SIEMENS

SIMATIC

PROFINET Schnittstellenbeschreibung Evaluation Kit ERTEC 200P PN IO V4.0

Programmier- und Bedienhandbuch

Vorwort

Einleitung 1

PROFINET IO-Device Software-Übersicht 2

Software-Erstellung von PROFINET IO-Devices 3

Schnittstellenbeschreibung 4

Anhang A

02/2013

A5E03855365-01

Rechtliche Hinweise

Warnhinweiskonzept

Dieses Handbuch enthält Hinweise, die Sie zu Ihrer persönlichen Sicherheit sowie zur Vermeidung von Sachschäden beachten müssen. Die Hinweise zu Ihrer persönlichen Sicherheit sind durch ein Warndreieck hervorgehoben, Hinweise zu alleinigen Sachschäden stehen ohne Warndreieck. Je nach Gefährdungsstufe werden die Warnhinweise in abnehmender Reihenfolge wie folgt dargestellt.

⚠GEFAHR
bedeutet, dass Tod oder schwere Körperverletzung eintreten **wird**, wenn die entsprechenden Vorsichtsmaßnahmen nicht getroffen werden.

⚠WARNUNG
bedeutet, dass Tod oder schwere Körperverletzung eintreten **kann**, wenn die entsprechenden Vorsichtsmaßnahmen nicht getroffen werden.

⚠VORSICHT
bedeutet, dass eine leichte Körperverletzung eintreten kann, wenn die entsprechenden Vorsichtsmaßnahmen nicht getroffen werden.

ACHTUNG
bedeutet, dass Sachschaden eintreten kann, wenn die entsprechenden Vorsichtsmaßnahmen nicht getroffen werden.

Beim Auftreten mehrerer Gefährdungsstufen wird immer der Warnhinweis zur jeweils höchsten Stufe verwendet. Wenn in einem Warnhinweis mit dem Warndreieck vor Personenschäden gewarnt wird, dann kann im selben Warnhinweis zusätzlich eine Warnung vor Sachschäden angefügt sein.

Qualifiziertes Personal

Das zu dieser Dokumentation zugehörige Produkt/System darf nur von für die jeweilige Aufgabenstellung **qualifiziertem Personal** gehandhabt werden unter Beachtung der für die jeweilige Aufgabenstellung zugehörigen Dokumentation, insbesondere der darin enthaltenen Sicherheits- und Warnhinweise. Qualifiziertes Personal ist auf Grund seiner Ausbildung und Erfahrung befähigt, im Umgang mit diesen Produkten/Systemen Risiken zu erkennen und mögliche Gefährdungen zu vermeiden.

Bestimmungsgemäßer Gebrauch von Siemens-Produkten

Beachten Sie Folgendes:

⚠WARNUNG
Siemens-Produkte dürfen nur für die im Katalog und in der zugehörigen technischen Dokumentation vorgesehenen Einsatzfälle verwendet werden. Falls Fremdprodukte und -komponenten zum Einsatz kommen, müssen diese von Siemens empfohlen bzw. zugelassen sein. Der einwandfreie und sichere Betrieb der Produkte setzt sachgemäßen Transport, sachgemäße Lagerung, Aufstellung, Montage, Installation, Inbetriebnahme, Bedienung und Instandhaltung voraus. Die zulässigen Umgebungsbedingungen müssen eingehalten werden. Hinweise in den zugehörigen Dokumentationen müssen beachtet werden.

Marken

Alle mit dem Schutzrechtsvermerk ® gekennzeichneten Bezeichnungen sind eingetragene Marken der Siemens AG. Die übrigen Bezeichnungen in dieser Schrift können Marken sein, deren Benutzung durch Dritte für deren Zwecke die Rechte der Inhaber verletzen kann.

Haftungsausschluss

Wir haben den Inhalt der Druckschrift auf Übereinstimmung mit der beschriebenen Hard- und Software geprüft. Dennoch können Abweichungen nicht ausgeschlossen werden, so dass wir für die vollständige Übereinstimmung keine Gewähr übernehmen. Die Angaben in dieser Druckschrift werden regelmäßig überprüft, notwendige Korrekturen sind in den nachfolgenden Auflagen enthalten.

Siemens AG
Industry Sector
Postfach 48 48
90026 NÜRNBERG
DEUTSCHLAND

A5E03855365-01
Ⓟ 02/2013 Änderungen vorbehalten

Copyright © Siemens AG 2013.
Alle Rechte vorbehalten

Vorwort

Vorwort

Zweck des Handbuchs

Die Anwenderdokumentation beschreibt die Software-Funktionalität des Evaluation Kit für ein PROFINET IO-Device:

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

Zielgruppe des Handbuchs

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

Dieses Handbuch gilt für ERTEC 200P-basierte Plattformen.

Aufbau des Handbuchs

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

- Kapitel 1: Einleitung
- Kapitel 2: PROFINET IO-Device Software-Übersicht
- Kapitel 3: Software-Erstellung von PROFINET IO-Devices
- Kapitel 4: Schnittstellenbeschreibung
- Anhang: Abkürzungen/Begriffsverzeichnis, Literaturverzeichnis

Dieses Handbuch enthält die Beschreibung des PROFINET IO-Stacks für das unterstützte Evaluation Kit zum Zeitpunkt der Herausgabe. Wir behalten uns vor, bei neueren Erzeugnisständen die Anwenderdokumentation 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 der im Handbuch enthaltenen Abbildungen und Tabellen.
- Im Anhang finden Sie ein Abkürzungs- und Begriffsverzeichnis, 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 im Anhang des Handbuchs den genauen Titel der Dokumente entnehmen.

Konventionen

Im vorliegenden Handbuch werden die Begriffe "ThreadId" und "TaskId" synonym verwendet.

Beachten Sie auch die folgendermaßen gekennzeichneten Hinweise:

Hinweis

Ein Hinweis enthält wichtige Informationen zum beschriebenen Produkt, zur Handhabung des Produkts oder zu dem Teil der Dokumentation, auf den besonders aufmerksam gemacht wird.

Weitere Unterstützung

Bei Fragen zur Nutzung des beschriebenen Evaluation Kit, 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 schicken Sie bitte schriftlich an die angegebene E-Mail-Adresse.

Zusätzlich erhalten Sie allgemeine Informationen, aktuelle Produkt-Informationen, FAQs und Downloads, die beim Einsatz nützlich sein können, im Internet (http://www.siemens.de/comdec).

Technischer Ansprechpartner für Deutschland/weltweit

Siemens AG
ComDeC (http://www.siemens.de/comdec)

Tel.: +49 911 750 2080
Tel.: +49 911 750 4384
Tel.: +49 911 750 2078
Fax: +49 911 750 2100
E-Mail: (mailto:ComDeC@siemens.com)

Hausadresse:
Würzburger Str. 121
D-90766 Fürth

Briefadresse:
Postfach 2355
D-90713 Fürth

Technischer Ansprechpartner für USA

PROFI Interface Center
(http://www.profiinterfacecenter.com)
One Internet Plaza
Johnson City, TN 37604

Tel.: +1 (423) 262-2576
Fax: +1 (678) 297-7289
E-Mail: (mailto:PIC.industry@siemens.com)

Inhaltsverzeichnis

Vorwort ... 3

1 Einleitung ... 13

1.1 Inhalt und Zielgruppe dieser Schnittstellenbeschreibung ... 13

1.2 Beispiel-Plattformen ... 14

1.3 Sonstige Hinweise ... 14

2 PROFINET IO-Device Software-Übersicht ... 15

2.1 Softwarearchitektur ... 15
2.1.1 Softwarearchitektur ... 16
2.1.1.1 ERTEC 200P ... 16

2.2 Komponenten des PROFINET IO-Stacks ... 17
2.2.1 EDDP (Ethernet Device Driver für ERTEC 200P) ... 17
2.2.2 ACP (Acyclic Communication Protocol) ... 17
2.2.3 CM (Context Manager) ... 18
2.2.4 CLRPC (Connectionless Remote Procedure Call) ... 18
2.2.5 DCP (Dynamic Configuration Protokoll) ... 18
2.2.6 GSY (Generic Sync Module) ... 18
2.2.7 LLDP (Link Layer Discovery Protokoll) ... 18
2.2.8 MRP (Media Redundancy Protocol ... 19
2.2.9 OHA (Objekt Handler) ... 19
2.2.10 POF (Polymeric optical fiber) ... 19
2.2.11 SOCK (Socket Interface) ... 19
2.2.12 TCP/IP-Stack ... 19
2.2.13 Systemanpassung (SYS, LSAS) ... 19
2.2.14 OS-Abstraction Layer (OS-Adapt) ... 20
2.2.15 PNDV ... 20
2.2.16 PNPB ... 20

2.3 Sonstige Software-Komponenten ... 20
2.3.1 Operating System ... 20
2.3.2 Board Support Package (BSP) ... 20

2.4 Applikationsbeispiele ... 21
2.4.1 Genereller Aufbau der Anwenderbeispiele ... 22
2.4.2 Taktsynchrone Applikation mit IRT, T_Input- und T_Output-Auswertung ... 24
2.4.3 App1: SI-basiertes Beispiel für RT und IRT ... 25
2.4.4 App2: DBAI-basiertes Beispiel für RT und IRT ... 26

3 **Software-Erstellung von PROFINET IO-Devices** 27

3.1 Verzeichnisstruktur des PROFINET IO-Quellcodes 27

3.2 Dateien der Beispielapplikationen 31
3.2.1 Dateien von App1_STANDARD 31
3.2.2 Dateien von App2_DBAI 31
3.2.3 Dateien von App3_IsoApp 32
3.2.4 Dateien von App_common 33
3.2.5 Anwenderschnittstelle 34
3.2.6 Anzupassende Module der Betriebssystem-Schnittstelle 34
3.2.7 Anzupassende Module der Socket-Schnittstelle 34
3.2.8 Module der BSP-Schnittstelle 35
3.2.9 Speicherung remanenter Daten 35
3.2.10 Sonstige Dateien der Systemanpassung 35

3.3 Wichtige Randbedingungen für das Einbinden einer Applikation 36

3.4 Portieren der PROFINET IO-Software auf eine andere Plattform 36
3.4.1 Portierung auf Kunden-Hardware unter Beibehaltung von Mikrocontroller und Betriebssystem 37
3.4.2 Verwendung anderer Compiler/Linker 37
3.4.2.1 Auswahl der Toolkette 37
3.4.2.2 Big oder little Endian 37
3.4.2.3 Data Alignment-Anforderungen 38
3.4.2.4 Datenverarbeitungsbreite 38
3.4.2.5 Speichermanagement 38
3.4.3 Verwendung anderer Betriebssysteme 39

3.5 Typischer Ablauf eines IO-Device-Anwenderprogramms 39
3.5.1 Initialisierungsphase 40
3.5.2 Produktivbetrieb 42
3.5.3 Abschlussphase 46

3.6 Prinzipieller Datenverkehr der IO-Device-Anwenderschnittstelle 47

3.7 Zyklischer IO-Datenverkehr der IO-Device-Anwenderschnittstelle 47
3.7.1 Zyklisches Schreiben mit Status 48
3.7.2 Zyklisches Lesen mit Status 49
3.7.3 Zyklischer Datenaustausch über optionales DBA-Interface 49

3.8 IO-Datenaustausch bei IRT Class 3 (ERTEC-basierte Plattformen) 50

3.9 Diagnosedaten verwalten 50
3.9.1 Channel Diagnosis Data 50
3.9.2 Manufactory Specified Diagnosis Data 52

3.10 Besonderheiten beim Ziehen und Stecken von Modulen im Produktivbetrieb 53
3.10.1 Besonderheiten bei "Return of Submodul" 54

3.11 Callback-Mechanismus 54

4 Schnittstellenbeschreibung 57
4.1 Upper Layer Schnittstellenfunktionen zur Applikation 57
4.1.1 Funktionen für den Systemanlauf 57
4.1.1.1 PNIO_init 57
4.1.1.2 PNIO_setup 58
4.1.1.3 PNIO_device_open 58
4.1.1.4 PNIO_async_appl_rdy 59
4.1.1.5 PNIO_device_close 59
4.1.1.6 PNIO_CP_register_cbf 60
4.1.2 Einstellungen von GeräteNamen und IP-Suite 60
4.1.2.1 PNIO_cbf_save_station_name 61
4.1.2.2 PNIO_cbf_save_ip_addr 61
4.1.2.3 PNIO_cbf_start_led_blink 61
4.1.2.4 PNIO_cbf_stop_led_blink 62
4.1.2.5 PNIO_cbf_reset_factory_settings 62
4.1.3 Speicherung remanenter Daten (REMA) 62
4.1.3.1 PNIO_cbf_store_rema_mem 63
4.1.3.2 PNIO_cbf_restore_rema_mem 63
4.1.4 IO-Device Konfiguration 64
4.1.4.1 PNIO_sub_plug 64
4.1.4.2 PNIO_sub_plug_list 65
4.1.4.3 PNIO_sub_pull 65
4.1.5 Diagnosedaten im Subslot ablegen 66
4.1.5.1 PNIO_diag_channel_add 66
4.1.5.2 PNIO_diag_channel_remove 66
4.1.5.3 PNIO_ext_diag_channel_add 67
4.1.5.4 PNIO_ext_diag_channel_remove 68
4.1.5.5 PNIO_diag_generic_add 69
4.1.5.6 PNIO_diag_generic_remove 70
4.1.6 Senden und Empfangen von Alarmen 70
4.1.6.1 PNIO_process_alarm_send 71
4.1.6.2 PNIO_upload_retrieval_alarm_send 72
4.1.6.3 PNIO_ret_of_sub_alarm_send 72
4.1.6.4 PNIO_cbf_dev_alarm_ind() 73
4.1.7 Quittierung von asynchronen Funktionen 73
4.1.7.1 PNIO_cbf_async_req_done 73
4.1.8 Lesen und Schreiben von Records 74
4.1.8.1 PNIO_cbf_rec_read 76
4.1.8.2 PNIO_cbf_rec_write 77
4.1.8.3 PNIO_rec_set_rsp_async 77
4.1.8.4 PNIO_rec_read_rsp 78
4.1.8.5 PNIO_rec_write_rsp 78
4.1.9 Zyklischer Datenaustausch über das Standard-Callbackinterface (SCI) 79
4.1.9.1 PNIO_initiate_data_read, PNIO_initiate_data_write 79
4.1.9.2 PNIO_cbf_data_write, PNIO_cbf_data_read 80
4.1.10 Zyklischer Datenaustausch über das optionale DBA-Interface 81
4.1.10.1 Zyklischer Datenaustausch über das optionale DBA-Interface 81
4.1.10.2 PNIO_dbai_enter 82
4.1.10.3 PNIO_dbai_exit 82
4.1.10.4 PNIO_dbai_buf_lock 83
4.1.10.5 PNIO_dbai_buf_unlock 83

4.1.11 Events und Alarme empfangen ... 84
4.1.11.1 PNIO_cbf_ar_connect_ind ... 84
4.1.11.2 PNIO_cbf_ar_ownership_ind ... 85
4.1.11.3 PNIO_cbf_ar_indata_ind ... 85
4.1.11.4 PNIO_cbf_ar_disconn_ind ... 86
4.1.11.5 PNIO_cbf_param_end_ind ... 86
4.1.12 Control-Funktionen ... 87
4.1.12.1 PNIO_set_dev_state ... 87
4.1.13 Hardware-Komparatoren für taktsynchronen Betrieb ... 88
4.1.13.1 Hardware-Komparatoren für taktsynchronen Betrieb ... 88
4.1.13.2 PNIO_IsoActivateIsrObj ... 90
4.1.13.3 PNIO_IsoActivateGpioObj ... 90
4.1.13.4 PNIO_IsoActivateTransEndObj ... 91
4.1.13.5 PNIO_IsoFreeObj ... 91
4.1.14 Fehlerbehandlung ... 92
4.1.14.1 PNIO_get_last_error ... 92
4.1.14.2 PNIO_Log ... 92
4.1.14.3 PNIO_set_iops ... 93
4.1.15 Sonstige Funktionen ... 94
4.1.15.1 PNIO_printf ... 94
4.1.15.2 PNIO_TrcPrintf ... 94
4.1.15.3 PNIO_get_version ... 94

