Standard-Ethernetcontroller.

1.3 Sonstige Hinweise

Bei der Portierung der Software auf andere Plattformen wird empfohlen, die zentralen Komponenten des PROFINET IO-Stacks unverändert zu lassen. Dies vereinfacht für den Anwender die Einspielung zukünftiger Versionen.

Das Anwenderbeispiel wurde auf der zugehörigen Beispielplattform (s.o.) getestet.

2 PROFINET IO-Device Software Übersicht

2.1 Softwarearchitektur

Die folgenden Bilder zeigen den hierarchischen Aufbau eines PROFINET Devices in einer Systemumgebung mit Echtzeit-Betriebssystem. Dabei muss sowohl zwischen der Hardwareplattform (ERTEC oder Standard-Ethernet Controller) als auch zwischen einem externen IP Stack (z.B. der native IP Stack des verwendeten Betriebssystems) oder den im ERTEC Development Kit enthaltenen Interniche IP Stack unterschieden werden. Bei Standard Ethernetcontrollern kann außerdem das Lower Layer Interface des IP Stacks auf dem EDD oder direkt auf dem BSP aufgesetzt sein. Folgende Kombinationen sind davon in den Development Kits enthalten:

	EB200/400, ECOS Plattform	**Standard Ethernet Controller, NETOS Plattform**
IP-Stack	INTERNICHE IP Stack, integriert in PNIO	Externer IP Stack der NETOS Plattform
SNMP MIB2 Agent	INTERNICHE IP Stack, integriert in PNIO	Externer MIB2 Agent der NETOS Plattform
IP Lower Layer Anbindung	Auf EDDI	Direkt auf NETOS BSP
EDD	EDDI	EDDS
SOCK	SOCK_lsa	SOCK_lit

Tabelle 1: Unterschiede der Stack-Layers in den verschiedenen Devkits

Die PROFINET IO-Device Software besteht aus folgenden Komponenten:

- PROFINET IO-Protokollsoftware (hellblau dargestellt)
- Systemanpassung (hellblau gepunktet)
- Echtzeit – Betriebssystem
- Board Support Package
- TCP/IP Stack (Für PROFINET IO wird nur UDP verwendet)
- Optional kann SNMP integriert werden
- plattformspezifische Adaptionsschicht
- Device-Applikation (gelb dargestellt)

Alle blau dargestellten Komponenten des PROFINET IO werden von Siemens bereitgestellt. Die grünen Komponenten werden in der Regel vom Betriebssystem- oder Controllerhersteller mitgeliefert. Nur die blau-grün gestreiften Komponenten müssen vom Anwender an die eigene Plattform angepasst werden, die anderen Komponenten können in der Regel unverändert übernommen werden. Für die Anpassung der blau-grün gestreiften Komponenten ist im PROFINET IO-Softwarestack Beispielcode enthalten, passend für die jeweilige Beispielplattform.

2.1.1 Softwarearchitektur für ERTEC basierte Plattformen

2.1.1.1 Integrierter Interniche IP stack

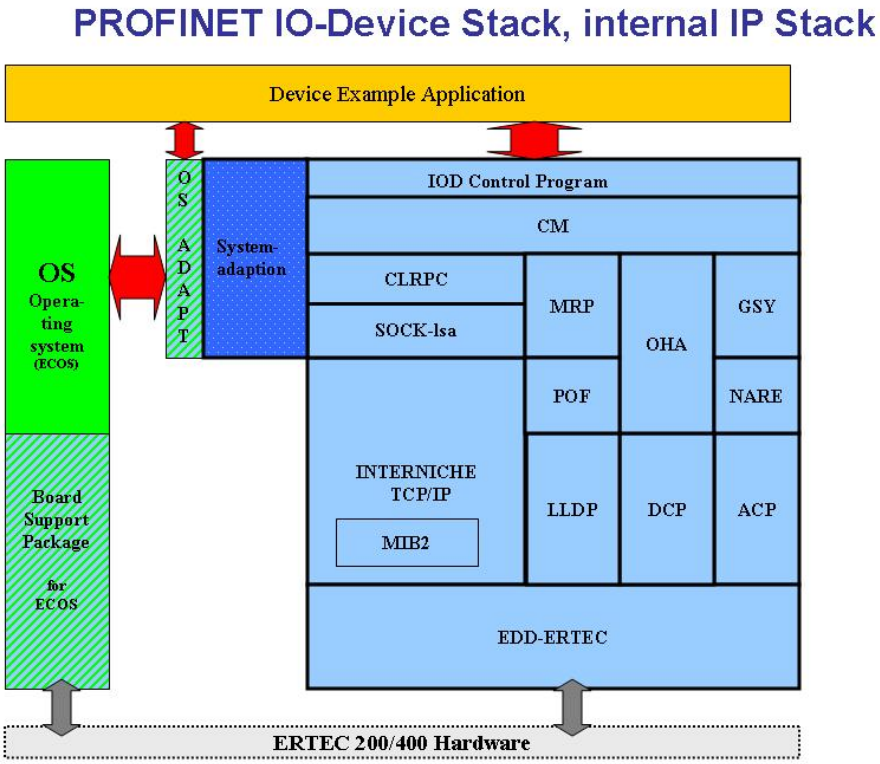


Abbildung 1: Systemumgebung für ein ERTEC basiertes IO-Device mit integriertem IP Stack

Diese Plattform hat die folgenden Eigenschaften:

- Komplettlösung für PROFINET inclusive Interniche IP Stack, SNMP MIB2 Agent
- Adaptionsschnittstellen für externe IP-Stacks und externe SNMP MIB2 Agent entfallen, der IP Stack ist direkt an das SOCK-lsa Paket adaptiert.
- Implementiert für ECOS Plattform

2.1.1.2 Externer IP stack

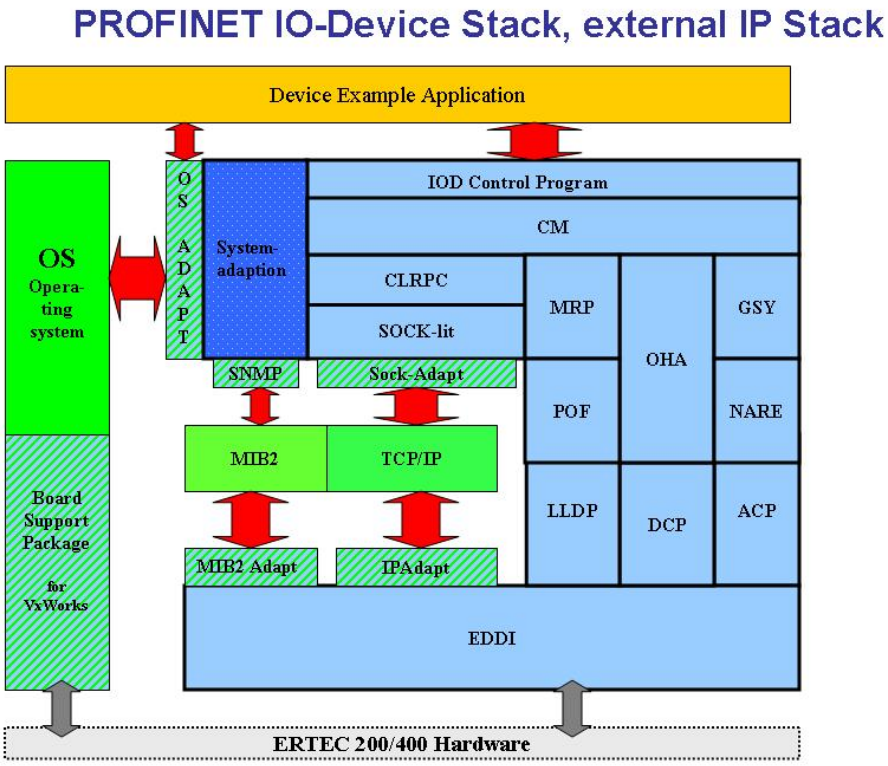


Abbildung 2: Systemumgebung für ein ERTEC basiertes PROFINET IO-Device mit externem IP Stack

Diese Plattform hat die folgenden Eigenschaften:

- Kein IP Stack oder SNMP-MIB2 Agent integriert, dieser muss extern über die entsprechenden Interfaces adaptiert werden.

2.1.1.3 Softwarearchitektur für Standard-Ethernetcontroller basierte Plattformen

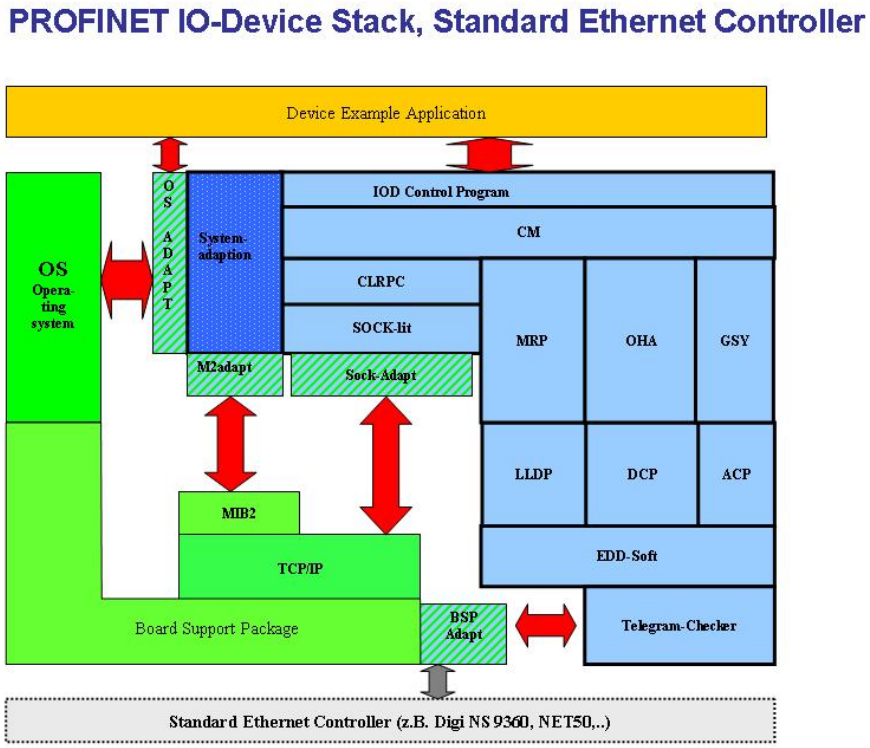


Abbildung 3: Systemumgebung für ein Standard-Ethernetcontroller basiertes PROFINET IO-Device

- Kein IP Stack oder SNMP-MIB2 Agent integriert, dieser muss extern adaptiert werden. Meist wird der vom Betriebssystemhersteller angebotene IP Stack verwendet, der bereits fertig an das Board Support Package adaptiert ist.
- Alle empfangenen Telegramme müssen zunächst an den Telegrammchecker geleitet. Dort wird entschieden, ob das Paket entweder an den IP Stack oder an den PROFINET EDD weitergeleitet wird.
- Implementiert für NETOS 6.3 Plattform.

2.2 Komponenten des PROFINET IO-Stacks

Das folgende Unterkapitel gibt eine kurze Übersicht über die Komponenten innerhalb des PROFINET IO-Stacks. Grundsätzlich können die Komponenten des Stacks nach folgenden Kategorien gegliedert werden:

- Systemunabhängige Basispakete mit einheitlicher Schnittstellen-Struktur. Dazu gehören ACP, CM, CLRPC, DCP, EDD, GSY, LLDP, MRP, OHA, SOCK. Die Basispakete stellen lediglich eine Art Funktionsbibliothek zur Verfügung, welche erst zusammen mit der Systemanpassung eine konkrete, lauffähige Systemimplementierung ergeben.
- Systemanpassung für alle enthaltenen Softwarepakete. Diese bildet die Schnittstelle zwischen den einzelnen systemunabhängigen Basispaketen und den Betriebssystem-Services wie Speichermanagemant, Taskmanagement, Interprozesskommunikation, Zeitmanagement. In der Systemanpassung ist auch die Softwarestruktur des IO-Stacks implementiert, d.h. welche Basispakete in welcher Task ablaufen und über welche Mechanismen die Tasks untereinander kommunizieren.
- OS-Abstraktion Layer. Sie bildet eine Low-Layer Abstraktionsschnittstelle zwischen der Systemanpassung und einem speziellen Betriebssystem. Bei der Portierung der Sofware auf ein anderes Betriebssystem muss dadurch nur die OS-Abstraktion Layer angepasst werden.
- Ergänzende Softwarepakete wie IOD-Control Programm.

2.2.1 EDD (Ethernet Device Driver)

Der EDD stellt Mechanismen bereit für das

- Selbständige Senden und Empfangen von zyklischen Realtime-Telegrammen
- Senden und Empfangen von azyklischen Realtime-Telegrammen
- Senden und Empfangen von Nicht - Realtime-Telegrammen

Er hat eine einheitliche LSA Schnittstelle zu überlagerten Clients (DCP, ACP) und eine Macro-Schnittstelle zur unterlagerten Komponente (Checker).

2.2.2 ACP (Acyclic Communication Protocol)

- Bearbeitung von Alarmen
- Schnittstelle zum Upper Layer (CM)
- Schnittstelle zum Lower Layer (EDD)
- Schneller Transfer von Buffer_Lock und Buffer_Unlock Services zwischen EDD und Context-Manager für den Austausch von zyklischen IO-Daten mit der Applikation.

2.2.3 CM (Context Manager)

- Aufbau und Verwalten von Kommunikationsbeziehungen zwischen IO-Device und IO-Controller. Requests für Aufbau von Kommunikationsbeziehungen werden vom remote IO-Controller mittels Connectionless Remote Procedure Calls über UDP und CLRPC in den Context Manager übertragen.
- Verwaltung der Ist-Konfiguration (gesteckte Module und Submodule).
- Schnittstelle zum Upper Layer (IOD)
- Schnittstelle zum Lower Layer (ACP, CLRPC, OHA, GSY)
- Schneller Transfer von Buffer_Lock und Buffer_Unlock Services zwischen ACP und Applikation für den Austausch von zyklischen IO-Daten mit der Applikation.

2.2.4 CLRPC (Connectionless Remote Procedure Call)

- Implementierung des Connectionless RPC Protokolls
- Schnittstelle zum Upper Layer (CM)
- Schnittstelle zum Lower Layer (SOCK)

2.2.5 DCP (Dynamic Configuration Protokoll)

- Vergabe von IP-Adressen und Gerätenamen über Ethernet
- Schnittstelle zum Upper Layer (OHA)
- Schnittstelle zum Lower Layer (EDD)

2.2.6 GSY (Generic Sync Module)

- Taktsynchronisation von PROFINET Geräten

2.2.7 LLDP (Link Layer Discovery Protokoll)

- Protokoll zum Austausch von Nachbarschaftsinformationen zur Topologieerkennung
- Zyklisches Senden von LLDP-Paketen mit eigenen Stationsdaten (Chassis-ID, Port-ID etc.).
- Empfang von LLDP-Paketen von anderen Stationen und lokale Speicherung
- Bereitstellung der empfangenen Daten mit zugehöriger Port-ID
- Empfangsüberwachung und, bei Änderung oder Ausfall der LLDP-Daten, Indication an Anwender

2.2.8 MRP (Media Redundancy Protocol

- Medienredundanz von PROFINET Geräten
- Es wird ein MRP Client unterstützt.

2.2.9 OHA (Objekt Handler)

- Auskunftsfunktionen für Applikation
- Generieren von Änderungsmeldungen für die Applikation
- „Applikation“ für DCP-Server und LLDP
- (SNMP Anbindung (Agent) über SOCK)

2.2.10 POF (Poymeric optical fiber)

- Unterstützung für optische Übertragungsmedien POF und PCF (Polymeric Cladded Fiber)

2.2.11 SOCK-lit, SOCK-lsa (Socket Interface)

- SOCK-lit und SOCK-lsa sind alternativ zu verwenden.SOCK-lit enthält ein Interface für die Adaption externer IP Stacks, SOCK-lsa hingegen ist in Kombination mit dem integrierten Interniche IP stack zu verwenden.
- Senden und Empfangen von TCP/IP-Telegrammen
- Asynchrone Schnittstelle zum Upper Layer (CLRPC)

2.2.12 SOCK-Adapter

- Implementierung der Lower Layer Schnittstelle des SOCK Paketes (nichtblockierende Aufrufe, Callback-Schnittstelle)
- Abbildung der UDP Socket Aufrufe auf einen speziellen TCP/IP Stack
- Serialisierung empfangener UDP Telegramme von unterschiedlichen Ports.

2.2.13 IP-Adapter

- Implementierung der Lower Layer Schnittstelle des TCP/IP Stacks
- Senden und Empfangen von Raw Ethernet IP-Telegrammen

2.2.14 SNMP-Adapter

- Schnittstelle des PROFINET IO Stacks zur Anbindung von zusätzlichen externen SNMP Agents wie z.B. ein MIB2 Agent.
- Weiterleiten von SNMP Requests an externe SNMP Agents, die vom PROFINET internen Agent nicht verarbeitet werden können.

2.2.15 MIB2-Adapter

- Lower Layer Schnittstelle Schnittstelle des PROFINET IO Stacks zur Anbindung eines externen MIB2 Agents.
- Bereitstellung von aktuellen Statistikinformationen aus dem ERTEC bzw. ERTEC-EDD (Anzahl gesendeter/empfangener Telegramme, Portlinkstatus,…) zur Verwendung in der MIB2 Datenbasis.

2.2.16 Systemanpassung

- Gemeinsame Implementierung der Systemanpassung der einzelnen Basispakete
- Routing von Betriebssystemdiensten für Speichermanagement, Task- und Timerhandling, Interprozesskommunikation an das OS-Abstraction Layer
- Debug-Logging und Error-Logging
- Implementierung der Tasks und Kommunikation der Tasks untereinander

2.2.17 OS-Abstraction Layer

- Betriebssystem-Abstraktionsschnittstelle für PNIO
- Alle Komponenten von PNIO greifen nie direkt auf einen Betriebssystemaufruf zu, sondern nur über das OS Abstraction Layer
- Bildet alle Anforderungen der Systemanpassung auf einfache Betriebssystemdienste ab, die bei den meisten Echtzeit-Betriebssystemen nahezu 1:1 in einen Serviceaufruf umsetzbar sind.
- Dadurch einfache Anpassbarkeit an ein anderes Echtzeit-Betriebssystem

2.2.18 IOD Control Programm

- API zur Implementierung einer konkreten PROFINET IO-Device Instanz
- bietet eine einfach zu handhabende Schnittstelle zur Applikation
- führt die Verwaltung von IO-spezifischen Datenstrukturen und das Handling von Requestblöcken und asynchronen Confirmations aus

2.3 Sonstige Softwarekomponenten

2.3.1 Operating System

Das Echtzeit-Betriebssystem ist nicht Bestandteil des PROFINET IO-Softwarestacks. Es wird in der Regel über einen Dritthersteller bezogen. Das mitgelieferte Anwenderbeispiel ist zugeschnitten auf die jeweilige Beispielplattform, siehe Kapitel 1.2.

2.3.2 TCP/IP Stack, SNMP MIB 2

Der TCP/IP-Stack ist nur für die ECOS Plattform Bestandteil des PROFINET IO-Softwarestacks. Ansonsten wird dieser über einen Dritthersteller bezogen. Für PROFINET IO wird lediglich die UDP-Funktionalität benötigt. Optional kann SNMP Support (MIB2,..) integriert werden. Dieser ist ebenfalls nicht Bestandteil des Developer Kit und kann in der Regel vom TCP/IP Stack Hersteller bezogen werden.

2.3.3 Board Support Package (BSP)

Das Board Support Package kapselt die hardwarespezifischen Betriebssystemaufrufe für eine vorgegebene Plattform. Ein plattformspezifisches BSP wird in der Regel vom Hersteller des Betriebssystems mitgeliefert. Für die Entwicklungsboards EB400 und EB200 ist ein BSP für das

verwendete Betriebssystem im Lieferumfang des Developer's Kit enthalten. Dieses kann als Beispielvorlage für die Anpassung an eine kundenspezifische Plattform verwendet werden.

2.4 Applikationsbeispiele

Im Development Kit wurden verschiedene Beispiel - Applikationen integriert, um PROFINET optimal an die unterschiedlichen Anforderungen anpassen zu können. Sie zeigen exemplarisch den Umgang mit der Anwenderschnittstelle und können als Template für die eigene Implementierung verwendet werden.

Für den IO-Datenzugriff wurden im PROFINET Stack folgende unterschiedliche Zugriffsmechanismen relaisiert:

- Standard-Interface (**SI**), universell einsetzbar, für einfaches Handling bei RT und IRT.
- Direct Buffer Access Interface (**DBAI**), mit Performancevorteilen bei einer großen Anzahl von Modulen/Submodulen, nutzbar für RT und IRT.

Die im Development Kit enthaltenen Anwenderbeispiele setzen bezüglich IO-Datenzugriff jeweils auf einem der o.g. Interfaces auf. Der Zugriff auf azyklische Dienste wie Anlauf des PROFINET Stacks, Verbindungsaufbau, Schreiben und Lesen von Records oder Alarmbehandlung ist für die o.g. Interfaces identisch. Weitere Informationen zu den Interfaces sowie die azyklischen Dienste finden Sie in im Kapitel 3.3 „Einbinden des PROFINET Stacks in eine Applikation".

Es wird empfohlen, bei der Erstellung einer eigenen Applikation von den mitgelieferten Anwenderbeispielen auszugehen. Anhaltspunkte für die Wahl der geeignetsten Beispielapplikation bietet die folgende Tabelle.

Applikation	Beschreibung	Eigenschaften
App1_Standard	Universelles Beispiel, basierend auf dem Standard-Interface (SI).	✓ empfohlenes Template für die meisten Anwendungen ✓ einfache und schnelle Implementierung ✓ verwendbar für RT und IRT ✓ verwaltet selbständig mehrere ARs ✓ Modul-/Submodul-orientierte Sichtweise der Applikation auf die IO-Daten, d.h. sie muss die AR's oder IOCR's nicht kennen. ✓ Datenkonsistenz automatisch gewährleistet durch gepufferten Zugriff
App2_DBAI	Direct Buffer Access Interface	✓ Performancevorteile gegenüber SI (nur) bei großer Modul-/Submodulanzahl ✓ verwendbar für RT und IRT ✓ IOCR-Sichtweise auf die IO-Daten, d.h. Applikation muss AR's und IOCR's selber verwalten. ✓ Datenkonsistenz automatisch gewährleistet durch gepufferten Zugriff.

2.4.1 Genereller Aufbau der Anwenderbeispiele:

Die generelle Softwarearchitektur des PNIO Stacks wurde bereits in Kapitel 2.1 „Softwarearchitektur" vorgestellt. Der Aufbau der Anwenderbeispiele ist weitgehend identisch. Die Applikation besteht im Wesentlichen aus folgenden Komponenten:

- Main-Task (Einsprungfunktion mainAppl ()) Hier findet zunächst die Initialisierung des PROFINET Stacks statt. Anschliessend wartet die Task in einer Endlosschleife auf Tastatureingaben über die Funktion OsGetChar(). Über ein an die RS232 angeschlossenes Terminal können so typische Kommandos zur Laufzeit ausgeführt werden wie z.B. Senden von Alarmen, Stecken/Ziehen von Modulen im laufenden Betrieb etc.

- IO_Cycle Task. Diese Task führt zyklisch einen IO-Datenaustausch zwischen PROFINET Stack und der Applikation durch. Als zyklischer Trigger dient hier entweder ein vom ERTEC abgeleitetes Ereignis (das sogenannte „TRANS_END“ Event) oder eine feste Wartezeit. Das TRANS_END Event zeigt das Ende der Übertragungsphase der zyklischen Daten bei IRT an, d.h. alle Provider-IOCRs wurden gesendet und alle Consumer-IOCRs empfangen. Es kann aber auch bei RT verwendet werden und signalisiert in diesem Fall, dass alle lokalen Provider-IOCRs gesendet wurden. Wird hingegen nur eine einfache Wartezeit als Trigger verwendet, so ist der Applikationszyklus nicht mit dem Buszyklus synchronisiert.
- Event Handler. Hier werden Callback-Funktionen vom PROFINET Stack aufgerufen, um die Applikation über wichtige Ereignisse wie Verbindungsauf- und Abbau, Lesen und Schreiben von Records, TRANS_END, etc. zu informieren. Die Eventhandler laufen im Kontext des PROFINET IO Stacks.

Das folgende Bild zeigt die implementierte Taskstruktur, sie gilt für alle Anwenderbeispiele. Die Blockpfeile stellen dar, wer die entsprechenden Tasks erzeugt und startet.

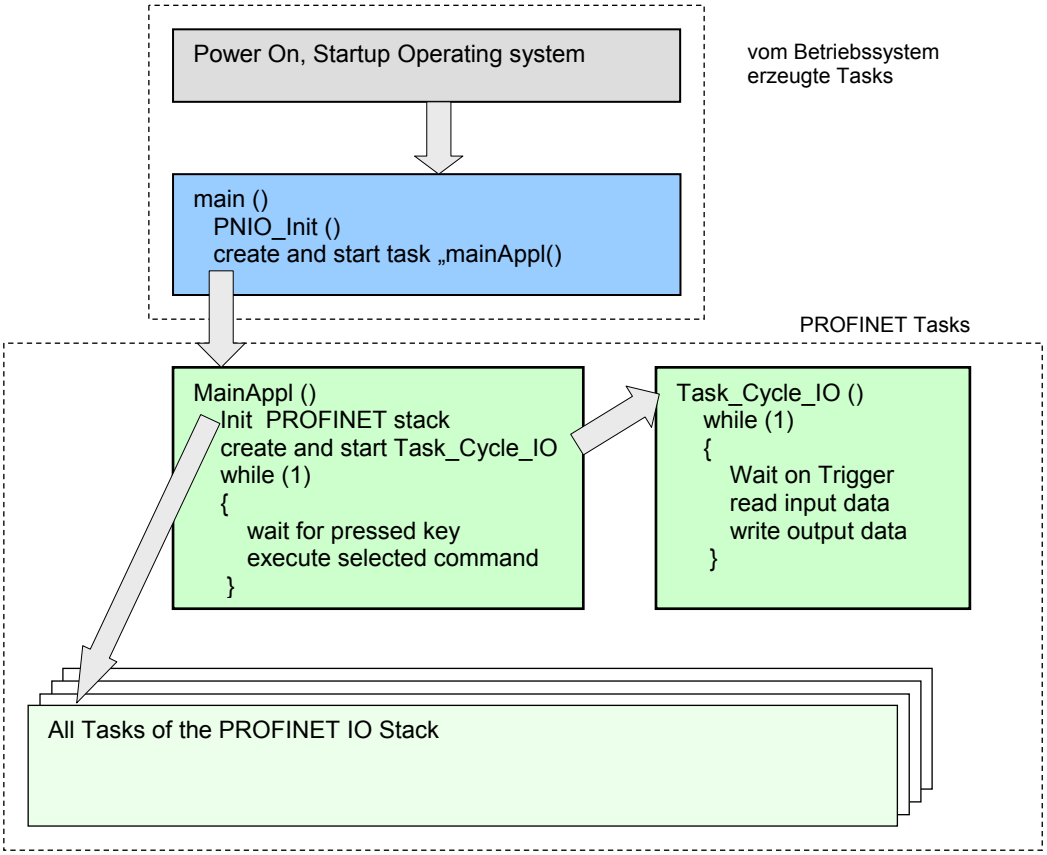


Abbildung 4: Tasks des PROFINET IO Anwenderbeispiels

Verzeichnisstruktur des Anwenderbeispiel-Sourcecodes
Alle Anwenderbeispiele liegen unter einem gemeinsamen Verzeichnis \Application. Für jedes Anwenderbeispiel wurde darin ein eigenes Unterverzeichnis erstellt. Von allen Anwenderbeispielen genutzte Funktionen und Headerfiles liegen in einem gemeinsamen Unterverzeichnis \Common. Eine Beschreibung der Verzeichnisstruktur des kompletten PROFINET-Stacks inclusive der Applikationsbeispiele finden Sie im Kapitel 3.1 „Verzeichnisstruktur des PROFINET IO Quellcodes“.

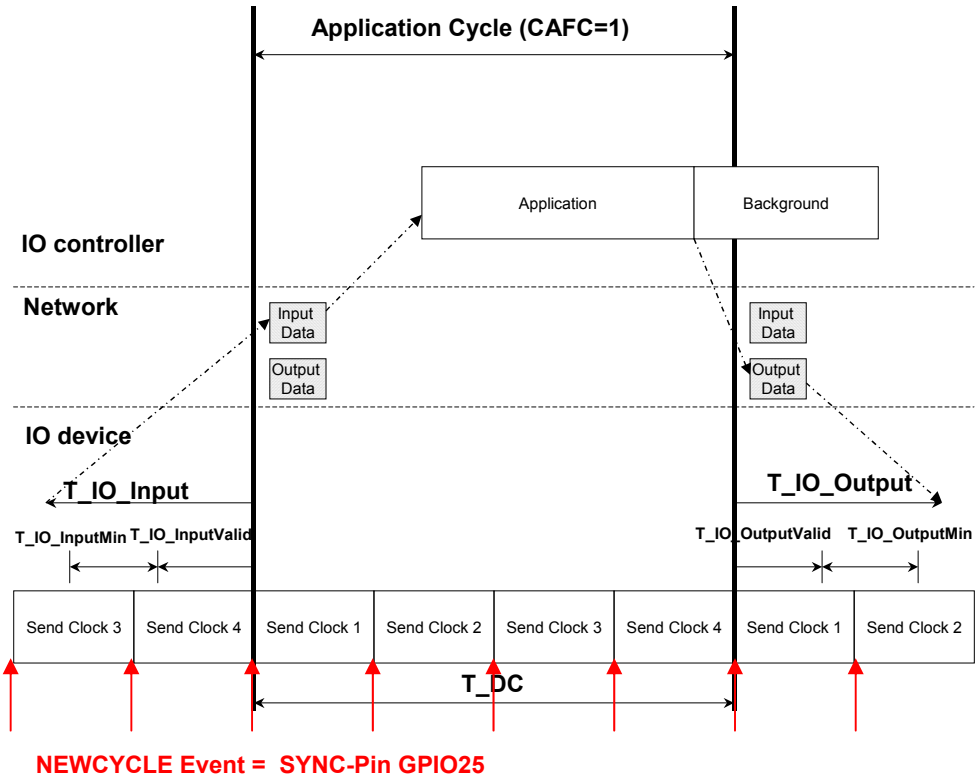
Die Auswahl des zu compilierenden Anwenderbeispiels erfolgt in der Headerdatei \application\common\usriod_cfg.h mittels folgendem Eintrag:

#define EXAMPL_DEV_CONFIG_VERSION 1 // the number 1..n specifies the selected example

Es ist daher einfach möglich, eigene Anwenderbeispiele unter einer neuen Nummer hinzuzufügen. Sie können so auch die mitgelieferten Beispiele kopieren und modifizieren, ohne dabei das Original zu verändern.

2.4.2 taktsynchrone Applikation mit IRT, T_Input und T_Output Auswertung:

IRT kann im Prinzip mit allen IO Modulen durchgeführt werden. Die im Anwenderbeispiel verwendeten Standardmodule (ID = 30h, 31h der Beispiel-GSD Datei) können sowohl für den RT als auch für den IRT Betrieb projektiert werden.
In der GSD Datei können aber optional auch Module definiert werden, die ausschließlich für einen IRT Betrieb projektierbar sind ((ID = 50h, 51h der Beispiel-GSD Datei). Für diese Module wird beim Verbindungsaufbau ein zusätzlicher Parameter-Anlaufrecord vom Controller ans Device übertragen (Record IsochronousModeData, Index 8030h). Dieser enthält zusätzliche Informationen über die IRT Zeitbedingungen wie T_IO_Input, T_IO_Output, T_IO_InputValid, T_IO_OutputValid. Das folgende Bild aus der PROFINET Spezifikation verdeutlicht die Zusammenhänge:



NEWCYCLE Event = SYNC-Pin GPIO25

Mit diesen Informationen ist es möglich, die Zeitpunkte für das Einlesen von Eingängen und das Aktivieren von Ausgängen exakt festzulegen und somit hochdynamische Prozesse zu steuern.
Die Applikationbeispiele werten diesen Record aus und zeigen die o.g. Werte auf der Konsole an. In einer realen Applikation könnte damit z.B. eine externe PLL konfiguriert werden, die über das Sync-Signal am GPIO25 des ERTEC getriggert wird. Die Sync-Signalflanke kommt bei jedem Beginn des Sendezyklus am Bus, also ohne Berücksichtigung eines evt. Reduction Ratio Wertes.
Hinweis: Dieses Verfahren wird in der Praxis für hochdynamische Prozesse mit Buszykluszeiten von 1ms oder darunter eingesetzt. Es ist daher auf einer Einprozessorlösung, d.h. wo neben dem PROFINET Stack auch die Applikation auf dem ARM946 des ERTEC läuft, meist nicht mit zufriedenstellender Genauigkeit bzw. Performance realisierbar. **Es wird empfohlen, in diesen Fällen auf einer Zweiprozessorlösung aufzusetzen**.