4.2 Lower Layer Schnittstellenfunktionen zum Board Support Package ... 95
4.2.1 BSP Funktionen für alle Plattformen ... 95
4.2.1.1 Bsp_Init ... 95
4.2.1.2 Bsp_GetMacAddr ... 95
4.2.1.3 Bsp_GetPortMacAddr ... 95
4.2.1.4 Bsp_EbSetLed (Implementierung optional) ... 96
4.2.2 Speicherung nichtflüchtiger Daten ... 96
4.2.2.1 Bsp_nv_data_clear ... 97
4.2.2.2 Bsp_nv_data_store ... 97
4.2.2.3 Bsp_nv_data_restore ... 97
4.2.2.4 Bsp_nv_data_memfree ... 98
4.2.3 Anbindung der ERTEC Switch Interrupts (nur für ERTEC-Plattformen) ... 98
4.2.4 GPIO-Anbindung (nur für ERTEC-Plattformen) ... 98
4.2.4.1 Bsp_ReadGPIOin_0_to_31 (Implementierung optional) ... 98
4.2.4.2 Bsp_SetGPIOout_0_to_31 (Implementierung optional) ... 99
4.2.4.3 Bsp_ClearGPIOout_0_to_31 (Implementierung optional) ... 99

4.3 Schnittstelle zum Betriebssystem ... 99
4.3.1 Schnittstelle zum Betriebssystem ... 99
4.3.2 Verwaltung von Ressourcen ... 100

4.3.3 Beschreibung der zu portierenden OS-Funktionen 100
4.3.3.1 OsInit() 100
4.3.3.2 OsAllocFX() 100
4.3.3.3 OsFreeX() 101
4.3.3.4 OsAllocTimer() 101
4.3.3.5 OsStartTimer() 102
4.3.3.6 OsStopTimer() 102
4.3.3.7 OsFreeTimer() 102
4.3.3.8 OsEnterX() 103
4.3.3.9 OsExitX 103
4.3.3.10 OsEnterShort 103
4.3.3.11 OsExitShort 103
4.3.3.12 OsAllocSemB 103
4.3.3.13 OsFreeSemB 104
4.3.3.14 OsTakeSemB 104
4.3.3.15 OsGiveSemB 104
4.3.3.16 OsSetThreadPrio 104
4.3.3.17 OsCreateThread 105
4.3.3.18 OsStartThread 105
4.3.3.19 OsWaitOnEnable() 105
4.3.3.20 OsGetThreadId() 106
4.3.3.21 OsCreateMsgQueue() 106
4.3.3.22 OsWait_ms() 106
4.3.3.23 OsGetTime_us() 107
4.3.3.24 OsGetUnixTime() 107
4.3.3.25 OsReadMessageBlocked() 107
4.3.3.26 OsReadMessageBlockedX() 107
4.3.3.27 OsSendMessage() 108
4.3.3.28 OsSendMessageX() 108
4.3.3.29 __InterlockedDecrement() 108
4.3.3.30 __InterlockedIncrement() 109
4.3.3.31 OsIntDisable() 109
4.3.3.32 OsIntEnable() 109
4.3.4 Kapselung von Funktionsaufrufen der Standard-Libraries 110
4.3.5 OS-Funktionen der Beispielapplikation 111

4.4 Wichtige Hinweise und Einschränkungen 111
4.4.1 Anzahl von IO-Devices 111
4.4.2 Anzahl von Modulen und Submodulen 111
4.4.3 Maximale Anzahl der Nutzdaten für ein Device 111
4.4.4 Funktionale Einschränkungen 112

A Anhang 113

A.1 Abkürzungen/Begriffsverzeichnis 113

A.2 Literaturverzeichnis 115

Tabellen

Tabelle 2- 1 Unterschiede der Stack-Layers in den verschiedenen Development Packages 15
Tabelle 3- 1 Struktur und Beschreibung der Verzeichnisse 27
Tabelle 3- 2 Dateien der Beispielapplikation 33
Tabelle 3- 3 Header-Dateien der Anwenderschnittstelle 34
Tabelle 3- 4 Dateien der Betriebssystem-Abstraktionsschnittstelle 34
Tabelle 3- 5 Dateien der Socket-Abstraktionsschnittstelle 34
Tabelle 3- 6 Dateien zur Anpassung an das Board Support Package 35
Tabelle 3- 7 Dateien zur Anpassung an remanente Daten 35
Tabelle 3- 8 Sonstige Dateien der Systemanpassung 35
Tabelle 3- 9 Aufzurufende API-Funktionen in der Anlaufphase 40
Tabelle 3- 10 Aufzurufende API-Funktionen zum Lesen von IO-Daten bei RT und IRT 42
Tabelle 3- 11 Aufzurufende API-Funktionen zum Schreiben von IO-Daten bei RT und IRT 43
Tabelle 3- 12 Aufzurufende API-Funktionen zum synchronen Lesen von Record-Daten 44
Tabelle 3- 13 Aufzurufende API-Funktionen zum asynchronen Lesen von Record-Daten 44
Tabelle 3- 14 Aufzurufende API-Funktionen zum synchronen Schreiben von Record-Daten 45
Tabelle 3- 15 Aufzurufende API-Funktionen zum asynchronen Schreiben von Record-Daten 45
Tabelle 3- 16 API-Callback-Funktionen bei Verbindungsauf- und abbau 46
Tabelle 3- 17 Aufzurufende API-Funktionen für die Erstellung eines Diagnose-Records 50
Tabelle 3- 18 Aufzurufende API-Funktionen für die Aktivierung eines erstellten Diagnose-Records 51
Tabelle 3- 19 Aufzurufende API-Funktionen für das Entfernen eines Diagnose-Records 51
Tabelle 3- 20 Aufzurufende API-Funktionen für die Erstellung eines generischen Diagnose-Records 52
Tabelle 3- 21 Aufzurufende API-Funktionen für die Aktivierung eines generischen Diagnose-Records 52
Tabelle 3- 22 Aufzurufende API-Funktionen für das Entfernen eines generischen Diagnose-Records 52
Tabelle 3- 23 Übersicht der Callback-Funktionen im IO-Device 54

Bilder

Bild 2-1 Systemumgebung für ein ERTEC basiertes IO-Device mit integriertem TCP/IP-Stack 16
Bild 2-2 Tasks des PROFINET IO-Anwenderbeispiels 23
Bild 4-1 Synchrone Read Record-Bearbeitung 74
Bild 4-2 Asynchrone Read Record-Bearbeitung 75

Einleitung 1

PROFINET ist ein Automatisierungskonzept für die Realisierung modularer, dezentraler Applikationen. Mit PROFINET erstellen Sie Automatisierungslösungen, wie sie Ihnen von PROFIBUS her bekannt und vertraut sind. Die Umsetzung von PROFINET wird einerseits durch den PROFINET-Standard für Automatisierungsgeräte und andererseits durch das Engineering Tool (STEP 7, TIA-Portal) realisiert. Das bedeutet, dass Sie im Engineering 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 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-Controllern
- 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 Firmwarestack implementieren zu können.

1.1 Inhalt und Zielgruppe dieser Schnittstellenbeschreibung

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

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

Diese Dokumentation beinhaltet **keine**:

- Übersicht über PROFINET
- Beschreibung des PROFINET-Protokolls
- detaillierte Beschreibung über Aufbau und Abläufe im PROFINET IO-Stack

1.2 Beispiel-Plattformen

Derzeit wird ausschließlich folgende Plattform unterstützt:

- ERTEC 200P, Betriebssystem eCos 3.0, Evaluation Board EB 200P

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 Beispiel-Plattform (s.o.) getestet.

PROFINET IO-Device Software-Übersicht 2

2.1 Softwarearchitektur

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

Tabelle 2- 1 Unterschiede der Stack-Layers in den verschiedenen Development Packages

	EB 200P, eCos Plattform	EB 200/400, eCos Plattform	Standard Ethernet Controller, NETOS Plattform
TCP/IP-Stack	Interniche TCP/IP-Stack, integriert in PNIO	Interniche TCP/IP-Stack, integriert in PNIO	Interniche TCP/IP-Stack, integriert in PNIO
SNMP MIB2 Agent	Interniche TCP/IP-Stack, integriert in PNIO	Interniche TCP/IP-Stack, integriert in PNIO	Interniche TCP/IP-Stack, integriert in PNIO
IP Lower Layer Anbindung	Auf EDDP	Auf EDDI	Auf EDDI
EDD	EDDP	EDDI	EDDS

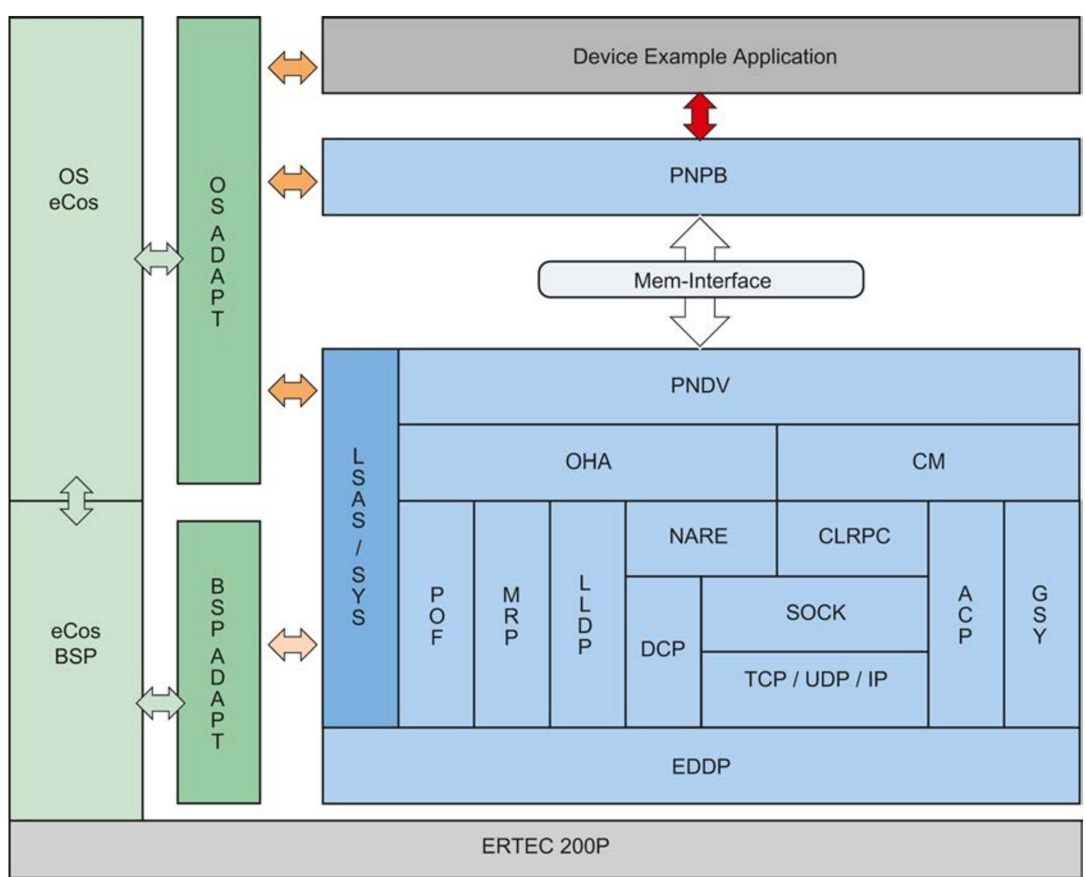
Die PROFINET IO-Device Software besteht aus folgenden Komponenten:

- PROFINET IO-Protokollsoftware (hellblau dargestellt)
- Systemanpassung (dunkelblau dargestellt)
- Echtzeit-Betriebssystem
- Board Support Package
- plattformspezifische Adaptionsschicht
- Device-Applikation (dunkelgrau dargestellt)

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

2.1.1 Softwarearchitektur

2.1.1.1 ERTEC 200P



Rote Pfeile	Applikations-Schnittstelle
Orangene Pfeile	Betriebssystem - Abstraktionsschnittstelle (OS-Adapt)
Hell orangener Pfeil	Hardware - Abstraktionsschnittstelle (BSP-Adapt)

Bild 2-1 Systemumgebung für ein ERTEC basiertes IO-Device mit integriertem TCP/IP-Stack

Diese Plattform hat die folgenden Eigenschaften:

- Komplettlösung für PROFINET inklusive Interniche IP-Stack
- SNMP Agent (MIB2, SNMP-MIB) im Stack enthalten
- Implementiert für eCos 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, POF (in Vorbereitung), LLDP, MRP, OHA, TCP/IP-Stack, SOCK. Die Basispakete stellen lediglich eine Art Funktionsbibliothek zur Verfügung, welche erst zusammen mit der Systemintegration eine konkrete, lauffähige Systemimplementierung ergeben.
- **Systemintegration (SYS, LSAS) für alle enthaltenen Softwarepakete**
  Diese bildet die Schnittstelle zwischen den einzelnen systemunabhängigen Basispaketen und den Betriebssystem-Services wie Speichermanagement, 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 Software auf ein anderes Betriebssystem muss dadurch nur der OS-Abstraktion Layer angepasst werden.
- **Ergänzende Softwarepakete wie PNDV und PNPB**

2.2.1 EDDP (Ethernet Device Driver für ERTEC 200P)

Der EDDP stellt Mechanismen bereit für:

- Selbstständiges Senden und Empfangen von zyklischen Realtime-Telegrammen.
- Senden und Empfangen von azyklischen Realtime-Telegrammen.
- Senden und Empfangen von Nicht - Realtime-Telegrammen.

Der EDDP hat eine einheitliche LSA-Schnittstelle zu überlagerten Clients (ACP, DCP, GSY, LLDP, MRP, ...).

2.2.2 ACP (Acyclic Communication Protocol)

Bearbeitung von:

- Diagnosealarmen
- Prozessalarmen
- Return-of-Submodul-Alarmen
- Upload-/Retrieval-Alarmen (Parameterserver)

Der ACP generiert die Alarm-Telegramme und überwacht den korrekten Ablauf des zugehörigen Ethernetprotokolls (Alarmquittungen, Timeout).

2.2.3 CM (Context Manager)

- Aufbau und Verwalten von Kommunikationsbeziehungen zwischen IO-Device und IO-Controller
  - Requests für Aufbau von Kommunikationsbeziehungen werden vom remoten 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 (PNDV)

2.2.4 CLRPC (Connectionless Remote Procedure Call)

- Implementierung des "Connectionless RPC" Protokolls

2.2.5 DCP (Dynamic Configuration Protokoll)

- Vergabe von IP-Adressen und Gerätenamen über Ethernet
- Auslesen von Bereitschafts-Informationen wie z. B.:
  - Welche PROFINET Devices sind am Netz aktiv
  - Welches PROFINET Device hat folgenden Gerätenamen
  - Hello-Telegramm für Fast Startup signalisiert Bereitschaft zum Verbindungsaufbau nach Power On

2.2.6 GSY (Generic Sync Module)

- Bearbeitung der Synchronisationstelegramme vom Sync-Master
- Leitungslängenmessung
- Synchronisationsüberwachung

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

2.2.10 POF (Polymeric optical fiber)

(In Vorbereitung, Funktion in dieser Version noch nicht enthalten)

2.2.11 SOCK (Socket Interface)

- Interne Adaptionsschnittstelle für die Behandlung von UDP-basierten Diensten im PROFINET Stack

2.2.12 TCP/IP-Stack

- Implementierung der TCP- und UDP-Funktionalität (PROFINET selbst verwendet lediglich UDP)
- basierend auf Interniche TCP/IP-Stack
- Senden und Empfangen von Raw Ethernet IP-Telegrammen
- Interne Adaptionsschnittstelle zu PROFINET Stack-internen SNMP MIB Agents (MIB2, LLDP-MIB)

2.2.13 Systemanpassung (SYS, LSAS)

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

2.2.14 OS-Abstraction Layer (OS-Adapt)

- 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

2.2.15 PNDV

- Implementierung der anwendungsneutralen Teile der Device-Applikation
- Start und Initialiserung des PROFINET Stacks
- Erzeugung der Tasks und Kommunikationskanäle innerhalb des PROFINET Stacks
- Kommuniziert mit darüberliegenden Schichten über ein stack-internes Speicherinterface

2.2.16 PNPB

- Implementierung der PROFINET IO-Device Anwenderschnittstelle zur Kundenapplikation
- Kommuniziert über ein internes Interface mit dem PNDV

2.3 Sonstige Software-Komponenten

2.3.1 Operating System

Das Echtzeit-Betriebssystem ist nicht Bestandteil des PROFINET IO-Software-Stacks. Es wird in der Regel über einen Dritthersteller bezogen. Das mitgelieferte Anwenderbeispiel ist zugeschnitten auf die jeweilige Beispiel-Plattform, siehe Kapitel Beispiel-Plattformen (Seite 14).

2.3.2 Board Support Package (BSP)

Das Board Support Package (BSP) kapselt die hardware-spezifischen Betriebssystemaufrufe für eine vorgegebene Plattform. Ein plattformspezifisches BSP wird in der Regel vom Hersteller des Betriebssystems mitgeliefert. Für das Evaluation Board EB 200P ist ein BSP für das verwendete Betriebssystem im Lieferumfang des Evaluation Kit enthalten. Dieses kann als Beispielvorlage für die Anpassung an eine kundenspezifische Plattform verwendet werden.

2.4 Applikationsbeispiele

Im Evaluation 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 realisiert:

- Standard-Interface (SI), universell einsetzbar, für einfaches Handling bei RT und IRT.
- Direct Buffer Access Interface (DBAI), mit Performance-Vorteilen 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 im Kapitel Wichtige Randbedingungen für das Einbinden einer Applikation (Seite 36).

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

Applikation	Beschreibung	Eigenschaften
App1_Standard	Universelles Beispiel, basierend auf dem Standard-Interface (SI).	• empfohlenes Template für die meisten Applikationen • 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 ARs oder IOCRs nicht kennen. • Datenkonsistenz automatisch gewährleistet durch gepufferten Zugriff
App2_DBAI	Direct Buffer Access Interface	• Performance-Vorteile gegenüber SI (nur) bei großer Modul-/Submodulanzahl • verwendbar für RT und IRT • IOCR-Sichtweise auf die IO-Daten, d. h. Applikation muss ARs und IOCRs selber verwalten. • Datenkonsistenz automatisch gewährleistet durch gepufferten Zugriff.
App3_IsoApp	Isochrone Applikation bei IRT	• Aufbau ähnlich "App1_Standard", d. h. gleiche Zugriffsmethode auf IO-Daten sowie azyklische Dienste • Verwendet IO-Module, die IRT voraussetzen, daher nur im IRT-Betrieb verwendbar • Verwaltet Iso-Record-Index 0x8030 zur Vorgabe von T_Input, T_Output und Zykluszeit • Auslösen von Interrupts oder Hardware-Signalen GPIO 5, 7 zu den Zeitpunkten T_Input und T_Output

2.4.1 Genereller Aufbau der Anwenderbeispiele

Die generelle Softwarearchitektur des PNIO-Stacks wurde bereits im Kapitel Softwarearchitektur (Seite 15) 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 = Zeitpunkt, an dem die aktuellen Ausgangsdaten im Device zur Verfügung stehen), 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 Event Handler laufen im Kontext des PROFINET IO-Stacks.

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

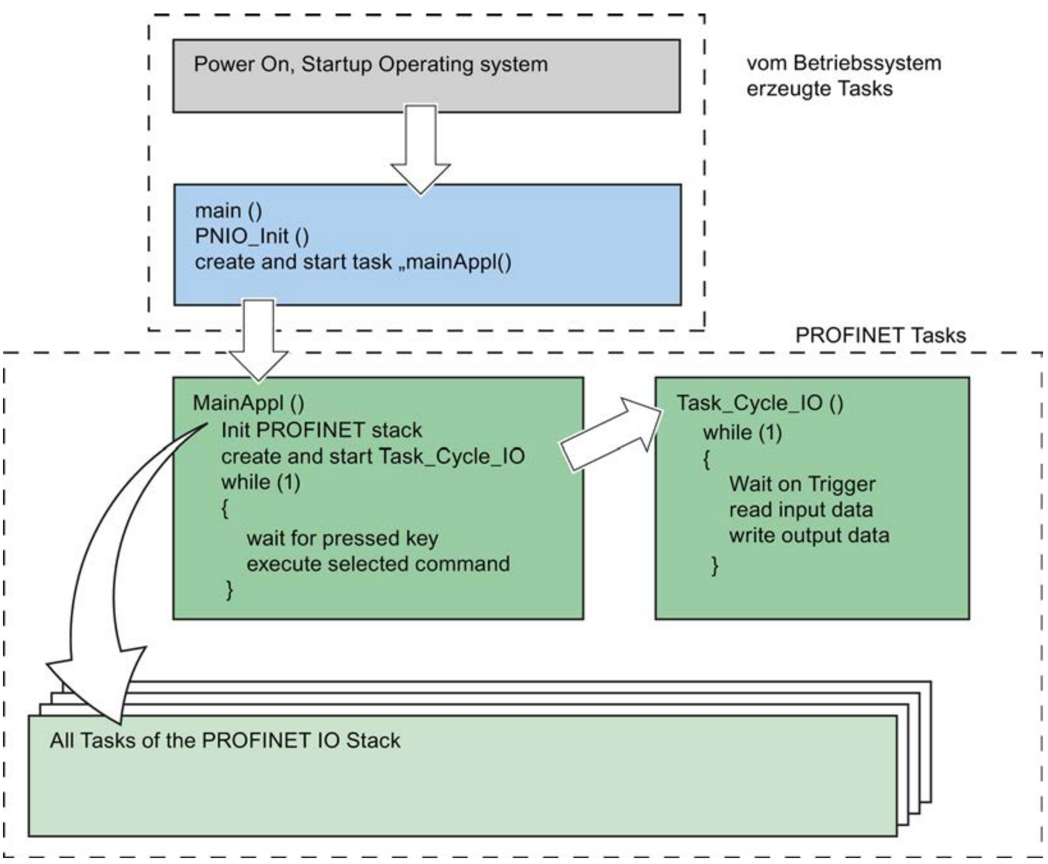


Bild 2-2 Tasks des PROFINET IO-Anwenderbeispiels

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 Header-Dateien liegen in einem gemeinsamen Unterverzeichnis "\Common".

Eine Beschreibung der Verzeichnisstruktur des kompletten PROFINET-Stacks inklusive der Applikationsbeispiele finden Sie im Kapitel Verzeichnisstruktur des PROFINET IO-Quellcodes (Seite 27).

Die Auswahl des zu kompilierenden Anwenderbeispiels erfolgt in der Header-Datei "\application\common\usrapp_cfg.h" mittels folgendem Eintrag:

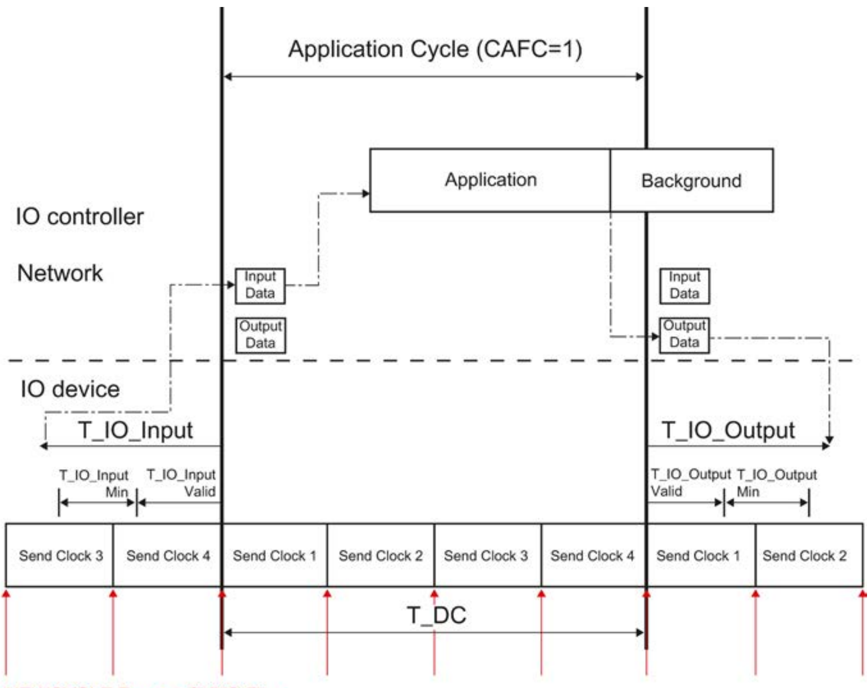
```
#define EXAMPL_DEV_CONFIG_VERSION 1 // the number 1..n specifies the selected example
```

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 IO-Controller ans Device übertragen (Record IsochronousModeData, Index 8030_H). 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:



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.

Zusätzlich werden die GPIOs 5 und 7 so konfiguriert, dass sie zum Zeitpunkt T_IO_Input bzw. T_IO_Output jeweils einen kurzen Puls ausgeben. Die Sync-Signalflanke kommt bei jedem Beginn des Sendezyklus am Bus, also ohne Berücksichtigung eines eventuellen Reduction Ratio-Wertes.

Die Auswahl des dafür zu kompilierenden Anwenderbeispiels erfolgt in der Header-Datei "\application\common\usrapp_cfg.h" mittels folgendem Eintrag:

```
#define EXAMPL_DEV_CONFIG_VERSION 3 // the number 1..n specifies the selected example
```

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

2.4.3 App1: SI-basiertes Beispiel für RT und IRT

Einsatzbereich

Dies ist das Standard-Applikationsbeispiel, welches für die meisten Applikation 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 ARs (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 Callback-Funktionen. 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 IO-Controller stammenden Output-Daten jedes einzelnen Submoduls wird anschließend vom PNIO-Stack die Callback-Funktion "PNIO_cbf_data_read()" aufgerufen, für die Input-Daten des Devices wird pro Submodul die Callback-Funktion "PNIO_cbf_data_write()" aufgerufen. Darin muss die Applikation jeweils die IO-Daten für genau ein Submodul lesen oder schreiben. Dabei werden auch die Providerstatus der Output-Daten vom IO-Controller sowie Provider- und Consumerstatus der Input-Daten vom Device als Übergabeparameter bzw. Returnwert der Callbacks übergeben.

2.4.4 App2: DBAI-basiertes Beispiel für RT und IRT

Einsatzbereich

Dieses Applikationsbeispiel bietet gegenüber dem Standard-Interface (SI) eventuell Performance-Vorteile, 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 Abbild der IOCRs und greift dort direkt auf IO-Daten bzw. Provider-/Consumerstatus IOPS/IOCS zu. Die Applikation baut also die IOCRs selber auf, muss dafür aber deren Aufbau kennen und verwalten. Die dazu notwendigen Informationen werden beim Verbindungsaufbau an die Device-Applikation in der Connect- sowie der Ownership-Indication übergeben.

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 Input-Daten, Consumer-IOCR für Output-Daten).

Nach Bearbeitung wird der Puffer durch Aufruf von "PNIO_dbai_buf_unlock()" wieder freigegeben. Bei einer Provider IOCR (d. h. Input-Daten auf dem Device) wird das neue IOCR-Abbild nun auf dem Bus aktiviert.

Software-Erstellung von PROFINET IO-Devices 3

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

3.1 Verzeichnisstruktur des PROFINET IO-Quellcodes

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

Tabelle 3- 1 Struktur und Beschreibung der Verzeichnisse

Verzeichnisse			Dateien	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 verwendet.**
ACP	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
CLRPC	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
CM	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
DCP	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes

Verzeichnisse			Dateien	Beschreibung
EDDP (nur für ERTEC 200P Plattform)	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
	\	xxx \	*.c	Weitere interne Unterverzeichnisse des Basispaketes, hier zusammengefasst dargestellt als \xxx\
EVMA (nur für ERTEC 200P-Plattform)	\	_com \	*.h	globale Header-Dateien der Komponente
	\	*.c, *.h		Quellcode und interne Header-Dateien der Komponente
GSY	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
	\	clp \	*.c	Quellcode und interne Header-Dateien des Basispaketes
	\	krisc32 \	*.c	Quellcode und interne Header-Dateien des Basispaketes
IOM	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
LLPD	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
LSAS	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	adapt	*.h	Quellcode für die Systemanpassung (Konfiguration) der einzelnen Basispakete
	\	Adapt_h	*.h	Header-Dateien für die Systemanpassung (Konfiguration) der einzelnen Basispakete
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
MRP	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
NARE	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes

Verzeichnisse			Dateien	Beschreibung
OHA	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
PCPNIO_LSA	\	*.c, *.h		Adaption der Trace-Schnittstelle für das (PNIO-Stack-interne) Debugging
Platform	\	*.h		Header-Dateien für die Auswahl der Plattform
PNDV	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
PNIO	\	_com \	*.h	globale Header-Dateien
	\	*.c		Versionseintrag für Software-Stack des Evaluation Kits
PNIO_API_inc	****	***.h**		**Header-Dateien der PROFINET IO-Applikationsschnittstelle**
PNPB	\	_adapt \	*.c, *.h	Quellcode und interne Header-Dateien für die Lower Layer Adaption
	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
POF	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	dmi \	*.c	Quellcode und interne Header-Dateien des Basispaketes
	\	edd \	*.c	Quellcode und interne Header-Dateien des Basispaketes
	\	prm\	*.c	Quellcode und interne Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
SOCK	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
SYS	\	_cfg \	*.h	Konfigurations-Dateien für ERTEC 200P, LSAS, PNDV, TRACE
	\	_inc \	*.h	Header-Dateien des Basispaketes
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes

Verzeichnisse			Dateien	Beschreibung
SysAdapt1	****	**cfg **	***.c, *.h**	**vom Anwender anzupassende Module** **Plattformabhängiger Beispielcode für** **OS–Abstraction Layer** **Upper- und Lower Layer Interface zum TCP/IP-Stack** **MIB2-Adaption** **BSP-Adaption**
	\	inc \	*.h	globale Header-Dateien der Systemanpassung
	\	src \	*.c	Quellcode der Systemanpassung, der i.d.R. unverändert übernommen werden kann
TCPIP	\	_com \	*.h	globale Header-Dateien des Basispaketes
	\	Allports \	*.c, *.h	Quellcode / Header-Dateien des Paketes
	\	h \	*.c, *.h	interne Header-Dateien des Basispaketes
	\	Ip \	*.c, *.h	Quellcode / Header-Dateien des IP-Protokolls
	\	Ipmc \	*.c, *.h	Quellcode / Header-Dateien des IPMC-Protokolls
	\	Misclib \	*.c, *.h	Quellcode / Header-Dateien zur Checksummenberechnung
	\	Net \	*.c, *.h	Quellcode / Header-Dateien Lower Layer-Adaption
	\	Snmp \	*.c, *.h	Quellcode / Header-Dateien des SNMP-Protokolls
	\	Snmpv1 \	*.c, *.h	Quellcode / Header-Dateien des SNMP-Protokolls
	\	tcp \	*.c, *.h	Quellcode / Header-Dateien des TCP-Protokolls
	\	*.c, *.h		Quellcode und interne Header-Dateien des Basispaketes
Trace_dk	\		trace_dk.c trace_dk.h	Speicherung von Fehlermeldungen in Umlaufpuffer oder Ausgabe auf die TeraTerm-Konsole (angeschlossen über die RS232-Schnittstelle) als Debug-Hilfe
	\	Traceout_con \	*.c, *.h	Ausgaben der Meldungen auf TeraTerm-Konsole
	\	Traceout_mem \	*.c, *.h	Ausgaben der Meldungen in einen Umlaufpuffer und zur TeraTerm-Konsolenausgabe (Default)

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 Class 2, IRT Class 3. 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 Class 2, IRT Class 3 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.
PnUsr_Api.c	Unterprogramme	Unterprogramme für das Anwenderbeispiel. Die enthaltenen Funktionen können bei Bedarf als Funktionsbibliothek in der Kundenapplikation verwendet werden.