Weitere Erläuterungen zu diesem Thema finden Sie auch in der PROFINET Spezifikation /1a/ bzw. /1b/.

2.4.3 App1 Standard Interfaced (SI) based example for RT and IRT

Einsatzbereich:
Dies ist das Standard-Applikationsbeispiel, welches für die meisten Anwendungen verwendet werden kann. Das SI kann für RT und IRT genutzt werden und besitzt für beide Betriebsarten eine einheitliche Anwenderschnittstelle. Die Applikation muss sich um den Aufbau der IOCR im Datentelegramm nicht kümmern, dies wird vom PNIO Stack übernommen. Auch die Verwaltung mehrerer AR's (Funktion Shared Device) führt der Stack selbständig durch. Die Applikation hat weitgehend nur die Sicht auf die IO-Module, unabhängig von der AR. Dadurch ist eine Applikation für das SI einfach zu implementieren.

Beschreibung des IO-Datenzugriffs:
Der IO Datenzugriff erfolgt submodulgranular über Callbackfunktionen. Ein IO-Datenaustausch wird dabei zyklisch durch die Applikation angestoßen mittels eines Aufrufs von PNIO_initiate_data_read() bzw. PNIO_initiate_data_write(). Für die vom Controller stammenden Outputdaten jedes einzelnen Submoduls wird anschließend vom PNIO Stack die Callbackfunktionen PNIO_cbf_data_read() aufgerufen, für die Inputdaten des Devices wird pro Submodul die Callbackfunktionen PNIO_cbf_data_write() aufgerufen. Darin muss die Applikation jeweils die IO-Daten für genau ein Submodul lesen oder schreiben. Dabei werden auch die Providerstati der Outputdaten vom Controller sowie Provider- und Consumerstati der Inputdaten vom Device als Übergabeparameter bzw. Returnwert der Callback übergeben.

2.4.4 App2 DBAI based example for RT and IRT

Einsatzbereich:
Dieses Applikationsbeispiel bietet gegenüber dem Standard-Interface SI evt. Performancevorteile, wenn sehr viele Submodule (z.B. bei einem Proxy) konfiguriert werden. Das DBAI kann wie das SI sowohl für RT wie IRT genutzt werden und besitzt für beide Betriebsarten eine einheitliche Anwenderschnittstelle. Beim Datenzugriff arbeitet die Applikation je auf einem gepufferten Abbbild der IOCR's und greift dort direkt auf IO Daten bzw. Provider-/Consumerstati IOPS/IOCS zu. Die Applikation baut also die IOCR's selber auf, muss dafür aber deren Aufbau kennen und verwalten.

Beschreibung des IO-Datenzugriffs:
Für jeden Zugriff auf eine IOCR ruft die Applikation zunächst die Funktion PNIO_dbai_buf_lock auf. Damit erhält Sie den Pointer auf eine gepufferte und konsistente IOCR. Die Applikation greift nun direkt auf die in der IOCR liegenden Submodul-IO Daten sowie die IOPS/IOCS lesend bzw. schreibend zu, je nach Datenrichtung (Provider-IOCR für Inputdaten, Consumer-IOCR für Outputdaten).
Nach Bearbeitung wird der Puffer durch Aufruf von PNIO_dbai_buf_unlock() wieder freigegeben. Bei einer Provider IOCR (d.h. Inputdaten auf dem Device) wird das neue IOCR-Abbild nun auf dem Bus aktiviert.

3 PROFINET IO-Softwareerstellung

Die folgenden Unterkapitel beschreiben die Verzeichnisstruktur der Software, die Interfaces und die Applikationsbeispiele. Fett dargestellte Module müssen evt. vom Anwender angepasst werden. Nicht fett dargestellte Module sollten vom Anwender nicht oder nur in Ausnahmefällen verändert werden.

3.1 Verzeichnisstruktur des PROFINET IO-Quellcode

Die Aufteilung des Quellcode in verschiedene Unterverzeichnisse orientiert sich an der Softwarestruktur des IO-Stacks für PNIO-Devices. Eine Übersicht gibt die folgende Tabelle.

Verzeichnisse			**Files**	**Beschreibung**
Appl_startup	\		main_xx.c	Einsprungpunkt in die PROFINET IO-Software. Start der IO Main-Task mainAppl()
Application	****	**App1_Standard**	***.c, *.h**	**Standard Applikationsbeispiel für RT und IRT, nutzt das Standard Interface (SI) für den IO Datenzugriff**
	****	**App2_DBAI**	***.c, *.h**	**Direct buffer access (DBAI) Applikationsbeispiel für RT und IRT**
	****	**App_Common**	***.c, *.h**	**Common modules, werden von den verschiedenen Applikationsbeispielen App1…App4 verwendet.**
ACP	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	Header Files des Basispaketes
	\	src \	*.c	Quellcode des Basispaketes
CLRPC	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	, *.h	globale Header Files des Basispaketes
	\	src \	*.c	Quellcode des Basispaketes
CM	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	src \	*.c	Quellcode des Basispaketes
Common	\		*.h	Common headerfiles im PNIO stack, die von verschiedenen Basispaketen genutzt werden
DCP	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	core \	*.c	Quellcode des Basispaketes
EDDi (nur für ERTEC Plattformen)	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	ebx00 \	*.h	Embedded Systemanbindung
	\	src \	*.c	Quellcode des Basispaketes
	\	xx_mem \	*.h	KRAM Speicherverwaltung
EDD_soft (nur NICHT ERTEC Plattformen)	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	src \	*.c	Quellcode des Basispaketes
	\	inc \	*.h	globale Header Files des Basispaketes

GSY	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	core \	*.c	Quellcode des Basispaketes
	\	cpc\	*.c	Quellcode des Basispaketes
IOD	\		*.c, *.h	IOD Control Programm, enthält die Implementierung der Device-Instanz und der Anwenderschnittstelle IODAPI
IOD_ERTEC				ERTEC 200/400 spezifische Komponenten des IOD für die Verwaltung des KRAM im ERTEC
LLPD	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	src \	*.c	Quellcode des Basispaketes
MRP	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	src \	*.c	Quellcode des Basispaketes
NARE	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	src \	*.c	Quellcode des Basispaketes
OHA	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	src \	*.c	Quellcode des Basispaketes
PCPNIO_LSA	\		*.c, *.h	Implementierung der für alle Basispakete gemeinsamen Systemanpassung. Hier sind in der Regel keine Anpassungen durch den Anwender erforderlich
PNIO_API_inc	\		*.h	Headerfiles der PROFINET IO-Applikationsschnittstelle
POF[1]	\	base\	*.c, *.h	Quellcode des Basispaketes
	\	cfg \	*.c, *.h	Call interface of the POF in the system adaptation
	\	dmi \	*.c	Quellcode des Basispaketes
	\	edd \	*.c	Quellcode des Basispaketes
	\	I2c\	*.c	Quellcode des Basispaketes
	\	Inc\	*.c	Headerfiles des Basispaketes
	\	prm\	*.c	Quellcode des Basispaketes
SOCK_lit [2]	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	src \	*.c	Quellcode des Basispaketes, Paket wird verwendet für die Adaption externer IP Stacks
SOCK_lsa (nur für internen Interniche IP Stack) [3]	\	cfg \	*.c, *.h	Systemanpassung des Paketes (unverändert übernehmbar)
	\	inc \	*.h	globale Header Files des Basispaketes
	\	core \	*.c	Quellcode des Basispaketes, Paket wird verwendet beim Einsatz des im PROFINET Stack enthaltenen Interniche IP Stacks.

[1] dieses Paket ist optional und nur für ERTEC 200 mit POF bzw. PCF Medien einsetzbar
[2] dieses Paket ist nicht im DK_ERTEC und SK_ERTEC enthalten
[3] dieses Paket ist nicht im DK_SW enthalten

SysAdapt1	****	**cfg **	***.c, *.h**	**vom Anwender anzupassende Module** **Plattformabhängiger Beispielcode für** **OS – Abstraction Layer** **Upper- und Lower Layer Interface zum TCP/IP Stack** **MIB 2 Adaption** **Bsp Adaption**
	\	inc \	*.h	globale Header Files der Systemanpassung
	\	src \	*.c	Quellcode der Systemanpassung, der i.d.R. unverändert übernommen werden kann
Trace_dk	\		trace_dk.c trace_dk.h	Speicherung von Fehlermeldungen in Umlaufpuffer oder Ausgabe auf die Console auf einem Hyperterminal (angschlossen über die RS232 Schnittstelle) als Debug-Hilfe.
	\	Traceout_con \	*.c, *.h	Ausgaben der Meldungen auf Hyperterminal-Console
	\	Traceout_mem \	*.c, *.h	Ausgaben der Meldungen in einen Umlaufpuffer

Tabelle 2: Struktur und Beschreibung der Verzeichnisse

3.2 Dateien der Beispielapplikationen

3.2.1 Dateien von App1_STANDARD

Modul	Inhalt	Beschreibung
usriod_main.c	Hauptprogramm für RT, IRT Beispiel	Standard Anwenderbeispiel Hauptprogramm für RT und IRT C2, IRT C3. Anlauf des IO-Stacks, Hauptschleife mit Start von Funktionen über Tastatur für eine RT-Applikation
iodapi_event.c	Meldung von Ereignissen an die Applikation	Event Handler für Anwenderbeispiele Standard RT, IRT C2, IRT C3 Enthält Funktionen, die der IO-Stack beim Auftreten von Ereignissen wie Verbindungsauf-/Abbau, Alarmempfang etc. aufruft und damit der Applikation mitteilt. Der Anwender muss diese Funktionen gemäß seinen Anforderungen implementieren.

3.2.2 Dateien von App2_DBAI

Modul	Inhalt	Beschreibung
usriod_main_dbai.c	Hauptprogramm für DBA Beispiel	DBAI-Anwenderbeispiel Hauptprogramm Anlauf des IO-Stacks, Hauptschleife mit Start von Funktionen über Tastatur für eine DBA Applikation (Direkt Buffer Access)
usriod_main_dbai.h	Headerfile	Headerfile zu usriod_main_dbai.c
iodapi_event_dbai.c	Meldung von Ereignissen an die Applikation	Event-Handler, nur für das Anwenderbeispiel in usriod_main_dbai.c.

3.2.3 Dateien von App_common

Modul	Inhalt	Beschreibung
Tcp_IF.c	FW-Download über TCP	TCP basierte Services zum Übertragen einer neuen Firmware
Tcp_ IF.h	FW-Download über TCP	Headerfile zu tcp_if.c
Tcp_Flash_fw.c	FW flashen	Übertragung und Flashen einer neuen Firmware über TCP/IP
usriod_cfg.h	Auswahl und Konfiguration des Beispiels	Über ein Define wird das entsprechende Anwenderbeispiel (RT, IRT Class3, DBA Interface) ausgewählt, außerdem Defines für die Konfiguration des Devices
usriod_diag.c	Applikationsprogram m für Diagnose	Beispiel für die Behandlung von Standard-Kanaldiagnose inclusive Diagnose-Alarm.
usriod_diag.h	Headerfile	Headerfile für usriod_diag.c, enthält u.a. Datenstrukturdefinitionen für die Standard-Kanaldiagnose.
usriod_im_func.c	I&M Funktionen	Beispiel für die Behandlung I&M Funktionen. Implementiert ist IM0, für IM1..3 ist lediglich ein Rahmen vorhanden, der bei Bedarf vom Anwender ausprogrammiert werden kann.
usriod_im_func.h	I&M Funktionen	Headerfile für useriod_im_func.c
usriod_PE.c	PROFIenergy	Anwenderbeispiel zur Behandlung des PROFIenergy Records
usriod_PE.h	PROFIenergy	Headerfile für usriod_PE.c
usriod_utils.c	Utilities	Hilfsprogramme zur Messung der Systemlast für Debug-Zwecke
usriod_utils.h	Headerfile	Headerfile für usriod_utils.c
iodapi_log.c	Logging von Debug- und Fehlermeldungen	Zentrales Melden von Fehlern und Notes an die Applikation, Logging für Debugzwecke oder Anstoss von Fehlerbehandlungsroutinen. Die Funktionen werden vom Stack aufgerufen und müssen vom Anwender gemäß seinen Anforderungen implementiert werden. Es können auch Leerfunktionen implementiert werden.

Modul	Inhalt	Beschreibung
Iodapi_rema.c	Remanente Daten	Übergabe von remanenten Daten (PDEV-Records) vom PNIO Stack an die Applikation zwecks Sicherung im nichtflüchtigen Speicher.
iodapi_event.h	Headerfile	Headerfile für iodapi_event.c. Hier sind normalerweise keine Anpassungen nötig.

Tabelle 3: Dateien der Beispielapplikation

3.2.4 Anwenderschnittstelle

Die Headerfiles zur Beschreibung der Anwenderschnittstelle sind im Unterverzeichnis \PNIO_API_inc abgelegt. Diese Files dürfen nicht verändert werden.

Modul	Inhalt	Beschreibung
pniousrd.h	Makros und Definitionen	Enthält globale Strukturen und Definitionen für die PROFINET IO-Anwenderprogrammierschnittstelle.
pniobase.h	Makros und Definitionen	Enthält Datentypen, Konstanten und Funktionsdeklarationen für die IO-Controller-Funktionalität der IO-Anwenderprogrammierschnittstelle.
pnioerrx.h	Makros und Definitionen	Enthält die Error-Codes
Pnio_trace.h	Makros und Definitionen	Trace Interface (Umlenkung auf LSA-Trace)
Iodapi_rema.h	Makros und Definitionen	Enthält Datentypen, Konstanten der REMA-Schnittstelle.

Tabelle 4: Headerdateien der Anwenderschnittstelle

3.2.5 Anzupassende Module der Betriebssystem-Schnittstelle

Modul	Inhalt	Beschreibung
xxx_os.c	OS Services	Abstraktionsschnittstelle für die Aufrufe von Betriebssystem-Services, xx steht dabei für die Plattform, (z.B: vxw = vxworks,.gh = NET+OS/Greenhill-Tools,...). Hier erfolgt die Abbildung von PNIO Aufrufen auf plattformabhängige Betriebssystemfunktionen.
os_cfg.h	OS Konfiguration	Systemkonfigurationen für PNIO: Einstellung von Taskprioritäten, Festlegung von Systemressourcen für PNIO
compiler.h	Compilerspezifische Definitionen	Festlegung von compilerspezifischen Einstellungen.
compiler_stdlibs.h	Einbindung von Standard-Headerfiles	Festlegung der einzubindenden Standardheaderfiles.

Tabelle 5: Dateien der Betriebssystem-Abstraktionsschnittstelle

3.2.6 Anzupassende Module der Socket- und SNMP Schnittstelle

Modul	Inhalt	Beschreibung
xx_ossock.c	Socket Adaption	Schnittstelle zum TCP/IP Stack. Hier erfolgt die Abbildung von PNIO Aufrufen auf plattformabhängige TCP/IP Stack Aufrufe.
xx_os_snmp.c	MIB2 Adaption	Schnittstelle Anbindung externer MIB-Agents, hier MIB2. Sie enthält (plattformabhängigen) Beispielcode für die Aktivierung der MIB2 Unterstützung sowie den Zugriff auf MIB2 Objekte.

Tabelle 6: Dateien der Socket-Abstraktionsschnittstelle

3.2.7 Anzupassende Module der IP Stack Lower Layer Schnittstelle

Modul	Inhalt	Beschreibung
xx_ipadapt_iniche.c	IP Stack Adaption	Lower Layer Adaption des im PNIO Stack enthaltenen Interniche TCP/IP Stacks

Tabelle 7: Dateien der Socket-Abstraktionsschnittstelle

3.2.8 Module der BSP-Schnittstelle

Modul	Inhalt	Beschreibung
xx_BspAdapt.c	BSP Adaption	Schnittstelle vom IO-Stack zum Ethernet-Controller
xx_BspAdapt_Ertec.c	BSP Adaption	Schnittstelle vom IO-Stack zum ERTEC

Tabelle 8: Dateien zur Anpassung an das Board Support Package

3.2.9 Speicherung remanenter Daten

Modul	Inhalt	Beschreibung
xx_nv_data.c	Speicherung remanenter Daten	Schnittstelle zur Speicherung nichtflüchtiger Daten. Diese Schnittstelle wird nur vom Anwenderbeispiel, nicht aber vom PNIO Stack selber vewendet und ist daher bei Bedarf änderbar.

Tabelle 9: Dateien zur Anpassung an remanente Daten

3.3 Wichtige Randbedingungen für das Einbinden einer Applikation

Bei der Einbindung des PROFINET IO-Stacks in eine Kundenapplikation sind folgende Randbedingungen zu beachten:

1. **Der in einer Callbackfunktion PNIO_cbf_xxxx() ausgeführte Anwendercode sollte möglichst kurz sein**, da alle vom CM eintreffenden Callbackereignisse in einer Message Queue sequentialisiert werden und der Anwendercode im Context der IOD Interfacetask „Pnio_Api“ aufgerufen wird. Eine Callbackfunktion kann daher erst aufgerufen werden, wenn die vorhergehende Callbackfunktion beendet wurde.
2. **In einer Callbackfunktion PNIO_cbf_xxxx() sollten keine API Funktionen PNIO_xxx() aufgerufen werden.** Zulässige Ausnahmen sind PNIO_rec_set_rsp_async(), PNIO_sub_set_state(), PNIO_Log(), PNIO_get_last_error(), PNIO_printf (debugging).
3. **Jede Anwendertask, aus deren Kontext PROFINET IO-Servicefunktionen PNIO_xxxx() aufgerufen werden, muss mit OsCreateThread( ) erzeugt und mit OsStartThread() gestartet werden.** Dabei wird dem Thread automatisch eine Message Queue zugewiesen, die für die Kommunikation mit IOD verwendet wird und ausschließlich dafür reserviert ist.
4. **OsAlloc() ,OsAllocX() OsAllocF() ,OsAllocFX() sollten nur vom IO-Stack verwendet werden, nicht aber von der Applikation.** Der Applikationsstack verwendet verschiedene Speicherpools, sofern diese Funktion vom Betriebssystem unterstützt wird. Eine zusätzliche Nutzung durch die Applikation könnte zu einen „NO MEMORY AVAILABLE“ Fehler führen. Falls diese Pools dennoch von der Applikation mittels OsAlloc oder OsAllocX benutzt werden sollen (z.B. aus Gründen einer plattformunabhängigen Betriebssystemschnittstelle), muss vom Anwender ein extra Pool für die Applikation implementiert werden oder der MEMPOOL_DEFAULT (s. os.h, os_cfg.h) verwendet werden. Der MEMPOOL_DEFAULT ist dabei entsprechend zu vergrößern. Es sind dazu Änderungen os_cfg.h und evt. auch in <Plattform>_os.c und os.h notwendig.
5. **Taskprioritäten** des PROFINET IO-Stacks sind in os_taskprio.h festgelegt. Hier sind plattformabhängig Änderungen notwendig, die Prioritätenhierarchie der PROFINET IO-Tasks untereinander darf dabei aber nicht verändert werden. Applikationstasks sollten nach Möglichkeit unterhalb der Stackprioritäten liegen, . **Bei höher prioren Anwendertasks ist zu beachten, dass das Laufzeitverhalten des Stacks negativ beeinflusst werden kann.**

3.4 Portieren der PROFINET IO-Software auf eine andere Plattform

Der folgende Abschnitt befasst sich mit der Portierung der PROFINET IO-Software auf eine beliebige Plattform. Die Applikation selbst wird dabei nicht betrachtet, da diese in jedem Fall (auf Basis der Beispielapplikation) durch eine Kundenapplikation ersetzt werden muss. Abhängig von der auszutauschenden Komponente sind unterschiedliche Änderungen in der Software durchzuführen. Grundsätzlich können dabei folgende Varianten betrachtet werden:

- Ersatz des Evaluation Boards durch Kundenhardware unter Beibehaltung des Mikrocontrollers und Betriebssystems (einfachster Fall)

- Verwendung anderer Compiler
- Verwendung anderer Mikrocontroller (nicht für ERTEC Plattformen)
- Verwendung anderer Betriebssysteme
- Verwendung anderer UDP/IP - Stacks

3.4.1 Portierung auf Kundenhardware unter Beibehaltung von Mikrocontroller und Betriebssystem

Um die Software auf Ihre eigene Hardwareplattform ohne Änderung des Betriebssystems zu portieren, müssen Sie das Board Support Package an Ihre Hardware anpassen. Sonstige Änderungen im Betriebssysteminterface, UDP Interface, BSP Interface oder Application Interface (siehe Abbildung 1) sind in der Regel nicht notwendig.

3.4.2 Verwendung anderer Compiler/Linker

Die Einstellung des Compilers wird in der Date **auto_platform_select.h** vorgenommen. Diese ist im Unterverzeichnis Pnio_src\Platform für jede Plattform getrennt vorhanden, wobei über Selection des Includepfades jeweils nur eine auto_platform_select.h included wird. Dort existieren zur Zeit Defines für ausgewählte Standard-Plattformen sowie eine generische Plattform.

3.4.2.1 Auswahl der Toolkette

Von den folgenden Definitionen muss genau eines verwendet werden (nicht benötigte Defines von TOOL_CHAIN_xxxx sind auszukommentieren).

```
 #define TOOL_CHAIN_GNU_ECOS              1
// #define TOOL_CHAIN_GNU_VXW              1
// #define TOOL_CHAIN_GH_NETOS             1
// #define TOOL_CHAIN_GENERIC_32BIT        1
```

Wenn Sie nicht die voreingestellte Toolkette verwenden, können sie Ihre eigenen compilerspezifischen Definitionen am einfachsten unter TOOL_CHAIN_GENERIC_32BIT eintragen. In der Regel müssen Sie dazu lediglich in der Datei compiler.h Anpassungen vornehmen.
Wenn Sie aber weder TOOL_CHAIN_GNU_VXW, TOOL_CHAIN_GH_NETOS etc. noch TOOL_CHAIN_GENERIC_32BIT verwenden wollen, können Sie auch einen eigenen Compilerschalter in compiler.h definieren. In diesem Fall müssen an mehreren Stellen im Code Ergänzungen gemacht werden. Suchen Sie daher alle Stellen im Quellcode, wo eines der o.g. Defines verwendet wird und fügen Sie dort ihren eigenen compilerschalter-abhängigen Code ein.

3.4.2.2 Big- oder little Endian

```
#define PNIO_BIG_ENDIAN      0      //  Little Endian, z.B. NETOS Plattform
#define PNIO_BIG_ENDIAN      1      //  Big Endian, z.B. ERTEC Plattform
```

3.4.2.3 Data Alignment Anforderungen

```
#define ALIGNMENT_ORDER_WORD   2  //  Wort Zugriff darf nur auf word aligned erfolgen
#define ALIGNMENT_ORDER_WORD   1  //  Wort Zugriff auch auf ungerade Adressen erlaubt

#define ALIGNMENT_ORDER_DWORD  4 //  DWORD Zugriff nur dword-aligned möglich
#define ALIGNMENT_ORDER_DWORD  1 //  DWORD Zugriff auch auf ungerade Adressen  möglich
```

Normalerweise müssten die oben angegebenen Defautl Einstellungen für ALIGNMENT_ORDER_xx auf nahezu jeder Plattform möglich sein.

3.4.2.4 Datenverarbeitungs-Breite

Die Software wurde bisher ausschließlich auf Controllern mit 32 Bit Datenverarbeitung portiert. Alle Daten- und Adresspointer sind ebenfalls 32 Bit breit.

3.4.2.5 Speichermanagement

Für den IO-Stack existieren keine besonderen Vorgaben für das Speichermanagement. Unter Berücksichtigung der Alignment- und Speichermanagement-Anforderungen der Hardware ist der IO-Stack damit frei lokatierbar. Da einige LSA Layer ein 8 Byte Alignment voraussetzen, wurde in der Beispielsystemanpassung ein 8 Byte Alignment für OsAllocX und OsFreeX implementiert. Damit ist diese Anforderung auch für Betriebssysteme erreicht, die selber diese Anforderung nicht unterstützen.

Für das dynamische Speichermanagement wurden in der Systemanpassung in os.h unterschiedliche Speicherpools definiert, welche in der Software beim Allokieren von Speicher über folgende Defines referenziert werden:

```
#define MEMPOOL_DEFAULT     0     // maybe cached
#define MEMPOOL_FAST        1     // maybe cached
#define MEMPOOL_RX_TX_BUF   2     // uncached
#define MEMPOOL_CACHED      3     // maybe cached
#define MEMPOOL_UNCACHED    4     // uncached
```

Sofern das verwendete Betriebssystem bzw. die Hardware dies unterstützt, können die verschiedenen Pools auf verschiedene physikalische Speicherbereiche abgebildet werden. Dazu muss eine entsprechende Implementierung der Funktionen OsAllocX() und OSFreeX() in xx_OS.C erfolgen.

Durch die Aufteilung kann eine Laufzeitoptimierung erreicht werden aufgrund

- Aufteilung in Speicher mit schneller und weniger schneller Zugriffszeit
- geringere Fragmentierung und schnellere Garbage Collection durch kleinere Speicherpools

ERTEC spezifische Einstellungen
Für den Austausch von NRT Telegrammen muss ein uncached Memory für den ERTEC vorgesehen werden. Im Beispiel wurden dazu 8 MByte vorgesehen, die nicht vom Betriebssystem verwaltet werden. Anfangsadresse und Größe werden in der Datei ertec_cfg.h eingestellt über folgende defines:

#define ERTEC_SDRAM_BASE_ADDR	Startadresse des SDRAM
#define ERTEC_UNCACHED_RAM_OFFSET	Offset der uncached Region im SDRAM
#define ERTEC_UNCACHED_RAM_SIZE	Größe des uncached Speicherbereiches

3.4.3 Verwendung anderer Mikrocontroller

Die PROFINET IO-Software kann im Prinzip auf jeden beliebigen 32 Bit Mikrocontroller portiert werden, sofern dafür ein entsprechender Compiler verfügbar ist. Die meisten Abhängigkeiten sind damit bereits durch den Compiler (siehe Kapitel 3.4.2 "Verwendung anderer Compiler") abgedeckt. Weitere Besonderheiten des Controllers müssen abhängig vom konkreten Fall bewertet werden.

3.4.4 Verwendung anderer Betriebssysteme

Die Taskprioritäten der PROFINET IO-Tasks können in **os_cfg.h** eingestellt werden. Dabei sollte die Reihenfolge aufsteigender Prioritäten nicht verändert werden. Es ist zu beachten, dass bei manchen Betriebssystemen der niedrigste Zahlenwert (Prio = 0) die höchste Priorität darstellt, bei anderen ist es umgekehrt.

Die Datei xx_OS.C beinhaltet eine Betriebssystem-Abstraktionsschnittstelle, welche vom Anwender an das verwendete Betriebssystem anzupassen ist. Die Funktionen der Betriebssystemschnittstelle sind detailliert in Kapitel **4.3** "Schnittstelle zum Betriebssystem" beschrieben.

3.4.5 Verwendung anderer TCP/IP Stacks

Das CLRPC Paket innerhalb des PROFINET IO-Stacks setzt auf einer UDP Socketschnittstelle auf. Während die Socketschnittstellen unterschiedlicher TCP/IP Stacks in den synchronen Standardfunktionen wie sendto() und recvfrom() relativ ähnlich sind, gibt es bisweilen größere Unterschiede bei der Konfigurierung (Options-Einstellung) und bei der Implementierung asynchroner Funktionen (zum Empfang auf mehreren Ports gleichzeitig).
Aus diesem Grunde wurde in OsSocket.h eine Socket-Abstraktionsschnittstelle (UDP-Interface, siehe Bild in Kapitel 2.1 "Softwarearchitektur") definiert und in xx_OsSock.C eine Beispielimplementierung integriert. Zur Anpassung an andere TCP/IP Stacks sind somit lediglich Anpassungen in xx_ossocket.c durchzuführen.
Die Funktionen des UDP-Interfaces sind in Kapitel 4.4 beschrieben.

3.5 Typischer Ablauf eines IO-Device-Anwenderprogramms

Überblick

Der typische Ablauf eines IO-Device Anwenderprogramms gliedert sich in 3 Phasen.

- Initialisierungsphase
- Produktivbetrieb
- Abschlussphase

IO Datenzugriff:

- Asynchroner IO-Datenzugriff mit Konsistenzmechanismen für RT
- Synchroner oder Asynchroner IO-Datenzugriff mit Konsistenzmechanismen für IRT Class 2
- Synchroner IO-Datenzugriff ohne Konsistenzmechanismen, nur für IRT Class3

Lesen Sie die nachfolgenden Details.

3.5.1 Initialisierungsphase

Beschreibung

Die Initialisierungsphase wird in mehrere Schritte gegliedert. Dabei ist zu unterscheiden zwischen Funktionsaufrufen die von dem IO-Device-Anwenderprogramm getätigt werden und Callback-Aufrufen die von der IO-Schnittstelle getätigt werden.

Schritt	Aktion	Zweck
Systemanlauf		
0	System läuft an und ruft seine Main()-Funktion auf	
1	PNIO_init( )	PNIO_init initialisiert u.a. das OS-Interface und muss daher einmalig vor allen anderen PNIO-Funktionen aufgerufen werden.
2	OsCreateThread(MainAppl) OsCreateMsgQueue OsStartThread	Mit OsCreateThread wird die erste PNIO-Anwendertask MainAppl() gestartet, die eine OS-Message Queue benötigt, um mit dem PNIO-Stack zu kommunizieren. **Alle weiteren PNIO-Api-Aufrufe müssen aus MainAppl() oder einer weiteren ebenfalls mit OsCreateThread() erzeugten Task gemacht werden.**
PROFINET-Stack Anlauf		
3	PNIO_set_eth_par( )	Vorgabe der IP-Suite (vor Anlauf des PNIO-Stacks bekanntgegeben werden)
4	PNIO_setup( )	Anlauf des PNIO Stacks, Start aller PNIO Tasks und Belegen der Ressourcen.
Instanzen für Device's, Api's, Module, Submodule erzeugen		
5	PNIO_device_open( )	Erzeugen einer Device-Instanz
6	PNIO_api_add( )	API (Application Process Identifier) anmelden.
7	PNIO_mod_plug( )	Stecken der PROFINET IO-Module des IO-Device
8	PNIO_sub_plug( )	Stecken der PROFINET IO-Submodule des IO-Devices
9	PNIO_set_dev_state( )	Setzen des Devices in den Zustand OPERATE
10	PNIO_netcom_enable( )	Aktivieren der Netzwerkkommunikation (Dummy Funktion bei ERTEC)
11	PNIO_device_start( )	IO-Device aktivieren, ab diesem Zeitpunkt ist das IO-Device für den IO-Controller sichtbar.
Warten auf Verbindungsaufbau vom Controller		
12	Warten auf Aufruf der PNIO_CBF_AR_ CHECK_IND( )-Callback-Funktion.	Dieser Callback wird von der IO-Schnittstelle aufgerufen, sobald ein IO-Controller eine Verbindung zu dem IO-Device-Anwenderprogramm aufbaut und seine erwartete Konfiguration für das IO-Device-Anwenderprogramm übermittelt hat. Durch Aufruf dieses Callback werden dem IO-Device-Anwenderprogramm Application-Relation-globale Parameter zur Prüfung übergeben. Das IO-Device-Anwenderprogramm kann bei Fehlern in den Application-Relation-globalen Parameter den Application-Relation-Aufbau abbrechen; siehe hierzu PNIO_dev_ar_abort( ).

Schritt	Aktion	Zweck
13	Reagieren auf eventuellen Aufruf der PNIO_CBF_CHECK_ IND( )-Callback-Funktion. (optional)	Dieser Callback wird für jedes Submodul von der IO-Schnittstelle aufgerufen, deren Konfiguration nicht mit der des IO-Controllers übereinstimmt. Damit wird dem Anwenderprogramm die Möglichkeit gegeben den Submodulausbau zu ändern bzw. die Submodule als kompatibel oder falsch zu markieren. Bei Übereinstimmung der Konfigurationen wird dieser Callback nicht aufgerufen.
14	Warten auf Aufruf der PNIO_CBF_AR_ INFO_IND( )-Callback-Funktion.	Dieser Callback wird von der IO-Schnittstelle aufgerufen sobald die Application Relation zum IO-Controller aufgebaut ist. Durch Aufruf dieses Callback wird dem IO-Device-Anwenderprogramm mitgeteilt welche Module und Submodule in dieser Application Relation betrieben werden und um welche RT Classe (1/2/3) es sich handelt.
Parametrieren der Submodule		
15	Reagieren auf Aufruf der PNIO_CBF_REC_ WRITE( )-Callback-Funktion.	Dieser Callback wird von der IO-Schnittstelle aufgerufen, für den Fall, dass ein IO-Controller einen Parametrierdatensatz für ein Submodul übermittelt. Durch Aufruf dieses Callback werden eventuelle Parametrierdaten pro Submodul an die IO-Device-Anwenderprogramm übermittelt.
16	PNIO_sub_set_state() (optional PNIO_cbf_sub_get_state)	Nachdem ein Modul parametriert wurde, teilt die Applikation dem Stack mit, ob die Parametrierung erfolgreich war und das Modul korrekt hochgelaufen ist („ready“ oder „fehler“). Alternative: PNIO_cbf_sub_get_state()
17	Warten auf Aufruf der PNIO_CBF_PARAM_ END_IND( )-Callback-Funktion.	Dieser Callback wird von der IO-Schnittstelle aufgerufen, sobald ein IO-Controller das Ende der Parametrierphase signalisiert. Falls ein Modul bis zu diesem Zeitpunkt noch nicht als „ready“ gemeldet wurde, ruft der Stack für dieses Modul PNIO_cbf_sub_get_state() auf, um den aktuellen Status zu erfragen.
18	PNIO_initiate_data_ write( )	Mit diesem Aufruf veranlasst das Anwenderprogramm das Aufrufen der PNIO_CBF_DATA_WRITE( )-Callbacks, damit das IO-Device-Anwenderprogramm die Eingangsdaten der funktionsfähigen Submodule initialisieren und die lokalen Status auf „GOOD“ setzen kann. Für alle nicht funktionsfähigen Submodule müssen die lokalen Status auf „BAD“ gesetzt werden. **Achtung** Die PROFINET IO-Norm verlangt, dass alle Ausgangsdaten aller funktionsfähigen Submodule vor dem Versenden des Application Ready (Schritt 20) auf gültige Werte gesetzt werden und der Lokale Status jeweils auf GOOD gesetzt wird.

Schritt	Aktion	Zweck
19	PNIO_initiate_data_ read( )	Mit diesem Aufruf veranlasst das Anwenderprogramm das Aufrufen des PNIO_CBF_DATA_READ-Callback, damit es für alle Ausgangsdaten der funktionsfähigen Submodule den lokalen Status auf „GOOD“ setzen kann. Für alle nicht funktionsfähigen Submodule müssen die lokalen Status auf „BAD“ gesetzt werden. **Achtung** Die PROFINET IO-Norm verlangt, dass alle funktionsfähigen Submodule vor dem Versenden des Application Ready (Schritt 20) die Lokalen Status auf GOOD gesetzt werden.
20	PNIO_set_appl_ state_ready( )	Das IO-Device-Anwenderprogramm meldet hiermit dem IO-Controller eine Liste der nicht funktionsfähigen Submodule und die Bereitschaft, in den Datenaustausch zu gehen. Für die Module, für die die Applikation noch nicht den Anlaufzustand („ready“ oder „Fehler“) mittels PNIO_sub_set_state() gemeldet hat, wird der Zustand vom Stack mittels PNIO_cbf_sub_get_state() explizit abgefragt. Der Stack benötigt spätestens jetzt diese Information, um die BadList im ApplicationReady Telegramm ausfüllen zu können.
21	Warten auf Aufruf der PNIO_CBF_AR_ INDATA_IND( )-Callback-Funktion.	Dieser Callback wird von der IO-Schnittstelle aufgerufen, sobald ein IO-Controller zum ersten mal die IO-Daten übermittelt hat. Signalisierung des Beginns des zyklischen Datenaustauschs.

Tabelle 10: Aufzurufende API Funktionen in der Anlaufphase

3.5.2 Produktivbetrieb

Überblick

Im Produktivbetrieb findet der Datenverkehr mit dem IO-Controller statt. Im Einzelnen sind dies:

IO-Daten lesen/schreiben

Alarme senden und deren Quittungen empfangen

Datensatz lesen/schreiben

Die Details des Datenverkehrs werden nachfolgend erläutert.

IO-Daten bei RT und IRT lesen

Das Lesen der IO-Daten (Ausgangsdaten aus Sicht des IO-Controllers) erfolgt in zwei Schritten unter Verwendung der Konsistenzsteuermechanismen des ERTEC400/200:

Schritt	Aktion	Zweck
1	Warten auf das PNIO_CP_CBE_TRANS_END_IND-Ereignis) (optional, nur für ERTEC)	Mit diesem Ereignis signalisiert die IO Schnittstelle dem Device Anwenderprogramm den Abschluss der IO Datenübertragung am Bus. Die zugehörige Callbackfunktion wird einmalig von der Applikation mittels PNIO_CP_cbf_register_Cbf angemeldet.
2	PNIO_initiate_data_read( )	Lesewunsch an die IO-Schnittstelle melden. Dies veranlasst die IO-Schnittstelle Schritt 2 auszuführen. Dieser Aufruf kehrt erst zurück nachdem alle Submodule in Schritt 2 abgearbeitet wurden.
3	PNIO_CBF_DATA_READ( )	Die IO-Schnittstelle ruft für jedes Submodul mit Ausgangsdaten diesen Callback auf und übergibt ihm, unter anderem, den Pointer auf einen Datenpuffer mit den vom IO-Controller empfangenen Ausgangsdaten.

Tabelle 11: Aufzurufende API Funktionen zum Lesen von IO-Daten bei RT und IRT

IO-Daten bei RT und IRT schreiben

Das Schreiben der IO-Daten (Eingangsdaten aus Sicht des IO-Controllers) erfolgt in zwei Schritten:

Schritt	Aktion	Zweck
1	Warten auf das PNIO_CP_CBE_TRANS_END_IND-Ereignis) (optional, nur für ERTEC)	Mit diesem Ereignis signalisiert die IO Schnittstelle dem Device Anwenderprogramm den Abschluss der IO Datenübertragung am Bus. Die zugehörige Callbackfunktion wird einmalig von der Applikation mittels PNIO_CP_cbf_register_Cbf angemeldet.
2	PNIO_initiate_data_write( )	Schreibwunsch an die IO-Schnittstelle melden. Dies veranlasst die IO-Schnittstelle Schritt 2 auszuführen. Dieser Aufruf kehrt erst zurück nachdem alle Submodule in Schritt 2 abgearbeitet wurden.
3	PNIO_CBF_DATA_WRITE( )	Die IO-Schnittstelle ruft für jedes Submodul mit Eingangsdaten diesen Callback auf, und übergibt ihr unter anderem den Pointer auf einen Datenpuffer, in den die Eingangsdaten kopiert werden sollen, die an den IO-Controller gesendet werden sollen.

Tabelle 12: Aufzurufende API Funktionen zum Schreiben von IO-Daten bei RT und IRT

Datensatz-lesen/schreiben-Auftrag des IO-Controllers bearbeiten

Bearbeiten eines Datensatz-lesen-Auftrags
Sobald die IO-Schnittstelle ein Datensatzleseauftrag vom IO-Controller empfängt, ruft sie die vom IO-Device-Anwenderprogramm angemeldete Callback Funktion PNIO_CBF_REC_READ( ) auf. Die Applikation kann die Recorddaten innerhalb der Callbackfunktion bereitstellen oder asynchron zu einem späteren Zeitpunkt liefern.

a) synchrones Lesen von Recorddaten:

Schritt	Aktion	Zweck
1	PNIO_cbf_rec_read()	Record-Leserequest an die Applikation. Die Applikation stellt die Daten innerhalb der Callbackfunktion bereit. Mit Beenden der Callback ist der Request aus Applikationssicht abgeschlossen.

Tabelle 13: Aufzurufende API Funktionen zum synchronen Lesen von Recorddaten

b) asynchrones Lesen von Recorddaten:

Schritt	Aktion	Zweck
1	PNIO_cbf_rec_read()	Record-Leserequest vom Stack an die Applikation. Die Applikation möchte die Daten asynchron bereitstellen
2	PNIO_rec_set_rsp_async()	Die Applikation teilt dem Stack mit, dass die Antwort asynchron erfolgt. PNIO_rec_set_rsp_async muss innerhalb der Callbackfunktion aufgerufen werden.
3	PNIO_rec_read_rsp ()	Die Applikation übergibt asynchron die geforderten Recorddaten an den Stack. PNIO_rec_read_rsp kann von jeder beliebigen Anwendertask aus aufgerufen werden.

Tabelle 14: Aufzurufende API Funktionen zum asynchronen Lesen von Recorddaten

Bearbeiten eines Datensatz-schreiben-Auftrags
Sobald die IO-Schnittstelle einen Datenssatzschreibenauftrag vom IO-Controller empfängt, ruft sie den vom IO-Device-Anwenderprogramm angemeldete Callback Funktion PNIO_CBF_REC_WRITE( ) auf. Die Applikation kann den Abschluss des Schreibvorgangs synchron mit Verlassen der Callbackfunktion an den Stack melden oder dies asynchron zu einem späteren Zeitpunkt durchführen.

a) synchrones Schreiben von Recorddaten:

Schritt	Aktion	Zweck
1	PNIO_cbf_rec_write()	Record-Schreibrequest an die Applikation. Die Applikation bearbeitet die Daten innerhalb der Callbackfunktion. Mit Beenden der Callback ist der Request aus Applikationssicht abgeschlossen.

Tabelle 15: Aufzurufende API Funktionen zum synchronen Schreiben von Recorddaten

b) asynchrones Schreiben von Recorddaten:

Schritt	Aktion	Zweck
1	PNIO_cbf_rec_write()	Record-Schreibrequest vom Stack an die Applikation. Die Applikation möchte die Antwort asynchron bereitstellen
2	PNIO_rec_set_rsp_async()	Die Applikation teilt dem Stack mit, dass die Antwort asynchron erfolgt. PNIO_rec_set_rsp_async muss innerhalb der Callbackfunktion aufgerufen werden.
3	PNIO_rec_write_rsp ()	Die Applikation meldet asynchron den Abschluss des Requests incl. Statusinfo an den Stack. PNIO_rec_write_rsp kann von jeder beliebigen Anwendertask aus aufgerufen werden.

Tabelle 16: Aufzurufende API Funktionen zum asynchronen Schreiben von Recorddaten

Alarme senden und Alarmquittierung empfangen

Für jeden versendeten Alarm erhält das IO-Device-Anwenderprogramm eine Alarmquittung indem die IO-Schnittstelle den PNIO_CBF_ASYNC_REQ_DONE( )-Callback aufruft. Die Zuordnung welche Quittung zu welchem Alarm gehört, geschieht über das User Handle. Dieses User Handle vergibt das IO-Device-Anwenderprogramm beim Senden des Alarms. Das PNIO_CBF_ASYNC_REQ_DONE( )-Callback erhält den User Handle als Übergabeparameter.

Bei PROFINET IO existiert eine Wettlaufsituation (Race Condition) im Zusammenhang mit Alarmen und dem Wiederaufbau der Application Relation zwischen IO-Controller und IO-Device. Dies kann am besten durch folgendes Beispiel verdeutlicht werden:

Angenommen es besteht eine Application Relation zwischen IO-Device und IO-Controller und das IO-Device möchte einen Diagnose-Alarm an den IO-Controller senden, weil ein Submodul wegen eines Wackelkontaktes kurzzeitig fehlerhaft ist. Noch bevor dieser Alarm über das Netz übertragen werden kann, baut der IO-Controller, z. B. wegen eines Neustarts, die bestehende Application Relation ab und baut sie anschließend erneut auf. Beim Aufbau der Application Relation mit dem IO-Controller könnte das Submodul aber bereits wieder voll funktionsfähig sein. Wenn jetzt der Alarm, der für die zuvor bestehende Application Relation an den IO-Controller gelangt, würde der IO-Controller fälschlicherweise davon ausgehen, dass das vorhandene Submodul fehlerhaft ist.

PROFINET IO löst solche Probleme durch Einführung des Session Key. Der Session Key ist ein Wert der bei jedem Aufbau der Application Relation erhöht wird. Ein Überlauf vom höchsten Wert zum niedrigsten Wert wird toleriert. Dieser Session Key muss bei jedem Alarm mitgegeben werden. Dadurch ist der IO-Controller in der Lage veraltete Alarme zu verwerfen.

Aus diesem Grund muss das IO-Device-Anwenderprogramm z. B. beim PNIO_CBF_CHECK_IND( )-Callback den übergebenen aktuellen Session Key speichern, um ihn beim Senden eines Alarmes (und auch Datensätze) anzugeben.

Hinweis

Alarme dürfen erst nach Aufruf PNIO_set_appl_ state_ready( ) von dem IO-Device-Anwenderprogramm versendet werden.

Callback-Ereignisse bei Verbindungsauf- und -abbau beim IO-Device

Beim Verbindungsaufbau werden von der IO-Schnittstelle Informationen per Callback zur Verfügung gestellt.

Die folgende Tabelle nennt die „gemeldeten Informationen“ und den zugehörigen Callback-Namen.

Callback-Namen	Gemeldete Information
PNIO_CBF_AR_CHECK_IND	Application-Relation-Information
PNIO_CBF_CHECK_IND	Check Indication – Tatsächlicher Modulausbau stimmt nicht mit dem projektieren Ausbau überein.
PNIO_CBF_AR_INFO_IND	Soll-Ausbau des IO-Device und RT-Klasse (RT class 1/2/3)
PNIO_CBF_PARAM_END_IND	Ende der Parametrierung durch IO-Controller
PNIO_CBF_AR_INDATA_IND	Application-Relation-InData – Erster Datenaustausch mit dem IO-Controller erfolgt.
PNIO_CBF_AR_ABORT_IND	Abort Event – Verbindungsabbruch, bevor bereits ein Datenaustausch stattgefunden hat.
PNIO_CBF_AR_OFFLINE_IND	Offline Event – Verbindungsabbruch, nach dem bereits ein Datenaustausch stattgefunden hat.
PNIO_CBF_APDU_STATUS_IND	Status des IO-Controllers

Tabelle 17: API Callback Funktionen bei Verbindungsauf- und Abbau

Hinweis

Bei diesen Diensten handelt es sich um passive Funktionalität. Alle diese Callbacks werden von der PROFINET-Bibliothek in der Regel als Reaktion auf PROFINET IO-Controller-Aktionen aufgerufen.

3.5.3 Abschlussphase

Beschreibung

Die Abschlussphase des IO-Device wird in drei Schritte gegliedert:

Schritt	Aktion	Zweck
1	PNIO_device_stop( )	IO-Device wird deakiviert und ist somit für den IO-Controller nicht mehr erreichbar. Dies führt zum Aufruf der PNIO_CBF_AR_OFFLINE_IND( ). Für den Fall, dass Datenaustausch zwischen IO-Controller und IO-Device stattfindet (PNIO_CBF_AR_INDATA_IND-Callback wurde von der Schnittstelle aufgerufen), wird von der IO-Schnittstelle PNIO_CBF_AR_abort_IND-Callback aufgerufen. Für den Fall, das noch kein Datenaustausch zwischen IO-Controller und IO-Device stattgefunden hat (PNIO_CBF_AR_INDATA_IND-Callback wurde von der Schnittstelle noch nicht aufgerufen) wird von der IO-Schnittstelle der PNIO_CBF_AR_OFFLINE_IND-Callback aufgerufen.
2	PNIO_mod_pull( )	Alle Module inklusive deren Submodule werden an der IO-Schnittstelle entfernt.
3	PNIO_api_remove( )	nicht implementiert.
3	PNIO_device_close( )	nicht implementiert.

Tabelle 18: Aufzurufende API Funktionen in der Abschlussphase

3.6 Prinzipieller Datenverkehr der IO-Device Anwenderschnittstelle

Beschreibung

Die IO-Device-Funktionen kennen grundsätzlich zwei Mechanismen des Datenverkehrs:

Zyklischer IO-Datenverkehr

- IO-Daten schreiben
- IO-Daten lesen

Der IO-Datenverkehr wird außerdem mit Statusinformationen begleitet.
Diese Besonderheit wird im folgenden Kapitel beschrieben.

Azyklischer Datenverkehr

- Schreiben und lesen von Datensätzen
- Alarme senden und deren Quittungen empfangen

Weitere Erläuterungen hierzu finden Sie im Kapitel 3.11, „Callback-Mechanismus“.

3.7 Zyklischer IO-Datenverkehr der IO-Device Anwenderschnittstelle

Grundsätzliche Arbeitsweise

Sowohl beim Schreiben als auch beim Lesen der IO-Daten durch das IO-Device-Anwenderprogramm wird immer nur das lokale Prozessabbild auf dem CP beschrieben bzw. ausgelesen. Es werden dabei keinerlei Daten über das Netz gesendet.

Den Datenverkehr zwischen dem lokalem Prozessabbild und dem IO-Controller wickeln die unterlagerten IO-Device-Funktionen bzw. die Hardware selbständig und zyklisch ab.
Die Details dieses Datenverkehrs werden in der Projektierung festgelegt.

Hinweis 1

Es ist nicht notwendig, dass das IO-Device-Anwenderprogramm in jedem Buszyklus die IO-Daten schreibt oder liest.

Hinweis 2

Es ist nicht notwendig, dass das IO-Device-Anwenderprogramm öfter auf das Prozessabbild zugreift, als der projektierte Zyklus.

Hinweis 3 (nur für ERTEC Plattformen)

Nur für ERTEC400:

Beim ERTEC 400 können maximal 254 Byte auf einmal konsistent übertragen werden. Falls die gesamten IO Daten einer IOCR grösser sind, so werden diese vom Stack in bis zu 254 Byte grosse Segmente unterteilt, wobei die Segmentgrenzen nur an Submodulgrenzen verlaufen. Damit ist zusätzlich sichergestellt, dass bei IO Datenmengen grösser 254 Byte die Daten innerhalb eines Submoduls immer konsistent sind.

Nur für ERTEC200: Die Gesamtgröße einer IOCR, d.h. die Summe aller IO Daten dieser IOCR inclusive der Qualifier IOPS und IOCS ist auf 256 Byte begrenzt.

IO-Daten und Datenstatus

Die Qualität der IO-Daten wird durch den Datenstatus beschrieben, der den Wert GOOD oder BAD annehmen kann.

Sowohl beim Schreiben als auch beim Lesen werden jeweils zwei Datenstatus ausgetauscht:

Lokaler Status (Status Ihres IO-Device-Anwenderprogramms)

Remoter Status (Status des Kommunikationspartners)

3.7.1 Zyklisches Schreiben mit Status

Ablauf des Schreibvorgangs in das Prozessabbild

Das IO-Device-Anwenderprogramm initiiert den Schreibvorgang durch das Aufrufen der Funktion PNIO_initiate_data_write( ). Daraufhin ruft die IO-Schnittstelle für jedes vom IO-Controller in Betrieb genommene Submodul den PNIO_CBF_DATA_WRITE( )-Callback auf. Mit Aufruf der Callback-Funktion PNIO_CBF_DATA_WRITE( ) werden die Eingabedaten und der zugehörige **lokale Datenstatus** dieser Daten für den Kommunikationspartner in das lokale Prozessabbild geschrieben.
Zusätzlich wird vom Kommunikationspartner der **remote Status** dieser Eingabedaten aus dem lokalen Prozessabbild gelesen.
Es sind beim zyklischen Schreiben also zwei Stati im Spiel.

Kommunikationsrichtung	Werte
zum Kommunikationspartner	● Eingabedaten ● Lokaler Status
vom Kommunikationspartner	Remoter Status

Lokaler Status

Im Normalfall wird der lokale Status vom IO-Device-Anwenderprogramm auf GOOD gesetzt.

Wenn die Eingangsdaten fehlerhaft oder ungültig sind, sollte das IO-Device-Anwenderprogramm den lokalen Status auf BAD setzen.
Der Kommunikationspartner könnte dann beispielsweise projektierte Ersatzwerte ausgeben.

Remoter Status des Kommunikationspartners

Mit dem remoten Status meldet der Kommunikationspartner, ob er die Eingangsdaten „gut“ verarbeiten konnte oder ob bei ihm eine Störung vorliegt.

Dieser Status bezieht sich auf früher übertragene Eingangsdaten des selben Moduls, nicht auf die des gerade angestoßenen Schreibauftrags.

3.7.2 Zyklisches Lesen mit Status

Ablauf der Funktion PNIO_CBF_DATA_READ( )

Das IO-Device-Anwenderprogramm initiiert den Lesevorgang durch das Aufrufen der Funktion PNIO_initiate_data_read( ). Daraufhin ruft die IO-Schnittstelle für jedes vom IO-Controller in Betrieb genommene Submodul die Callback-Funktion PNIO_CBF_DATA_READ( ) auf. Mit Aufruf der Callback-Funktion werden die Ausgabedaten und der zugehörige **remote Datenstatus** vom Kommunikationspartner aus dem lokalen Prozessabbild (stackintern) gelesen und an die Applikation übergeben.
Zusätzlich wird für Kommunikationspartner der eigene **lokale Status** dieser Ausgabedaten in das lokale Prozessabbild geschrieben.
Es sind beim zyklischen Lesen also zwei Stati im Spiel.

Kommunikationsrichtung	Werte
vom Kommunikationspartner	• Ausgabedaten • Remoter Status
zum Kommunikationspartner	Lokaler Status

Remoter Status des Kommunikationspartners

Mit dem remoten Status meldet der Kommunikationspartner die Qualität der Ausgabedaten (GOOD oder BAD).

Wenn der Kommunikationspartner BAD meldet, kann das IO-Device-Anwenderprogramm zum Beispiel Ersatzwerte weiterverarbeiten.

Lokaler Status

Im Normalfall wird der lokale Status vom IO-Device-Anwenderprogramm auf GOOD gesetzt.

Wenn aber das IO-Device-Anwenderprogramm die gelieferten Ausgabedaten nicht weiterverarbeiten kann, sollte beim Leseauftrag der lokalen Status auf BAD gesetzt werden. Damit kann der Kommunikationspartner, sobald er diesen Status empfängt, erkennen, ob seine gesendeten Ausgabedaten korrekt weiterverarbeitet wurden.

3.7.3 Zyklischer Datenaustausch über das optionale DBA Interface

Ab der Version 2.2 können die IO Daten auch über ein optionales zusätzliches DBA Interface (Direct Buffer Access) ausgetauscht werden. Dabei erhält der Anwender direkten Zugriff auf die IOCR und kann die IO Daten und IOxS der einzelnen Submodule direkt dort eintragen bzw. auslesen. Die Funktion bringt Performancevorteile bei einer großen Anzahl von Modulen, da nicht für jedes Submodul eine Callbackfunktion aufgerufen wird. Nähre Informationen dazu finden Sie in Kapitel 4.1.10 „Zyklischer Datenaustausch über das optionale DBA Interface“.

Das DBA Interface kann in allen RT Klassen (RT, IRT C2, IRT C3) verwendet werden.

3.8 IO Datenaustausch bei IRT Class 3 (ERTEC basierte Plattformen)

Ab der Version 3.1.0 ist die Anwenderschnittstelle für RT und IRT identisch. In beiden Fällen kann der IO-Datenaustausch nun durch dieselbe Callbackfunktion zum Ereignis „Transmission End“ getriggert werden. Für die Applikation ist damit der Mechanismus des IO Datenzugriffs identisch für RT, IRT C2, IRT C3. Die folgende Tabelle zeigt die Unterschiede der alten Versionen bis 2.2.0.3 gegenüber der aktuellen Version:

Funktion	Version 2.2.0.3 und älter	ab Version 3.1.0
Beginn des IO Datenaustauschs RT	asynchron	synchron (empfohlen) durch TRANS_END-Event getriggert oder asynchron

Beginn des IO Datenaustauschs IRT Klasse 2	(nicht unterstützt)	synchron (empfohlen) durch TRANS_END-Event getriggert oder asynchron
Beginn des IO Datenaustauschs IRT Klasse 3	nur synchron, getriggert durch StartOp-Event. Muss von Applikation durch OpDone abgeschlossen werden vor Beginn des nächsten NewCycle.	synchron (empfohlen) durch TRANS_END-Event getriggert oder asynchron
einheitliches API für RT, IRT	nein	ja

3.8.1 Besonderheiten beim Aufbau einer IRT Class 3 AR

Die Inputdaten bei einer IRT Class 3 AR dürfen von der Applikation nicht aktualisiert werden, bevor die Applikation vom Stack mittels der Funktion PNIO_cbf_ar_in_data() benachrichtigt wurde. *Dies gilt nicht für RT und IRT Class 2 AR's.*

Nutzung des Standard IO data exchange Interfaces (PNIO_initiate_data_write):

Der PNIO Stack verriegelt selber die o.g. Bedingung. Wenn die Applikation beim IRT Class 3 Verbindungsaufbau „zu früh", d.h. vor dem IN-Data Event einen Datenaustausch mittels PNIO_initiate_data_write() startet, so wird vom Stack für die entsprechenden IO Submodule keine PNIO_cbf_data_write() Callback generiert und die Applikation kann die IO Daten nicht aktualisieren.
Damit ist das Interface kompatibel zum RT bzw. IRT Class 2 Data Access und die Applikation muss nichts weiter beachten.

Nutzung des Direct Buffer Access Interfaces (DBA):

Hier ist die Applikation selber verantwortlich, dass bei einer IRT C3 AR die Inputdaten nicht aktualisiert werden, bevor vom Stack PNIO_cbf_ar_in_data() aufgerufen wurde.

3.9 Diagnosedaten verwalten

Beschreibung

Bei PROFINET IO gibt es zwei verschiedene Diagnoseverfahren.

Channel Diagnosis Data

Manufactory Specified Diagnosis Data

3.9.1 Channel Diagnosis Data

In PROFINET IO werden Diagnosedaten als „Channel Diagnosis Data“ kodiert. Bestandteil der „Channel Diagnose Data“ sind die „Channel Properties“. Diese kann das IO-Device-Anwenderprogramm mit Hilfe der Funktion PNIO_build_channel_properties( ) generieren. Genaue Details siehe /1/.

Bei PROFINET IO kann ein Submodul aus mehreren Kanälen bestehen. Je Kanal können mehrere „Channel Diagnosis Data“ existieren. Diese kann das IO-Device-Anwenderprogramm mit der Funktion PNIO_diag_channel_add( ) beim Submodul ablegen.

Ist ein „Channel Diagnosis Data“ nicht mehr gültig, muss das IO-Device-Anwenderprogramm mit der Funktion PNIO_diag_channel_remove( ) beim Submodul entfernen.

Für die Verwaltung der Channel-Diagnosis-Data-Diagnosedaten stehen folgende Funktionen zur Verfügung:

Funktion	Zweck
PNIO_build_channel_properties( )	Generieren der Channel Properties
PNIO_diag_channel_add( )	Kanal-Diagnosedaten im Subslot ablegen
PNIO_diag_channel_remove( )	Kanal-Diagnosedaten vom Subslot entfernen

Tabelle 19: Aufzurufende API Funktionen für die Erstellung eines Diagnosedatensatzes

Das Setzen von Channel Diagnosis Data erfolgt in vier Schritten

Schritt	Aktion	Zweck
1	PNIO_build_channel_properties( )	Generieren der Channel Properties
2	PNIO_diag_channel_add( )	Kanal-Diagnosedaten im Subslot ablegen.
3	PNIO_diag_alarm_send( )	Melden des kommenden Kanalfehlers als Diagnosealarm zum IO-Controller.
4	PNIO_CBF_ASYNC_REQ_DONE( )	Quittung auswerten.

Tabelle 20: Aufzurufende API Funktionen für die Aktivierung eines erstellten Diagnosedatensatzes

Das Entfernen von Channel Diagnosis Data erfolgt in drei Schritten

Schritt	Aktion	Zweck
1	PNIO_diag_channel_remove( )	Kanal-Diagnosedaten vom Submodul entfernen.
2	PNIO_diag_alarm_send( )	Melden des gehenden Kanalfehlers als Diagnosealarm zum IO-Controller senden.
3	PNIO_CBF_ ASYNC_REQ_DONE( )	Quittung auswerten.

Tabelle 21: Aufzurufende API Funktionen für das Entfernen eines Diagnosedatensatzes

3.9.2 Manufactory Specified Diagnosis Data

Manufactory Specified Diagnosis Data bietet dem IO-Device-Anwenderprogramm die Möglichkeit eigene herstellerspezifische Diagnosedaten für ein Submodul abzulegen. Innerhalb der Manufactory Specified Diagnosis Data gibt es keine Strukturfestlegung.

Die Channel Properties müssen auch für herstellerspezifische Diagnosedaten angegeben werden. Genaue Details siehe /1/.

Für die Verwaltung der Manufactory Specified Diagnosis Data-Diagnosedaten stehen folgende Funktionen zur Verfügung:

Funktion	Zweck
PNIO_build_channel_properties( )	Generieren der Channel Properties
PNIO_diag_generic_add( )	Herstellerspezifische Diagnosedaten im Subslot ablegen
PNIO_diag_generic_remove( )	Herstellerspezifische Diagnosedaten vom Subslot entfernen

Tabelle 22: Aufzurufende API Funktionen für die Erstellung eines generischen Diagnosedatensatzes

Das Setzen von Manufactory Specified Diagnosis Data erfolgt in vier Schritten

Schritt	Aktion	Zweck
1	PNIO_build_channel_properties( )	Generieren der Channel Properties
2	PNIO_diag_generic_add( )	Herstellerspezifische Kanal-Diagnosedaten im Subslot ablegen.
3	PNIO_diag_alarm_send( )	Melden des kommenden Kanalfehlers als Diagnosealarm zum IO-Controller.
4	PNIO_CBF_ ASYNC_REQ_DONE ( )	Quittung auswerten.

Tabelle 23: Aufzurufende API Funktionen für die Aktivierung eines generischen Diagnosedatensatzes

Das Entfernen von Manufactory Specified Diagnosis Data erfolgt in drei Schritten

Schritt	Aktion	Zweck
1	PNIO_diag_generic_remove( )	Kanal-Diagnosedaten vom Submodul entfernen.
2	PNIO_diag_alarm_send( )	Melden des gehenden Kanalfehlers als Diagnosealarm zum IO-Controller senden
3	PNIO_CBF_ ASYNC_REQ_DONE ( )	Quittung auswerten.

Tabelle 24: Aufzurufende API Funktionen für das Entfernen eines generischen Diagnosedatensatzes

3.10 Besonderheiten beim Ziehen und Stecken von Modulen im Produktivbetrieb

Alarm beim Modul/Submodul Ziehen

Die IO-Schnittstelle generiert einen PROFINET IO Ziehenalarm, sobald das IO-Device-Anwenderprogramm mit Aufruf der folgenden Funktionen ein Modul bzw. ein Submodul zieht.

PNIO_mod_pull( )

PNIO_sub_pull( )

Alarm beim Modul/Submodul Stecken

Die IO-Schnittstelle generiert ein PROFINET IO-Steckenalarm, sobald das IO-Device-Anwenderprogramm mit Aufruf der folgenden Funktionen ein Modul bzw. ein Submodul steckt.

PNIO_mod_plug( )

PNIO_sub_plug( )

Hinweis

Zwischen dem Callback „PNIO_CBF_AR_CHECK_IND“ und „PNIO_CBF_PARAM_END_IND“ dürfen keine Module gesteckt oder gezogen werden.