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	Header-Datei	Header-Datei 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 App3_IsoApp

Modul	Inhalt	Beschreibung
usriod_main_isoapp.c	Hauptprogramm für RT-, IRT-Beispiel	Standard-Anwenderbeispiel, Hauptprogramm für IRT Class 3 mit taktsynchronen IO-Submodulen
iodapi_event_isoapp.c	Meldung von Ereignissen an die Applikation	Event Handler für Anwenderbeispiele Standard IRT C3 mit taktsynchronen IO-Submodulen. Aktivieren der ERTEC-Komparatoren zur Behandlung von T_Input- und T_Output-Zeiten gemäß den Daten in Record-Index 0x8030. Dabei werden auch die notwendigen Konfigurationen der GPIOs (5, 7) mittels der Funktion "PNIO_IsoActivateGpioObj()" durchgeführt. **Hinweis**: Zeitgesteuerte Interrupts können mittels "PNIO_IsoActivateIsrObj()" aktiviert werden. Ein Beispiel dazu finden Sie in "usrio_main_isoapp.c", Taste "W" bzw. "w" auf der Konsole.

3.2.4 Dateien von App_common

Tabelle 3- 2 Dateien der Beispielapplikation

Modul	Inhalt	Beschreibung
iodapi_event.h	Header-Datei	Header-Datei für iodapi_event.c. Hier sind normalerweise keine Anpassungen nötig.
iodapi_rema.c	Remanente Daten	Übergabe von remanenten Daten (PDEV-Records) vom PNIO Stack an die Applikation zwecks Sicherung im nichtflüchtigen Speicher.
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.
Perform_measure.c	Messen der Prozessorauslastung	Optionale Performancemessung in der Idle Task (nur für Anwenderbeispiel vorgesehen, nicht für ein reales Device).
Perform_measure.h	Messen der Prozessorauslastung	Header-Datei zu perform_measure.h
Tcp_flash_fw.c	FW-Download über TCP	Hauptprogramm für die TCP-basierten Services zum Übertragen und Flashen einer neuen Firmware.
TCP_IF.c	FW-Download über TCP	TCP-basierte Services zum Übertragen einer neuen Firmware.
Tcp_IF.h	FW-Download über TCP	Header-Datei zu tcp_if.c
Tcp_Flash_fw.c	FW flashen	Übertragung und Flashen einer neuen Firmware über TCP/IP
usriapp_cfg.h	Auswahl eines Anwenderbeispiels	Über ein Define wird das entsprechende Anwenderbeispiel (RT, IRT Class3, DBA Interface) ausgewählt.
usriod_cfg.h	Konfiguration des Beispiels	Defines für die Konfiguration des Devices
usriod_diag.c	Applikationsprogramm für Diagnose	Beispiel für die Behandlung von Standard-Kanaldiagnose inklusive Diagnose-Alarm.
usriod_diag.h	Header-Datei	Header-Datei für usriod_diag.c, enthält u. a. Datenstrukturdefinitionen für die Standard-Kanaldiagnose
usriod_PE.c	PROFIenergy	Anwenderbeispiel zur Behandlung des PROFIenergy Records
usriod_PE.h	PROFIenergy	Header-Datei für usrioed_PE.c
usriod_utils.c	Utilities	Hilfsprogramme zur Messung der Systemlast für Debug-Zwecke
usriod_utils.h	Header-Datei	Header-Datei für usriod_utils.c

3.2.5 Anwenderschnittstelle

Die Header-Dateien zur Beschreibung der Anwenderschnittstelle sind im Unterverzeichnis "...\PNIO_API_inc" abgelegt. Diese Dateien dürfen nicht verändert werden.

Tabelle 3- 3 Header-Dateien der Anwenderschnittstelle

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.

3.2.6 Anzupassende Module der Betriebssystem-Schnittstelle

Tabelle 3- 4 Dateien der Betriebssystem-Abstraktionsschnittstelle

Modul	Inhalt	Beschreibung
xxx_os.c	OS Services	Abstraktionsschnittstelle für die Aufrufe von Betriebssystem-Services, xx steht dabei für die Plattform (z. B. ecos, ...). Hier erfolgt die Abbildung von PNIO-Aufrufen auf plattformabhängige Betriebssystemfunktionen.
os_cfg.h	OS Konfiguration	Systemkonfigurationen für PNIO: Festlegung von Systemressourcen für PNIO (z. B. Mutex).
os_taskprio.h	OS Konfiguration	Systemkonfigurationen für PNIO: Einstellung von Taskprioritäten
compiler.h	Compilerspezifische Definitionen	Festlegung von compilerspezifischen Einstellungen.
compiler_stdlibs.h	Einbindung von Standard-Header-Dateien	Festlegung der einzubindenden Standard-Header-Dateien.

3.2.7 Anzupassende Module der Socket-Schnittstelle

Tabelle 3- 5 Dateien der Socket-Abstraktionsschnittstelle

Modul	Inhalt	Beschreibung
xx_ossock_iniche.c	Socket Adaption	Optional, wird nur für Logadapt verwendet. Hier sind normalerweise keine Änderungen notwendig.

3.2.8 Module der BSP-Schnittstelle

Tabelle 3- 6 Dateien zur Anpassung an das Board Support Package

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

3.2.9 Speicherung remanenter Daten

Tabelle 3- 7 Dateien zur Anpassung an remanente 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 verwendet und ist daher bei Bedarf änderbar.
xx_flash.h	Speicherung remanenter Daten	Header-Dateien mit Funktionsdeklarationen für die Behandlung des Flashs. Normalerweise sind hier keine Anpassungen notwendig.

3.2.10 Sonstige Dateien der Systemanpassung

Tabelle 3- 8 Sonstige Dateien der Systemanpassung

Modul	Inhalt	Beschreibung
hamaport.c	GPIO Settings	Funktionen zur Konfiguration der GPIOs. Normalerweise sind hier keine Anpassungen notwendig.
ecos_os_debug.c	Optionale Tools	Erweiterte Ausgaben über verwendete Ressourcen des Betriebssystems (Tasks, Speicher, ...). Nur für eCos-Plattform verwendbar.
Os_utils.c, Os_utils.h	Optionale Tools	Optionale Tools zur Verwaltung von Umlaufpuffer, die für Debugging-Zwecke verwendet werden können.

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 **Callback-Funktion** "PNIO_cbf_xxxx()" ausgeführte **Anwendercode** sollte möglichst kurz sein, da alle vom CM eintreffenden Callback-Ereignisse in einer Message Queue sequenziert werden und der Anwendercode im Kontext einer PNPB-Interfacetask aufgerufen wird. Eine Callback-Funktion kann daher erst aufgerufen werden, wenn die vorhergehende Callback-Funktion beendet wurde.
2. In einer **Callback-Funktion** "PNIO_cbf_xxxx()" sollten **keine API-Funktionen** "PNIO_xxx()" aufgerufen werden. Zulässige Ausnahmen sind "PNIO_rec_set_rsp_async()", "PNIO_get_last_error()", "PNIO_printf (debugging)" sowie Stecken/Ziehen von Submodulen im Kontext der Ownership-Indication.
3. Jede **Anwender-Task**, 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. **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. Applikations-Tasks sollten nach Möglichkeit unterhalb der Stackprioritäten liegen.
   Bei höher prioren Anwender-Tasks ist zu beachten, dass das **Laufzeitverhalten des Stacks** negativ beeinflusst werden kann.

3.4 Portieren der PROFINET IO-Software auf eine andere Plattform

Für eine ERTEC-basierte Plattform ist die Portierung auf ein anderes Betriebssystem normalerweise nicht notwendig. Es wird daher empfohlen, möglichst die bereits adaptierte eCos-Betriebssystemplattform zu nutzen.
Dennoch ist eine Portierung grundsätzlich möglich. Der folgende Abschnitt befasst sich daher 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 Kunden-Hardware unter Beibehaltung des Mikrocontrollers und Betriebssystems (einfachster Fall)
- Verwendung anderer Compiler
- Verwendung anderer Mikrocontroller (nicht für ERTEC-Plattformen)
- Verwendung anderer Betriebssysteme

3.4.1 Portierung auf Kunden-Hardware unter Beibehaltung von Mikrocontroller und Betriebssystem

Um die Software auf Ihre eigene Hardware-Plattform ohne Änderung des Betriebssystems zu portieren, müssen Sie das Board Support Package an Ihre Hardware anpassen. Sonstige Änderungen im Betriebssystem-Interface, BSP-Interface oder Application Interface (siehe Bild 2-1 (Seite 16)) sind in der Regel nicht notwendig.

Siehe auch

ERTEC 200P (Seite 16)

3.4.2 Verwendung anderer Compiler/Linker

Die Einstellung des Compilers wird in der Datei "auto_platform_select.h" vorgenommen. Diese ist im Unterverzeichnis "(...)Pnio_src\Platform" für jede Plattform getrennt vorhanden, wobei über Selektion des Include-Pfades jeweils nur eine "auto_platform_select.h" eingebunden wird.

3.4.2.1 Auswahl der Toolkette

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

```
#define TOOL_CHAIN_GNU_ECOS              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.

Es ist prinzipiell auch möglich, einen eigenen Compilerschalter in "compiler.h" zu 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. ERTEC-Plattform
#define PNIO_BIG_ENDIAN     1     // Big Endian
```

3.4.2.3 Data Alignment-Anforderungen

Die Festlegung des Data-Alignments wird in verschiedenen Compilern oft unterschiedlich behandelt. Manche Compiler verwenden eine #pragma pack() bzw. #pragma unpack()-Anweisung, wobei alle Definitionen zwischen beiden Anweisungen entsprechend gepackt werden.

Andere Compiler hingegen erwarten eine entsprechende Definition bei jeder Datenstruktur, die (wiederum compilerabhängig) vor oder nach der eigentlichen Definition steht.
Für alle drei genannten Fälle wurden im PN-Stack die folgenden Mechanismen implementiert:

- #include "sys_pck.h", #include "sys_unpck.h"
- #define ATTR_PNIO_PACKED_PRE
- #define ATTR_PNIO_PACKED

Abhängig vom Compiler ist dabei genau eine der genannten Möglichkeiten auszuwählen. Die beiden anderen sind als Leermakros oder leere Header-Datei zu implementieren. Die Makros ATTR_PNIO_PACKED bzw. ATTR_PNIO_PACKED_PRE sind in der Datei "pnio_src\sysadapt1\cfg\compiler.h" implementiert, die #pragma pack/unpack-Anweisungen hingegen in der Datei "pnio_src\sysadapt1\inc\sys_pck.h" bzw. "sys_unpck.h".

3.4.2.4 Datenverarbeitungsbreite

Die Software wurde bisher ausschließlich auf IO-Controllern mit 32 bit-Datenverarbeitung portiert. Alle Daten- und Addresspointer 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 können prinzipiell in der Systemanpassung in "os.h" unterschiedliche Speicherpools definiert werden, welche in der Software beim Allokieren von Speicher referenziert werden. Dieser Mechanismus wird derzeit allerdings nicht verwendet, es ist lediglich folgender Pool definiert:

```
#define MEMPOOL_DEFAULT        0                // maybe cached
```

3.4.3 Verwendung anderer Betriebssysteme

Die Taskprioritäten der PROFINET IO-Tasks können in "os_taskprio.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 Schnittstelle zum Betriebssystem (Seite 99) 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 und IRT (RT_CLASS3)
- Synchroner IO-Datenzugriff ohne Konsistenzmechanismen, nur für IRT (RT_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.

Tabelle 3- 9 Aufzurufende API-Funktionen in der Anlaufphase

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-Anwender-Task "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_setup()	Anlauf des PNIO-Stacks, Start aller PNIO-Tasks und Belegen der Ressourcen
Instanzen für Devices, APIs, Module, Submodule erzeugen		
4	PNIO_device_open()	Erzeugen einer Device-Instanz
5	PNIO_sub_plug()	Stecken der PROFINET IO-Submodule des IO-Devices
6	PNIO_set_dev_state()	Setzen des Devices in den Zustand OPERATE
Warten auf Verbindungsaufbau vom IO-Controller		
7	Warten auf Aufruf der PNIO_cbf_ar_ connect_ind()-Callback-Funktion.	Dieser Callback wird von der IO-Schnittstelle aufgerufen, sobald ein IO-Controller eine Verbindung zu dem IO-Device-Anwenderprogramm aufbaut. Durch Aufruf dieses Callback werden dem IO-Device-Anwenderprogramm Application-Relation-globale Parameter zur Prüfung übergeben.
8	Warten auf Aufruf der PNIO_cbf_ar_ownership_ind( )-Callback-Funktion.	Nach dem Verlassen der Connect Indication wird die Ownership Indication aufgerufen. Hier wird der Applikation eine Liste aller Submodule, deren Eigenschaften (Steckplatz, Modul/Submodul-ID, OwnerSessionKey etc.) übergeben. Nur wenn die Applikation die Ownership für einen Subslot ablehnt und damit das Submodul nicht bearbeiten möchte, so setzt es in dieser Callback-Funktion den OwnerSessionKey auf 0. Andernfalls bleibt der OwnerSessionKey unverändert. Ist ein falsches, nicht kompatibles Submodul gesteckt, so muss die Applikation den Parameter IsWrongSubmod auf PNIO_TRUE setzen. Alternativ kann die Prüfung der Submodul-ID auch im PN-Stack durchgeführt werden. Details siehe Funktionsbeschreibung von PNIO_cbf_ar_ownership_ind().

Schritt	Aktion	Zweck
Parametrieren der Submodule		
9	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 Parametrier-Record für ein Submodul übermittelt. Durch Aufruf dieses Callback werden eventuelle Parametrierdaten pro Submodul an das IO-Device-Anwenderprogramm übermittelt.
10	Warten auf Aufruf der PNIO_cbf_param_ end_ind()-Callback-Funktion.	Dieser Callback wird von der IO-Schnittstelle einzeln für jedes parametrierte Submodul aufgerufen, sobald ein IO-Controller das Ende der Parametrierphase signalisiert. Im Returnwert gibt die Applikation zurück, ob das Modul ordnungsgemäss arbeitet oder nicht. Details siehe Funktionsbeschreibung von PNIO_cbf_param_end_ind().
11	PNIO_initiate_data_write()	Mit diesem Aufruf veranlasst das Anwenderprogramm das Aufrufen der "PNIO_CBF_DATA_WRITE()"-Callback, 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 ApplicationReady auf gültige Werte gesetzt werden und der lokale Provider-Status jeweils auf "GOOD" gesetzt wird.
12	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 für alle funktionsfähigen Submodule vor dem Versenden des ApplicationReady die lokalen Consumer-Status auf "GOOD" gesetzt wurden.
13	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

3.5.2 Produktivbetrieb

Überblick

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

- IO-Daten lesen/schreiben
- Alarme senden und deren Quittungen empfangen
- Record 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 PNIO-Controllers) erfolgt in drei Schritten unter Verwendung der Konsistenzsteuermechanismen des ERTEC 200P:

Tabelle 3- 10 Aufzurufende API-Funktionen zum Lesen von IO-Daten bei RT und IRT

Schritt	Aktion	Zweck
1	Warten auf das Ereignis PNIO_CP_CBE_TRANS_END_IND	Mit diesem Ereignis signalisiert die IO-Schnittstelle dem Device-Anwenderprogramm den Abschluss des Sendens der Input-Daten am Bus. Die zugehörige Callback-Funktion 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.