Neuparametrierung nach Stecken

Nach jedem PNIO_sub_plug( ) wird das zugehörige Submodul vom IO-Controller neu parametriert. Dies bedeutet, dass die IO-Schnittstelle für jeden vom IO-Controller übertragenen Parametrierdatensatz den PNIO_CBF_REC_WRITE( )-Callback aufruft.

Das Ende der Parametrierung wird von der IO-Schnittstelle durch Aufruf des PNIO_CBF_PARAM_END_IND( )-Callback signalisiert.

Nach der Parametrierung ermittelt das IO-Device-Anwenderprogramm, ob das gesteckte Submodul mit der übertragenen Parametrierung funktionsfähig ist.

Wenn „JA“, muss das IO-Device-Anwenderprogramm die zu sendenden Eingangsdaten und den lokalen Status für die Ein- und Ausgänge dieses Submoduls auf GOOD setzen. Danach muss das IO-Device-Anwenderprogramm die Funktion PNIO_set_appl_state_ready( ) aufrufen. **Der Vorgang ist damit beendet.**

Wenn „NEIN“, muss das IO-Device-Anwenderprogramm die lokalen Status für die Ein- und Ausgänge diese Submoduls auf BAD stehen lassen. (BAD wurde vor dem Ziehen gesetzt; siehe Hinweis oben). Das IO-Device-Anwenderprogramm muss in diesem Fall die Funktion PNIO_set_appl_state_ready( ) aufrufen. **Der Vorgang ist damit mit Fehler beendet**.

3.10.1 Besonderheiten bei „Return of Submodul“

Beschreibung

Bei Funktionsstörung eines gesteckten Submoduls darf das IO-Device-Anwenderprogramm die lokalen Status der Ein- und Ausgangsdaten auf BAD setzen. Dies bewirkt beim zugeordneten IO-Controller, dass die Ein- und Ausgangsdaten für das Anwenderprogramm auf dem IO-Controller nicht mehr gültig sind.

Ist das Submodul wieder funktionsfähig, muss das IO-Device-Anwenderprogramm die lokalen Status der Ein- und Ausgangsdaten auf GOOD setzen. Danach muss das IO-Device-Anwenderprogramm den Übergang von BAD nach GOOD dem IO-Controller durch Aufrufen der Funktion PNIO_ret_of_sub_alarm_send( ) signalisieren. Der IO-Controller wird auf Grund des Return-of-Submodule-Alarms das Submodul nicht neu parametrieren. **Das Submodul ist damit wieder funktionsfähig.**

3.11 Callback-Mechanismus

Funktionsweise

Callback-Funktionen werden von der Komponente IOD vorgegeben. Der Name einer Callback-Funktion kann beliebig gewählt werden.

Ein Callback-Ereignis ist ein asynchrones Ereignis, welches durch IOD gestartet wird. Es unterbricht den Ablauf des Anwenderprogramms und startet die Callback-Funktion in einem eigenen Thread. Synchronisierungstechniken sind deshalb erforderlich.

Callback-Funktionen im IO-Device

Folgender Tabelle entnehmen Sie die Callback-Ereignisse und -Ereignistypen im IO-Device. Sie zeigt auch, womit Sie eine Callback-Funktion anmelden können und wodurch ein Callback-Ereignis ausgelöst wird:

Callback-Ereignis (asynchron)	Callback-Ereignistyp	Registriert durch ...	Ausgelöst durch ...
Daten lesen	PNIO_CBF_ DATA_READ	PNIO_device_open	Anwenderprogramm durch Aufruf von PNIO_initiate_data_read
Daten schreiben	PNIO_CBF_ DATA_WRITE	PNIO_device_open	Anwenderprogramm durch Aufruf von PNIO_initiate_data_write
Datensatz lesen	PNIO_CBF_ REC_READ	PNIO_device_open	IO-Controller
Datensatz schreiben	PNIO_CBF_ REC_WRITE	PNIO_device_open	IO-Controller
Quittung für einen Alarmsendeauftrag	PNIO_CBF_ ASYNC_REQ_DONE	PNIO_device_open	Anwenderprogramm durch Aufruf von: ● PNIO_process_alarm_ send ● PNIO_diag_alarm_ send ● PNIO_ret_of_sub_ alarm_send
Ausfall des IO-Controllers	PNIO_CBF_DEV_ALARM_ IND	PNIO_device_open	Von der IO-Schnittstelle lokal generierter Alarm
Vergleich des Sollausbaus	PNIO_CBF_ CHECK_IND	PNIO_device_open	CM
Verbindungsaufbau	PNIO_CBF_AR_INFO_IND	PNIO_device_open	IO-Controller
Beginn des Datenaustausch	PNIO_CBF_AR_ INDATA_IND	PNIO_device_open	IO-Controller
Verbindungsabbau ohne zuvor existierendem Datenaustausch	PNIO_CBF_AR_ABORT_IND	PNIO_device_open	IO-Controller, Anwenderprogramm
Verbindungsabbau nach zuvor existierendem Datenaustausch	PNIO_CBF_AR_OFFLINE_ IND	PNIO_device_open	IO-Controller, Anwenderprogramm

Fortsetzung der Tabelle auf der nächsten Seite

Fortsetzung

Callback-Ereignis (asynchron)	Callback-Ereignistyp	Registriert durch ...	Ausgelöst durch ...
Statusänderung des IO-Controllers	PNIO_CBF_APDU_STATUS_ IND	PNIO_device_open	IO-Controller
Ender der Parametrierphase	PNIO_CBF_PRM_END_IND	PNIO_device_open	IOD
LED Blinken einschalten	PNIO_CBF_START_LED_BLINK	PNIO_device_open	IOD
LED Blinken ausschalten	PNIO_CBF_STOP_LED_BLINK	PNIO_device_open	IOD
Generische Alarme	PNIO_CBF_DEV_ALARM_IND	PNIO_device_open	IOD
Neue IP-Adresse speichern	PNIO_CBF_SAVE_IP_ADDR	PNIO_device_open	OHA
Neuer Stationsname speichern	PNIO_CBF_SAVE_STATION_NAME	PNIO_device_open	OHA
Modulzustand ready/not ready abfragen	PNIO_CBF_SUB_GET_STATE	PNIO_device_open	IOD
REMA Daten speichern	PNIO_CBF_STORE_REMA_MEM	PNIO_device_open	IOD
Übertragung zykl. Daten beendet	PNIO_CP_CBE_TRANS_END_IND	PNIO_CP_register_cbf	IOD, nur für ERTEC verfügbar

Tabelle 25: Übersicht der Callback Funktionen im IO-Device

Hinweis 1

Binden Sie beim Kompilieren Ihres Anwenderprogramms multithreading-fähige Standardbibliotheken.

Hinweis 2

Innerhalb einer Callback-Funktion sind Funktionsaufrufe verboten, die zum Aufruf der selben Callback-Funktion führen.

So dürfen zum Beispiel auch die hier beschriebenen Funktionen der IO-Device-Anwenderprogrammierschnittstelle nicht aufgerufen werden, wenn nicht ausdrücklich zugelassen!

Ablaufkoordination bei Callbacks

Eine Callback-Funktion kann das IO- Device-Anwenderprogramm zu jedem Zeitpunkt unterbrechen. Außerdem können sich Callback-Funktionen für unterschiedliche Ereignisse gegenseitig unterbrechen. Deshalb muss eine Callback-Funktion für die gleichzeitige, mehrfache Abarbeitung ausgelegt sein („reentrant“), weil sie aus verschiedenen Threads aufgerufen werden kann. **Praktisch bedeutet dies, dass das Schreiben und Lesen von gemeinsam verwendeten Variablen durch Synchronisierungsmechanismen geschützt werden muss**.

Vermeiden Sie Wartezeit in Callback-Funktionen, besonders beim Eintritt in Critical Sections. Ein erneuter Aufruf dieser und weiterer Callback-Funktionen kann hierdurch blockiert werden. Verwenden Sie stattdessen möglichst getrennte Datenhaltung.

4 Schnittstellenbeschreibung

4.1 Upper Layer Schnittstellenfunktionen zur Applikation

Vom Anwender zu implementierende Funktionen

Alle Funktionen mit der Kennzeichnung "**Funktionsaufruf: IO-Stack ➔ Applikation**" werden nicht von der Applikation, sondern vom Stack aufgerufen und müssen vom Anwender implementiert werden. Alle diese Funktionen beginnen mit dem Namenspräfix „PNIO_cbf_“, da es sich, logisch betrachtet, um eine Callbackfunktion handelt. Für alle diese Funktionen ist bereits eine einfache Beispielimplementierung enthalten, die aber in der Regel vom Anwender erweitert werden muss.

Synchrone und asynchrone Funktionen

Bei einer synchronen Funktion ist die Bearbeitung beim Return der Funktion bereits abgeschlossen.
Bei asynchronen Funktionen hingegen wird der Abschluss der Bearbeitung über eine Callback Funktion angezeigt.
Ob eine Funktion synchron oder asynchron arbeitet, ist in der Beschreibung der einzelnen Funktionen enthalten.

4.1.1 Funktionen für den Systemanlauf

4.1.1.1 PNIO_init

PNIO_init ()		. Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Diese Funktion initialisiert die Adaptions-Schnittstelle des PNIO Stacks zum Betriebssystem (OS-Interface) und das BSP-Interface. Sie muss daher einmalig im Anlauf zuerst aufgerufen werden, bevor irgendeine andere PNIO-Funktion aufgerufen wird, z.B. bevor die erste PNIO-Task mit OsCreateThread() erzeugt wird.		
Input	---	
Output	---	

4.1.1.2 PNIO_setup

PNIO_setup ()	. Funktionsaufruf: Applikation ➔ IO-Stack, synchron

Startet den IO-Stack. Die Funktion wird einmalig im Anlauf aus der ersten PNIO-Task heraus aufgerufen, die mit OsCreateThread erzeugt werden musste.

Als Übergabeparameter sind Stationsname und Stationstyp sowie die Logging Level des Anwenderinterfaces und des IO-Stacks vorzugeben. Zulässige Werte für Logging Level sind:

1: Alle Fehler werden ausgegeben

2: Alle Fehler und wichtige Notes werden ausgegeben

3: Alle Fehler, wichtige und normale Notes werden ausgegeben

4: Alle Fehler, wichtige und normale Notes, Speicher-Notes werden ausgegeben

Hinweis: Die Logging Level größer 1 sollten nur in Ausnahmefällen für das Debugging eingestellt werden. Falls zu viele Ausgaben durch einen hohen Logging-Level gemacht werden, ist evt. die Echtzeitfähigkeit und damit die Lauffähigkeit des Systems nicht mehr gewährleistet. Dies ist system- und implementierungsabhängig.

Input	pStationName	Zeiger auf den Stationsnamen (kann ein nicht NULL-terminierter String sein)
	StationNameLen	Länge des Station Name String
	pStationType	Zeiger auf den Stationstyp (NULL-terminierter String)
	PnioApiLogLevel	Logging Level des Applikations-Interfaces
	PnioStackLogLevel	Logging Level des PROFINET IO-Stack
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.1.3 PNIO_set_eth_par

PNIO_set_eth_par ()	Funktionsaufruf: Applikation ➔ IO-Stack, synchron

Übergabe der (z.B. im NVRAM) persistent gespeicherten Verbindungsdaten IP-Adresse, Subnet-Mask, Router Adresse an den IO-Stack. Die Funktion wird einmalig im Anlauf vor PNIO_setup () aufgerufen, um dem IO-Stack vor dem Start der IO Threads die persistent gespeicherten Ethernetparameter mitzuteilen

Input	IpAddr	IP-Adresse als 32 Bit Wert
	SubnetMask	Subnetz-Maske als 32 Bit Wert (z.B. 0xffffff00 für 255.255.255.0)
	DefRouterAddr	Default Router Adresse als 32 Bit Wert
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.1.4 PNIO_device_open

PNIO_device_open ()		. Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Erzeugt eine Device-Instanz. Die Funktion wird einmalig im Anlauf (nach PNIO_setup()) für jede Device- instanz aufgerufen. Als Parameter werden Device- Vendor- und Instanz-ID, sowie max. Anzahl von AR, die Device Annotation und (optional aus Gründen der Schnittstellenkompatibilität zu EB400 und CP1616, über #define in compiler.h einstellbar) eine Liste von Callback-Funktionen übergeben. Der Stack erzeugt ein Handle für das Device und schreibt dieses an die mit pDevHndl vorgegebene Adresse. Das Device-Handle ist in der Applikation zu speichern und bei den meisten PNIO_-Funktionen als Parameter anzugeben. Hinweis: Derzeit ist die Multidevicefunktionalität noch nicht implementiert, dennoch ist das Handle korrekt anzugeben.		
Input	Reserved 1	reserviert
	Reserved 2	reserviert
	VendorId	Hersteller-Id für das Device, muss bei PNO beantragt werden
	DeviceId	Device-ID, muss eindeutig innerhalb der PNIO Produkte eines Herstellers sein.
	InstanceId	Instanz ID für das Device (derzeit nur InstanceId = 1 zulässig)
	NumOfSlots	Maximale Anzahl von Slots pro Device (inclusive DAP und PDEV Interface), die gesteckt werden können. . Die verwendeten Slotnummern dürfen Lücken beinhalten, so dass dieser Wert nicht identisch mit der maximal verwendeten Slotnummer+1 sein muss.
	NumOfSubslots	Maximale Anzahl von Subslots pro Slot. Die verwendeten Subslotnummern dürfen Lücken beinhalten, so dass dieser Wert nicht identisch mit der maximal verwendeten Subslotnummer+1 sein muss.
	MaxAr_Class123	Zulässige Anzahl von AR's der RT-Klasse 1, 2, 3 , die gleichzeitig aktiv sein können
	MaxAr_DevAcc	Zulässige Anzahl von zusätzlichen AR's für Device Access
	pDevAnnotation	Annotation Struktur, enthält Device-Typ, Bestellnummer, Ausgabestand,.. etc.
	pSnmpPar	Zeiger vom Typ PNIO_SNMP_LLDP auf Snmp Objekte, die in die LLDP MIB eingetragen werden.
	pCbf	Liste von Callback functions (hier NULL)
	MrpCapabilityActive	PNIO_TRUE: MRP Fähigkeit aktiviert, PNIO_FALSE: MRP Fähigkeit ist gesperrt.
	pDevHndl	Zeiger auf Adresse, in der der IO-Stack das Device-Handle an die Applikation zurückliefert.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.1.5 PNIO_api_add

PNIO_api_add ()		. Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Erzeugt eine API-Instanz. Die Funktion wird einmalig im Anlauf (nach PNIO_device_open ()) für jedes API in der Device- instanz aufgerufen..		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	API	API Nummer. Default API ist 0, weitere sind derzeit noch nicht implementiert.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.1.6 PNIO_netcom_enable

PNIO_netcom_enable ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
EDD-ERTEC: Dummy Funktion, wird leer implementiert. EDD-soft: Die Funktion aktiviert den Empfang von Ethernet Telegrammen auf unterster Ebene im Checker. Die Funktion sollte als letzte Funktion im Anlauf aufgerufen werden, d.h. nachdem der IO-Stack hochgelaufen ist.		
Input	-----	
Output	return	Ausführungsstatus: PNIO_OK

4.1.1.7 PNIO_device_start

PNIO_device_start ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Die Funktion aktiviert die Bereitschaft zum Verbindungsaufbau. Erst nach dem Aufruf kann vom Controller eine Verbindung aufgebaut werden.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
Output	return	Ausführungsstatus: PNIO_OK , PNIO_NOT_OK

4.1.1.8 PNIO_device_stop

PNIO_device_stop ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Die Funktion bricht alle laufenden PROFINET Verbindungen ab. Erst nach erneutem Aufruf von PNIO_device_start () kann wieder eine Verbindung aufgebaut werden.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
Output	return	Ausführungsstatus: PNIO_OK , PNIO_NOT_OK

4.1.1.9 PNIO_set_appl_state_ready

PNIO_set_appl_state_ready ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Hiermit teilt die Applikation dem Stack die Bereitschaft zum Datenaustausch mit. Erst danach kann ein Controller eine Verbindung zum Device aufbauen. Hinweis: In der bestehenden Implementierung wird ein Appl-StateReady Request bereits automatisch vom IOD Control Programm an den CM geschickt, wenn ein vom IO-Controller gesendetes PARAM_END Telegramm empfangen wird und die anschließend vom Stack aufgerufene Funktion PNIO_cbf_param_end_ind () vom Anwender mit return = PNIO_TRUE beendet wird. Es ist also in diesem Fall nicht notwendig, diese Funktion in der Applikation nocheinmal aufzurufen. **Hinweis:** Wird ein Submodul bei laufender AR gesteckt, so kehrt PNIO_sub_plug() (dasgleiche gilt für PNIO_sub_plug_ext) erst zurück, nachdem das Submodul vom Controller neu parametriert wurde und der Vorgang durch Application-Ready abgeschlossen wurde. Falls daher das asynchrone ApplicationReady verwendet wird, so darf PNIO_set_appl_state_ready () nicht im gleichen Taskkontext wie PNIO_sub_plug() aufgerufen werden,		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
	ArNumber	Application-Relation-Number, die der Applikation mittels PNIO_cbf_ar_info_ind() mitgeteilt wurde.
	SessionKey	Session Key für die aktuelle Application Relation, die der Applikation mittels PNIO_cbf_ar_info_ind() mitgeteilt wurde.
Output	return	Ausführungsstatus: PNIO_OK , PNIO_NOT_OK

4.1.1.10 PNIO_device_close

PNIO_device_close ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
wird noch nicht unterstützt.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
Output		Ausführungsstatus: PNIO_OK

4.1.1.11 PNIO_CP_register_cbf

PNIO_CP_register_cbf ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
(Nur für ERTEC verfügbar) Mit PNIO_register_cbf() können Callbackfunktionen für zyklische Ereignisse beim PNIO Stack angemeldet werden, um die Applikation mit der Datenübertragung am Bus zu synchronisieren. Derzeit ist das Ereignis PNIO_CP_CBE_TRANS_END_IND implementiert. Damit wird der Applikation das Ende der Datenübertragung für den laufenden IO Zyklus angezeigt. Die Applikation kann nun auf die IO Daten mittels PNIO_initiate_data_read() bzw. PNIO_initiate_data_write() zugreifen.		
Input	CbeType	Spezifiziert das Ereignis,bei dem die Callback aufgerufen wird. Derzeit ist PNIO_CP_CBE_TRANS_END_IND implementiert.
	pCbf	Startadresse der vom PNIO Stack aufzurufenden Anwenderfunktion, wenn das o.g. Ereignis eingetreten ist.
Output		Ausführungsstatus: PNIO_OK

4.1.2 Knotentaufe

Bei der Knotentaufe können die Ethernetparameter (IP-Adresse, Subnet-Maske, Default Router Adresse) des IO-Devices von einem PROFINET IO-Controller über Ethernet eingestellt werden. Stationsnamen und Stationstyp des Device müssen zuvor mit dem Projektierungstool eingestellt worden sein.
Zur Übergabe dieser Informationen an die Applikation werden vom IO-Stack Funktionen aufgerufen, welche vom Anwender zu implementieren sind. Die Applikation muss diese Daten in einem nichtflüchtigen Speicher (NV-RAM, Flash EPROM,..) ablegen und beim nächsten Systemanlauf diese dem Stack mittels der Funktionen

- PNIO_device_open ()
- PNIO_set_eth_par ()

wieder übergeben.

Für die Übergabe von Stationsnamen und Stationstyp ist ein Funktionsrahmen in iodapi_event.c für folgende Funktionen hinterlegt, welcher vom Anwender zu implementieren ist:

- PNIO_cbf_save_station_name ()

Die Einstellung der Ethernet-Parameter läuft nach dem gleichen Prinzip ab, hier wird allerdings nicht die Applikation, sondern das BSP über die Funktion

- Bsp_SetIpAddr ()

benachrichtigt. Dort muss vom Anwender neben der Sicherung der Ethernetparameter im nichtflüchtigen Speicher auch die Umstellung der IP-Adresse des TCP/IP Stacks durchgeführt werden. Die Umstellung der IP-Adresse für den TCP/IP Stack ist dabei auf folgenden Wegen möglich:

1. Umstellung der IP Parameter im laufenden Betrieb. Falls der TCP/IP Stack diese Möglichkeit bietet, so sollte diese verwendet werden.
2. Herunterfahren und Neuanlauf des TCP/IP Stacks mit neuen IP Parametern.
3. Neuanlauf des Systems mit neuen IP Parametern.

Im Engineering Tool kann eine Blinkfunktion gestartet werden. Das angesprochene IO-Device identifiziert sich daraufhin optisch durch eine blinkende LED. Die Ansteuerung der LED ist plattformabhängig und daher nicht direkt im IO-Stack enthalten. Der IO-Stack ruft in diesem Fall folgende Funktionen auf, welche vom Anwender entsprechend zu implementieren sind:

- PNIO_cbf_start_led_blink (Blinkfrequenz)
- PNIO_cbf_stop_led_blink ()

Mit der folgenden Funktion müssen die Werkseinstellungen wiederhergestellt werden.

- PNIO_cbf_reset_factory_settings ()

4.1.2.1 PNIO_cbf_save_station_name

PNIO_cbf_save_station_name ()		Funktionsaufruf: IO-Stack ➔ Applikation , synchron
Wenn über Ethernet eine neuer Stationsname für das Device vergeben werden soll, so wird vom IO-Stack die Funktion PNIO_cbf_save_station_name () aufgerufen. Die Applikation muss den übergebenen Stationsnamen in einem nichtflüchtigen Speicher ablegen, wenn der Parameter Remanent ungleich Null ist. Der Wert wird beim nächsten Systemanlauf von der Applikation aus dem nichtflüchtígen Speicher gelesen und mit der Funktion PNIO_setup () wieder an den Stack übergeben.		
Input	pStationName	Zeiger auf den String, welcher den Stationsnamen beinhaltet. (Der Stationsname muss nicht zwingen null-terminiert sein)..
	NameLength	Länge des Strings in Byte
	Remanent	<> 0: Daten müssen remanent gespeichert werden == 0: Wert darf nicht remanent gespeichert werden
Output	return	PNIO_OK, PNIO_NOT_OK

4.1.2.2 PNIO_cbf_save_ip_addr

PNIO_cbf_save_ip_addr ()		Funktionsaufruf: IO-Stack ➔ Applikation , synchron
Wenn über Ethernet eine neue IP-Adresse für das Device vergeben werden soll, so wird vom IO-Stack die Funktion PNIO_cbf_save_ip_addr() aufgerufen. Die Applikation muss die Parameter IP-Addresse, Subnet Mask und Default Router einem nichtflüchtigen Speicher ablegen, wenn der Parameter "Remanent" ungleich Null ist. Der Wert wird beim nächsten Systemanlauf von der Applikation aus dem nichtflüchtígen Speicher gelesen und an mit der Funktion PNIO_set_eth_par () wieder an den Stack übergeben.		
Input	NewIpAddr	Neue IP-Adresse
	SubnetMask	Neuer Wert für Subnet Mask
	DefaultRouterAddr	Neuer Wert für Default Router
	Remanent	<> 0: Daten müssen remanent gespeichert werden == 0: Wert darf nicht remanent gespeichert werden
Output	return	PNIO_OK, PNIO_NOT_OK

4.1.2.3 PNIO_cbf_start_led_blink

PNIO_cbf_start_led_blink ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Wenn im Engineering Tool die Blink-Funktion gestartet wird, so ruft der IO-Stack diese Funktion auf. Die Applikation kann daraufhin eine LED (falls vorhanden) in den Blinkmodus in der vorgegebenen Frequenz versetzen. Der Blinkvorgang dauert ca. 3 Sekunden, dann wird automatisch vom Stack PNIO_cbf_stop_led_blink aufgerufen und damit der Blinkvorgang beendet.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Frequency	vorgegebene Blinkfrequenz in Hertz
Output	return	must be PNIO_OK

4.1.2.4 PNIO_cbf_stop_led_blink

PNIO_cbf_stop_led_blink ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Blinkmodus der LED wieder abschalten.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
Output	return	must be PNIO_OK

4.1.2.5 PNIO_cbf_reset_factory_settings

PNIO_cbf_stop_reset_factory_settings ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Rücksetzen auf Werkseinstellung. Die Applikation muss alle permanent gespeicherten Parameter (Gerätename, IP Suite, REMA Daten,.. etc) auf Werkseinstellungen zurücksetzen. Anschließend muss die Applikation einen Anlauf des Systems durchführen, damit der PNIO Stack seine Daten intern zurücksetzt.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
Output	return	must be PNIO_OK

.

4.1.3 Speicherung remanenter Daten (REMA)

Neben den Daten für die Knotentaufe (Gerätename, IP-Suite) müssen auch die Records des Physical Device (PDEV-Records) auf dem Gerät nichtflüchtig gespeichert werden. Die PDEV-Records werden zunächst nach dem Aufbau der AR vom PNIO Controller an das Device mittels Record-Write Funktionen übertragen.
Die Records werden zunächst vom PNIO Stack zwischengespeichert und dort bearbeitet. Wenn alle PDEV Records empfangen wurden, so werden diese vom PNIO Stack in einen zusammenhängenden Speicherbereich eingetragen und in einen einmaligen Aufruf an die Applikation zur persistenten Speicherung übertragen. Die Applikation muss also die PDEV Records dafür nicht interpretieren, sondern lediglich den gesamten Datenblock übernehmen und im nichtflüchtigen Speicher ablegen. Die Übergabe des PDEV-Datenblockes vom PNIO Stack erfolgt mit der Funktion

PNIO_cbf_store_rema_mem ()

Als Übergabeparameter wird dabei ein Zeiger auf die Daten, die Datenlänge sowie (rein informative für die Applikation) die Anzahl der enthaltenen Records übertragen.

Beim nächsten Geräteanlauf liest die Applikation diesen Datenblock vom nichtflüchtigen Speicher und übergibt diesen nach dem Anlauf des PNIO Stack (also z.B. nach dem Stecken aller Module) mittels der Funktion

PNIO_restore_rema_mem ()

wieder an den PNIO Stack. Als Übergabeparameter werden dabei lediglich der Zeiger auf die Daten und die Datenlänge übergeben. ***Auch wenn keine gültigen REMA-Daten vorliegen, muss PNIO_restore_rema_mem() mit einem NULL-Pointer aufgerufen werden, um die ordnungsgemässe Funktion des Stacks zu gewährleisten.***

4.1.3.1 PNIO_cbf_store_rema_mem

PNIO_cbf_store_rema_mem ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Hier werden der Applikation alle empfangenen PDEV-Records in einem zusammenhängenden Datenblock zur nichtflüchtigen Speicherung übergeben. Die Applikation muss den Datenblock lediglich ohne Änderung oder Interpretation im NV Speicher ablegen und beim nächsten Anlauf mittels PNIO_restore_rema_mem() wieder an den PNIO Stack übergeben.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
	InterfaceID	Interface ID, derzeit nur Wert 1 möglich
	MemSize	Größe des Datenblocks in Byte
	pMem	Zeiger auf den Datenblock
Output	return	PNIO_OK, PNIO_NOT_OK

4.1.3.2 PNIO_restore_rema_mem

PNIO_restore_rema_mem ()		Funktionsaufruf: Applikation ➔ IO Stack, synchron
Mit dieser Funktion übergibt die Applikation den im NV Ram abgelegten Datenblock mit den gespeicherten PDEV-Records wieder an den PNIO Stack. Die Funktion ist einmalig im Hochlauf von der Applikation aufzurufen, nachdem der Stack hochgelaufen ist und die PDEV Module gesteckt wurden. ***Auch wenn keine gültigen REMA-Daten vorliegen, muss PNIO_restore_rema_mem() mit einem NULL-Pointer aufgerufen werden, um die ordnungsgemässe Funktion des Stacks zu gewährleisten.***		
Input	MemSize	Größe des Datenblocks in Byte
	pMem	Zeiger auf den Datenblock
Output	return	PNIO_OK, PNIO_NOT_OK

4.1.4 I/O Device Konfiguration

4.1.4.1 PNIO_mod_plug

PNIO_mod_plug ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Stecken eines neuen Moduls. Die Funktion wird von der Applikation im Anlauf aufgerufen zur Vorgabe der aktuellen Konfiguration an den IO-Stack. Sie kann außerdem im laufenden Betrieb bei Änderungen der aktuellen Konfiguration aufgerufen werden. An den IO-Controller wird in diesem Fall vom IO-Stack automatisch ein Plug-Alarm gesendet. Wenn die gesteckte Konfiguration nicht mit der projektierten Konfiguration übereinstimmt, so wird vom Stack die Funktion PNIO_cbf_check_ind () aufgerufen, damit die Applikation den Konflikt bereinigen kann.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Moduls, hier ist nur der Slot relevant. Slot Nummer = 1....N sind zulässig, die maximal Slotnummer wurde in PNIO_device_open () festgelegt. Als Type muss PNIO_ADDR_GEO eingetragen werden.
	ModIdent	Module Indentfier (ist in der GSD Datei hinterlegt)
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.4.2 PNIO_sub_plug

PNIO_sub_plug ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Stecken eines neuen Submoduls in einen Subslot. Die Funktion wird aufgerufen im Anlauf zur Vorgabe der aktuellen Konfiguration an den IO-Stack. Sie kann außerdem im laufenden Betrieb bei Änderung der aktuellen Konfiguration aufgerufen werden, d.h. wenn ein ausgefallenes oder gezogenes Submodul wieder funktionsfähig ist. An den IO-Controller wird in diesem Fall vom IO-Stack automatisch ein Plug-Alarm gesendet. Für die IM0-Filtertabelle wird automatisch PNIO_IM0_SUBMODULE eingetragen, d.h. **dieses Submodul muss das Lesen von IM0 Daten unterstützen**. Siehe dazu auch Funktion PNIO_sub_plug_ext(). **Hinweis:** Wird ein Submodul bei laufender AR gesteckt, so kehrt PNIO_sub_plug() erst zurück, nachdem das Submodul vom Controller neu parametriert wurde und der Vorgang durch Application-Ready abgeschlossen wurde. Falls daher das asynchrone ApplicationReady verwendet wird, so darf PNIO_set_appl_state_ready () nicht im gleichen Taskkontext wie PNIO_sub_plug() aufgerufen werden,		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Slot Nummer 1....N sind zulässig, die maximal Slotnummer wurde in PNIO_setup() festgelegt. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	SubIdent	Submodule Identfier (ist in der GSD Datei hinterlegt)
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.4.3 PNIO_sub_plug_ext

PNIO_sub_plug_ext ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Erweiterung der Funktion PNIO_sub_plug().Im Unterschied zur Funktion PNIO_sub_plug( ) kann mit dieser Funktion dem IO-Controller mitgeteilt werden, dass ein neues Submodul gesteckt wurde, das von der Projektierung aus gesehen falsch ist. Das heißt, dass die neue Submodul-ID weder die selbe Submodul-ID wie das alte Submodul hat noch kompatibel zum alten Submodul ist. In diesem Fall wird vom Device statt einem „Plug Submodule Alarm“ ein „Plug Wrong Submodule Alarm“ zum IO-Controller gesendet. Bei AlarmType= PNIO_ALARM_TYPE_PLUG oder AlarmType= PNIO_ALARM_TYPE_PLUG_WRONG wird keine Soll-Ist Prüfung durchgeführt und daher bei Inkonsistenz auch nicht PNIO_cbf_check_ind() aufgerufen. Bei Vorgabe von AlarmType=PNIO_ALARM_TYPE_DEPENDS_ON_CHECK wird hingegen eine Soll-Ist Prüfung durch den Stack vorgenommen, wenn das Submodul bei laufender AR gesteckt wird. Bei Inkonsistenz wird in diesem Fall PNIO_cbf_check_ind() aufgerufen. **Hinweis:** Wird ein Submodul bei laufender AR gesteckt, so kehrt PNIO_sub_plug_ext() erst zurück, nachdem das Submodul vom Controller neu parametriert wurde und der Vorgang durch Application-Ready abgeschlossen wurde. Falls daher das asynchrone ApplicationReady verwendet wird, so darf PNIO_set_appl_state_ready () nicht im gleichen Taskkontext wie PNIO_sub_plug() aufgerufen werden,		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Slot Nummer 1....N sind zulässig, die maximal Slotnummer wurde in PNIO_setup() festgelegt. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	SubIdent	Submodule Identfier (ist in der GSD Datei hinterlegt)
	Im0Support	Informationen für die IM0FilterData gemäß PNIO-Spezifikation /1/ , mögliche Werte siehe enum PNIO_IM0_SUPP_ENUM in File pniousrd.h. Hier kann für jedes Submodul festgelegt werden, ob dieses IM0 unterstützt oder nicht.
	AlarmType	PNIO_ALARM_TYPE_PLUG PNIO_ALARM_TYPE_PLUG_WRONG PNIO_ALARM_TYPE_DEPENDS_ON_CHECK
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.4.4 PNIO_mod_pull