IO-Daten bei RT und IRT schreiben

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

Tabelle 3- 11 Aufzurufende API-Funktionen zum Schreiben von IO-Daten bei RT und IRT

Schritt	Aktion	Zweck
1	Warten auf das Ereignis PNIO_CP_CBE_TRANS_END_IND	Mit diesem Ereignis signalisiert die IO-Schnittstelle dem Device-Anwenderprogramm den Abschluss der IO-Datenübertragung am Bus. Die zugehörige Callback-Funktion 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 ihm, unter anderem, den Pointer auf einen Datenpuffer, in den die Eingangsdaten kopiert werden sollen, die an den IO-Controller gesendet werden sollen.

Record-lesen/schreiben-Auftrag des PNIO-Controllers bearbeiten

Bearbeiten eines Record-lesen-Auftrags

Sobald die IO-Schnittstelle einen Record-lesen-Auftrag vom IO-Controller empfängt, ruft sie die vom IO-Device-Anwenderprogramm angemeldete Callback-Funktion "PNIO_CBF_REC_READ()" auf. Die Applikation kann die Record-Daten innerhalb der Callback-Funktion bereitstellen oder asynchron zu einem späteren Zeitpunkt liefern.

- Synchrones Lesen von Record-Daten:

Tabelle 3- 12 Aufzurufende API-Funktionen zum synchronen Lesen von Record-Daten

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

- Asynchrones Lesen von Record-Daten:

Tabelle 3- 13 Aufzurufende API-Funktionen zum asynchronen Lesen von Record-Daten

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 Callback-Funktion aufgerufen werden.
3	PNIO_rec_read_rsp()	Die Applikation übergibt asynchron die geforderten Record-Daten an den Stack. "PNIO_rec_read_rsp" kann von jeder beliebigen Anwender-Task aus aufgerufen werden.

Bearbeiten eines Record-schreiben-Auftrags

Sobald die IO-Schnittstelle einen Record-Schreibauftrag vom IO-Controller empfängt, ruft sie die Callback-Funktion "PNIO_CBF_REC_WRITE()" auf. Die Applikation kann den Abschluss des Schreibvorgangs synchron mit Verlassen der Callback-Funktion an den Stack melden oder dies asynchron zu einem späteren Zeitpunkt durchführen.

- Synchrones Schreiben von Record-Daten:

Tabelle 3- 14 Aufzurufende API-Funktionen zum synchronen Schreiben von Record-Daten

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

- Asynchrones Schreiben von Record-Daten:

Tabelle 3- 15 Aufzurufende API-Funktionen zum asynchronen Schreiben von Record-Daten

Schritt	Aktion	Zweck
1	PNIO_cbf_rec_write()	Record-Schreibauftrag 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 Callback-Funktion aufgerufen werden.
3	PNIO_rec_write_rsp()	Die Applikation meldet asynchron den Abschluss des Requests inkl. Statusinfo an den Stack. "PNIO_rec_write_rsp" kann von jeder beliebigen Anwender-Task aus aufgerufen werden.

Alarme senden und Alarmquittierung empfangen

Für jeden versendeten Alarm erhält das IO-Device-Anwenderprogramm eine Alarmquittung, indem die IO-Schnittstelle die Funktion "PNIO_cbf_async_req_done()" aufruft. Die Zuordnung, welche Quittung zu welchem Alarm gehört, geschieht über den AlarmTyp sowie den Fehlerort "AR-Nummer/API/Slot/Subslot."

Hinweis

Alarme dürfen erst nach Verlassen der Funktion PNIO_cbf_param_end() 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 enthält die "gemeldeten Informationen" und den zugehörigen Callback-Namen.

Tabelle 3- 16 API-Callback-Funktionen bei Verbindungsauf- und abbau

Callback-Name	Gemeldete Information
PNIO_cbf_ar_connect_ind	Verbindungsaufbauwunsch vom IO-Controller
PNIO_cbf_ar_ownership_ind	Soll-Ausbau des PNIO-Device, RT-Klasse (RT Class 1/3) und weitere Submodul-Eigenschaften
PNIO_cbf_rec_write_ind	Schreiben eines Parameter-Records für einen Subslot
PNIO_cbf_param_end_ind	Ende der Parametrierung durch PNIO-Controller
PNIO_cbf_indata_ind	ApplicationRelation-InData – Erste gültige Daten wurden vom PNIO-Controller empfangen.
PNIO_cbf_disconn_ind	DisconnectEvent – Verbindungsabbruch

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

(Die Abschlussphase des IO-Device ist derzeit nicht implementiert.)

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 Records
- Alarme senden und deren Quittungen empfangen

Weitere Erläuterungen hierzu finden Sie im Kapitel Callback-Mechanismus (Seite 54).

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 Device beschrieben bzw. ausgelesen. Es werden dabei keinerlei Daten über das Netz gesendet.

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

Hinweis

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

Hinweis

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

Hinweis

Die Daten eines Submoduls werden immer konsistent übertragen. Gemäß PROFINET IO-Norm muss ein PROFINET-Gerät (Controller und Device) mindestens 254 byte konsistent übertragen können. Dies ist im Zusammenspiel mit einem PROFINET IO-Controller zu beachten.

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 des 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 die Callback-Funktion "PNIO_cbf_data_write()" auf. Darin werden die Input-Daten und der zugehörige **lokale Providerstatus** dieser Daten in das lokale Prozessabbild geschrieben.

Lokaler Status

Im Normalfall wird der lokale Providerstatus der Input-Daten vom IO-Device-Anwenderprogramm auf "GOOD" gesetzt.

Wenn die Input-Daten fehlerhaft oder ungültig sind, muss das IO-Device-Anwenderprogramm den lokalen Providerstatus auf "BAD" setzen.
Der Kommunikationspartner könnte dann beispielsweise projektierte Ersatzwerte ausgeben.

Remoter Status des Kommunikationspartners

Mit dem "remote Consumerstatus" meldet der Kommunikationspartner, ob er die Input-Daten "gut" verarbeiten konnte oder ob bei ihm eine Störung vorliegt.

Bei der Benutzung des Standard-Interfaces (IO-Datenaustauschanstoß mittels "PNIO_initiate_data_read" und "PNIO_initiate_data_write") werden diese Informationen nicht zur Applikation übertragen, da diese dort in der Regel auch nicht benötigt werden. Die remoten Consumerstatus der Input-Daten können aber beim DBA-Interface aus den IOCR-Daten ausgelesen werden..

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 mit Output-Daten die Callback-Funktion "PNIO_cbf_data_read()" auf. Mit Aufruf der Callback-Funktion werden die Ausgabedaten und der zugehörige **remote Providerstatus** vom Kommunikationspartner aus dem lokalen Prozessabbild (stack-intern) gelesen und an die Applikation übergeben.
Zusätzlich wird für Kommunikationspartner der **lokale Consumerstatus** dieser Ausgabedaten in das lokale Prozessabbild geschrieben.
Es sind beim zyklischen Lesen also zwei Status im Spiel:

Kommunikationsrichtung	Werte
vom Kommunikationspartner	• Ausgabedaten • Remoter Providerstatus
zum Kommunikationspartner	Lokaler Consumerstatus

Remoter Status des Kommunikationspartners

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

Wenn der Kommunikationspartner den Providerstatus "BAD" meldet, dürfen die IO-Daten im IO-Device-Anwenderprogramm nicht weiterverarbeitet werden; das IO-Device-Anwenderprogramm könnte dann beispielsweise Ersatzwerte ausgeben.

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 lokale 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 optionales DBA-Interface

Alternativ können die IO-Daten auch über ein optionales DBA-Interface (DirectBufferAccess) 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 Callback-Funktion aufgerufen wird. Nähere Informationen dazu finden Sie in Kapitel Zyklischer Datenaustausch über das optionale DBA-Interface (Seite 81).

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

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

Die Anwenderschnittstelle für RT und IRT ist identisch. In beiden Fällen kann der IO-Datenaustausch durch dieselbe Callback-Funktion zum Ereignis "TransmissionEnd" getriggert werden.

3.9 Diagnosedaten verwalten

Beschreibung

Bei PROFINET IO gibt es folgende Möglichkeiten, eine Diagnose zu melden:

- Standard-Kanaldiagnose
- Erweiterte Kanaldiagnose
- Herstellerspezifische Diagnose

Der PROFINET-Stack sendet automatisch beim Erhalt eines neuen Diagnoseeintrags einen entsprechenden "Diagnosealarm-kommend" an den PROFINET IO-Controller. Bei der Wegnahme der Diagnose wird analog dazu ein entsprechender "Diagnosealarm-gehend" versendet.

3.9.1 Channel Diagnosis Data

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()" den Diagnoseeintrag aus dem Submodul entfernen.

Für die Verwaltung der Diagnosedaten stehen folgende Funktionen zur Verfügung:

Tabelle 3- 17 Aufzurufende API-Funktionen für die Erstellung eines Diagnose-Records

Funktion	Zweck
PNIO_diag_channel_add()	Kanal-Diagnosedaten im Subslot ablegen
PNIO_diag_channel_remove()	Kanal-Diagnosedaten vom Subslot entfernen
PNIO_ext_diag_channel_add()	Kanal-Diagnosedaten im Subslot ablegen
PNIO_ext_diag_channel_remove()	Kanal-Diagnosedaten im Subslot entfernen

Setzen von Channel Diagnosis Data

Das Setzen von Channel Diagnosis Data erfolgt in zwei Schritten:

Tabelle 3- 18 Aufzurufende API-Funktionen für die Aktivierung eines erstellten Diagnose-Records

Schritt	Aktion	Zweck
1	PNIO_diag_channel_add()	Kanal-Diagnosedaten im Subslot ablegen. Der Stack generiert dazu automatisch einen "Diagnosealarm-kommend" an den IO-Controller.
2	PNIO_cbf_async_req_done()	"Diagnosealarm-kommend" - Quittung auswerten.

Entfernen von Channel Diagnosis Data

Das Entfernen von Channel Diagnosis Data erfolgt in zwei Schritten:

Tabelle 3- 19 Aufzurufende API-Funktionen für das Entfernen eines Diagnose-Records

Schritt	Aktion	Zweck
1	PNIO_diag_channel_remove()	Kanal-Diagnosedaten vom Submodul entfernen. Der Stack generiert dazu einen "Diagnosealarm-gehend" an den IO-Controller.
2	PNIO_cbf_async_req_done()	"Diagnosealarm-gehend" - Quittung auswerten.

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 dieser Diagnosedaten stehen folgende Funktionen zur Verfügung:

Tabelle 3- 20 Aufzurufende API-Funktionen für die Erstellung eines generischen Diagnose-Records

Funktion	Zweck
PNIO_diag_generic_add()	Herstellerspezifische Diagnosedaten im Subslot ablegen. Der Stack generiert dazu automatisch einen "Generic Alarm-kommend" an den IO-Controller.
PNIO_diag_generic_remove()	Herstellerspezifische Diagnosedaten vom Subslot entfernen. Der Stack generiert dazu automatisch einen "Generic Alarm-gehend" an den IO-Controller.

Setzen von Manufactory Specified Diagnosis Data

Das Setzen von Manufactory Specified Diagnosis Data erfolgt in zwei Schritten:

Tabelle 3- 21 Aufzurufende API-Funktionen für die Aktivierung eines generischen Diagnose-Records

Schritt	Aktion	Zweck
1	PNIO_diag_generic_add()	Herstellerspezifische Diagnosedaten im Subslot ablegen. Der Stack generiert dazu automatisch einen "Generic Alarm-kommend" an den IO-Controller.
2	PNIO_cbf_ async_req_done()	Generic Alarm-kommend - Quittung auswerten.

Entfernen von Manufactory Specified Diagnosis Data

Das Entfernen von Manufactory Specified Diagnosis Data erfolgt in zwei Schritten:

Tabelle 3- 22 Aufzurufende API-Funktionen für das Entfernen eines generischen Diagnose-Records

Schritt	Aktion	Zweck
1	PNIO_diag_generic_remove()	Herstellerspezifische Diagnosedaten vom Submodul entfernen. Der Stack generiert dazu automatisch einen "Generic Alarm-gehend" an den IO-Controller.
2	PNIO_cbf_ async_req_done()	Generic Alarm-gehend - Quittung auswerten.

3.10 Besonderheiten beim Ziehen und Stecken von Modulen im Produktivbetrieb

Alarm beim Ziehen von Submodulen

Der PROFINET-Stack generiert einen PROFINET IO-Ziehenalarm, sobald das IO-Device-Anwenderprogramm mit Aufruf der folgenden Funktion(en) ein Modul bzw. ein Submodul zieht:

- PNIO_sub_pull()

Alarm beim Stecken von Submodulen

Die IO-Schnittstelle generiert einen PROFINET IO-Steckenalarm, sobald das IO-Device-Anwenderprogramm mit Aufruf der folgenden Funktion(en) ein Modul bzw. ein Submodul steckt:

- PNIO_sub_plug()

Hinweis

Zwischen dem Aufruf von "PNIO_cbf_ar_ownership_ind" und dem Abschluss der Funktion "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 der PNIO-Stack für jeden vom IO-Controller übertragenen Parametrier-Record die Funktion "PNIO_cbf_rec_write()" aufruft.

Das Ende der Parametrierung wird von der IO-Schnittstelle durch Aufruf der Funktion "PNIO_cbf_param_end()" 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_cbf_param_end()" mit Return (PNIO_SUBMOD_STATE_RUN) abschließen.
- Wenn "NEIN", muss das IO-Device-Anwenderprogramm die lokalen Status für die Ein- und Ausgänge dieses Submoduls auf "BAD" setzen und anschließend die Funktion "PNIO_cbf_param_end()" mit Return (PNIO_SUBMOD_STATE_STOP) abschließen. Wenn dann zu einem späteren Zeitpunkt bei laufender AR das Modul gültige Daten liefern kann, muss der Status auf "GOOD" gesetzt werden und ein Return-of-Submodul-Alarm ausgelöst werden (siehe folgendes Kapitel).

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-Submodul"-Alarms das Submodul nicht neu parametrieren.

Das Submodul ist damit wieder funktionsfähig.

3.11 Callback-Mechanismus

Funktionsweise

Callback-Funktionen werden von der Komponente PNPB des PNIO-Stacks vorgegeben.

Ein Callback-Ereignis ist ein asynchrones Ereignis, welches durch den PNIO-Stack 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.