PNIO_mod_pull ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Ziehen eines gesteckten Moduls. An den IO-Controller wird in diesem Fall vom IO-Stack automatisch ein Pull-Alarm gesendet.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Moduls, hier ist nur der Slot relevant. Slot Nummer = 1....N sind zulässig, die maximal Slotnummer wurde in PNIO_setup() festgelegt. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.4.5 PNIO_sub_pull

PNIO_sub_pull ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Ziehen eines gesteckten Submoduls. An den IO-Controller wird in diesem Fall vom IO-Stack automatisch ein Pull-Alarm gesendet.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Slot Nummer 1....N sind zulässig, die maximal Slotnummer wurde in PNIO_setup() festgelegt. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.4.6 PNIO_sub_set_state

PNIO_sub_ set_state ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Hiermit teilt die Applikation dem Stack mit, dass ein Submodul nach dessen Parametrierung ordnungsgemäss in den RUN gehen konnte. Der Stack generiert anhand dieser Information die Bad-List, die mittels des „Application ready" Telegramms an den Controller gesendet wird. Wurde hingegen ein Submodul bis zum Sendezeitpunkt von ApplicationReady noch nicht mit PNIO_sub_set_state als „RUN" gemeldet, so fragt der Stack vor Versenden des Application ready explizit bei der Applikation mittels PNIO_cbf_sub_get_state() den Zustand genau dieses Moduls ab. Die Applikation gibt dabei als Returnwert PNIO_SUBMOD_STATE_STOP oder PNIO_SUBMOD_STATE_RUN zurück. Es ist also nicht zwingend notwendig, die Funktion PNIO_sub_set_state (PNIO_SUBMOD_STATE_GOOD) zu verwenden, da andernfalls der Stack selber bei der Applikation mittels PNIO_cbf_sub_get_state() für alle Module mit Zustand ungleich RUN nachfragt.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Slot Nummer 1....N sind zulässig, die maximal Slotnummer wurde in PNIO_setup() festgelegt. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	SubmodState	PNIO_SUBMOD_STATE_RUN (ein Aufruf mit SubState = PNIO_SUBMOD_STATE_STOP bewirkt keine Aktion, da keine Zustandsänderung stattfindet).
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.4.7 PNIO_cbf_sub_get_state

PNIO_cbf_sub_ get_state ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Wenn ein Application Ready an den Controller gesendet werden soll (z.B. durch Aufruf von PNIO_set_appl_state_ready()), so wird für alle diejenigen Submodule, die bis dahin noch nicht den Zustand „RUN" erreicht haben, die Callbackfunktion PNIO_cbf_sub_get_state() aufgerufen. Die Applikation gibt dabei als Returnwert PNIO_SUBMOD_STATE_STOP oder PNIO_SUBMOD_STATE_RUN zurück.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Slot Nummer 1....N sind zulässig, die maximal Slotnummer wurde in PNIO_setup() festgelegt. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	*pSubmodState	Pointer, wohin die Applikation eine der folgenden Antworten schreiben muss: PNIO_SUBMOD_STATE_RUN, wenn das Submodul im Zustand „RUN" ist PNIO_SUBMOD_STATE_STOP wenn das Submodul im Zustand „STOP" ist.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.5 Diagnosedaten im Subslot ablegen

4.1.5.1 PNIO_diag_channel_add

PNIO_diag_channel_add ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Herunterladen eines Diagnosedatensatzes in einen Subslot. Die Diagnosedaten können über einen speziellen Recordaufruf (siehe /1/) vom Controller ausgelesen werden. Auf einen Subslot können mehrere Diagnosedatensätze geladen werden, die über einen vom Anwender zu vergebenden Tag eindeutig referenziert werden. Über diese Referenz kann der Diagnosesatz mit der Funktion PNIO_diag_channel_remove() wieder entfernt werden .		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	ChannelNum	Kanalnummer, Inhalt und Aufbau siehe auch „ChannelNumber“ in /1/
	ChannelProp	Spezifiziert die Channel Properties. Diese werden mit der Funktion PNIO_build_channel_properties() vorher generiert. Inhalt und Aufbau siehe auch „ChannelProperties“ in /1/
	ChannelErrType	Channel Error Type, Inhalt und Aufbau siehe „ChannelErrorType“ in /1/
	DiagTag	vom Anwender vorgegebene eindeutige Referenz auf den Datensatz. Dieser wird benötigt, um den Diagnosedatensatz mit PNIO_ext_diag_channel_remove() wieder zu löschen.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.5.2 PNIO_diag_channel_remove

PNIO_diag_channel_remove ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Löschen eines mit PNIO_diag_channel_add() heruntergeladenen Diagnosedatensatzes .		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	DiagTag	vom Anwender vorgegebene eindeutige Referenz auf den Datensatz
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.5.3 PNIO_ext_diag_channel_add

PNIO_ext_diag_channel_add ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Herunterladen eines erweiterten Diagnosedatensatzes in einen Subslot. Über diese Referenz kann der Diagnosesatz mit der Funktion PNIO_ext_diag_channel_remove() wieder entfernt werden .		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	ChannelNum	Kanalnummer, Inhalt und Aufbau siehe auch „ChannelNumber“ in /1/
	ChannelProp	Spezifiziert die Channel Properties. Diese werden mit der Funktion PNIO_build_channel_properties() vorher generiert. Inhalt und Aufbau siehe auch „ChannelProperties“ in /1/
	ChannelErrType	Channel Error Type, Inhalt und Aufbau siehe „ChannelErrorType“ in /1/
	ExtChannelErrType	Extended Channel Error Type, Inhalt und Aufbau siehe „ExtChannelErrorType“ in /1/
	ExtChannelAddValue	Additional Value, Inhalt und Aufbau siehe „ExtChannelAddValue“ in /1/
	DiagTag	vom Anwender vorgegebene eindeutige Referenz auf den Datensatz. Dieser wird benötigt, um den Diagnosedatensatz mit PNIO_ext_diag_channel_remove() wieder zu löschen.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.5.4 PNIO_ext_diag_channel_remove

PNIO_ext_diag_channel_remove ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Löschen eines mit PNIO_ext_diag_channel_add() heruntergeladenen Diagnosedatensatzes .		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	DiagTag	vom Anwender vorgegebene eindeutige Referenz auf den Datensatz
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.5.5 PNIO_diag_generic_add

PNIO_diag_generic_add ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Herunterladen eines herstellerspezifischen Diagnosedatensatzes in einen Subslot. Die Diagnosedaten können über einen speziellen Recordaufruf (siehe /1/) z.B. von einem Diagnosetool ausgelesen werden. Auf einen Subslot können mehrere Diagnosedatensätze geladen werden, die über einen vom Anwender vorgebbaren Tag referenziert werden. Über diese Referenz kann der Diagnosesatz mit der Funktion PNIO_diag_generic_remove() wieder entfernt werden .		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	ChannelNum	Kanalnummer
	ChannelProp	Diagnosedaten. Diese werden mit der Funktion PNIO_build_channel_properties () vorher generiert.
	DiagTag	vom Anwender vorgegebene eindeutige Referenz auf den Datensatz
	UserStructIdent	UserStructureIdentifier, s. /1/: 0..7fff : Herstellerspezifische Daten in pInfoData
	pInfoData	Zeiger auf die Diagnosedaten
	InfoDataLen	Länge der Diagnosedaten in Bytes
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.5.6 PNIO_diag_generic_remove

PNIO_diag_generic_remove ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Löschen eines mit PNIO_generic_add() heruntergeladenen Diagnosedatensatzes		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	DiagTag	vom Anwender vorgegebene eindeutige Referenz auf den Datensatz
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.5.7 PNIO_build_channel_properties

PNIO_build_channel_properties ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Generieren eines info_tag für Standard-Diagnosedatensätze, dieses kann anschließend mit PNIO_diag_channel_add() auf einen Slot heruntergeladen werden. Kodierung der Parameter Type, Spec und Dir siehe auch /1/ "Coding of the field ChannelProperties"		
Input	Type	channel properties.type (Datentyp 1Bit … 64Bit). Mögliche Werte sind DIAG_CHANPROP_TYPE_1BIT DIAG_CHANPROP_TYPE_2BIT DIAG_CHANPROP_TYPE_4BIT DIAG_CHANPROP_TYPE_BYTE DIAG_CHANPROP_TYPE_WORD DIAG_CHANPROP_TYPE_DWORD DIAG_CHANPROP_TYPE_LWORD DIAG_CHANPROP_TYPE_OTHERS
	Spec	channel properties.specifier (1: Fehler kommt; 2: Fehler geht, keine weiteren Fehler; 3: Fehler geht, weitere Fehler stehen an)
	Dir	channel properties.dir (0: herstellerspezifisch; 1: Input; 2: Output; 3: Input/Output
	Accumulative	reserviert (muss 0 sein)
	MaintenanceRequired	reserviert (muss 0 sein)
	MaintenanceDemanded	reserviert (muss 0 sein)
Output	return	Wert für ChannelProp, welcher als Aufrufparameter für PNIO_diag_channel_add() verwendet wird.

4.1.6 Senden und Empfangen von Alarmen

Über das IOD-API können folgende subslotspezifischen Alarme ausgelöst werden:

- Prozessalarme (optional)
- Diagnosealarme (optional)
- „Return of Submodule" – Alarme (mandatory)

Prozess- und Diagnosealarme können von der Applikation ausgelöst werden, wenn diese ein entsprechendes Ereignis erkannt hat. „Return of Submodule"-Alarme müssen von der Applikation ausgelöst werden, wenn der Nutzdatenbegleiter (IOPS, IOCS) während des laufenden Betriebs von BAD auf GOOD wechselt.
Die Implementierung der Alarmfunktionen ist **asynchron**, d.h. die Funktion wartet nicht, bis der Alarm vom IO-Controller quittiert wurde. Stattdessen wird nach Empfang der Alarmquittung vom IO-Stack die Callbackfunktion PNIO_cbf_async_req_done () aufgerufen, welche vom Anwender zu implementieren ist.

4.1.6.1 PNIO_process_alarm_send

PNIO_process_alarm_send ()		Funktionsaufruf: Applikation ➔ IO-Stack, asynchron
Senden eines (submodul-spezifischen) Prozessalarms an einen IO-Controller. Die beiliegenden Daten sind im Alarmtelegramm an den Controller enthalten, werden aber nicht lokal im Submodul gespeichert. Eine Remove-Funktion ist deshalb nicht notwendig.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	*pData	Zeiger auf Alarmdaten
	DataLen	Länge der Alarmdaten in Byte
	UserStructIdent	UserStructureIdentifier, s. /1/: 0..7fff : Herstellerspezifische Daten in pData
	UserHndl	Vom Anwender frei vergebbares Handle, um mehrere parallel laufende Alarm-Send Aufrufe zu unterscheiden. Das Handle wird der Applikation nach Empfang der Alarmquittung als Übergabeparameter der Callback-Funktion PNIO_cbf_async_req_done() übergeben.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.6.2 PNIO_diag_alarm_send

PNIO_diag_alarm_send ()		Funktionsaufruf: Applikation ➔ IO-Stack, asynchron
Senden eines submodul-spezifischen Diagnosealarms an einen IO-Controller. Die beiliegenden Daten werden mit dem Alarmtelegramm an den Controller übergeben, werden aber im Gegensatz zu den mit PNIO_diag_channel_add() übergebenen Diagnosedatensätzen nicht im IO-Stack gespeichert. Diagnosealarme können im Gegensatz zu Prozessalarmen kommend oder gehend sein. Vor einem "Kommend-Alarm" muss der entsprechende Diagnosedatensatz gemäß Standardformat (s. IEC 61158) mit PNIO_diag_channel_add() an den IO-Stack übergeben werden, damit die Alarmdaten auch nachträglich z.B. mit einem Diagnosetool auslesbar sind. Nach Auslösung eines "Gehend-Alarms" muss ein mit PNIO_diag_channel_add hinterlegter Diagnosedatensatz mit PNIO_diag_channel_remove() wieder gelöscht werden. Der Gehend-Alarm ist dabei als Sammelalarm zu sehen, d.h. wenn mehrere Diagnosedatensätze eines Subslots auf einmal gelöscht werden, so muss der "Gehend-Alarm" nur einmalig pro betroffenenem Subslot gesendet werden.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	AlarmState	PNIO_STATE_ALARM_APPEARS, wenn ein neuer Alarm kommt. PNIO_STATE_ALARM_DISAPPEARS, wenn ein anstehender Alarm verschwindet.
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	*pData	Zeiger auf Alarmdaten. Hier sollten dieselben Daten übergeben werden, die zuvor mit PNIO_DiagChannelAdd() für diesen Subslot übergeben wurden. Bei einem "Gehend-Alarm" ist pData = NULL zu setzen.
	DataLen	Länge der Alarmdaten in Byte. Bei einem "Gehend-Alarm" ist DataLen = 0 zu setzen.
	UserStructIdent	UserStructureIdentifier, s. /1/: 0..7fff : Herstellerspezifische Diagnosedaten in pData 0x8000 : Kanaldiagnosedaten in pData 0x8001: Kanaldiagnose mehrfach vorhanden 0x8002: Erweiterte Diagnose
	UserHndl	Vom Anwender frei vergebbares Handle, um mehrere parallel laufende Alarm-Send Aufrufe zu unterscheiden. Das Handle wird der Applikation nach Empfang der Alarmquittung als Übergabeparameter der Callback-Funktion PNIO_cbf_async_req_done() übergeben.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.6.3 PNIO_ret_of_sub_alarm_send

PNIO_ret_of_sub_alarm_send ()		Funktionsaufruf: Applikation ➔ IO-Stack, asynchron
Senden eines „Return of Submodul“ – Alarms an den Controller. Der Alarm muss vom Device abgesetzt werden, wenn sich der Status des Nutzdatenbegleiters IOPS/IOCS vom Zustand BAD in den Zustand GOOD ändert. Die Reaktion auf dem IO-Controller ist ähnlich wie bei einem Stecken-Alarm, allerdings wird bei PNIO_ret_of_sub_alarm_send () das Submodul nicht neu vom Controller parametriert. Hinweis: Wenn der IOPS/IOCS von GOOD nach BAD ändert, muss hingegen kein Alarm ausgelöst werden.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	API number (default API is 0)
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	UserHndl	Vom Anwender frei vergebbares Handle, um mehrere parallel laufende Alarm-Send Aufrufe zu unterscheiden. Das Handle wird der Applikation nach Empfang der Alarmquittung als Übergabeparameter der Callback-Funktion PNIO_cbf_async_req_done() übergeben
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.6.4 PNIO_cbf_dev_alarm_ind ()

PNIO_cbf_dev_alarm_ind ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Mit dieser Funktion wird die Applikation über einen Alarm informiert, der vom Controller empfangen wurde.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	*pAlarm	Zeiger auf Alarmdaten vom Typ PNIO_DEV_ALARM_DATA. Darin werden weitere Informationen über den Alarm geliefert.
Output	---	-

4.1.7 Quittierung von asynchronen Funktionen

4.1.7.1 PNIO_cbf_async_req_done

PNIO_cbf_async_req_done ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Wenn ein asynchroner Request von der Applikation an den Stack gestellt wurde, so erfolgt die Quittung durch Aufruf der Funktion PNIO_cbf_async_req_done. Als Übergabeparameter wird neben dem Status (OK/nicht OK) ein UserHandle übergeben, welches von der Applikation frei vergeben wurde. Damit kann die Applikation die Quittung dem entsprechenden Request zuordnen, falls mehrere Requests parallel an den IO-Stack gestellt wurden. Derzeit sind nur die Alarme als asynchrone Requests implementiert.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	UserHndl	Handle, welches vom Anwender beim Request vergeben wurde
	Status	PNIO_OK PNIO_NOT_OK
Output	---	

4.1.8 Lesen und Schreiben von Records

Neben der synchronen Bearbeitung (d.h. komplette Bearbeitung des Rd/Wr Record Requests innerhalb der Callbackfunktion PNIO_cbf_rec_read() oder PNIO_cbf_rec_write() kann die Bearbeitung von Read- oder Write-

record Requests auch asynchron erfolgen. Die Applikation teilt dazu innerhalb der Callbackfunktion dem Stack mit, dass die Antwort auf den Request später, d.h. asynchron erfolgt.

Das Szenario für einen **synchronen Read Record Request** sieht folgendermassen aus:

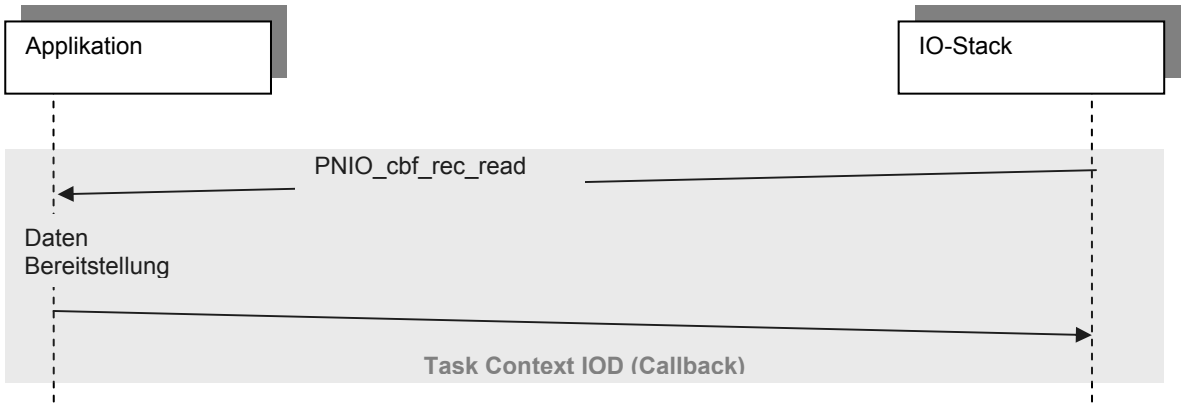


Abbildung 5: synchrone Read Record Bearbeitung

- Ein Record Read Request vom IO-Controller wird empfangen und vom IO-Stack ausgewertet.
- IO-Stack ruft PNIO_cbf_rec_read auf
- Applikation stellt die geforderten Record Daten sowie Fehlerstatus innerhalb der Callbackfunktion bereit und schreibt diesen an die vom IO-Stack vorgegebenen Adressen.
- Mit Verlassen der Callback Funktion ist der Request für die Applikation abgeschlossen.

Das Szenario für einen **asynchronen Request** sieht folgendermassen aus:

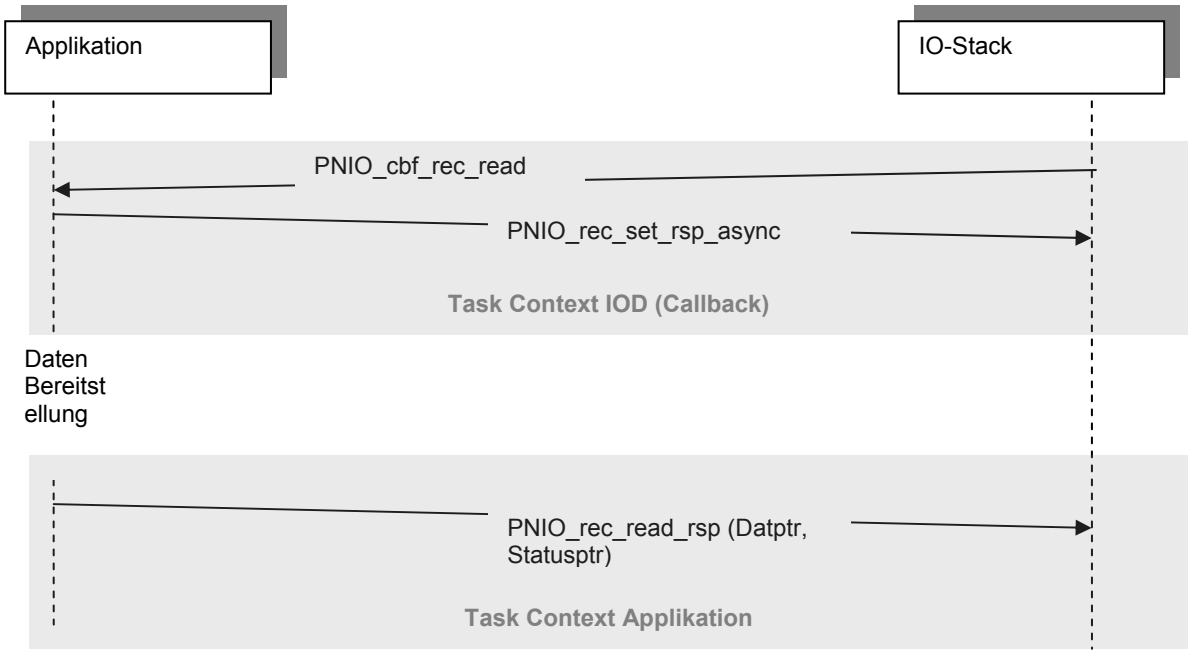


Abbildung 6: asynchrone Read Record Bearbeitung

- Ein Record Read Request vom IO-Controller wird empfangen und vom IO-Stack ausgewertet.
- IO-Stack ruft PNIO_cbf_rec_read() auf.
- Die Applikation ruft aus der Callback Funktion heraus PNIO_rec_set_rsp_async () auf und teilt damit dem Stack mit, dass der Request asynchron erfolgt und nicht mehr innerhalb der Callbackfunktion abgeschlossen werden muss (aber kann). Als Returnwert wird der Applikation ein Requesthandle übergeben, um bei der späteren Übergabe der Daten an den Stack diesen Request referenzieren zu können. Die vom IO-Stack vorgegebenen Adressen für Recorddaten und Fehlerzustand werden anschliessend von der Applikation nicht weiter verwendet.
- Die Applikation stellt die geforderten Record Daten, Datenlänge und Fehlerstatus innerhalb eines beliebigen Taskkontextes an einer beliebigen Speicheradresse bereit.
- Die Applikation ruft PNIO_rec_read_rsp() auf und übergibt dabei Recorddaten, Datenlänge und Fehlerstatus an den Stack.

Synchrone bzw. asynchrone Write Record Requests funktionieren in der gleichen Art und Weise.

4.1.8.1 PNIO_cbf_rec_read

PNIO_cbf_rec_read ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Die Funktion wird vom IO-Stack aufgerufen, wenn der IO-Controller eine Read Record Data Anforderung an das Device gesendet hat. Adressiert wird ein Record über SlotNummer – Subslotnummer - Index. Die Applikation liest die geforderten Daten aus dem Subslot und schreibt sie an die mit pBuf vorgegebene Adresse. Die maximal zulässige Datenlänge in Bytes wird vom Stack in *pBufLen übergeben, die tatsächlich übertragene Byteanzahl wird von der Applikation ebenfalls in *pBufLen zurückgemeldet. PNIO_cbf_rec_read wird vom IO-Stack nur für gesteckte Submodule aufgerufen.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
	Api	Reserviert für spätere Versionen. Derzeit ist nur Api = 0 möglich.
	ArHndl	AR-Nummer
	SessionKey	Session Key
	SequenceNum	Fortlaufende Nummerierung (Lücken in der Nummerierung sind möglich). Kann auf Applikationsebene als Referenz mehrerer quasi gleichzeitig auftretender Requests verwendet werden.
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden
	RecordIndex	Record Index der zu lesenden Daten
	*pBufLen	(nur für synchronen Betrieb, don't care bei asynchronem Betrieb) Zeiger auf die Anzahl der Recorddaten in Bytes. Der Stack übergibt hier die maximal zulässige Recorddaten-Länge, die Applikation gibt die tatsächlich geschriebene Anzahl zurück. **Die vom Stack vorgegebene maximale Datenlänge darf dabei keinesfalls überschritten werden.**
	*pBuffer	(nur für synchronen Betrieb, don't care bei asynchronem Betrieb) Zeiger, wohin die Applikation die gelesenen Recorddaten kopieren muss.
	*pPnioState	(nur für synchronen Betrieb, don't care bei asynchronem Betrieb) Zeiger auf 4 byte PNIO Status, enthält ErrCode, ErrDecode, ErrCode1, ErrCode2, AddValue 1, AddValue 2 gemäß IEC 61158, siehe /2/
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.8.2 PNIO_cbf_rec_write

PNIO_cbf_rec_write ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Die Funktion wird vom IO-Stack aufgerufen, wenn der IO-Controller eine Write Record Data Anforderung an das Device gesendet hat. Adressiert wird ein Record über SlotNummer – Subslotnummer - Index. Die Applikation übernimmt die Recorddaten ab der mit pBuf vorgegebene Adresse. Die Datenlänge in Bytes wird vom Stack in *pBufLen übergeben, die tatsächlich übernommene Byteanzahl wird von der Applikation ebenfalls in *data_length zurückgemeldet. PNIO_cbf_rec_write wird vom IO-Stack nur für gesteckte Submodule aufgerufen.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
	Api	Reserviert für spätere Versionen. Derzeit ist nur Api = 0 möglich.
	ArHndl	AR-Nummer
	SessionKey	Session Key
	SequenceNum	Fortlaufende Nummerierung (Lücken in der Nummerierung sind möglich). Kann auf Applikationsebene als Referenz mehrerer quasi gleichzeitig auftretender Requests verwendet werden.
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	RecordIndex	Record Index der zu lesenden Daten
	*pBufLen	Zeiger auf die Anzahl der Recorddaten in Bytes. Der Stack übergibt hier die maximal zulässige Recorddaten-Länge. Im synchronen Betrieb gibt die Applikation die tatsächlich geschriebene Anzahl zurück. **Die vom Stack vorgegebene maximale Datenlänge darf dabei keinesfalls überschritten werden.**
	*pBuffer	Zeiger auf die Recorddaten.
	*pPnioState	(nur für synchronen Betrieb, don't care bei asynchronem Betrieb) Zeiger auf 4 byte PNIO Status, enthält ErrCode, ErrDecode, ErrCode1, ErrCode2, AddValue 1, AddValue 2 gemäß IEC 61158, siehe /2/
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.8.3 PNIO_rec_set_rsp_async

PNIO_rec_set_rsp_asnyc ()		Funktionsaufruf: Applikation ➔ IO-Stack (darf nur aus PNIO_cbf_rec_read oder PNIO_cbf_rec_write aufgerufen werden)
Hiermit teilt die Applikation aus der Callbackfunktion PNIO_cbf_rec_read() oder PNIO_cbf_rec_write() heraus mit, dass die Bereitstellung der Daten asynchron erfolgt. Wird PNIO_rec_set_rsp_async() nicht aufgerufen, so geht der Stack von einer synchronen Bearbeitung aus, damit ist das Interface kompatibel zu älteren Versionen des Developer's Kit.		
Input	---	
Output	return	Handle, um den Request bei der späteren Übergabe der Daten an den Stack referenzieren zu können

4.1.8.4 PNIO_rec_read_rsp

PNIO_rec_read_rsp		Funktionsaufruf: Applikation ➔ IO-Stack
Nur für den asynchronen Betrieb wird mittels PNIO_rec_read_rsp() ein Record Read Request abgeschlossen. Dabei werden die Daten, der Fehlerstatus sowie die Länge der tatsächlich vom Submodul gelesenen Daten an den Stack übergeben. Die Funktion kann aus einer beliebigen Anwendertask heraus aufgerufen werden.		
Input	pRqHnd	Handle für die Referenzierung des Requests. Der Handle wurde beim vorherigen Aufruf von PNIO_rec_set_rsp_async() als Returnwert an die Applikation übergeben
	pDat	Zeiger auf die von der Applikation bereitgestellten Daten. Der Speicher für die Recorddaten wird im asynchronen Betrieb von der Applikation und nicht wie beim synchronen Betrieb vom IO-Stack vorgegeben.
	NettoDatLength	Länge der tatsächlich vom Submodul gelesenen Recorddaten
	pPnioState	Zeiger vom Typ PNIO_ERR_STAT auf die bereitgestellten Daten, bestehend aus 4 byte PNIO Status (enthält ErrCode, ErrDecode, ErrCode1, ErrCode2), sowie je 2 Byte AddValue 1, AddValue 2 gemäß IEC 61158, siehe /2/. Der Speicher für die Fehlerstatusdaten wird im asynchronen Betrieb von der Applikation und nicht wie beim synchronen Betrieb vom IO-Stack vorgegeben.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.8.5 PNIO_rec_write_rsp

PNIO_rec_write_rsp		Funktionsaufruf: Applikation ➔ IO-Stack
Für den asynchronen Betrieb wird mittels PNIO_rec_write_rsp() ein Record Write Request abgeschlossen. Dabei wird der Fehlerstatus sowie die Länge der tatsächlich ins Submodul geschriebenen Daten an den Stack übergeben. Die Funktion kann aus einer beliebigen Anwendertask heraus aufgerufen werden.		
Input	pRqHnd	Handle für die Referenzierung des Requests. Der Handle wurde beim vorherigen Aufruf von PNIO_rec_set_rsp_async() als Returnwert an die Applikation übergeben
	NettoDatLength	Länge der tatsächlich ins Submodul geschriebenen Recorddaten
	pPnioState	Zeiger vom Typ PNIO_ERR_STAT auf die bereitgestellten Daten, bestehend aus 4 byte PNIO Status (enthält ErrCode, ErrDecode, ErrCode1, ErrCode2), sowie je 2 Byte AddValue 1, AddValue 2 gemäß IEC 61158, siehe /2/. Der Speicher für die Fehlerstatusdaten wird im asynchronen Betrieb von der Applikation und nicht wie beim synchronen Betrieb vom IO-Stack vorgegeben.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.9 Zyklischer Datenaustausch über das Standard-Callbackinterface (SCI)

4.1.9.1 PNIO_initiate_data_read, PNIO_initiate_data_write

PNIO_initiate_data_read()	Funktionsaufruf: Applikation ➔ IO-Stack, synchron

Mit dieser Funktion wird einmalig der Austausch von IO Outputdaten (Daten-Übertragungsrichtung: Controller → Device) zwischen IO-Stack und Applikation für alle AR's (RT, IRT C2, IRT C3) durchgeführt. Das IOD Control Program reserviert beim CM einen Datenpuffer. Daraufhin ruft der IO-Stack für alle Submodule mit Ausgangsdaten, für die eine aktive IO-AR zu einem IO-Controller besteht, die Funktion PNIO_cbf_data_read (). Darin wird die Applikation aufgefordert, die entsprechenden IO Outputdaten vom IO-Stack zu lesen und auf die Ausgänge des Submoduls zu schreiben. Ebenfalls ausgetauscht werden dabei IO Consumer Status und IO Provider Status.

PNIO_initiate_data_read () verhält sich dabei synchron, d.h. erst nach Aufruf aller PNIO_cbf_data_read() Aufrufe kehrt PNIO_initiate_data_read() zurück . Die aktualisierten IO Daten werden mit dem nächsten Übertragungszyklus an den IO-Controller gesendet.

Der Anwender braucht sich dabei um das Pufferhandling nicht zu kümmern, dies erledigt das IOD Control Programm. Er muss lediglich die Funktion PNIO_cbf_data_read () implementieren. **Ein Funktionsrumpf dazu ist im Modul iodapi_event.c hinterlegt**.

Die Datenkonsistenz wird durch Lock-Mechanismen (ERTEC400: max. 254 Byte konsistent) gewährleistet.

Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

PNIO_initiate_data_write()	Funktionsaufruf: Applikation ➔ IO-Stack, synchron

Mit dieser Funktion wird einmalig der Austausch von IO Inputdaten (Daten-Übertragungsrichtung: Device → Controller) zwischen IO-Stack und Applikation für alle AR's (RT, IRT) durchgeführt. Das IOD Control Program reserviert beim CM einen Datenpuffer. Daraufhin ruft der IO-Stack für alle Submodule mit Inputdaten, für die eine aktive IO-AR zu einem IO-Controller besteht, die Funktion PNIO_cbf_data_write (). Darin wird die Applikation aufgefordert, die entsprechenden IO Inputdaten vom Submodul zu lesen und in den vom IO-Stack vorgegebenen Puffer zu schreiben. Ebenfalls ausgetauscht werden dabei IO Consumer Status und IO Provider Status.