Tabelle 3- 23 Übersicht der Callback-Funktionen im IO-Device

Callback-Ereignis (asynchron)	Callback-Ereignistyp	Ausgelöst durch ...
Daten lesen	PNIO_cbf_data_read	Anwenderprogramm durch Aufruf von PNIO_initiate_data_read
Daten schreiben	PNIO_cbf_data_write	Anwenderprogramm durch Aufruf von PNIO_initiate_data_write
Record lesen	PNIO_cbf_rec_read	IO-Controller
Record schreiben	PNIO_cbf_rec_write	IO-Controller

Callback-Ereignis (asynchron)	**Callback-Ereignistyp**	**Ausgelöst durch ...**
Quittung für einen Alarmsendeauftrag	PNIO_cbf_async_req_done	Anwenderprogramm durch Aufruf von: • PNIO_process_alarm_send • PNIO_diag_alarm_send • PNIO_ext_diag_alarm_send • PNIO_ret_of_sub_alarm_send • PNIO_upload_retrieval_alarm_send
Alarm vom IO-Controller an Device	PNIO_cbf_dev_alarm_ind	Alarme vom IO-Controller an das Device (reserved)
Vergleich des Soll-Ausbaus	PNIO_cbf_check_ind	PNPB, optional wenn PNIO_cbf_ar_ownership_ind() mit Return (DO_EXEC_CHECK_IND) verlassen wurde.
Verbindungsaufbau Schritt 1	PNIO_cbf_ar_connect_ind	IO-Controller
Verbindungsaufbau Schritt 2	PNIO_cbf_ar_ownership_ind	IO-Controller
Verbindungsabbau	PNIO_cbf_ar_disconn_ind	IO-Controller, Anwenderprogramm
Ende der Parametrierphase	PNIO_cbf_param_end_ind	PNPB
LED Blinken einschalten	PNIO_cbf_start_led_blink	PNPB
LED Blinken ausschalten	PNIO_cbf_stop_led_blink	PNPB
Neue IP-Adresse speichern	PNIO_cbf_save_ip_addr	PNPB
Neuen Stationsnamen speichern	PNIO_cbf_save_station_name	PNPB
REMA-Daten speichern	PNIO_cbf_store_rema_mem	PNPB
Übertragung zyklischer Daten beendet	PNIO_CP_CBE_TRANS_END_IND	PNDV

Hinweis

Binden Sie beim Kompilieren Ihres Anwenderprogramms multithreading-fähige Standardbibliotheken.

Hinweis

Innerhalb einer Callback-Funktion sind Funktionsaufrufe verboten, die zum Aufruf derselben 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.

Schnittstellenbeschreibung

4

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 Callback-Funktion 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 IP-Suite (IP-Adresse, Subnet Mask, Default Router Adresse) vorzugeben.			
Input	PNIO_INT8*	pStationName	Zeiger auf den Stationsnamen (kann ein nicht NULL-terminierter String sein)
	PNIO_UINT32	StationNameLen	Länge des Station Name String
	PNIO_INT8*	pStationType	Zeiger auf den Stationstyp (NULL-terminierter String)
	PNIO_UINT32	IpAddr	IP-Adresse des Gerätes
	PNIO_UINT32	SubnetMask	IP Subnet Mask
	PNIO_UINT32	DefRouterAddr	IP Default Router
Output	PNIO_UINT32	return	Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"

4.1.1.3 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 EB 200P 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 Multidevice-Funktionalität noch nicht implementiert, dennoch ist das Handle korrekt anzugeben.			
Input	PNIO_UINT16	VendorId	Hersteller-ID für das Device, muss bei PNO beantragt werden
	PNIO_UINT16	DeviceId	Device-ID, muss eindeutig innerhalb der PNIO-Produkte eines Herstellers sein.
	PNIO_UINT16	InstanceId	Instanz-ID für das Device (derzeit nur InstanceId = 1 zulässig)
	PNIO_ANNOTATION*	pDevAnnotation	Annotation Struktur, enthält Device-Typ, Bestellnummer, Ausgabestand, etc.
	PNIO_SNMP_LLDP*	pSnmpPar	Zeiger vom Typ "PNIO_SNMP_LLDP" auf SNMP Objekte, die in die LLDP-MIB eingetragen werden.
	PNIO_BOOL	MrpCapabilityActive	"PNIO_TRUE": MRP-Fähigkeit aktiviert, "PNIO_FALSE": MRP-Fähigkeit ist gesperrt.
	PNIO_UINT32*	pDevHndl	Zeiger auf Adresse, in der der IO-Stack das Device-Handle an die Applikation zurückliefert.
Output	PNIO_UINT32	return	Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"

4.1.1.4 PNIO_async_appl_rdy

<table>
<tr><th colspan="3">PNIO_async_appl_rdy()</th><th>Funktionsaufruf: Applikation > IO-Stack, synchron</th></tr>
<tr><td colspan="4">Die Funktion "PNIO_async_appl_rdy()" ist nur erforderlich, falls das ApplicationReady für ein Submodul verzögert werden soll, da die Parametrierung dieses Submoduls noch nicht abgeschlossen ist.
Normalerweise wird die Information, ob ein Submodul nach Abschluss der Parametrierung korrekt angelaufen ist oder nicht, bereits im Returnwert der Callback-Funktion "PNIO_cbf_param_end_ind()" an den PNIO-Stack übergeben. Dazu wird von der Applikation "PNIO_SUBMOD_STATE_RUN" (Submodul OK) oder "PNIO_SUBMOD_STATE_STOP" (Submodul nicht OK) im Returnwert zurückgeliefert. Der Stack kann dann automatisch ein ApplikationReady-Telegramm an den PNIO-Controller generieren und dieses im Fehlerfall in den im Telegramm enthaltenen Moduldiffblock eintragen.
Dauert hingegen die Parametrierphase eines Submoduls länger und soll die Callback-Funktion "PNIO_cbf_param_end_ind()" nicht solange verzögert werden, so kann das ApplikationReady für dieses Modul auch später geschickt werden. Dazu wird zunächst von "PNIO_cbf_param_end_ind()" der Returnwert "PNIO_SUBMOD_STATE_APPL_RDY_FOLLOWS" zurückgeliefert. Ist die Parametrierung dann abgeschlossen, so wird diese Information für die betroffenen Submodule mittels "PNIO_async_appl_rdy()" "nachgeliefert".</td></tr>
<tr><td rowspan="7">Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td>PNIO_UINT16</td><td>ArNum</td><td>Nummer der betroffenen AR</td></tr>
<tr><td>PNIO_UINT32</td><td>Api</td><td>API-Nummer</td></tr>
<tr><td>PNIO_UINT16</td><td>SlotNum</td><td>Slot-Nummer</td></tr>
<tr><td>PNIO_UINT16</td><td>SubslotNum</td><td>Subslot-Nummer</td></tr>
<tr><td>PNIO_SUBMOD_STATE</td><td>SubState</td><td>Definiert, ob das Modul korrekt angelaufen ist oder nicht. Mögliche Werte sind:
• "PNIO_SUBMOD_STATE_RUN"
• "PNIO_SUBMOD_STATE_STOP"</td></tr>
<tr><td>PNIO_BOOL</td><td>MoreFollows</td><td>• "PNIO_TRUE": Es folgen noch "PNIO_async_appl_rdy()"-Aufrufe für weitere Submodule dieser AR. Die Requests werden im PNIO-Stack lediglich zwischengespeichert.
• "PNIO_FALSE": Es folgen keine weiteren "PNIO_async_appl_rdy()"-Aufrufe. Alle vorherigen im PNIO-Stack zwischengespeicherten Requests (MoreFollows = PNIO_TRUE) werden bearbeitet und das ApplicationReady-Telegramm für den PNIO-Controller erzeugt und versendet.</td></tr>
<tr><td>Output</td><td>PNIO_UINT32</td><td>return</td><td>Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"</td></tr>
</table>

4.1.1.5 PNIO_device_close

<table>
<tr><th colspan="3">PNIO_device_close()</th><th>Funktionsaufruf: Applikation > IO-Stack, synchron</th></tr>
<tr><td colspan="4">Wird noch nicht unterstützt.</td></tr>
<tr><td>Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td>Output</td><td>PNIO_UINT32</td><td></td><td>Ausführungsstatus: "PNIO_OK"</td></tr>
</table>

4.1.1.6 PNIO_CP_register_cbf

PNIO_CP_register_cbf()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Mit "PNIO_register_cbf()" können Callback-Funktionen 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	PNIO_CP_CBE_TYPE	CbeType	Spezifiziert das Ereignis, bei dem die Callback aufgerufen wird. Derzeit ist "PNIO_CP_CBE_TRANS_END_IND" implementiert.
	PNIO_CP_CBF	pCbf	Startadresse der vom PNIO-Stack aufzurufenden Anwenderfunktion, wenn das o.g. Ereignis eingetreten ist.
Output	PNIO_UINT32		Ausführungsstatus: "PNIO_OK"

4.1.2 Einstellungen von GeräteNamen und IP-Suite

Die Ethernetparameter (IP-Adresse, Subnetz-Maske, Default Router-Adresse) des IO-Device können von einem PROFINET IO-Controller über Ethernet eingestellt werden. Der Stationsname des Device muss 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 Funktion

- "PNIO_setup()"

wieder übergeben.

Für die Übergabe des Stationsnamens ist ein Funktionsrahmen in "iodapi_event.c" für folgende Funktion hinterlegt, welche vom Anwender zu implementieren ist:

- "PNIO_cbf_save_station_name()"

Die Einstellung der Ethernet-Parameter läuft nach dem gleichen Prinzip ab durch Aufruf der Funktion

- "PNIO_cbf_save_ip_addr()"

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 nichtflüchtigen Speicher gelesen und mit der Funktion "PNIO_setup()" wieder an den Stack übergeben.			
Input	PNIO_INT8*	pStationName	Zeiger auf den String, welcher den Stationsnamen beinhaltet. (Der Stationsname muss nicht zwingend null-terminiert sein.)
	PNIO_UINT16	NameLength	Länge des Strings in byte
	PNIO_UINT8	Remanent	<> 0: Daten müssen remanent gespeichert werden == 0: Wert darf nicht remanent gespeichert werden und ein bereits gespeicherter Name muss gelöscht werden
Output	PNIO_UINT32	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 in einem nichtflüchtigen Speicher ablegen, wenn der Parameter "Remanent" ungleich Null ist. Der Wert wird beim nächsten Systemanlauf von der Applikation aus dem nichtflüchtigen Speicher gelesen und mit der Funktion "PNIO_set_eth_par()" wieder an den Stack übergeben.			
Input	PNIO_UINT32	NewIpAddr	Neue IP-Adresse
	PNIO_UINT32	SubnetMask	Neuer Wert für Subnet Mask
	PNIO_UINT32	DefRouterAddr	Neuer Wert für Default Router
	PNIO_UINT8	Remanent	<> 0: Daten müssen remanent gespeichert werden == 0: Wert darf nicht remanent gespeichert werden und bereits gespeicherte Werte müssen gelöscht werden
Output	PNIO_UINT32	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	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	PortNum	Portnummer (1...n)
	PNIO_UINT32	Frequency	vorgegebene Blinkfrequenz in Hertz
Output	PNIO_UINT32	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	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	PortNum	Portnummer (1...n)
Output	PNIO_UINT32	return	must be "PNIO_OK"

4.1.2.5 PNIO_cbf_reset_factory_settings

PNIO_cbf_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	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
Output	PNIO_UINT32	return	must be "PNIO_OK"

4.1.3 Speicherung remanenter Daten (REMA)

Neben dem Gerätenamen und der 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 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-Datenblocks vom PNIO-Stack erfolgt mit der Funktion

- "PNIO_cbf_store_rema_mem()".

Als Aufrufparameter wird dabei ein Pointer auf die Daten sowie die Datenlänge übergeben.

Beim nächsten Geräteanlauf fragt der PNIO-Stack diese Daten von der Applikation ab. Der Stack ruft dazu die Funktion

- "PNIO_cbf_restore_rema_mem()"

Als Aufrufparameter wird dabei die Adresse eines Zeigers (**ppMem) übergeben. Die Applikation trägt dort die tatsächliche Adresse der REMA-Daten ein. Außerdem wird ein Zeiger auf die Datenlänge übergeben, die Applikation trägt dort die tatsächliche Datenlänge ein. Es ist zu beachten, dass die REMA-Daten dabei nicht kopiert werden und daher auch nach Beendigung von "PNIO_cbf_restore_rema_mem()" gültig bleiben müssen (static data).

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	PNIO_UINT32	MemSize	Größe des Datenblocks in byte
	PNIO_UINT8*	pMem	Zeiger auf den Datenblock
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.1.3.2 PNIO_cbf_restore_rema_mem

PNIO_cbf_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.			
Input	PNIO_UINT32*	pMemSize	Zeiger auf Größe des Datenblocks in byte, hier muss die Applikation die tatsächliche Datenlänge eintragen.
	PNIO_UINT8**	ppMem	Adresse des Zeigers auf den Datenblock. Die Applikation muss hier die tatsächliche Adresse der REMA-Daten eintragen. **Die REMA-Daten müssen static sein, d. h. auch nach Beendigung von "PNIO_cbf_restore_rema_mem" gültig bleiben.**
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.1.4 IO-Device Konfiguration

4.1.4.1 PNIO_sub_plug

<table>
<tr><th colspan="3">PNIO_sub_plug()</th><th>Funktionsaufruf: Applikation > IO-Stack, synchron</th></tr>
<tr><td colspan="4">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.
Eine Liste von Modulen kann gesteckt werden durch Setzen des Parameters "MoreFollows" = "PNIO_TRUE". In diesem Fall werden die Module zunächst im PNDV zwischengespeichert und erst beim nächsten Stecken-Aufruf mit "MoreFollows=PNIO_TRUE" an den CM weitergereicht. Die Rückmeldung in "pError" ist daher erst anschließend gültig und kann von der Applikation abgefragt werden.
Hinweis: Im Anlauf müssen zunächst der DAP und die PDEV-Daten "PNIO_sub_plug_list()" gesteckt werden. Erst danach dürfen weitere Module gesteckt werden, entweder als weitere Einträge in der Submodul-Liste "pIoSublist" oder einzeln mittels der Funktion "PNIO_sub_plug()".</td></tr>
<tr><td rowspan="8">Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td>PNIO_UINT32</td><td>Api</td><td>API number (default API is 0)</td></tr>
<tr><td>PNIO_DEV_ADDR*</td><td>pAddr</td><td>Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Slot Nummer 1....N sind zulässig, die maximale Slotnummer wurde in "PNIO_setup()" festgelegt. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.</td></tr>
<tr><td>PNIO_UINT32</td><td>ModIdent</td><td>Module Indentifier (ist in der GSD-Datei hinterlegt)</td></tr>
<tr><td>PNIO_UINT32</td><td>SubIdent</td><td>Submodule Identifier (ist in der GSD-Datei hinterlegt)</td></tr>
<tr><td>PNIO_IM0_SUPP_ENUM</td><td>Im0Support</td><td>Legt fest, ob und in welcher Form IM0 unterstützt wird, siehe enum "PNIO_IM0_SUPP_ENUM" in Datei "pniousrd.h".</td></tr>
<tr><td>IM0_DATA*</td><td>pIm0Dat</td><td>Wenn das Modul IM0 unterstützt (Im0Support <> PNIO_IM0_NOTHING), dann wird hier ein Zeiger auf die IM0-Daten übergeben. Die weitere Behandlung der IM0-Daten erfolgt stack-intern.</td></tr>
<tr><td>PNIO_UINT8</td><td>IopsIniVal</td><td>Für Submodule ohne IO-Daten (z. B. PDEV-Submodule) wird hier der initiale IOPS-Wert für Input-Module übergeben (gemäß PNIO-Norm verhält sich ein Submodul ohne Daten wie ein Input-Modul mit Datenlänge 0).</td></tr>
<tr><td>Output</td><td>PNIO_UINT32</td><td>return</td><td>Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"</td></tr>
</table>

4.1.4.2 PNIO_sub_plug_list

PNIO_sub_plug_list()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Stecken einer Liste von Submodulen in Subslots. 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. An den IO-Controller werden in diesem Fall vom IO-Stack automatisch Plug-Alarme gesendet. **Hinweis**: Im Anlauf **müssen** zunächst der DAP und die PDEV-Daten mit der Funktion "**PNIO_sub_plug_list()**" gesteckt werden. Erst danach dürfen weitere Module gesteckt werden (entweder als weitere Einträge in der Submodul-Liste "pIoSublist" oder einzeln mittels der Funktion "PNIO_sub_plug()". Entweder der DAP oder das Interface (Subslot 0x8000) muss als IM0-Stellvertreter für das Device stehen, d. h. er muss eigene IM0-Daten besitzen und es muss in der Submodulliste das Element "Im0Support" = (PNIO_IM0_SUBMODULE + PNIO_IM0_DEVICE) gesetzt sein. Ein Anwenderbeispiel dazu finden Sie in Datei "usriod_main.c", Struktur "IoSubList[]".			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_SUB_LIST_ENTRY*	pIoSubList	Liste von Submodulen vom Typ "PNIO_SUB_LIST_ENTRY". Diese enthält u. a. Steckplatznummer, Modul- und Submodul-IDs, IM0-Support ja/nein.
	PNIO_UINT32	NumOfSublistEntries	Anzahl der Einträge in der Liste von Submodulen
	PNIO_IM0_LIST_ENTRY*	pIm0List	Liste von IM0-Daten für diejenigen Submodule, die eigene IM0-Daten besitzen
	PNIO_UINT32	NumOfIm0ListEntries	Anzahl der Einträge in der Liste von IM0-Daten
	PNIO_UINT32*	pStatusList	Liste von Statusrückmeldungen ("PNIO_OK", "PNIO_NOT_OK") für jedes Submodul in "pIoSubList". Die Statusliste hat somit die gleiche Anzahl Einträge wie die Submodulliste.
Output	PNIO_UINT32	return	Sammelstatus: "PNIO_OK", wenn alle Einzelrückmeldungen in "pStatusList" ebenfalls "PNIO_OK" enthalten, andernfalls wird "PNIO_NOT_OK" zurückgemeldet.