PNIO_initiate_data_write () verhält sich dabei synchron, d.h. erst nach Aufruf aller PNIO_cbf_data_write() Aufrufe kehrt PNIO_initiate_data_write () zurück.

Der Anwender braucht sich dabei um das Pufferhandling nicht zu kümmern, dies erledigt das IOD Control Programm. Er muss lediglich die Funktion PNIO_cbf_data_write () implementieren. **Ein Funktionsrumpf dazu ist im Modul iodapi_event.c hinterlegt**.

Die Datenkonsistenz wird durch Lock-Mechanismen (ERTEC400: max. 254 Byte konsistent) gewährleistet.

Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
Output	return	Ausführungsstatus: PNIO_OK, PNIO_NOT_OK

4.1.9.2 PNIO_cbf_data_write, PNIO_cbf_data_read

PNIO_cbf_data_write ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Einlesen der physikalischen Eingangsdaten vom Submodul und Schreiben in den vom IO-Stack vorgegebenen Puffer. Die Funktion wird vom IO-Stack aufgerufen, nachdem die Applikation einen Datenaustausch mit PNIO_initiate_data_write () angestossen hat. Die Applikation muss die physikalischen Eingänge des in Slot/Subslot gesteckten Submoduls lesen und auf die vom Stack vorgebebene Pufferadresse kopieren. Achtung: **Die mit DataLen vorgegebene Länge darf dabei in keinem Fall überschritten werden. Dies liegt in der Verantwortung des Anwenders.**		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	BufLen	Länge der Daten in Byte.
	*pBuffer	Zeiger auf Datenpuffer, dorthin muss die Applikation die IO-Inputdaten kopieren. **Die in DataLen vorgegebene Datenlänge darf dabei keinesfalls überschritten werden.**
	IOremState	Remote Consumerstatus, wurde vom IO-Controller zusammen mit den Ausgangsdaten gesendet. Definiert in iodapi.h sind derzeit PNIO_S_ GOOD, PNIO_S_ BAD, weitere Informationen siehe /1/ und /2/.
Output	Iops	IO Providerstatus. Definiert in iodapi.h sind derzeit derzeit PNIO_S_ GOOD, PNIO_S_ BAD, weitere Informationen siehe /1/ und /2/.

PNIO_cbf_data_read()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Schreiben der IO Output Daten auf die physikalischen Ausgänge des Submoduls. Die Funktion wird vom IO-Stack aufgerufen, nachdem die Applikation einen Datenaustausch mit PNIO_initiate_data_read () angestossen hat. Die Applikation muss die in pData spezifizierten Ausgangsdaten lesen und auf die physikalischen Ausgänge des spezifizierten Submoduls ausgeben. Im Returnwert der Funktion wird der Consumerstatus an den Stack zurückgemeldet. Dieser wird im nächsten zyklischen RT Telegramm an den IO-Controller übertragen und kann dort ausgewertet werden.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	BufLen	Länge der Daten in Byte.
	*pBuffer	Zeiger auf Datenpuffer, von dort muss die Applikation die IO-Outputdaten lesen.
	IOremState	Remote Providerstatus, wurde vom IO-Controller zusammen mit den Ausgangsdaten gesendet. Definiert in iodapi.h sind derzeit PNIO_S_GOOD, PNIO_S_BAD, weitere mögliche Werte siehe /1/ und /2/.
Output	Iocs	IO Consumerstatus. Definiert in iodapi.h sind derzeit PNIO_S_GOOD, PNIO_S_BAD, weitere mögliche Werte siehe /1/ und /2/.

4.1.10 Zyklischer Datenaustausch über das optionale DBA-Interface

Alternativ zum Standard Callbackinterface (SCI) können IO Daten sowie Provider/Consumerstatus auch über das Direct Buffer Access (DBA)- Interface ausgetauscht werden. Dabei wird jeweils der komplette IOCR-Datenblock, welcher die IO Daten, IOPS und IOCS der Submodule enthält, der Applikation zur Verfügung gestellt. Diese extrahiert selber jeweils die für die IO-Daten, IOPS und IOCS für die einzelnen Submodule aus der IOCR. Eine IOCR ist entweder eine Input CR (Device→Controller) oder Output CR (Controller→Device)
Das DBA-Interface beinhaltet folgende Funktionen:

PNIO_dbai_enter()	Sperren des Aufrufs von PNIO_cbf-Funktionen und Sperren des Schreibzugriffs auf interne AR-Verwaltungsdaten, z.B: bei Abbruch der AR
PNIO_dbai_buffer_lock()	Anfordern eines IOCR-Puffers (entweder Input IOCR oder Output IOCR)

PNIO_dbai_buffer_unlock()	Freigeben eines vorher belegten IOCR Puffers nach der Bearbeitung
PNIO_dbai_exit()	Freigabe der stackinternen mit PNIO_dbai_enter() belegten Semaphore

Für den Aufruf von PNIO_dbai_buffer_lock() sowie die Extraktion der IO-Daten und IOPS/IOCS Informationen aus der IOCR sind in der Applikation zusätzliche Informationen notwendig, die vom Stack bei Aufruf der Callback PNIO_ar_info_ind() an die Applikation übergeben werden und dort gespeichert werden müssen. Sie sind gültig, bis die AR wieder abgebaut wird.

Folgende Informationen werden beim Aufruf von PNIO_dbai_buffer_lock übergeben:

- Pointer auf die entsprechende IOCR (Die Applikation darf lesend auf den bei PNIO_cbf_ar_info_ind() erhaltenen IOCR Pointer zugreifen, wenn dieser durch PNIO_dbai_enter() geschützt ist. Sie muss also nicht eine Kopie dieser Daten halten.
- Übertragungsrichtung der IOCR (Input/Output)

Folgende Informationen werden für den Zugriff auf die submodulspezifischen IO-Daten sowie IOPS/IOCS einer **Input** –IOCR (Übertragungsrichtung Device → Controller) benötigt:

- Slotnummer, Subslotnummer der projektierten Input- und Output Submodule
- Lage (Offset, Länge) der IO Daten der projektierten Input-Submodule
- Lage (Offset) der (loca) IOPS der projektierten Inputmodule
- Lage (Offset) der (local) IOCS der projektierten Outputmodule

Folgende Informationen werden für den Zugriff auf die submodulspezifischen IO-Daten sowie IOPS/IOCS einer **Output** –IOCR (Übertragungsrichtung Controller → Device) benötigt:

- Slotnummer, Subslotnummer der projektierten Input- und Output Submodule
- Lage (Offset, Länge) der IO Daten der projektierten Output-Submodule
- Lage (Offset) der (remote) IOPS der projektierten Outputmodule
- Lage (Offset) der (remote) IOCS der projektierten Inputmodule

Die Applikation muss für einen Pufferzugriff auf eine IOCR (Input oder Output) folgende Schritte durchführen:

PNIO_dbai_enter()
Prüfen, ob IOCR noch gültig ist
Wenn gültig
    PNIO_dbai_buffer_lock()
    Puffer bearbeiten
    PNIO_dbai_buffer_unlock()
PNIO_dbai_exit()

4.1.10.1 PNIO_dbai_enter

PNIO_dbai_enter ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Diese Funktion sollte vor PNIO_dbai_buffer_lock() aufgerufen werden, um sowohl den Aufruf von Callbackfunktionen PNIO_cbf_xxx als auch die Veränderung von AR-Verwaltungsdaten durch den Stack zu sperren. Dies ist notwendig, damit bei einem plötzlichen Abbruch der AR nicht diese Informationen gelöscht werden, während die Applikation noch darauf zugreift. Durch Verwendung dieses Locking Mechanismus kann die Applikation lesend direkt auf die bei der Ar-Info Indication übergebenen Pointer zugreifen, die bis zum Abbruch der AR gültig bleiben.		
Input	---	
Output	---	

4.1.10.2 PNIO_dbai_exit

PNIO_dbai_exit ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Freigabe der mit PNIO_dbai_enter() gelockten Ressourcen des PNIO Stack. Diese Funktion muss aufgerufen werden, nachdem der Puffer mit dbai_buffer_unlock() wieder freigegeben wurde. Hinweis: Zwischen PNIO_dbai_enter und PNIO_dbai_exit werden keine Callbackfunktionen PNIO_cbf_xxx() vom Stack aufgerufen. Die Pufferbearbeitung durch die Applikation sollte also möglichst kurz sein.		
Input	---	
Output	---	

4.1.10.3 PNIO_dbai_buffer_lock

PNIO_dbai_buffer_lock ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Anfordern eines IOCR Puffers vom PNIO Stack. Dieser enthält die aktuellen IO-Daten, IOPS und IOCS der projektierten Submodule. Die Daten werden über eine Struktur vom Typ PNIO_BUFFER_LOCK_TYPE übergeben. Die Input-Elemente dieser Struktur erhält die Applikation durch PNIO_cbf_ar_info_ind(). Als Output werden der Zeiger auf den Puffer sowie der Cycle Counter aus dem APDU Status übergeben. Dieser kann bei IRT z.B. bei einem eingestellten Reduction Ratio > 1 zur Ermittlung der Phase dienen, in der die IO Daten tatsächlich übertragen werden.		
Input	pAccBuf	Zeiger auf Struktur vom Typ PNIO_BUFFER_LOCK_TYPE. Diese enthält - AR Handle - Session Key - Übertragungsrichtung (PNIO_IOCR_TYPE_ENUM IocrType) - Offset der zu lockenden IO Daten in IOCR (nur EB400 bei Länge > 255 Byte, sonst 0) - Länge der zu lockenden Daten (EB400: max. 254 Byte auf einmal) - [out] pBuf Zeiger auf den IOCR Puffer - [out] APDU Status, enthält den Cycle Counter (bit 0…15)
Output	Return	PNIO_OK, PNIO_NOT_OK

4.1.10.4 PNIO_dbai_buffer_unlock

PNIO_dbai_buffer_lock ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Damit wird nach der Bearbeitung durch die Applikation ein mit PNIO_dbai_buffer_lock() angeforderter Puffer wieder an den Stack zurückgegeben.		
Input	pAccBuf	Zeiger auf dieselben Daten von PNIO_dbai_buffer_lock
Output	Return	PNIO_OK, PNIO_NOT_OK

4.1.11 Events und Alarme empfangen

4.1.11.1 PNIO_cbf_ar_check_ind

PNIO_cbf_ar_check_ind ()	Funktionsaufruf: IO-Stack ➔ Applikation, synchron

Beim Verbindungsaufbau durch den IO-Controller schickt dieser die mittels HW-Config parametrierte Sollkonfiguration an das Device. Die für die Applikation (eventuell) interessanten Informationen werden dieser mittels der Funktion PNIO_cbf_ar_check_ind() mitgeteilt. Erst danach wird bei einer Differenz von Soll- und Istausbau submodulgranular die Funktion PNIO_cbf_check_ind() aufgerufen, wenn das entsprechende Submodul in Soll- und Istausbau differiert.

PNIO_cbf_ar_check_ind() wird bei jedem Verbindungsaufbau aufgerufen und liefert alle gesteckten Module und Submodule. , während PNIO_cbf_check_ind() nur aufgerufen wird, wenn eine Soll-Istdifferenz der gesteckten Submodule existiert.

Hinweis 1: Solange die Funktion PNIO_cbf_check_ind() nicht verlassen wurde, dürfen Module und Submodule gezogen und gesteckt werden. Ab V3.2.0 ist dies auch direkt in der Callback zulässig und muss nicht mehr in einem anderen Taskcontext erfolgen.

Hinweis 2: Der Compilerschalter PNIO_CHECK_IND_ALL ist ab V3.2.0 nicht mehr notwendig und wurde entfernt.

Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	HostIp	IP-Adresse der Remote Station (in der Regel der IO-Controller)
	ArType	Siehe Struktur cm_ar_type_enum, Werte sind: CM_AR_TYPE_SINGLE CM_AR_TYPE_SINGLE_RTC3 CM_AR_TYPE_CIR CM_AR_TYPE_IOC_REDUNDANT CM_AR_TYPE_IOD_REDUNDANT CM_AR_TYPE_SUPERVISOR
	ArUUID	AR UUID
	ArProperties	Siehe Struktur cm_ar_properties_enum
	CmiObjUUID	Object UUID
	CmiStationNameLength	Länge des Host - Stationsnamens
	pCmiStationName	Zeiger auf den Stationsnamen
	pAr	Zeiger vom Typ PNIO_AR_TYPE auf die Eigenschaften der AR. Diese enthält u.a. jeweils einen weiteren Zeiger auf ein Array der IOCR's sowie der Module Hier werden alle Module angezeigt, um z.B. die Istkonfiguration der projektierten Konfiguration anzupassen.
Output	----	

4.1.11.2 PNIO_cbf_check_ind

PNIO_cbf_check_ind ()	Funktionsaufruf: IO-Stack ➔ Applikation, synchron

Beim Verbindungsaufbau durch den IO-Controller schickt dieser die mittels HW-Config parametrierte Sollkonfiguration an das Device. Der CM vergleicht diese Sollkonfiguration mit seiner Istkonfiguration, die von der Applikation mittels PNIO_mod_plug () und PNIO_sub_plug() an den Stack übergeben wurde. Falls in der Sollkonfiguration ein Modul/Submodul enthalten ist welches nicht auf Deviceseite mittels PNIO_mod_plug () und PNIO_sub_plug () gesteckt wurde, so wird vom Stack für das falsche Submodul die Funktion PNIO_cbf_check_ind() aufgerufen.

Die Applikation muss für den spezifizierten Slot/Subslot die Modul-Identifikation , Submodulidentifikaton sowie den Modul- und Submodul-Status des tatsächlich gesteckten Submodules eintragen und damit die Istkonfiguration an die Sollkonfiguration anpassen. Als Submodul-State sind folgende Werte zurückzugeben

PNIO_SUB_STATE_NO_SUBMODULE, wenn kein Submodul gesteckt ist
PNIO_SUB_STATE__WRONG_SUBMODULE, wenn (irrtümlich) ein falsches Submodul gesteckt wurde
PNIO_SUB_STATE__SUBSTITUTED_SUBMODULE, wenn (korrekterweise) ein Ersatzsubmodul (anderer Typ) gesteckt wurde
PNIO_SUB_STATE__STATE_ PROPER_SUBMODULE , wenn das projektierte Submodul identisch zum gesteckten ist.

Hinweis 1: Solange die Funktion PNIO_cbf_check_ind() nicht verlassen wurde, dürfen Module und Submodule gezogen und gesteckt werden. Ab V3.2.0 ist dies auch direkt in der Callback zulässig und muss nicht mehr in einem anderen Taskcontext erfolgen.

Hinweis 2: PNIO_xxx PROPER_MODULE bzw. PNIO_xxx PROPER_SUBMODULE sind Sonderfälle, die nur beim Ziehen und Stecken während des Betriebes systembedingt auftreten können. In diesem Fall wird eine Check-Indication ausgelöst, obwohl das projektierte Modul mit dem tatsächlich gesteckten Modul übereinstimmt.

Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	Api	Reserviert für spätere Versionen. Derzeit ist nur Api = 0 möglich.
	ArHndl	AR-Nummer
	SessionKey	Session Key
	*pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss PNIO_ADDR_GEO eingetragen werden.
	*pModIdent	Zeiger auf Puffer, in den die Applikation die Modul-Identifikation des tatsächlich gesteckten Moduls schreiben muss
	*pSubIdent	Zeiger auf Puffer, in den die Applikation die Submodul-Identifikation des tatsächlich gesteckten Submoduls schreiben muss.
	*pSubState	Zeiger auf Puffer, in den die Applikation den Status des entspr. Submoduls schreiben muss. Mögliche Werte für Submodule sind: PNIO_SUB_STATE_NO_SUBMODULE PNIO_SUB_STATE_WRONG_SUBMODULE PNIO_SUB_STATE_SUBSTITUTED_SUBMODULE PNIO_SUB_STATE_PROPER_SUBMODULE
Output	----	

4.1.11.3 PNIO_cbf_ar_info_ind

PNIO_cbf_ar_info_ind ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Der Stack teilt der Applikation mit, dass vom IO-Controller ein Verbindungswunsch für eine neue Application-Relation empfangen wurde. Im Übergabeparameter pAR wird ein Zeiger auf eine verkettete Struktur übergeben, welche alle wichtigen Informationen über die aufzubauende AR enthält. Zur späteren Referenzierung der AR wird pAR->ArHndl an die Applikation übergeben, dieses wird auch bei den Funktionen PNIO_cbf_ar_abort_ind(), PNIO_cbf_ar_offline_ind (), PNIO_cbf_apdu_status_ind () und PNIO_cbf_ar_indata_ind () als Parameter verwendet. Damit kann bei mehreren AR's die entsprechende AR bei Bedarf zugeordnet werden. PNIO_cbf_ar_info_ind () hat für die Applikation nur Informationscharakter, d.h. für das Handling des IO-Stacks ist sie nicht zwingend notwendig. Der Anwender kann hier auch eine Leerfunktion programmieren.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	*pAr	Zeiger auf eine Struktur vom Typ PNIO_AR_TYPE mit Informationen über die AR. Der Aufbau der Struktur ist in iodapi.h definiert. Das Element pNextAr der Struktur PNIO_AR_TYPE ist für die Applikation nicht relevant. **Auf die gesamte Struktur darf nur lesend zugegriffen werden, Daten dürfen nicht verändert werden!**
Output	---	

4.1.11.4 PNIO_cbf_ar_in_data_ind

PNIO_cbf_ar_indata_ind ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Der Stack teilt der Applikation durch eine „AR-InData“ - Indication mit, dass der zyklische Datenaustausch begonnen wurde, d.h. dass ein erstes IO-Datentelegramm vom IO-Controller nach ApplicationReady empfangen wurde.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
	ArHndl	Spezifiziert die AR-Nummer, diese wurde mit PNIO_cbf_ar_info_ind() als Element der Struktur PNIO _AR_TYPE übergeben.
	SessionKey	Session Key
Output	---	

4.1.11.5 PNIO_cbf_ar_abort_ind

PNIO_cbf_ar_abort_ind ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Der Stack teilt der Applikation mit, dass ein ABORT-Event aufgetreten ist. Ein Abort Event tritt auf, wenn eine bestehende Verbindung unterbrochen wurde oder explizit vom Controller abgebaut wurde, bevor AR-InData gemeldet wurde.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open () erzeugt wurde.
	ArHndl	Spezifiziert die AR-Nummer, diese wurde mit PNIO_cbf_ar_info_ind() als Element der Struktur PNIO_AR_TYPE übergeben.
	SessionKey	Session Key
	ReasonCode	Beschreibt den Grund für den Verbindungsabbruch, mögliche Reason Codes für einen Abbruch sind in iodapi.h in der enum-Typdefinition PNIO_AR_REASON {..} hinterlegt.
Output	---	

4.1.11.6 PNIO_cbf_ar_offline_ind

PNIO_cbf_ar_offline_ind ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Der Stack teilt der Applikation mit, dass ein Offline-Event aufgetreten ist. Ein Offline Event tritt auf, wenn eine bestehende Verbindung unterbrochen wurde oder explizit vom Controller abgebaut wurde, nachdem bereits AR-InData gemeldet wurde.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	ArHndl	Spezifiziert die AR-Nummer, diese wurde mit PNIO_cbf_ar_info_ind() als Element der Struktur PNIO_AR_TYPE übergeben.
	SessionKey	Session Key
	ReasonCode	Beschreibt den Grund für den Verbindungsabbruch, mögliche Reason Codes für einen Abbruch sind in iodapi.h in der enum-Typdefinition PNIO_AR_REASON {..} hinterlegt.
Output	---	

4.1.11.7 PNIO_cbf_param_end_ind

PNIO_cbf_param_end_ind ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Der Stack teilt der Applikation mit, dass die Parametrierung aller Module abgeschlossen wurde. Die Applikation quittiert diese Funktion durch return (PNIO_TRUE), damit wird automatisch vom Stack die "Application-Ready" Meldung an den Controller übertragen. In diesem Fall darf PNIO_set_appl_state_ready() **nicht** von der Applikation aufgerufen werden. Nur wenn die Applikation zu diesem Zeitpunkt noch nicht fertig ist, kann Sie mit return (PNIO_FALSE) quittieren und muss dann zu einem späteren Zeitpunkt selber die Funktion PNIO_set_appl_state_ready() aufrufen.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	ArHndl	Spezifiziert die AR-Nummer, diese wurde mit PNIO_cbf_ar_info_ind() als Element der Struktur PNIO_AR_TYPE übergeben.
	SessionKey	Session Key
	Api	API Nummer (nur gültig, wenn SubslotNum ungleich 0)
	SlotNum	Slotnummer (nur gültig, wenn SubslotNum ungleich 0)
	SubslotNum	== 0: ParamEnd für alle Module, <> 0: ParamEnd nur für das spezifizierte Modul
Output	return	PNIO_TRUE, wenn die Applikation READY ist (Standardfall) PNIO_FALSE, wenn die Applikation noch nicht READY ist.

4.1.11.8 PNIO_cbf_apdu_status_ind

PNIO_cbf_apdu_status_ind ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Der Stack teilt der Applikation mit, dass sich der Status des IO-Controllers geändert hat.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	ArHndl	Spezifiziert die AR-Nummer, diese wurde mit PNIO_cbf_ar_info_ind() als Element der Struktur PNIO_AR_TYPE übergeben.
	SessionKey	Session Key
	ApduStatus	neuer Status des IO-Controllers. Mögliche Werte sind PNIO_EVENT_APDU_STATUS_STOP PNIO_EVENT_APDU_STATUS_RUN PNIO_EVENT_APDU_STATUS_STATION_OK PNIO_EVENT_APDU_STATUS_STATION_PROBLEM PNIO_EVENT_APDU_STATUS_PRIMARY PNIO_EVENT_APDU_STATUS_BACKUP und in der enum-Typdefinition PNIO_APDU_STATUS_IND hinterlegt.
Output	---	

4.1.11.9 PNIO_cbf_edd_link_state_ind

PNIO_cbf_link_state_ind ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Der Stack teilt der Applikation mit, dass sich der Linkstatus eines Ethernetports geändert hat.		
Input	pLinkStat	Pointer auf Struktur vom Typ PNIO_EDD_LINKSTAT_TYPE, diese enthält folgende Elemente mit den dazugehörigen Wertebereichen: - PortID Port Nummer, 1…EDD_MAX_PORT_CNT - State EDD_LINK_UP, EDD_ LINK_DOWN, EDD_LINK_UP_CLOSED, EDD_LINK_UP_DISABLED - Speed EDD_LINK_SPEED_10, EDD_LINK_SPEED_100, EDD_LINK_SPEED_1000 - Mode EDD_LINK_MODE_HALF, EDD_LINK_MODE_FULL
Output	return	PNIO_OK

4.1.11.10 PNIO_cbf_ pd_sync_info

PNIO_cbf_pd_sync_info ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Der Stack teilt der Applikation mit, dass sich der Synchronisationsstatus des Gerätes geändert hat.		
Input	IsRateValid	1: Es wurden Sync-Frames empfangen 0: Es wurden keine Sync-Frames empfangen
	SyncLedInfo	PNIO_PD_SYNC_INFO_NOT_SYNC: IRT ist konfiguriert, aber das Gerät ist nicht synchron ode res stehen entsprechende RT-Class 3 Diagnosen an. PNIO_PD_SYNC_INFO_SYNC: Das Gerät ist synchron in Class 3.
Output	return	PNIO_OK

4.1.12 Control-Funktionen

4.1.12.1 PNIO_set_dev_state

PNIO_set_dev_state ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Setzt den Zustand des Device auf OPERATE / CLEAR und auf STATION_OK / STATION_PROBLEM		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	DevState	neuer Status des Device. Mögliche Werte sind PNIO_DEVSTAT_OPERATE PNIO_DEVSTAT_CLEAR Hinweis: die Defines früherer Versionen PNIO_DEVSTAT_CLEAR_STATION_PROBLEM und PNIO_DEVSTAT_OPERATE_STATION_PROBLEM wurden gestrichen. Der Problem-Indikator wird vom Stack automatisch gesetzt.
Output	return	PNIO_OK Auftrag wurde ordnungsgemäß ausgeführt PNIO_NOT_OK Fehler bei Auftragsausführung aufgetreten

4.1.12.2 PNIO_device_ar_abort

PNIO_ar_abort ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Diese Funktion bricht die Application Relation ab. Dadurch wird die Verbindung zwischen dem IO-Controller und dem IO-Device abgebaut. Für den Fall, dass Datenaustausch zwischen IO-Controller und IO-Device stattfindet (PNIO_cbf_ar_indata_ind-Callback wurde von der Schnittstelle aufgerufen), wird von der IO-Schnittstelle PNIO_cbf_ar_offline_ind-Callback aufgerufen. Für den Fall, das noch kein Datenaustausch zwischen IO-Controller und IO-Device stattgefunden hat (PNIO_cbf_ar_indata_ind-Callback wurde von der Schnittstelle noch nicht aufgerufen) wird von der IO-Schnittstelle der PNIO_cbf_ar_abort_ind -Callback aufgerufen.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	ArHndl	Spezifiziert die AR-Nummer, diese wurde mit PNIO_cbf_ar_info_ind() als Element der Struktur PNIO_AR_TYPE übergeben.
	Session Key	Session Key
Output	return	PNIO_OK Auftrag wurde ordnungsgemäß ausgeführt PNIO_NOT_OK Fehler bei Auftragsausführung aufgetreten

4.1.13 Fehlerbehandlung

4.1.13.1 PNIO_get last_error

PNIO_get_last_error ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Liest den letzten, bei einem Aufruf einer PNIO_xxxx() Funktion aufgetretenen Fehler, der von CM in der Response gemeldet wurde und stellt diesen der Applikation zur Verfügung. Da die meisten Funktionen nur eine Sammel-Fehlerinformation (OK, NOT_OK) zurückliefern, kann hier bei Bedarf eine detailliertere Fehlerinformation angefordert werden. Wenn ein Fehler aufgetreten ist, so wird dieser gespeichert und bleibt so lange erhalten, bis ein neuer Fehler aufgetreten ist. Ein korrekt und mit PNIO_OK abgeschlosser Funktionsaufruf von PNIO_xxxxx löscht nicht den gespeicherten Fehlerwert, d.h. PNIO_get_last_error sollte nur aufgerufen werden, wenn ein vorheriger Auftrag mit Fehler, also einem Wert ungleich PNIO_OK, quittiert wurde.		
Input	----	
Output	return	Die möglichen Werte sind in der enum-Typdefinition PNIO_ERR_ENUM hinterlegt: PNIO_OK PNIO_ERR_xxxxxx

Die Definition PNIO_ERR_ENUM ist im File pnioerrx.h hinterlegt.

4.1.13.2 PNIO_log

PNIO_log ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron
Hiermit teilt der Stack der Applikation mit, dass ein Fehler oder Logging Auftrag stattgefunden hat. Es wird die Fehlerklasse (Fatal Error, Error, Logging,..) mitgeteilt sowie eine Referenz auf Source File (Package ID, Module ID) und die Linenumber im Code. Die Applikation kann z.B. abhängig von der Fehlerklasse Fehlerbehandlungen anstossen.		
Input	DevHndl	Device-Handle, welches vom Stack mit PNIO_device_open() erzeugt wurde.
	ErrLevel	Kennzeichnet die Fehlerklasse. Die Logging Level sind in folgende Ebenen unterteilt: PNIO_LOG_DEACTIVATED PNIO_LOG_CHAT PNIO_LOG_CHAT_HIGH PNIO_LOG_NOTE PNIO_LOG_NOTE_HIGH PNIO_LOG_WARNING PNIO_LOG_WARNING_HIGH PNIO_LOG_ERROR PNIO_LOG_ERROR_FATAL
	PackId	Kennzeichnet das Paket, in dem der Fehler aufgetreten sind. Definierte Werte sind PNIO_PACKID_ACP PNIO_PACKID_CLRPC PNIO_PACKID_CM PNIO_PACKID_DCP PNIO_PACKID_EDD PNIO_PACKID_GSY PNIO_PACKID_IOD PNIO_PACKID_LLDP PNIO_PACKID_MRP PNIO_PACKID_NARE PNIO_PACKID_OHA PNIO_PACKID_OS PNIO_PACKID_POF PNIO_PACKID_SOCK PNIO_PACKID_BSPADAPT PNIO_PACKID_REMA PNIO_PACKID_OTHERS
	ModId	Kennzeichnet zusammen mit PackId das Modul, indem der Fehler aufgetreten ist. Die Modul-Id ist indeutig innerhalb eines Paketes.
	LineNum	Line Number, in dem der Fehler aufgetreten ist.
Output	---	

4.1.13.3 PNIO_set_log_level

PNIO_set_log_level ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Setzen des Logging Levels für ein spezielles Paket. Es werden nur die Meldungen ausgegeben, deren Level größer oder gleich dem eingestellten Loglevel ist. Bsp: Nache Setzen von PNIO_LOG_IMPORTANT werden für dieses Paket alle important Loggings (ILOG) sowie Fehler (ELOG, FATAL_ERROR) ausgegeben. Hinweis: Aus Laufzeit- und Speicherplatzgründen sind derzeit nur ELOGs und FATALs per Compilerschalter in den Code hineincompilert. Alle LOG, MLOG und ILOG von CM, CLRPC, ACP, DCP und SOCK sind im compilierten Code nicht enthalten. Für deren Aktivierung muss in compiler.h #define *DEBUG*_LOGGING_xxx aktiviert werden. **Achtung: Logging Ausgaben können die Echzeitfähigkeit des Systems negativ beeinflussen.**		
Input	NewLogLevel	Kennzeichnet die Fehlerklasse. Die Logging Level sind in folgende Ebenen unterteilt: PNIO_LOG_DEACTIVATED PNIO_LOG_CHAT PNIO_LOG_CHAT_HIGH PNIO_LOG_NOTE PNIO_LOG_NOTE_HIGH PNIO_LOG_WARNING PNIO_LOG_WARNING_HIGH PNIO_LOG_ERROR PNIO_LOG_ERROR_FATAL
	PackId	Kennzeichnet das Paket, in dem der Fehler aufgetreten sind. Definierte Werte sind PNIO_PACKID_ACP PNIO_PACKID_CLRPC PNIO_PACKID_CM PNIO_PACKID_DCP PNIO_PACKID_EDD PNIO_PACKID_GSY PNIO_PACKID_IOD PNIO_PACKID_LLDP PNIO_PACKID_MRP PNIO_PACKID_NARE PNIO_PACKID_OHA PNIO_PACKID_OS PNIO_PACKID_POF PNIO_PACKID_SOCK PNIO_PACKID_BSPADAPT PNIO_PACKID_REMA PNIO_PACKID_OTHERS
Output	---	

4.1.14 Sonstige Funktionen

4.1.14.1 PNIO_print

PNIO_print ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron Applikation ➔ Applikation, synchron
Zentrale Behandlungsfunktion für alle Print-Ausgaben. Die Daten werden in standardisierter Form, bestehend aus einem Formatstring und einer Argumentenliste, übergeben. Der Anwender kann damit in beliebiger Weise weiterverfahren. Bei entsprechender Unterstützung durch das Betriebssystem bzw. Standard C-Library kann zum Beispiel eine printf-Ausgabe mittels der Funktion vfprintf(stdout, fmt, argptr); durchgeführt werden. Siehe dazu Beispielcode der Funktion PNIO_Printf (..) im Modul PNIO_Log.c.		
Input	*fmt	Formatstring, wie bei Printf
	argptr	Argumentenliste im standardisierten Format (va_list).
Output	---	

4.1.14.2 PNIO_printf

PNIO_printf ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron Applikation ➔ Applikation, synchron
Zentraler Ersatz für eine printf () –Funktion. Zur Weitergabe der variablen Parameter werden diese in einen Formatstring und eine Argumentenliste vom Typ va_list gewandelt und an die unterlagerte und zentrale Ausgabefunktion PNIO_print() weitergeleitet.		
Input	*fmt	Formatstring, wie bei Printf
	x, y, ...	Variable Argumentenliste wie bei printf()
Output	---	

4.1.14.3 PNIO_TrcPrintf

PNIO_TrcPrintf ()		Funktionsaufruf: IO-Stack ➔ Applikation, synchron Applikation ➔ Applikation, synchron
Zentrale Ausgabe von Meldungen auf PNIO_print() oder zur späteren Auswertung in als ASCII Zeichen einen Umlaufpuffer. Zur Weitergabe der variablen Parameter werden diese in einen Formatstring und eine Argumentenliste vom Typ va_list gewandelt und an die unterlagerte und zentrale Ausgabefunktion PNIO_print() oder TrcPrint () weitergeleitet. Der Anwender kann den Umlaufpuffer im Debugger auswerten oder mittels von Ihm zu implementierender Funktionen nach Belieben exportieren (z.B. über TCP/IP o.ä.).		
Input	*fmt	Formatstring, wie bei Printf
	x, y, ...	Variable Argumentenliste wie bei printf()
Output	---	

4.1.14.4 PNIO_get_version

PNIO_get_version ()		Funktionsaufruf: Applikation ➔ IO-Stack, synchron
Auslesen der Versionskennung des Devkit.		
Input	*pVersion	Zeiger auf die Versionskennung vom Typ PNIO_VERSION. Die Funktion kopiert die Version an die vorgegeben Adresse.
Output	return	PNIO_OK

4.2 Lower Layer Schnittstellenfunktionen zum Board Support Package

4.2.1 BSP Funktionen für alle Plattformen

4.2.1.1 Bsp_Init

Bsp_Init ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Init Funktion, wird vom Stack im Anlauf aufgerufen. Hier sind evt. notwendige Initialisierungen für das BspAdapt-interface einzutragen.		
Input	---	
Output	return	PNIO_OK

4.2.1.2 Bsp_GetMacAddr

Bsp_GetMacAddr ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Auslesen der lokalen Geräte-Macadresse vom Controller		
Input	*pDevMacAddr	Puffer für die 6 Byte lange MAC Adresse. Die Funktion schreibt anschließend die MAC-Adresse in den vorgegebenen Puffer
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.1.3 Bsp_GetPortMacAddr

Bsp_GetPortMacAddr ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Auslesen der lokalen Port-Macadresse vom Controller. Neben der Geräte-Macadresse muss es bei PROFINET für jeden Port eine weitere Macadresse geben.		
Input	*pPortMacAddr	Puffer für die 6 Byte lange MAC Adresse. Die Funktion schreibt anschließend die MAC-Adresse in den vorgegebenen Puffer
	PortNum	Portnummer (1…N, N = Portanzahl), für die die Portmacadresse gelesen werden soll.
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.1.4 Bsp_SetIpAddr

Bsp_SetIpAddr ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Wenn über Ethernet neue Ethernetparameter (IP-Adresse, Subnet-Mask, Router Adr.) für das Device vergeben werden sollen, so wird vom IO-Stack die Funktion Bsp_SetIpAddr () aufgerufen. Die Applikation muss die Parameter im TCP/IP Stack umstellen. Sie muss weiterhin die Parameter in einem nichtflüchtigen Speicher ablegen, wenn der Parameter remanent ungleich Null ist. Beim nächsten Systemanlauf werden die Parameter von der Applikation aus dem nichtflüchtígen Speicher gelesen und an mit der Funktion PNIO_SetEthPar () wieder an den Stack übergeben.		
Input	NewIpAddr	neue einzustellende IP-Adresse
	SubnetMask	Subnetz-Maske
	DefRouterAddr	default Router Adresse
	remanent	ungleich 0: Daten müssen remanent gespeichert werden 0: Wert darf nicht remanent gespeichert werden
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.1.5 Bsp_EbSetLed (Implementierung optional)

Bsp_EbSetLed ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Optionale Funktion zum Ein/Ausschalten der LEDs (rot, grün) auf dem Evaluation board. Die Funktion wird nicht im PNIO Stack selbst, sondern lediglich in der Beispielapplikation verwendet, um Betriebsbereitschaft und zykl. Datenaustausch anzuzeigen.		
Input	Led	Spezifiziert die rote oder grüne LED (EB_LED_GREEN, EB_LED_RED)
	Val	1: LED einschalten, 2: LED ausschalten
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.2 BSP Funktionen nur für DK_SW (Standard-Ethernetcontroller-Plattformen)

4.2.2.1 Bsp_MC_Enable

Bsp_MC_Enable ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Fügt eine Multicast Ethernetadresses hinzu		
Input	*pMac	Pointer auf 6 Byte lange Ethernetadresse
Output	return	LSA_OK LSA_RET_ERR_PARAM

4.2.2.2 Bsp_MC_Disable

Bsp_MC_Disable ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Entfernt eine Multicast Ethernetadresses		
Input	*pMac	Pointer auf 6 Byte lange Ethernetadresse
Output	return	LSA_OK LSA_RET_ERR_PARAM

4.2.2.3 Bsp_GetPDevLinkMode

Bsp_GetPDevLinkMode		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Abfragen des Linkmodues und PHY-Types		
Input	Port	Portnummer (1…N, N = Portanzahl)
	pLinkMode	Speicheradresse, in der das BSP den eingestellten Linkmode an den PNIO Stack zurückgibt. Zulässige Werte siehe Parameter LinkMode in der Funktion Bsp_SetPDevLinkMode().
	pPhyPower	Speicheradresse, in der das BSP den eingestellten PHY-Powermodus an den PNIO Stack zurückgibt. Zulässige Werte siehe Parameter PhyPower in der Funktion Bsp_SetPDevLinkMode().
Output	return	LSA_OK LSA_RET_ERR_PARAM

4.2.2.4 Bsp_SetPDevLinkMode

Bsp_SetPDevLinkMode ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Setzen des Link-Modus und der PHY-Power.		
Input	Port	Portnummer (1…N, N = Portanzahl)
	LinkMode	Folgende Werte sind zulässig: PDEV_LINK_AUTONEG PDEV_LINK_10MBIT_HALF PDEV_LINK_10MBIT_FULL PDEV_LINK_100MBIT_HALF PDEV_LINK_100MBIT_FULL PDEV_LINK_1GBIT_HALF PDEV_LINK_1GBIT_FULL
	PhyPower	Folgende Werte sind zulässig: PDEV_PHY_POWER_ON PDEV_PHY_POWER_OFF PDEV_PHY_POWER_UNCHANGED
Output	return	LSA_OK LSA_RET_ERR_PARAM

4.2.2.5 Bsp_ReinitTransmitUnit

Bsp_ReinitTransmitUnit ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Reinitialisierung der Sendehardware		
Input	-----	
Output	return	LSA_OK

Empfangen von Ethernet telegrammen:

4.2.2.6 ENIC_RecDataX

ENIC_RecDataX ()		Funktionsaufruf: BSP ➔ IO-Stack, synchron
ENIC_RecDataX () ersetzt die bisherige Funktion ENIC_RecData (), welche nur noch aus Kompatibilitätsgründen in Altsystemen zu verwenden ist. In ENIC_RecDataX wurde ein zusätzlicher Parameter PortID definiert, ansonsten ist das Verhalten beider Funktionen identisch. Die Funktion wird vom BSP aufgerufen, wenn ein neues Ethernet Telegramm vom Controller empfangen wurde. Anschließend wird anhand des Telegrammtyps geprüft, ob das Telegramm an den EDD oder den TCP/IP Stack weitergeleitet werden muss (Checker-Funktionalität). Im Returnwert wird dem BSP mitgeteilt, ob er das Telegramm an den TCP IP Stack weiterleiten muss oder ob das Telegramm zu verwerfen ist und der Puffer vom BSP wieder freizugeben ist.		
Input	*pRxBuf	Zeiger auf Empfangspuffer
	Datlen	Länge des empfangenen Telegramms in Byte (ohne FCS)
	PortID	Portnummer (1…x) des Ports, auf dem das Telegramm empfangen wurde. Bei einem System mit nur einem Port ist PNIO_DEFAULT_PORT_ID zu verwenden.
Output	return	TELEGRAM_FORWARD weiterleiten an TCP/IP Stack TELEGRAM_DISCARD BSP muss Telegramm verwerfen

Die Funktion ENIC_RecData sowie die Rückmeldewerte TELEGRAM_FORWARD und TELEGRAM_DISCARD sind definiert in enic.h.

Senden von Ethernet telegrammen:

4.2.2.7 Bsp_SendData

Bsp_SendData ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Die Funktion wird vom IO-Stack aufgerufen, wenn ein Telegramm zum Senden an den Ethernetcontroller geschickt werden soll.		
Input	pBuffer	Zeiger auf Sendepuffer
	DatLen	Länge des zu sendenden Telegramms in Byte
	Client	CLIENT_EDD wenn aufrufendes Progr. der EDD ist CLIENT_TCPIP_STACK wenn aufrufendes Progr. der TCP/IP Stack ist
	PortID	Portnummer (1…x) des Ports, auf dem das Telegramm empfangen wurde. Bei einem System mit nur einem Port ist PNIO_DEFAULT_PORT_ID zu verwenden.
Output	return	LSA_OK, LSA_NOT_OK

4.2.2.8 Enic_SendDat_Conf

Enic_SendDat_Conf ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
Mit dieser Funktion wird dem EDD mitgeteilt, dass das zu sendende Telegramm abgeschickt wurde. Der EDD gibt daraufhin den Telegrammspeicher wieder frei. Enic_SendDat_Conf() wird in Bsp_SendData () aufgerufen, nachdem das Telegramm mit Bsp_SndDatReq() an das BSP übergeben wurde. Dies kann bei Bedarf vom Anwender geändert werden.		
Input	pTxBuf	Zeiger auf den Sendepuffer
Output	return	LSA_OK, LSA_NOT_OK

4.2.3 IP Stack Lower Layer Adaption (nur für ERTEC-Plattform)

Das Lower Layer Interface eines beliebigen TCP/IP Stacks kann über folgende Funktionen an PROFINET angebunden werden:

IpAdapt_Start	Initialisierungsfunktion für das IP Lower Layer Adapter
IpAdapt_Stop	Close-Funktion für das IP Lower Layer Adapter
IpAdapt_AllocSendBuffer	Allokieren eines Sendepuffers von PROFINET
IpAdapt_SendNrt	Übergabe eines zu sendenden IP Telegramms an PROFINET
IpAdapt_cbf_RecvNrt	Übergabe eines empfangenen IP Telegramms von PROFINET

4.2.3.1 IpAdapt_Start

IpAdapt_Start ()		Funktionsaufruf: IO-Stack ➔ BSP, synchron
In dieser Funktion wird die Initialisierung des TCP/IP Adapters durchgeführt. Sie wird im Anlauf vom PROFINET IO Stack aufgerufen und muss vom Anwender implementiert werden.		
Input	---	
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.3.2 IpAdpat_Stop

IpAdapt_Stop()		Funktionsaufruf: : IO-Stack ➔ BSP, synchron
In dieser Funktion wird das TCP/IP Adapter geschlossen. Die Funktion wird vom PROFINET IO Stack aufgerufen. Hier können vom Anwender für das Schliessen des Adapters notwendige Funktionalitäten implementiert werden.		
Input	---	
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.3.3 IpAdapt_AllocSendBuffer

IpAdapt_AllocSendBuffer (void** ppSendBuf, LSA_UINT32 frameLen) ()	Funktionsaufruf: TCP/IP-Stack ➔ IO-Stack, synchron

Bevor der TCP/IP Stack ein Ethernettelegramm über PROFINET versendet, muss er zunächst einen Datenpuffer für dieses Telegramm von PROFINET anfordern und diesen füllen. Das Allokieren wird mit dieser Funktion durchgeführt. Vorgegeben wird vom aufrufenden Programm die geforderte Pufferlänge sowie ein Zeiger, wohin PROFINET die Pufferadresse schreiben soll (ähnlich wie bei OsAlloc).

Input	ppSendBuf	Zeiger, wohin PROFINET die Adresse des allokierten Telegrammpuffers schreiben soll
	FrameLen	Länge des allokierten Puffers
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.3.4 IpAdapt_SendNrt

IpAdapt_SendNrt ()	Funktionsaufruf: TCP/IP-Stack ➔ IO-Stack, synchron

Mit dieser Funktion kann der TCP/IP Stack ein Ethernettelegramm versenden, dessen Puffer er sich vorher mit IpAdapt_AllocSendBuffer() allokiert und mit Daten gefüllt hat.

Input	pSendBuf	Zeiger auf den zu versendenden Puffer
	FrameLen	Länge des zu versendenden Telegramms
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.3.5 IpAdapt_cbf_RecvNrt

IpAdapt_cbf_RecvNrt ()	Funktionsaufruf: IO -Stack ➔ TCP/IP -Stack, synchron

Diese Callbackfuntion wird vom PROFINET Stack aufgerufen, wenn ein neues IP Telegramm empfangen wurde. Es wird das empfangene Telegramm übergeben. Der Empfangspuffer wird vom Stack bereitgestellt. Das Telegramm muss bei der Übergabe kopiert werden, da Profinet nachRetun von IpAdapt__cbf_RecvNrt() diesen Puffer wieder freigibt.

Input	pRecBuf	Zeiger auf den Empfangspuffer
	FrameLen	Länge des empfangenen Telegramms
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.4 Speicherung nichtflüchtiger Daten

Folgende Informationen (NV-Datentypen) müssen auf einem PNIO Device in einem nichtflüchtigen Speicher abgelegt werden:

- Gerätename
- IP Suite (IP-Adresse, Subnet-Mask, Default Router Adresse)
- Records des Physical Device (PDEV), die bei Verbindungsaufbau vom PNIO Controller geschrieben wurden

Diese NV-Datentypen werden beim nächsten Hochlauf aus dem nichtflüchtigen Speicher gelesen und an den PROFINET IO Stack übergeben.

Für diese Funktionalität stehen im PROFINET Anwenderbeispiel folgende Beispielfunktionen zur Verfügung:

Bsp_nv_data_clear () Rücksetzen aller NV-Datentypen auf Werkseinstellung

Bsp_nv_data_store ()	Speichern eines NV-Datentyps im nichtflüchtigen Speicher
Bsp_nv_data_restore ()	Rücklesen eines NV-Datentyps im nichtflüchtigen Speicher
Bsp_nv_data_memfree	Freigabe des von Bsp_nv_data_restore allokierten Speichers

Die oben genannten Funktionen werden nicht vom PROFINET Stack selber, sondern nur innerhalb des Anwenderbeispiels für das Handling der nichtflüchtigen Daten verwendet. Der Anwender kann die definierte NV-Daten-Schnittstelle also in seiner Software als Beispielvorlage übernehmen, aber auch völlig anders gestalten.

4.2.4.1 Bsp_nv_data_clear

Bsp_nv_data_clear()		Funktionsaufruf: IO Stack ➔ Bsp, synchron
Die Funktion wird von der Kundenapplikation aufgerufen, wenn alle nichtflüchtigen NV-Datentypen (non volatile data) auf die Werkseinstellungen zurückgesetzt werden sollen. Die Funktion kann z.B. direkt in der Funktion PNIO_reset_factory_settings aufgerufen werden, so dass sie automatisch ausgeführt wird, wenn von einem Engineering System das Rücksetzen auf Werkseinstellungen angefordert wird. Die Funktion muss vom Anwender implementiert werden.		
Input	---	

Output	return	PNIO_OK, PNIO_NOT_OK

4.2.4.2 Bsp_nv_data_store

Bsp_nv_data_store()		Funktionsaufruf: IO Stack ➔ Bsp, synchron
Die Funktion wird von der Kundenapplikatioin aufgerufen, wenn ein NV-Datentyp im nichtflüchtigen Speicher abgelegt werden soll. Dies kann z.B. ein Gerätename, eine IP Suite oder die Summe aller PDEV-Records sein. Dabei wird der Datentyp, der Zeiger auf die zu speichernden Daten und die Länge der zu speichernden Daten übergeben. Die Funktion muss vom Anwender implementiert werden.		
Input	NvDataType	Spezifiert den NV-Datentyp, zulässig sind folgende Typen: PNIO_NVDATA_DEVICENAME, PNIO_NVDATA_IPSUITE, PNIO_NVDATA_PDEV_RECORD
	pMem	Zeiger auf die zu speichernden Daten
	MemSize	Länge der zu speichernden Daten
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.4.3 Bsp_nv_data_restore

Bsp_nv_data_restore()		Funktionsaufruf: Applikation ➔ Bsp, synchron
Die Funktion wird von der Kundenapplikation im Anlauf aufgerufen, um die nichtflüchtigen Daten wie Gerätename, IP-Suite und PDEV-Records vom NV-Speicher zu lesen und anschliessend an den PNIO Stack zu übergeben. Die Funktion liefert immer einen gültigen Datentyp zurück. Sollten keine gültigen Daten im NV-Speicher vorliegen, so werden die Werkseinstellungen zurückgeliefert. Die Funktion liefert einen Zeiger zurück, wohin sie die gewüschten Daten kopiert hat. Da das aufrufende Programm im Vorfeld die Länge dieser Daten (z.B. bei PDEV-Records) nicht kennen kann, wird der Speicher dazu von Bsp_nv_data_restore () mit OsAlloc allokiert und **muss nach Gebrauch vom aufrufenden Programm** (also von der Applikation) mit Bsp_nv_data_memfree wieder freigegeben werden.		
Input	NvDataType	Spezifiert den NV-Datentyp, zulässig sind folgende Typen: PNIO_NVDATA_DEVICENAME, PNIO_NVDATA_IPSUITE, PNIO_NVDATA_PDEV_RECORD
	ppMem	Zeiger, wohin die Daten von Bsp_nv_data_restore kopiert wurden. Der Speicher muss nach Gebrauch vom aufrufenden Programm durch Aufruf von Bsp_nv_data_memfree() wieder freigegeben werden.
	MemSize	Länge der zu speichernden Daten
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.4.4 Bsp_nv_data_memfree

Bsp_nv_data_memfree()		Funktionsaufruf: Applikation ➔ Bsp, synchron
Die Funktion muss aufgerufen werden, um den beim Aufruf von Bsp_nv_data_restore() erhaltenen Datenblock nach Gebrauch wieder freizugeben. Dieser Block darf **nicht** mit OsFree() freigegeben werden.		
Input	pMem	Zeiger auf die von Bsp_nv_data_restore erhaltenen Daten. .
Output	return	PNIO_OK, PNIO_NOT_OK

4.2.5 Anbindung der ERTEC Switch Interrupts (nur für ERTEC Plattformen)

Der im ERTEC 400/200 integrierte Switch besitzt je einen hochprioren und einen niederprioren Interrupt, die an den PROFINET IO Stack angebunden werden müssen. In PROFINET IO Stack sind dazu folgende Interrupthandler enthalten, die bei Auftreten des entsprechenden Interrupts aufgerufen werden:

Interrupt (IRQ)	**Funktion**	**Aufzurufender Handler**
12	Hochpriorer Switch Interrupt	Bsp_ErtecIntSwiHandlerH ()
13	Niederpriorer Switch Interrupt	Bsp_ErtecIntSwiHandlerL ()

Um die o.g. Handler an die Interrupts zu binden, ruft der Stack folgende Interrupt-Anmeldefunktion auf, die vom Anwender zu implementieren ist:

Bsp_ErtecSwiIntConnect ()		Funktionsaufruf: IO Stack ➔ Bsp, synchron
Diese Funktion bindet je einen Interrupthandler an den hoch.- und niederprioren Switch Interrupt des ERTEC 400/200. Die Funktion wird vom PROFINET IO Stack aufgerufen und muss vom Anwender implementiert werden		
Input	pErtecSwiIntH	Adresse des Interrupthandlers für den hochprioren Switch Interrupt. Der Zeiger ist vom Typ PNIO_CBF_ERTEC_SWI_INT_H
	pErtecSwiIntL	Adresse des Interrupthandlers für den niederprioren Switch Interrupt. Der Zeiger ist vom Typ PNIO_CBF_ERTEC_SWI_INT_L
Output	return	PNIO_OK, PNIO_NOT_OK

4.3 Schnittstelle zum Betriebssystem

Die folgenden Schnittstellenfunktionen abstrahieren eine konkrete Betriebssystemschnittstelle. Sie müssen vom Anwender an das spezielle Betriebssystem angepasst werden. In vielen Fällen kann der Aufruf direkt auf einen Betriebssystemaufruf abgebildet werden. Alle Betriebssystemabstraktionsfunktionen sind im Modul **xxx_OS.C** implementiert. Alle notwendigen Schnittstellendefinitionen und Defines dazu sind in **OS.H** enthalten.

4.3.1 Verwaltung von Ressourcen

Die Verwaltung von Ressourcen und die Referenzierung von Ressourcen sind in den Betriebssystemen oft sehr unterschiedlich ausgelegt. Manche liefern z.B. als Referenz auf eine Ressource wie Thread, Mailbox etc. einen Pointer auf diese Ressource, andere einen beliebigen Index zurück. Aus diesem Grunde werden in der OS-Abstraktionsschnittstelle eigene Indizes generiert, so dass die tatsächlichen Betriebssystemreferenzen für die Applikation transparent sind. Die generierten ID's liegen in einem fortlaufenden Wertebereich von 0....N, so dass die erzeugte ID direkt als Referenz auf den internen Verwaltungsblock dienen kann und damit ein Zugriff einfach und schnell erfolgt.

4.3.2 Beschreibung der zu portierenden OS - Funktionen

4.3.2.1 OsInit ()

OsInit ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Die Funktion wird einmalig im Anlauf vom IO-Stack zur Initialisierung der Betriebssystem-Abstraktionsschnittstelle aufgerufen. OsInit() muss vor Aufruf aller anderen Osxxx-Funktionen abgeschlossen sein.		
Input	---	
Output	return	PNIO_OK,PNIO_NOT_OK

4.3.2.2 OsAllocFX ()

OsAllocFX()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Allokierfunktion für dynamischen Speicher. Der Speicher wird nicht vorbelegt.		
Input	ppMem	Zeiger, wohin die Adresse des allokierten Speichers zu schreiben ist
	Length	Länge des Speichers in Byte
	PoolID	Zur Laufzeitoptimierung können (aber müssen nicht) verschiedene Pools vorgegeben werden. Mögliche Defines sind MEMPOOL_DEFAULT MEMPOOL_FAST MEMPOOL_CACHED MEMPOOL_UNCACHED MEMPOOL_RX_TX_BUF Hinweis: in der Systemanpassung des Anwenderbeispiels dürfen alle o.g. Pools gecached sein.
Output	return	PNIO_OK,PNIO_NOT_OK

4.3.2.3 OsCalloc ()

OsCalloc()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Allokiert ein Feld, bestehend aus N Elementen der Grösse SizeOfElem, wird intern abgebildet auf OsAllocX (size=N * SizeOfElem)		
Input	NumOfElem	Anzahl der Feldelemente
	SizeOfElem	Grösse eines Elementes
Output	return	Startadresse des allokierten Speichers

4.3.2.4 OsFreeX ()

OsFreeX ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Rückgabe eine allokierten Speichers. Hier muss derselbe Pool angegeben werden, der auch beim Allokieren verwendet wurde. **Hinweis:** die Funktion **OsFree()** setzt auf OsFreeX() auf und kann unverändert übernommen werden.		
Input	pMem	Adresse des allokierten Speichers
	PoolID	Pool-ID, die beim Allokieren verwendet wurde
Output	return	LSA_OK, LSA_NOT_OK

4.3.2.5 OsAllocTimer()

OsAllocTimer ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Belegen eines Timers. Der Timer kann als zyklischer oder One-Shot Timer konfiguriert werden. Bei Ablauf des Timers wird die vorgegebene Callback-Funktion aufgerufen. Der Timer kann mit OsStartTimer() gestartet werden. Beim Ablauf werden der Callback-Funktion als Parameter die Timer_Id und eine User_Id übergeben. Die User-ID wird bei OsTimerStart() vorgegeben.		
Input	timer_id_ptr	Adresse, ab der die Timer –ID als Rückgabeparameter abgelegt wird
	timer_type	Timer Typ (zyklischer oder one-shot Timer). Mögliche Werte sind LSA_TIMER_TYPE_ONE_SHOT LSA_TIMER_TYPE_CYCLIC
	timer_base	Zeitbasis für den Timer. Mögliche Werte sind LSA_TIME_BASE_1MS LSA_TIME_BASE_10MS LSA_TIME_BASE_100MS LSA_TIME_BASE_1S
	callback_timeout	Callback-Funktion, welche bei Ablauf des Timers aufgerufen wird. Übergabeparameter dabe sind Timer-ID und User-Id. Timer-ID wird vom Stack bei OsAllocTimer() vergeben, die User-Id ist vom Anwender frei wählbar.
Output	return	LSA_OK LSA_RET_ERR_PARAM

4.3.2.6 OsStartTimer()

OsStartTimer()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Input	timer_id	Referenz auf den zu startenden Timer, wurde bei OsAllocTimer() generiert und ist Übergabeparameter der Callback Funktion, die bei Ablauf des Timers aufgerufen wird.
	user_id	frei vom Anwender vergebbare User Id, ist ebenfalls Übergabeparameter der Callback Funktion
	delay	bestimmt die Laufzeit des Timers, bezogen auf die in OsAllocTimer angegebene Zeitbasis. Beispiel: Bei einer Zeitbasis von 10 ms und einem vorgegebenen delay von 5 würde die Callback Funktion nach 50msec aufgerufen.
Output	return	LSA_OK ok, Timer wurde gestartet LSA_RET_ERR_PARAM Parametrierfehler

4.3.2.7 OsStopTimer()

OsStopTimer ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Anhalten eines laufenden Timers		
Input	timer_id	Referenz des Timers
Output	return	LSA_OK ok, Timer wurde angehalten LSA_RET_ERR_PARAM Parametrierfehler

4.3.2.8 OsFreeTimer()

OsFreeTimer()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Einen mit OsAllocTimer belegten Timer wieder freigegben		
Input	timer_id	Referenz des Timers
Output	return	LSA_OK LSA_RET_ERR_TIMER_IS_RUNNING

4.3.2.9 OsEnterX()

OsEnterX ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Belegen eines Mutex. Es sind maximal MAXNUM_OF_NAMED_MUTEXES belegbar, welche von xxx_os.c verwaltet werden.		
Input	MutexId	Identifier des Mutex. Mögliche Werte sind 0…. (MAXNUM_OF_NAMED_MUTEXES - 1)
Output	----	

4.3.2.10 OsExitX

OsExitX()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Freigeben eines belegten Mutex.		
Input	MutexId	Identifier des Mutex. Mögliche Werte sind 0…. (MAXNUM_OF_NAMED_MUTEXES - 1)
Output	----	

4.3.2.11 OsEnterShort

OsEnterShort()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
wird nicht auf ERTEC Plattformen verwendet, nur beim Soft_EDD vorhanden.		
Input	---	
Output	---	

4.3.2.12 OsExitShort

OsExitShort()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
wird nicht auf ERTEC Plattformen verwendet, nur beim Soft_EDD vorhanden.		
Input	---	
Output	---	

4.3.2.13 OsAllocSemB

OsAllocSemB()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Erzeugt ein binäres Semaphore. Das Semaphore muss im Initialzustand leer sein.		
Input	pSemId	Zeiger, ab dem die Semaphore ID zurückgeliefert wird
Output	return	PNIO_OK, PNIO_NOT_OK

4.3.2.14 OsFreeSemB

OsFreeSemB()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Löscht ein binäres Semaphore, welches zuvor mit OsAllocSemB() erezeugt wurde.		
Input	SemId	Semaphore ID, die von OsAllocSemB() erzeugt wurde.
Output	return	PNIO_OK, PNIO_NOT_OK

4.3.2.15 OsTakeSemB

OsTakeSemB()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Belegt ein binäres Semaphore. Die Funktion blockiert, wenn das Semaphore zuvor bereits anderweitig belegt wurde.		
Input	SemId	Semaphore ID, die von OsAllocSemB() erzeugt wurde.
Output	return	PNIO_OK, PNIO_NOT_OK

4.3.2.16 OsGiveSemB

OsGiveSemB()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Gibt ein mit OsTakeSemB() belegtes binäres Semaphore wieder frei.		
Input	SemId	Semaphore ID, die von OsAllocSemB() erzeugt wurde.
Output	return	PNIO_OK, PNIO_NOT_OK

4.3.2.17 OsSetThreadPrio

OsSetThreadPrio()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
wird nicht auf ERTEC Plattformen verwendet, nur beim Soft_EDD vorhanden. Ändern der Taskpriorität. Im IO-Stack wird damit die Threadpriorität des EDD-Low Context Thread vorübergehend erhöht, um zu verhindern, dass diese durch den EDD-High Context Thread unterbrochen wird. Verwendet wird OsSetThreadPrio dazu in den Ausgangsmacros EDD_ENTER_HIGH und EDD_EXIT_HIGH. Dort wird die Priorität der Task vorübergehend auf TASK_PRIO_HIGHEST angehoben.		
Input	ThreadId	ID des Thread, dessen Priorität verändert werden soll
	NewThreadPrio	Neuer einzustellender Prioritätswert
Output	Return	LSA_OK, LSA_NOT_OK

Hinweis: Eine Synchronisation von EDD-High und EDD-Low Thread mittels OsSetThreadPrio() sorgt dafür, dass der EDD Low Context nicht vom EDD High Context unterbrochen werden kann. Der High-Thread kann hingegen schon aufgrund seiner höheren Anfangspriorität nicht vom Low-Thread unterbrochen werden.

4.3.2.18 OsCreateThread

OsCreateThread()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Erzeugen eines Threads (Task). Alle Tasks laufen im selben Adressraum.		
Input	*pThreadEntry	Einsprungadresse für die Task
	*pThreadName	Angabe eines Namens für den Thread. Dies ist nur für Debugzwecke gedacht und kann entfallen, wenn das Betriebssystem diese Möglichkeit nicht unterstützt.
	ThreadPrio	Priorität der Task. Hinweis: Die Prioritäten der Tasks des IO-Stacks werden in der Datei os_cfg.h eingestellt.
	*pThreadId	hier wird die Adresse der zurückgelieferten Thread-ID vorgegeben
Output	return	LSA_OK, LSA_NOT_OK

4.3.2.19 OsStartThread

OsStartThread ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Starten des mit OsCreateThead() erzeugten Threads. Hinweis: bei manchen Betriebssystemen wird mit dem Create-Aufruf der Thread automatisch gestartet, bei anderen ist ein zusätzlicher Start Aufruf erforderlich. Zur Vereinheitlichung sieht das hier verwendete Modell grundsätzlich einen extra Startaufruf vor. Sollte das Betriebssystem die Task automatisch starten, wird diese z.B. über ein Wait-Flag in einem wartenden Zustand gehalten, bis OsStartThread () erfolgt ist.		
Input	ThreadID	Thread-ID, wurde bei OsCreateThread () als Rückgabeparameter übergeben
Output	return	LSA_OK, LSA_NOT_OK

4.3.2.20 OsWaitOnEnable ()

OsWaitOnEnable()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Hiermit wird eine erzeugte Task solange in einem wartenden Zustand gehalten, bis ein OsStartThread () erfolgt ist. OsWaitOnEnable() sollte daher als erster Aufruf in einer erzeugten Task ausgeführt werden.		
Input	---	
Output	return	LSA_OK, LSA_NOT_OK

4.3.2.21 OsGetThreadId ()

OsGetThreadId ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Auslesen der Thread ID für die laufende Task. Hier wird nicht die vom Betriebssystem gelieferte Thread ID direkt zurückgeliefert, sondern eine vom OS-Paket erzeugte Referenz (Tabellenindex). Bei OS unbekannten Tasks wird der Wert von TskIdPost zurückgeliefert. Hinweis: Da im Rahmen der LSA-Timer OsGetThreadId() je nach Implementierung der Os-Timer Funktionen auch von Systemtasks oder ISR's aufgerufen werden kann, wurde von OS für diesen Fall keine ThreadId vergeben, da Systemtasks oder ISR's natürlich nicht mit OsCreateThread erzeugt wurden. Das bedeutet für den Anwender lediglich, dass er als Dummy-ThreadID diejenige vom Post-thread (TaskIdPost) zurückliefern muss.		
Input	---	
Output	return	Thread ID der laufenden Task. Bei unbekannten Threads, d.h. solche, die nicht mit OSCreateThread() erzeugt wurden, muss TskIdPost zurückgeliefert werden.

4.3.2.22 OsCreateMessageQueue ()

OsCreateMessageQueue()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Erzeugen einer Message Queue. Eine Message Queue ist hier immer fest einem Thread zugeordnet. Einem Thread darf höchstes eine Message Queue zugeordnet werden. Jede Message enthält als Dateninformation 2 Pointer. Bei einer Pointerbreite von 4 Byte ist damit eine Message jeweils 8 Byte lang. Der Grund dafür ist, dass die Funktion OsCreateMessageQueue in der Systemanpassung des IO-Stacks häufig dazu benutzt wird, um eine Funktion in einem anderen Thread-Kontext aufzurufen . Beispiel: Um die Funktion fx (pData) im ThreadKontext Thread_1 aufzurufen, wird OsSendMessageX (Thread_1, fx, pdata, OS_MBX_PRIO_NORM) aufgerufen.		
Input	ThreadID	ID des Thread, der die Message Queue zugeordnet werden soll.
Output	return	LSA_OK Message Queue konnte zugeordnet werden LSA_NOT_OK Fehler ist aufgetreten.

4.3.2.23 OsWait_ms ()

OsWait_ms()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Warten einer vorgegebenen Zeit		
Input	Time_ms	Wartezeit in Millisekunden
Output	---	

4.3.2.24 OsGetTime_us ()

OsGetTime_us()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Auslesen der aktuellen Zeit in Mikrosekunden seit dem letzten Systemstart		
Input	---	
Output	return	aktuelle Zeit in Mikrosekunden

4.3.2.25 OsGetTime_us_hw ()

OsGetTime_us_hw()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Optionaler hochgenauer Hardwarezähler (falls vorhanden), Zähler in Mikrosekunden seit dem letzten Systemstart. Dieser kann im System zu Testzwecken für hochgenaue Hardwaremessungen verwendet werden. Auf einer ERTEC Plattform wird dieser auf die 32 Bit Hardwareclock im ERTEC abgebildet, die in 10nsec Schritten incrementiert wird.		
Input	---	
Output	return	aktuelle Zeit in Mikrosekunden

4.3.2.26 OsGetUnixTime ()

OsGetUnixTime()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Auslesen der aktuellen Zeit in Sekunden. Hier wird die Anzahl Sekunden seit dem Systemstart angegeben.		
Input	---	
Output	return	aktuelle Zeit in Sekunden

4.3.2.27 OsReadMessageBlocked ()

OsReadMessageBlocked ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Blockierendes Lesen einer 4 Byte langen Message für den spezifizierten Thread, die mit OsSendMessage() versandt wurde. Eine Message enthält einen Pointer. Die Thread-ID wird als Verwaltungsinformation benötigt, um die Performance im Betrieb zu optimieren. Jeder Thread sollte also einmal im Anlauf seine Thread-ID mit OsGetThreadId() lesen und sich merken. Note: In Version 3.0.0 und neuere des Development Kits wurde die stackinterne zeitkritische Kommunikation von 8 Byte auf 4 Byte lange Nachrichten umgestellt, um auch auf solche Betriebssysteme zeitoptimiert aufsetzen zu können, die nur 4 Byte statt 8 Byte lange Nachrichten unterstützen.		
Input	ppMessage	PtrPtr auf Message
	TaskId	ID des empfangenden (und damit die ID des eigenen) Threads
Output	return	LSA_OK, LSA_NOT_OK

4.3.2.28 OsReadMessageBlockedX ()

OsReadMessageBlockedX ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Blockierendes Lesen einer 8 Byte langen Message für den spezifizierten Thread, die mit OsSendMessageX() versandt wurde. Eine Message enthält jeweils 2 Pointer. Die Thread-ID wird als Verwaltungsinformation benötigt, um die Performance im Betrieb zu optimieren. Jeder Thread sollte also einmal im Anlauf seine Thread-ID mit OsGetThreadId() lesen und sich merken.		
Input	ppMessage1	PtrPtr auf Message1
	ppMessage2	PtrPtr auf Message2
	TaskId	ID des empfangenden (und damit die ID des eigenen) Threads
Output	return	LSA_OK, LSA_NOT_OK

4.3.2.29 OsSendMessage ()

OsSendMessage ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Senden einer Nachricht mit nur einem statt zwei Pointern. Empfangsseitig wird die Nachricht mit OsReadMessageBlocked() empfangen.		
Input	ThreadId	Id des Empfänger Threads
	pMessage	Zeiger auf die eigentliche Nachricht
	MsgPrio	Der Message kann eine Priorität zugeordnet werden. Innerhalb der Systemimplementierung des PNIO Threads haben alle Nachrichten die gleiche Priorität. Der Parameter kann aber genutzt werden, wenn dies im Rahmen einer speziellen Portierung des IO-Stacks von Vorteil ist.
Output	return	LSA_OK LSA_NOT_OK

4.3.2.30 OsSendMessageX ()

OsSendMessageX ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Extended OsSendMessage-Funktion: Senden einer Nachricht mit 2 Pointern. Empfangsseitig wird die Nachricht mit OsReadMessageBlockedX() empfangen. Damit können 2 getrennte Nachrichten mit einem OsSendMessageX() Aufruf übertragen werden. Begründung siehe Beschreibung der Funktion OsCreateMessageQueue ().		
Input	ThreadId	Id des Empfänger Threads
	pMessage1	Zeiger auf die eigentliche Nachricht 1
	pMessage2	Zeiger auf die eigentliche Nachricht 2
	MsgPrio	Der Message kann eine Priorität zugeordnet werden. Innerhalb der Systemimplementierung des PNIO Threads haben alle Nachrichten die gleiche Priorität. Der Parameter kann aber genutzt werden, wenn dies im Rahmen einer speziellen Portierung des IO-Stacks von Vorteil ist.
Output	return	LSA_OK LSA_NOT_OK

4.3.2.31 __InterlockedDecrement ()

__InterlockedDecrement ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Dekrementieren eines Long-Wertes unter Interruptsperre		
Input	pVal	Adresse des zu dekrementierenden Wertes
Output	*pVal	der Wert ab der vorgegebenen Adresse wird dekrementiert
	return	zusätzlich wird das Ergebnis (*pVal) auch als Rückgabewert übertragen

4.3.2.32 __InterlockedIncrement ()

__InterlockedIncrement ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Inkrementieren eines Long-Wertes unter Interruptsperre		
Input	pVal	Adresse des zu inkrementierenden Wertes
Output	*pVal	der Wert ab der vorgegebenen Adresse wird inkrementiert
	return	zusätzlich wird das Ergebnis (*pVal) auch als Rückgabewert übertragen

4.3.2.33 OsIntDisable()

OsIntDisable ()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Sperren der Interrupts des Systems		
Input	--	
Output	--	

4.3.2.34 OsIntEnable()

OsIntEnable()		Funktionsaufruf: IO-Stack ➔ Betriebssystem
Freigeben der Interrupts des Systems		
Input	---	
Output	---	

4.3.3 Kapselung von Funktionsaufrufen der Standardlibraries

Um eine möglichst grosse Plattformunabhängigkeit zu erreichen, werden im PNIO Stack keine Standardlibrary-Funktionen direkt aufgerufen. Stattdessen werden diese ebenfalls über das OS Abstraktionsinterface geleitet. Damit müssen die zugehörigen Standard-Headerfiles auch nur in wenigen Modulen des Devkits eingebunden werden.
Die Aufrufe werden in der Regel unverändert an die Standardlibrary weitergeleitet, so dass sich eine detaillierte Beschreibung dieser Funktionen erübrigt.
Folgende Funktionen des OS Interfaces wurden dazu definiert:

- OsAtoi
- OsHtons, OsHtonl
- OsNtohs, OsNtohl
- OsMemCpy
- OsMemMove

- OsMemSet
- OsMemCmp
- OsStrCmp
- OsStrnCmp
- OsStrCpy
- OsStrnCpy
- OsStrChr
- OsStrLen
- OsRand
- OsSrand

Die o.g. Funktionen können in der Regel unverändert übernommen werden, da sie von fast allen Plattformen in gleicher Weise unterstützt werden.

Weitere hier nicht aufgeführte und in xx_OS.C enthaltene Funktionen können ebenfalls unverändert übernommen werden, da sie nicht direkt auf Betriebssystemfunktionen zugreifen. So wird beispielsweise OsAlloc () zunächst auf OsAllocX () mit PoolId = DEFAULT umgeleitet und muss nicht angepasst werden.

4.3.4 OS-Funktionen der Beispielapplikation

Um eine möglichst grosse Plattformunabhängigkeit zu erreichen, wurden auch plattformspezifische Aufrufe der Beispielapplikation in die OS-Abstraktionsschicht xx_os.c integriert (xx steht für den Plattformnamen).
Die Funktionen dieses Abschnitts werden nicht vom PROFINET IO Stack selbst aufgerufen, sondern lediglich von der Beispielapplikation.

- OsGetChar liest ein ASCII-Zeichen von der Standardkonsole
- OsReboot Führt einen Wiederanlauf des Systems durch

4.4 OsSocket-Interface, SNMP Interface

Das OsSocket-Interface abstrahiert eine UDP-Socket Schnittstelle. Hier müssen vom Anwender Anpassungen an die Socket Schnittstelle des verwendeten TCP/IP Stacks durchgeführt werden. Alle Socket-Abstraktionsfunktionen sind im Modul **<Plattform>_OSSOCKET.C** implementiert. Alle notwendigen Schnittstellendefinitionen und Defines dazu sind in **OS_SOCKET.H** enthalten.
Viele Funktionen werden nahezu direkt auf einen BSD Socket Aufruf abgebildet, dadurch ist die Software leicht auf andere TCP/IP Stacks portierbar. Die Funktion OsSockRecvFrom() hingegen muss für PROFINET IO asynchron implementiert werden. Das OsSocket-Interface ruft eine Callback-Funktion auf, wenn ein Telegramm empfangen wurde.
Für jeden Empfangssocket wird eine eigene Task gestartet, so dass auch hier letztendlich wieder eine Abbildung auf einen einfachen synchronen BSD Aufruf recvfrom() erfolgen kann.
Die Struktur sockaddr_in enthält plattformabhängig nicht immer namensgleiche Elemente. Um hier größtmögliche Portierbarkeit zu erreichen, verwendet der IO-Stack eine eigene Definition namens OS_SOCKADDR_IN.

Das OsSocket-Interface ist nur dann notwendig, wenn ein externer IP Stack über das OsSocket-Interface an die SOCK-lit Komponente angebunden werden muss.
Bei Verwendung des im PNIO Stack enthaltenen Interniche-IP-Stacks (nur für ECOS) ist hingegen kein OsSocket Interface zwischen IP-Stack und SOCK-lsa Komponente eingebunden.

4.4.1.1 OsSockInit ()

OsSockInit ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Die Funktion initialisiert das OS-Socket Interface.		
Input	---	
Output	return	LSA_OK

4.4.1.2 OsSockCleanup()

OsSockCleanup ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Schließen aller mit OsSockUdpOpen() geöffneten Sockets		
Input	---	
Output	return	LSA_OK

4.4.2 Standard Socket Funktionalität

4.4.2.1 OsSockUdpOpen ()

OsSockUdpOpen ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Öffnet und konfiguriert einen UDP Socket. Der Stack liefert ein Socket-Handle an die Applikation zurück. Zur Vereinheitlichung der Schnittstelle und zur Laufzeitoptimierung enthält XXX_OsSocket.c eine eigene Handleverwaltung. Für jeden Socket wird in der GH-Implementierung eine Task gestartet, um gleichzeitigen von mehreren Ports Daten empfangen zu können. Damit ist eine Abbildung eines OsSocketRecv () auf einen einfachen synchronen BSD Socket call recvfrom() möglich. Die Anzahl maximal gleichzeitig geöffneter Sockets ist in der GH Implementierung über das Define #define MAXNUM_SOCKETS 10 begrenzt.		
Input	*SocketId	Adresse, auf der das Socket Handle an die Applikation zurückgeliefert wird.
	OsSockType	OSSOCK_TYPE_ASYNCHRON, OSSOCK_TYPE_SYNCHRON
Output	return	LSA_OK, LSA_NOT_OK

4.4.2.2 OsSockUdpClose ()

OsSockUdpClose ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Schließt den über das Socket-Handle referenzierten UDP Socket. Die bei OsSockUdpOpen () erzeugte Task wird in der GH-Implementierung nicht beendet, sondern würde bei einem nächsten OsSockUdpOpen() einem neuen Socket zugeordnet (Laufzeitoptimierung).		
Input	SocketId	Socket-Handle als Referenz auf den mit OsSockUdpOpen () erzeugten Socket
Output	return	LSA_OK, LSA_NOT_OK

4.4.2.3 OsSockBind ()

OsSockBind ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Binden eines Port an den Socket		
Input	SocketId	Socket-Handle als Referenz auf den mit OsSockUdpOpen () erzeugten Socket
	*pSockAdr	Zeiger auf Datenstruktur mit Socket Adresse. Die Struktur ist vom Typ OS_SOCKADDR_IN . OsSockBind() wandelt diese Struktur in eine platformabhängige Struktur sockaddr_in und reicht diese an einen BSD Socket call bind() Aufruf weiter.
	pSockAdrLen	Länge in Bytes der mit *pSockAdr referenzierten Daten
Output	return	LSA_OK, LSA_NOT_OK

4.4.2.4 OsGetSockName ()

OsGetSockName ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Abbildung auf einen BSD Socket call getsockname() – Aufruf.		
Input	SocketId	Socket-Handle als Referenz auf den mit OsSockUdpOpen () erzeugten Socket
	*pSockAdr	Zeiger auf Datenstruktur vom Typ sock_addr_in. Die Struktur ist in den Header Files des verwendeten TCP/IP Stacks definiert.
	*pSockAdrLen	Hier wird die Länge in Bytes der mit *pSockAdr referenzierten Daten zurückgeliefert.
Output	return	LSA_OK, LSA_NOT_OK

4.4.2.5 OsSockSendTo ()

OsSockSendTo ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Senden eines UDP Telegramms über eine Standard sendto() Socketfunktion. Alle Daten (außer SocketId wegen eigener Handle-Verwaltung in xxx_OsSocket.C) werden direkt an einen BSD Socket call sendto() weitergereicht.		
Input	SocketId	Socket-Handle als Referenz auf den mit OsSockUdpOpen () erzeugten Socket
	pBuf	Zeiger auf Puffer mit den zu sendenden Daten
	len	Länge der zu sendenden Daten in Bytes
	flags	werden direkt an Standard-Socketaufruf sendto() weitergereicht
	*pSockAdr	Zeiger auf Datenstruktur vom Typ OS_SOCKADDR_IN.
	SockAdrLen	Hier wird die Länge in Bytes der mit *pSockAdr referenzierten Daten vorgegeben.
Output	return	LSA_OK, LSA_NOT_OK

4.4.2.6 OsSockRecvFrom ()

OsSockRecvFrom ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Empfangen von UDP Telegrammen über den spezifizierten Socket. Die Funktion ist asynchron, es wird die mit pCbf spezifierte Callback Funktion aufgerufen, wenn ein UDP Telegramm empfangen wurde. Die Funktion wird in der Systemanpassung auf OsSockRecvFromSync() abgebildet.		
Input	SocketId	Socket-Handle als Referenz auf den mit OsSockUdpOpen () erzeugten Socket
	pBuf	Zeiger auf Puffer für die zu empfangenden Daten
	len	Länge des Empfangspuffers in Bytes (d.h. Maximallänge der zu empfangenden Daten).
	flags	werden direkt an Standard-Socketaufruf recvfrom () weitergereicht
	*pSockAdr	Zeiger auf Datenstruktur vom Typ OS_SOCKADDR_IN.
	SockAdrLen	Hier wird die Länge in Bytes der mit *pSockAdr referenzierten Daten vorgegeben.
	pCbf	Callback Funktion, die beim Empfang eines UDP Telegramms aufgerufen wird. Als Übergabeparameter wird dabei ein vom User frei vergebbarer Context, die Länge der tatsächlich empfangenen Daten sowie ein Status übergeben.
	*pUserContext	Vom Anwender frei vergebbarer User Context, der beim Aufruf der Callback Funktion übergeben wird.
Output	return	LSA_OK, LSA_NOT_OK

4.4.2.7 OsSockRecvFromSync ()

OsSockRecvFrom ()		Funktionsaufruf: IO-Stack ➔ TCP/IP Stack
Empfangen von UDP Telegrammen über den spezifizierten Socket. Die Funktion ist synchron und kann damit leicht auf den Standardsocketaufruf recvfrom() abgebildet werden.		
Input	SocketId	Socket-Handle als Referenz auf den mit OsSockUdpOpen () erzeugten Socket
	pBuf	Zeiger auf Puffer für die zu empfangenden Daten
	pLen	Pointer auf Länge des Empfangspuffers in Bytes (d.h. Maximallänge der zu empfangenden Daten). Nach Telegrammempfang ist hier die tatsächliche Länge eingetragen.
	flags	werden direkt an Standard-Socketaufruf recvfrom () weitergereicht
	*pSockAdr	Zeiger auf Datenstruktur vom Typ OS_SOCKADDR_IN.
	SockAdrLen	Hier wird die Länge in Bytes der mit *pSockAdr referenzierten Daten vorgegeben.
Output	return	LSA_OK, LSA_NOT_OK

4.4.3 SNMP Support bei Nutzung eines externen IP Stacks

Bei der Verwendung von SNMP bei Profinet werden folgende MIB's verwendet:

- PROFINET MIB
- MIB2
- optional weitere MIB's (außerhalb von PROFINET)

Zugriff auf die MIBs erhält ein Client über sogenannte MIB Agents, wobei die Profinet MIB und die MIB 2 jeweils einen eigenen Agent besitzen. Da alle SNMP Aufrufe über den Standard UDP Port 161 abgewickelt werden, muss ein Verfahren definiert sein, wie der SNMP Request an den zuständigen MIB Agent weitergeleitet wird. Im Profinet Stack ist dazu folgendes Verfahren implementiert:

- Alle SNMP Requests werden zunächst vom PROFINET MIB Agent entgegengenommen.
- Der PROFINET MIB Agent prüft, ob er selber Adressat dieses Requests ist. Wenn ja, führt er die Bearbeitung durch und sendet das Antworttelegramm (UDP Telegramm) direkt an den TCP/IP Stack zurück (blau gestrichelte Linie).
- Falls der PROFINET MIB Agent nicht der Adressat ist, leitet er das UDP Telegramm unverändert mittels der Funktion OsSnmpMibxRequest an die Applikation weiter. Die weitere Bearbeitung dieses SNMP Requests verläuft außerhalb von PROFINET (blau gepunktete Linie).
- Die Applikation ermittelt nun den zuständigen MIB Agent und leitet das Telegramm an diesen zur Bearbeitung weiter. Falls es (außer dem PROFINET MIB Agent) nur einen weiteren Agent gibt, ist dies sehr einfach, denn es muss der MIB 2 Agent sein.
- Der zuständige MIB Agent bearbeitet den SNMP Request und schickt die Antwort direkt an den TCP/IP Stack.

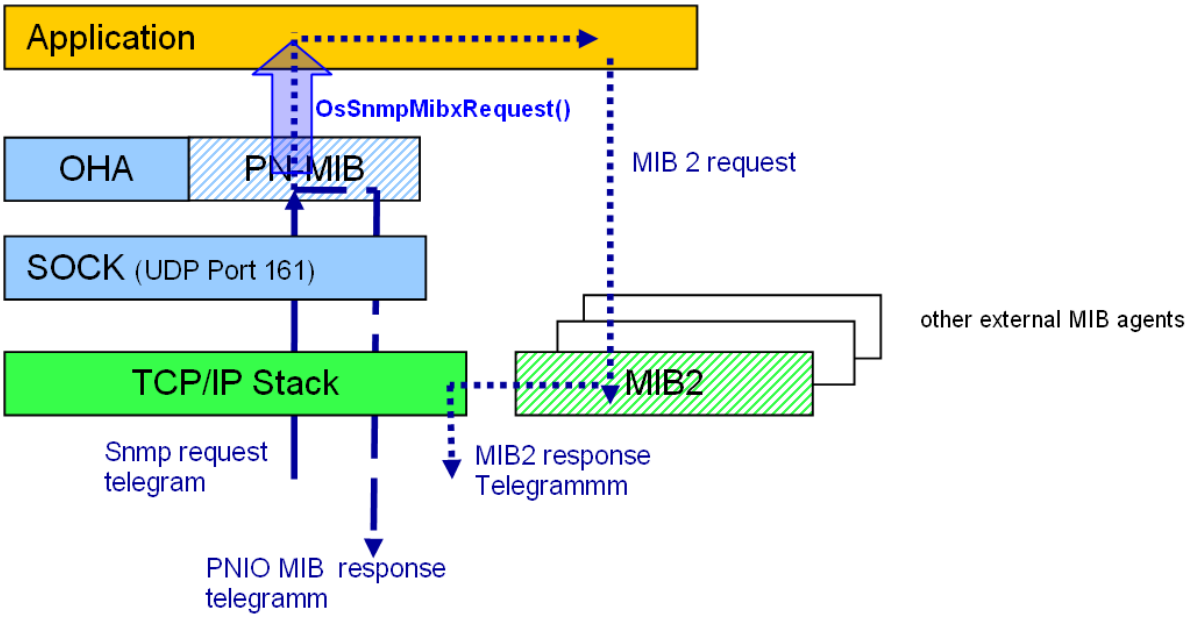


Abbildung 7: Anbindung externer SNMP Agents an PROFINET

Wie das SNMP Telegramm von der Applikation zum MIB2 Agent übertragen wird, ist plattformspezifisch und muss daher außerhalb von PROFINET vom Anwender implementiert werden.

4.4.3.1 Anbindung an SOCK-lit mit externem TCP/IP Stack

Die oben beschriebene Implementierung der SNMP Anpassung ist im Modul <platform>_os_snmp.c (derzeit für NETOS 6.3) hinterlegt. Dieses muss vom Anwender bei einer Portierung auf eine andere Plattform angepasst werden.

4.4.3.2 Anbindung an SOCK-lit mit internem INTERNICHE-TCP/IP Stack (ECOS)

Derzeit nicht verwendet

4.4.3.3 OsSnmpStartMibxSupport ()

OsSnmpStartMibxSupport ()	Funktionsaufruf: Application ➔ IO-Stack	
Die Funktion initialisiert das OS-SNMP Interface. Sie muss einmalig im Anlauf aufgerufen werden. Die Funktion ist rein applikativ, sie wird (derzeit) vom IO-Stack selber nicht verwendet. Es wird aber aus Gründen der Einheitlichkeit empfohlen, notwendige Initialisierungen für den SNMP Support darin zu implementieren.		
Input	---	
Output	return	LSA_OK

4.4.3.4 OsSnmpMibxRequest ()

OsSnmpMibxRequest ()	Funktionsaufruf: IO-Stack ➔ Application	
Jedes eintreffende UDP Telegramm auf Port 161 (SNMP Port) wird zunächst an den IO-Stack weitergeleitet, falls dort SNMP Support aktiviert wurde. Falls das Telegramm vom IO-Stack MIB Agent nicht bearbeitet werden konnte, so ruft der IO-Stack die Callbackfunktion OsSnmpMibxRequest() auf, um das Telegramm zur weiteren Bearbeitung an die Applikation zu übergeben. Diese muss das Telegramm an den zuständigen SNMP Agent weiterreichen (z.B. MIB2 Agent), welcher das Telegramm weiter bearbeitet und anschließend die Antwort direkt an den SNMP Client (über UDP Socket) zurücksendet.		
Input	pBuf	Zeiger auf den Telegrammpuffer
	len	Länge des Telegramms
	pSockAdr	Adresse des Absenders des Telegramms (SNMP Client)
Output	return	LSA_OK

4.5 Wichtige Hinweise und Einschränkungen

4.5.1 Anzahl von IO-Devices

Die derzeitige Implementierung der Systemanpassung lässt nur eine Deviceinstanz zu. Multidevice-Funktionalität ist noch nicht in der Systemanpassung implementiert.

4.5.2 Anzahl von Modulen und Submodulen

Das Mengengerüst und die maximalen Werte für Slot- und Subslotnummern werden in der Datei iod_cfg2.h im Unterverzeichnis sysadapt1\cfg festgelegt.
Ab der Version 3.0.0 können beliebige Slot- und Subslotnummern mit Lücken verwendet werden. Dadurch wird daher nicht mehr der Wertebereich der Slot- und Subslotnummern eingeschränkt, sondern lediglich die maximale Anzahl festgelegt.

4.5.3 Maximale Anzahl der Nutzdaten für ein Device

Die maximale Anzahl der Prozessdaten wird durch die maximale Telegrammgröße (netto 1434 Byte) und die Anzahl der „PhysicalSlots“ (Parameter in der GSDML-Datei) bestimmt. Die maximale Datenlänge in Byte (Attribute „MaxInputLength" und „MaxOutputLength“ in der GSDML-Datei) ist 1434 Byte Daten minus 2 mal Anzahl der „PhysicalSlots“.
Beispiel für ein IO-Device mit 16 „PhysicalSlots“: 1402 Byte maximale Nutzdaten
Beispiel für ein IO-Device mit 32 „PhysicalSlots“: 1370 Byte maximale Nutzdaten

Hinweis, gültig für den ERTEC200: Es wird empfohlen, für den ERTEC200 eine maximale Größe 512 Byte für die Summe der Input- und Output CR (also z.B. 256 Byte Input, 256 Byte Output) bei maximal 2 gleichzeitig aufgebauten ARs nicht zu überschreiten.
Andernfalls kann es aufgrund der begrenzten KRAM-Speichergrösse bei bestimmten Netzlastszenarien zu Problemen kommen. Eine IOCR beinhaltet neben den eigentlichen IO Daten auch die Provider- und Consumerstati sowie die PDEV-Module (DAP + Interface + Ports), die bei der Ermittlung der IOCR-Größe zu berücksichtigen sind.

Grösse_ Input CR = Sume_aller_Input-data + Anzahl_aller_IOxS

Grösse_ Output CR = Summe_aller_Output-data + Anzahl_aller_IOxS

mit Anzahl_aller_IOxS = Anzahl_input_submodule
+ Anzahl_output_submodule
+ Anzahl_input_output_submodule
+ Anzahl_ports + 2 // ports + interface + dap

Hinweis: Die in der Software festgelegten Werte (IOCR_IN_MAX_LENGTH, IOCR_OUT_MAX_LENGTH, IOD_CFG_NUMOF_AR in Datei iod_cfg2.h) müssen zu den Einstellungen in der GSD Datei passen.

4.5.4 Funktionale Einschränkungen

Folgende Funktionen sind in der aktuellen Version nicht enthalten bzw. wurden noch nicht getestet:

- Übernahme von Modulen durch Supervisor
- Querverkehr (Multicast)
- RT over UDP
- Shared input

5 Sonstiges

5.1 Abkürzungen/ Begriffsverzeichnis:

ACP	**A**cyclic **C**ommunication **P**rotocol, bezeichnet eines der Software Basispakete des IO-Stacks
API	**A**pplication **P**rocess **I**dentifier
AR	**A**pplication **R**elationship
BSP	**B**oard **S**upport **P**ackage
CLRPC	**C**onnection**l**ess **R**emote **P**rocedure **C**all, bezeichnet eines der Software Basispakete des IO-Stacks
CM	**C**ontext **M**anagement, bezeichnet eines der Software Basispakete des IO-Stacks
DAP	**D**evice **A**ccess **P**oint, spezifischer Eintrag im GSD File
DCP	**D**iscovery and basic **C**onfiguration **P**rotokoll, bezeichnet eines der Software Basispakete des IO-Stacks
DK_SW	**D**evelopment **K**it **S**oft**w**are (Entwicklungskit für Plattformen, basierend auf Standard-Ethernetcontrollern)
EB 400/200	**E**valuation **B**oard für ERTEC 400/200 Baustein (Bestandteil der ERTEC 400/200 Entwicklungskits)
EDDI	**E**thernet **D**evice **D**river für **I**RTE switch im ERTEC200/400 (frühere Versionen: EDD_ERTEC)
EDDS	**E**thernet **D**evice **D**river für **S**tandard Ethernet Controller (frühere Versionen: EDD_soft)
EDD	**E**thernet **D**evice **D**river, bezeichnet eines der Software Basispakete des IO-Stacks
ELOG	**E**rror **Log**ging für Debug-Zwecke (Level 1 = nur Fehler ausgeben)
GH	**G**reen**H**ill In diesem Dokument verwendete Abkürzung für den in der NET+ARM Beispielimplementierung verwendeten Compiler/Linker.
GSD	**G**eneric **S**tation **D**escription
GSDML	**GSD M**arkup **L**anguage
GSY	**G**eneric **Sy**nc Modul, bezeichnet eines der Software Basispakete des IO-Stacks
ILOG	**I**mportant **Log**ging für Debug-Zwecke (Level 2 = Ausgabe von ELOG + ILOG)
IOCR	**I**nput**/O**utput **C**ommunication **R**elationship
IOCS	**I**nput**/O**utput Object **C**onsumer **S**tatus
IOD	**IO-Device** (Instanz) , bezeichnet eines der Software Pakete des IO-Stacks
IOPS	**I**nput**/O**utput Object **P**rovider **S**tatus
IRT	**I**sochrone **R**eal**T**ime, Class 2 (IRT C2) oder Class 3 (IRT C3)
LLDP	**L**ink **L**ayer **D**iscovery **P**rotocol (IEEE 802.1AB, Allows stations to exchange chassis and port information)
LOG	**Log**ging für Debug-Zwecke (Level 3 = Ausgabe von ELOG + ILOG + LOG)
LSA	**L**ayer **S**tructure **A**rchitecture
MIB	**M**anagement **I**nformation **B**ase. Datenbasis für SNMP Dienste
MLOG	**M**emory **Log**ging für Debug-Zwecke (Level 4 = Ausgabe von ELOG + ILOG + LOG + MLOG)
MRP	**M**edia **R**edundancy **P**rotocol
NARE	**N**ame **A**ddress **R**esolution
NRT	**N**on **Real**time ist ein Oberbegriff für alle Nicht – Realtime Telegramme (nicht Typ 0x8892)
OHA	**O**bject **Ha**ndler
OS	**O**perating **S**ystem, bezeichnet hier die Abstraktionsschicht für ein beliebiges Betriebssystem, auf welches der IO-Stack portiert werden soll.
PDEV	**P**hysical **Dev**ice
PN-IO	**PROFINET IO**
PNO	**P**ROFIBUS **N**utzer **O**rganisation
PCF	**P**olymeric **C**ladded **F**iber (optisches Übertragungsmedium)
POF	**P**olymeric **O**ptical **F**iber (optisches Übertragungsmedium)
RT	**Real**time is a generic term for acyclic and cyclic realtime
RT	**Real**time ist ein Oberbegriff für azyklische und zyklische Realtime
SNMP	**S**imple **N**etwork **M**anagement **P**rotokoll
SOCK	UDP **Sock**et Interface für PROFINET IO, bezeichnet eines der Software Pakete des IO-Stacks
UDP	**U**ser **D**atagram **P**rotokol
UUID	**U**niversal **U**nique **Id**entifier

5.2 Literaturverzeichnis:

/1a/ **PROFINET IO Specification IEC 61158 – part5**
PROFINET IO Application Layer Service Specification
(Downloadbar von der PNO Website http://profibus.com)

/1b/ **PROFINET IO Specification IEC 61158 – part6**
PROFINET IO Application Layer Protocol Specification
(Downloadbar von der PNO Website http://profibus.com)

/2/ **GSDML Specification for PROFINET IO**
Version 2.25, Order No: 2.352
PROFIBUS Nutzerorganisation e.V.
(Downloadbar von der PNO Website http://profibus.com)

/3/ **Industrielle Kommunikation mit PROFINET**
Manfred Popp
PROFIBUS Nutzerorganisation e.V.
Bestellnummer 4.182

/4/ **PROFINET Technologie und Anwendung**
Systembeschreibung
(Downloadbar von der PNO Website http://profibus.com)

/5/ **SIMATIC PROFINET Systembeschreibung**
Systemhandbuch
Zeichnungsnummer A5E00298287-02
Teil der Dokumentationspakete mit Bestellnummern 6ES7398-8FA10-8BA0 und 6ES7151-1AA10-8BA0