4.1.4.3 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	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	API number (default API is 0)
	PNIO_DEV_ADDR*	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Slot Nummer 1....N sind zulässig, die maximale Slotnummer wurde in "PNIO_setup()" festgelegt. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
Output	PNIO_UINT32	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 Diagnose-Record in einen Subslot. Ist das Device Owner in einer AR für diesen Subslot, so wird automatisch ein "Diagnosealarm-kommend" an den IO-Controller geschickt. Der Diagnose-Record kann nach Beseitigung des Problems mit der Funktion "PNIO_diag_channel_remove()" wieder entfernt werden .			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	API number (default API is 0)
	PNIO_DEV_ADDR*	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT16	ChannelNum	Kanalnummer, Inhalt und Aufbau siehe auch "ChannelNumber" in /1/
	PNIO_UINT16	ErrorNum	Fehlernummer, siehe dazu auch /1/
	DIAG_CHANPROP_DIR	ChainDir	Datenrichtung (IN/OUT/INOUT), siehe enum "DIAG_CHANPROP_DIR" in Datei "pniousrd.h"
	DIAG_CHANPROP_TYPE	ChanType	Datentyp (1bit, 2bit, ... BYTE, WORD, DWORD), siehe enum "DIAG_CHANPROP_TYPE" in Datei "pniousrd.h"
	PNIO_BOOL	MaintenanceReq	PNIO_TRUE: Maintenance required, else PNIO_FALSE
	PNIO_BOOL	MaintenanceDem	PNIO_TRUE: Maintenance demanded, else PNIO_FALSE
Output	PNIO_UINT32	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 Diagnose-Records. Zur Referenzierung müssen die gleichen Werte angegeben werden wie beim zugehörigen "PNIO_diag_channel_add"-Aufruf.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	API number (default API is 0)
	PNIO_DEV_ADDR*	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT16	ChannelNum	Kanalnummer, Inhalt und Aufbau siehe auch "ChannelNumber" in /1/
	PNIO_UINT16	ErrorNum	Fehlernummer, siehe dazu auch /1/
	DIAG_CHANPROP_DIR	ChanDir	Datenrichtung (IN/OUT/INOUT), siehe enum "DIAG_CHANPROP_DIR" in Datei "pniousrd.h"
	DIAG_CHANPROP_TYPE	ChanTyp	Datentyp (1bit, 2bit, ... BYTE, WORD, DWORD), siehe enum "DIAG_CHANPROP_TYPE" in Datei "pniousrd.h"
Output	PNIO_UINT32	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 Diagnose-Records in einen Subslot. Ist das Device Owner in einer AR für diesen Subslot, so wird automatisch ein erweiterter "Diagnosealarm-kommend" an den IO-Controller geschickt. Der Diagnose-Record kann nach Beseitigung des Problems mit der Funktion "PNIO_ext_diag_channel_remove()" wieder entfernt werden.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	API number (default API is 0)
	PNIO_DEV_ADDR*	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT16	ChanNum	Kanalnummer, Inhalt und Aufbau siehe auch "ChannelNumber" in /1/
	PNIO_UINT16	ErrorNum	Fehlernummer, siehe dazu auch /1/
	DIAG_CHANPROP_DIR	ChanDir	Datenrichtung (IN/OUT/INOUT), siehe enum "DIAG_CHANPROP_DIR" in Datei "pniousrd.h"
	DIAG_CHANPROP_TYPE	ChanTyp	Datentyp (1bit, 2bit, ... BYTE, WORD, DWORD), siehe enum "DIAG_CHANPROP_TYPE" in Datei "pniousrd.h"
	PNIO_UINT16	ExtChannelErrType	Extended Channel Error Type, Inhalt und Aufbau siehe "ExtChannelErrorType" in /1/
	PNIO_UINT32	ExtChannelAddValue	Additional Value, Inhalt und Aufbau siehe "ExtChannelAddValue" in /1/
	PNIO_BOOL	MaintenanceReg	PNIO_TRUE: Maintenance required, else PNIO_FALSE
	PNIO_BOOL	MaintenanceDem	PNIO_TRUE: Maintenance demanded, else PNIO_FALSE
Output	PNIO_UINT32	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 Diagnose-Records.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	API number (default API is 0)
	PNIO_DEV_ADDR*	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT16	ChannelNum	Kanalnummer, Inhalt und Aufbau siehe auch „ChannelNumber“ in /1/
	PNIO_UINT16	ErrorNum	Fehlernummer, siehe dazu auch /1/
	DIAG_CHANPROP_DIR	ChanDir	Datenrichtung (IN/OUT/INOUT), siehe enum "DIAG_CHANPROP_DIR" in Datei "pniousrd.h"
	DIAG_CHANPROP_TYPE	ChanTyp	Datentyp (1bit, 2bit, ... BYTE, WORD, DWORD), siehe enum "DIAG_CHANPROP_TYPE" in Datei "pniousrd.h"
	PNIO_UINT16	ExtChannelErrType	Extended Channel Error Type, Inhalt und Aufbau siehe „ExtChannelErrorType“ in /1/
	PNIO_UINT32	ExtChannelAddValue	Additional Value, Inhalt und Aufbau siehe „ExtChannelAddValue“ in /1/
Output	PNIO_UINT32	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 Diagnose-Records in einen Subslot. Die Diagnosedaten können über einen speziellen Record-Aufruf (siehe /1/) z. B. von einem Diagnosetool ausgelesen werden. Auf einen Subslot können mehrere Diagnose-Records geladen werden, die über einen vom Anwender vorgebbaren Tag referenziert werden. Über diese Referenz kann der Diagnose-Record mit der Funktion "PNIO_diag_generic_remove()" wieder entfernt werden.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	API number (default API is 0)
	PNIO_DEV_ADDR*	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT16	ChannelNum	Kanalnummer
	DIAG_CHANPROP_DIR	ChanDir	Datenrichtung (IN/OUT/INOUT), siehe enum "DIAG_CHANPROP_DIR" in Datei "pniousrd.h"
	DIAG_CHANPROP_TYPE	ChanTyp	Datentyp (1bit, 2bit, ... BYTE, WORD, DWORD), siehe enum "DIAG_CHANPROP_TYPE" in Datei "pniousrd.h"
	PNIO_UINT16	DiagTag	vom Anwender vorgegebene eindeutige Referenz auf den Record
	PNIO_UINT16	UserStructIdent	UserStructureIdentifier, s. /1/: 0..7fff: Herstellerspezifische Daten in "pInfoData"
	PNIO_UINT8*	pInfoData	Zeiger auf die Diagnosedaten
	PNIO_UINT32	InfoDataLen	Länge der Diagnosedaten in bytes
	PNIO_BOOL	MaintenanceReq	PNIO_TRUE: Maintenance required, else PNIO_FALSE
	PNIO_BOOL	MaintenanceDem	PNIO_TRUE: Maintenance demanded, else PNIO_FALSE
Output	PNIO_UINT32	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_diag_generic_add()" heruntergeladenen Diagnose-Records.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	API number (default API is 0)
	PNIO_DEV_ADDR*	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT16	ChanNum	Kanalnummer
	DIAG_CHANPROP_DIR	ChanDir	Datenrichtung (IN/OUT/INOUT), siehe enum "DIAG_CHANPROP_DIR" in Datei "pniousrd.h"
	DIAG_CHANPROP_TYPE	ChanTyp	Datentyp (1bit, 2bit, ... BYTE, WORD, DWORD), siehe enum "DIAG_CHANPROP_TYPE" in Datei "pniousrd.h"
	PNIO_UINT16	DiagTag	vom Anwender vorgegebene eindeutige Referenz auf den Record
	PNIO_UINT16	UserStructIdent	UserStructureIdentifier, s. /1/: 0..7fff: Herstellerspezifische Daten in "pInfoData"
Output	PNIO_UINT32	return	Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"

4.1.6 Senden und Empfangen von Alarmen

Über das IOD-API können folgende subslot-spezifische Alarme ausgelöst werden:

- Prozessalarme (optional)
- Diagnosealarme (optional)
- "Return of Submodule"-Alarme (obligatorisch)

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 Callback-Funktion "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 IO-Controller enthalten, werden aber nicht lokal im Submodul gespeichert. Eine Remove-Funktion ist deshalb nicht notwendig.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	API number (default API is 0)
	PNIO_DEV_ADDR*	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT8*	pData	Zeiger auf Alarmdaten
	PNIO_UINT32	DataLen	Länge der Alarmdaten in byte
	PNIO_UINT16	UserStructIdent	UserStructureIdentifier, s. /1/: 0..7fff: Herstellerspezifische Daten in "pData"
	PNIO_UINT32	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	PNIO_UINT32	return	Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"

4.1.6.2 PNIO_upload_retrieval_alarm_send

<table>
<tr><th colspan="3">PNIO_upload_retrieval_alarm_send ()</th><th>Funktionsaufruf: Applikation > IO-Stack, asynchron</th></tr>
<tr><td colspan="4">Mit einem Upload-/Retrieval-Alarm wird ein Event an einen zentralen Parameterserver (iPar-Server) versendet. Dieses Event veranlasst den iPar-Server, einen Parametersatz vom Device mittels eines Record Read-Services auszulesen oder einen Parametersatz mittels eines Record Write-Services zum Device zu übertragen. Welcher Record-Index dabei verwendet wird und ob die Daten geschrieben oder gelesen werden, ist in einem Header, der Teil der Alarmdaten ist, hinterlegt.
Weitere Informationen zur Funktion und Behandlung des Upload-/Retrieval-Alarms finden Sie im Dokument "45841087_FB24iParServ_DOKU_V10_de.pdf" im Unterverzeichnis "\tools\TCI\iPar-Server" auf der Produkt-CD.</td></tr>
<tr><td>Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td></td><td>PNIO_UINT32</td><td>Api</td><td>API number (default API is 0)</td></tr>
<tr><td></td><td>PNIO_DEV_ADDR*</td><td>*pAddr</td><td>Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.</td></tr>
<tr><td></td><td>PNIO_UINT8*</td><td>*pData</td><td>Zeiger auf Alarmdaten.
Die Daten beginnen mit einem 24 byte langen Header. Dieser legt die Übertragungsrichtung, die Datenlänge sowie den verwendeten Record-Index fest. Der Aufbau des Headers ist in der PNIO-Spezifikation definiert. Ein Anwenderbeispiel dazu ist im Applikationscode enthalten, siehe Datei "usriod_main.c", Datenstrukturen "UploadAlarmDate[]" sowie "RetrievalAlarmData[]".</td></tr>
<tr><td></td><td>PNIO_UINT32</td><td>DataLen</td><td>Länge der Alarmdaten in byte.</td></tr>
<tr><td></td><td>PNIO_UINT32</td><td>UserHndl</td><td>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.</td></tr>
<tr><td>Output</td><td>PNIO_UINT32</td><td>return</td><td>Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"</td></tr>
</table>

4.1.6.3 PNIO_ret_of_sub_alarm_send

<table>
<tr><th colspan="3">PNIO_ret_of_sub_alarm_send()</th><th>Funktionsaufruf: Applikation > IO-Stack, asynchron</th></tr>
<tr><td colspan="4">Senden eines "Return of Submodul"-Alarms an den IO-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 IO-Controller parametriert.
Hinweis: Wenn sich der IOPS/IOCS von "GOOD" nach "BAD" ändert, muss hingegen kein Alarm ausgelöst werden.</td></tr>
<tr><td>Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td></td><td>PNIO_UINT32</td><td>Api</td><td>API number (default API is 0)</td></tr>
<tr><td></td><td>PNIO_DEV_ADDR*</td><td>pAddr</td><td>Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.</td></tr>
<tr><td></td><td>PNIO_UINT32</td><td>UserHndl</td><td>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.</td></tr>
<tr><td>Output</td><td>PNIO_UINT32</td><td>return</td><td>Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"</td></tr>
</table>

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 IO-Controller empfangen wurde (reserviert, derzeit werden noch keine Alarme vom Controller an das Device gesendet, die zur Applikation hochgereicht werden).			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_DEV_ALARM_DATA*	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	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	ArNum	AR number (1…N)
	PNIO_ALARM_TYPE	AlarmType	Alarmtyp (PNIO_ALM_CHAN_DIAG, PNIO_ALM_EXT_CHAN_DIAG, ...), siehe enum "PNIO_ALARM_TYPE" in Datei "pniousrd.h"
	PNIO_UINT32	Api	API number (Application Process Identifier)
	PNIO_DEV_ADDR*	pAddr	Slot/Subslot
	PNIO_UINT32	Status	"PNIO_OK" "PNIO_NOT_OK"
Output	-	-	

4.1.8 Lesen und Schreiben von Records

Neben der synchronen Bearbeitung (d. h. komplette Bearbeitung des Read/Write Record-Requests innerhalb der Callback-Funktion "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 Callback-Funktion 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:

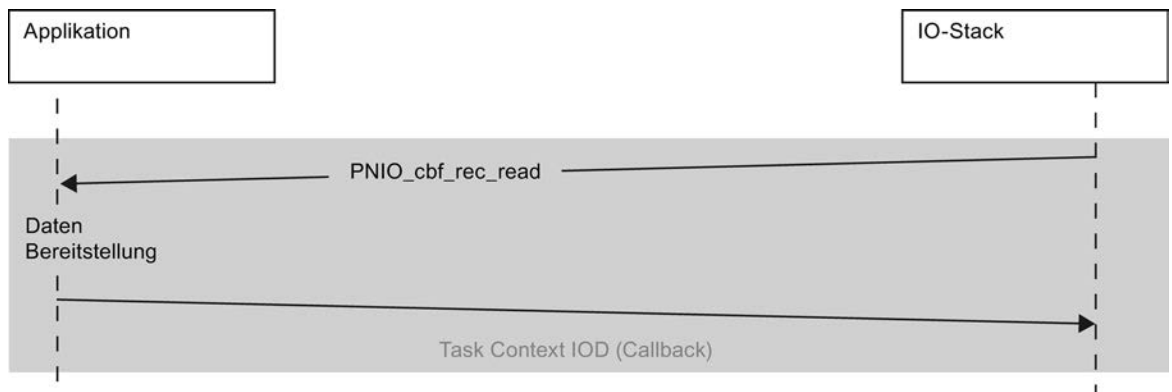


Bild 4-1 Synchrone Read Record-Bearbeitung

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

Das Szenario für einen **asynchronen Read Record-Request** sieht folgendermaßen aus:

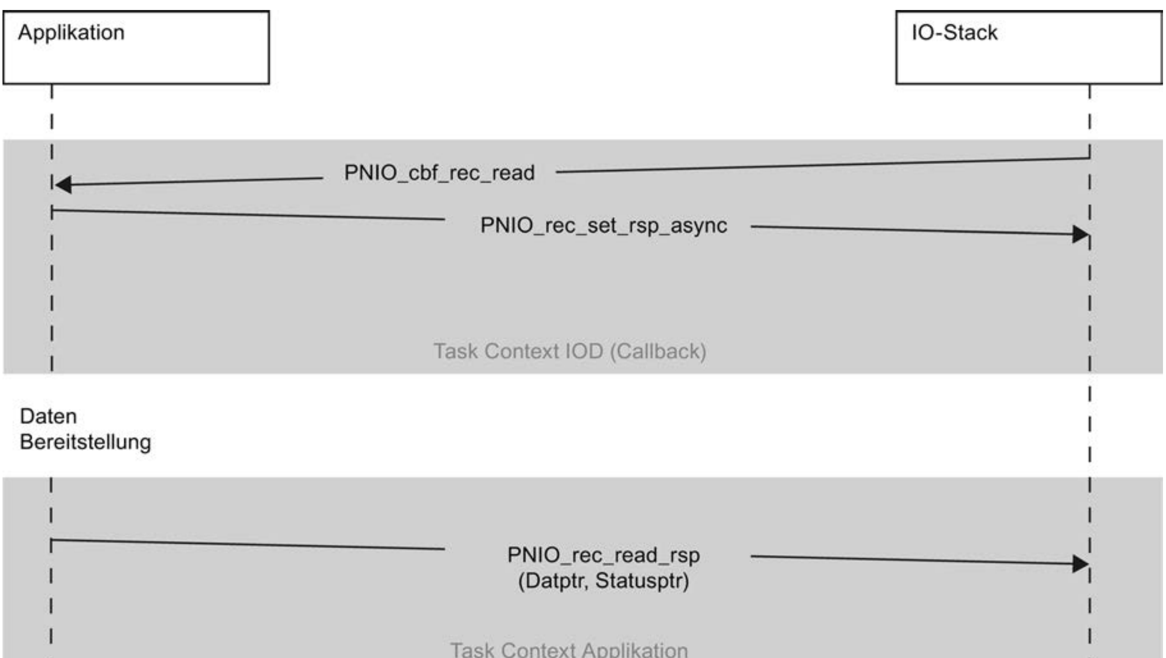


Bild 4-2 Asynchrone Read Record-Bearbeitung

- Ein Read Record-Request vom IO-Controller wird empfangen und vom IO-Stack ausgewertet.
- Der 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 Callback-Funktion abgeschlossen werden muss (aber kann). Als Returnwert wird der Applikation ein Request Handle übergeben, um bei der späteren Übergabe der Daten an den Stack diesen Request referenzieren zu können. Die vom IO-Stack vorgegebenen Adressen für Record-Daten und Fehlerzustand werden anschließend von der Applikation nicht weiter verwendet.
- Die Applikation stellt die geforderten Record-Daten, die Datenlänge und Fehlerstatus innerhalb eines beliebigen Taskkontextes an einer beliebigen Speicheradresse bereit.
- Die Applikation ruft "PNIO_rec_read_rsp()" auf und übergibt dabei Record-Daten, 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

<table>
<tr><th colspan="3">PNIO_cbf_rec_read()</th><th>Funktionsaufruf: IO-Stack > Applikation, synchron</th></tr>
<tr><td colspan="4">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 Byte-Anzahl wird von der Applikation ebenfalls in "*pBufLen" zurückgemeldet.
"PNIO_cbf_rec_read()" wird vom IO-Stack nur für gesteckte Submodule aufgerufen.</td></tr>
<tr><td rowspan="10">Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td>PNIO_UINT32</td><td>Api</td><td>Application Process Identifier, spezifiziert ein Profil</td></tr>
<tr><td>PNIO_UINT16</td><td>ArNum</td><td>AR-Nummer</td></tr>
<tr><td>PNIO_UINT16</td><td>SessionKey</td><td>Session Key</td></tr>
<tr><td>PNIO_UINT32</td><td>SequenceNum</td><td>Fortlaufende Nummerierung (Lücken in der Nummerierung sind möglich).
Kann auf Applikationsebene als Referenz mehrerer quasi gleichzeitig auftretender Requests verwendet werden.</td></tr>
<tr><td>PNIO_DEV_ADDR *</td><td>pAddr</td><td>Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.</td></tr>
<tr><td>PNIO_UINT32</td><td>RecordIndex</td><td>Record-Index der zu lesenden Daten</td></tr>
<tr><td>PNIO_UINT32*</td><td>pBufLen</td><td>(Nur für synchronen Betrieb, don’t care bei asynchronem Betrieb)
Zeiger auf die Anzahl der Record-Daten in bytes. Der Stack übergibt hier die maximal zulässige Record-Datenlänge, die Applikation gibt die tatsächlich geschriebene Anzahl zurück. Die vom Stack vorgegebene maximale Datenlänge darf dabei keinesfalls überschritten werden.</td></tr>
<tr><td>PNIO_UINT8*</td><td>pBuffer</td><td>(Nur für synchronen Betrieb, don’t care bei asynchronem Betrieb)
Zeiger, wohin die Applikation die gelesenen Record-Daten kopieren muss.</td></tr>
<tr><td>PNIO_ERR_STAT *</td><td>pPnioState</td><td>(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/</td></tr>
<tr><td>Output</td><td>PNIO_UINT32</td><td>return</td><td>Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"</td></tr>
</table>

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 Record-Daten ab der mit "pBuf" vorgegebene Adresse. Die Datenlänge in bytes wird vom Stack in "*pBufLen" übergeben, die tatsächlich übernommene Byte-Anzahl wird von der Applikation ebenfalls in "*pBufLen" zurückgemeldet. "PNIO_cbf_rec_write()" wird vom IO-Stack nur für gesteckte Submodule aufgerufen.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	Api	Application Process Identifier, spezifiziert ein Profil
	PNIO_UINT16	ArNum	AR-Nummer
	PNIO_UINT16	SessionKey	Session Key
	PNIO_UINT32	SequenceNum	Fortlaufende Nummerierung (Lücken in der Nummerierung sind möglich). Kann auf Applikationsebene als Referenz mehrerer quasi gleichzeitig auftretender Requests verwendet werden.
	PNIO_DEV_ADDR *	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT32	RecordIndex	Record-Index der zu lesenden Daten
	PNIO_UINT32*	pBufLen	Zeiger auf die Anzahl der Record-Daten in bytes. Der Stack übergibt hier die maximal zulässige Record-Datenlänge. Im synchronen Betrieb gibt die Applikation die tatsächlich geschriebene Anzahl zurück. **Die vom Stack vorgegebene maximale Datenlänge darf dabei keinesfalls überschritten werden.**
	PNIO_UINT8*	pBuffer	Zeiger auf die Record-Daten.
	PNIO_ERR_STAT *	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	PNIO_UINT32	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 Callback-Funktion "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 Development Kit.			
Input	-	-	
Output	PNIO_VOID*	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 Read Record-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 Anwender-Task heraus aufgerufen werden.			
Input	PNIO_VOID*	pRqHnd	Handle für die Referenzierung des Requests, wurde beim vorherigen Aufruf von "PNIO_rec_set_rsp_async()" als Returnwert an die Applikation übergeben.
	PNIO_UINT8*	pDat	Zeiger auf die von der Applikation bereitgestellten Daten. Der Speicher für die Record-Daten wird im asynchronen Betrieb von der Applikation und nicht, wie beim synchronen Betrieb, vom IO-Stack vorgegeben.
	PNIO_UINT32	NettoDatLength	Länge der tatsächlich vom Submodul gelesenen Record-Daten
	PNIO_ERR_STAT *	pPnioState	Zeiger 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	PNIO_UINT32	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 Write Record-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 Anwender-Task heraus aufgerufen werden.			
Input	PNIO_VOID*	pRqHnd	Handle für die Referenzierung des Requests, wurde beim vorherigen Aufruf von "PNIO_rec_set_rsp_async()" als Returnwert an die Applikation übergeben.
	PNIO_UINT32	NettoDatLength	Länge der tatsächlich ins Submodul geschriebenen Record-Daten
	PNIO_ERR_STAT *	pPnioState	Zeiger 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	PNIO_UINT32	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

<table>
<tr><th colspan="3">PNIO_initiate_data_read()</th><th>Funktionsaufruf: Applikation > IO-Stack, synchron</th></tr>
<tr><td colspan="4">Mit dieser Funktion wird einmalig der Austausch von IO-Outputdaten (Datenübertragungsrichtung: IO-Controller > Device) zwischen IO-Stack und Applikation für alle ARs (RT, IRT) durchgeführt. Der PN-Stack reserviert 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 der PN-Stack. Der Anwender 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 gewährleistet (max. 254 byte konsistent).</td></tr>
<tr><td rowspan="2">Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td>PNIO_UINT32*</td><td>pApduStatus</td><td>Zeiger auf 4 byte langen APDU-Status (siehe auch PNIO-Spezifikation /1b/). Dieser enthält:
• Byte 0, 1: Cycle Counter (Big Endian Format)
• Byte 2: APDU-Statusbyte, siehe define PNIO_APDU_STATUSBYTE_MASK
• Byte 3: Transfer-Status</td></tr>
<tr><td>Output</td><td>PNIO_UINT32</td><td>return</td><td>Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"</td></tr>
</table>

<table>
<tr><td colspan="3">PNIO_initiate_data_write()</td><td>Funktionsaufruf: Applikation > IO-Stack, synchron</td></tr>
<tr><td colspan="4">Mit dieser Funktion wird einmalig der Austausch von IO-Inputdaten (Datenübertragungsrichtung: Device > IO-Controller) zwischen IO-Stack und Applikation für alle ARs (RT, IRT) durchgeführt. Der PN-Stack reserviert zunächst 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 der PN-Stack. Der Anwender 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 gewährleistet (max. 254 byte konsistent).</td></tr>
<tr><td>Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td>Output</td><td>PNIO_UINT32</td><td>return</td><td>Ausführungsstatus: "PNIO_OK", "PNIO_NOT_OK"</td></tr>
</table>

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 im Slot/Subslot gesteckten Submoduls lesen und auf die vom Stack vorgegebene Pufferadresse kopieren. **Achtung: Die mit "DataLen" vorgegebene Länge darf dabei in keinem Fall überschritten werden. Dies liegt in der Verantwortung des Anwenders.**			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_DEV_ADDR *	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT32	BufLen	Länge der Daten in byte.
	PNIO_UINT8*	pBuffer	Zeiger auf Datenpuffer, dorthin muss die Applikation die IO-Inputdaten kopieren. **Die in "DataLen" vorgegebene Datenlänge darf dabei keinesfalls überschritten werden.**
	PNIO_IOXS	Iocs	Remote Consumer-Status, 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	PNIO_IOXS	Iops	IO-Provider-Status. 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 Consumer-Status an den Stack zurückgemeldet. Dieser wird im nächsten zyklischen RT-Telegramm an den IO-Controller übertragen und kann dort ausgewertet werden.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_DEV_ADDR *	pAddr	Zeiger auf die Steckaddresse (Slot/Subslot) des Submoduls. Als Typ muss "PNIO_ADDR_GEO" eingetragen werden.
	PNIO_UINT32	BufLen	Länge der Daten in byte.
	PNIO_UINT8*	pBuffer	Zeiger auf Datenpuffer, von dort muss die Applikation die IO-Outputdaten lesen.
	PNIO_IOXS	Iops	Remote Provider-Status, 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	PNIO_IOXS	Iocs	IO Consumer-Status. 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

4.1.10.1 Zyklischer Datenaustausch über das optionale DBA-Interface

Alternativ zum Standard-Callbackinterface (SCI) können IO-Daten sowie Provider/Consumer-Status 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 IO-Daten, IOPS und IOCS für die einzelnen Submodule aus der IOCR. Eine IOCR ist entweder eine Input-CR (Device > IO-Controller) oder Output-CR (IO-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_buf_lock()	Anfordern eines IOCR-Puffers (entweder Input-IOCR oder Output-IOCR)
PNIO_dbai_buf_unlock()	Freigeben eines vorher belegten IOCR-Puffers nach der Bearbeitung
PNIO_dbai_exit()	Freigabe der Stack-internen, mit "PNIO_dbai_enter()" belegten Semaphore

Für den Aufruf von "PNIO_dbai_buf_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_buf_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 > IO-Controller) benötigt:

- Slotnummer, Subslotnummer der projektierten Input- und Output-Submodule
- Lage (Offset, Länge) der IO-Daten der projektierten Input-Submodule
- Lage (Offset) der (local) 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 IO-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_buf_lock()
       Puffer bearbeiten
       PNIO_dbai_buf_unlock()
PNIO_dbai_exit()
```

4.1.10.2 PNIO_dbai_enter

PNIO_dbai_enter()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Diese Funktion sollte vor "PNIO_dbai_buf_lock()" aufgerufen werden, um sowohl den Aufruf von Callback-Funktionen "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 Lock-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	-	-	-

4.1.10.3 PNIO_dbai_exit

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

4.1.10.4 PNIO_dbai_buf_lock

PNIO_dbai_buf_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 einer eingestellten Reduction Ratio > 1 zur Ermittlung der Phase dienen, in der die IO-Daten tatsächlich übertragen werden.			
Input	PNIO_BUFFER_LOCK_TYPE*	pLock	Zeiger auf Struktur, diese enthält: • AR Handle • Session Key • Übertragungsrichtung ("PNIO_IOCR_TYPE_ENUM" IocrType) • Offset der zu sperrenden IO-Daten in IOCR (nur EB 200P bei Länge > 255 byte, sonst 0) • Länge der zu sperrenden Daten (EB 200P: max. 254 byte auf einmal) • [out] pBuf-Zeiger auf den IOCR-Puffer • [out] APDU-Status, enthält den Cycle Counter (bit 0…15)
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.1.10.5 PNIO_dbai_buf_unlock

PNIO_dbai_buf_unlock()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Damit wird nach der Bearbeitung durch die Applikation ein mit "PNIO_dbai_buf_lock()" angeforderter Puffer wieder an den Stack zurückgegeben.			
Input	PNIO_BUFFER_LOCK_TYPE*	pLock	Zeiger auf dieselben Daten von "PNIO_dbai_buf_lock"
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.1.11 Events und Alarme empfangen

4.1.11.1 PNIO_cbf_ar_connect_ind

PNIO_cbf_ar_connect_ind()			**Funktionsaufruf: IO-Stack > Applikation, synchron**
Die Funktion signalisiert der Applikation, dass eine neue Verbindung zu einem IO-Controller aufgebaut wurde.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	HostIp	IP-Adresse der Remote Station (in der Regel der IO-Controller)
	PNIO_UINT32	ArType	Siehe Struktur "cm_ar_type_enum", möglich Werte sind: • "CM_AR_TYPE_SINGLE" • "CM_AR_TYPE_SINGLE_RTC3" • "CM_AR_TYPE_SINGLE_SYSRED" (derzeit nicht implementiert) • "CM_AR_TYPE_SUPERVISOR" (derzeit nicht implementiert)
	PNIO_UINT32	ArNum	AR-Nummer (1...N)
Output	-	-	-

4.1.11.2 PNIO_cbf_ar_ownership_ind

<table>
<tr><th colspan="3">PNIO_cbf_ar_ownership_ind()</th><th>Funktionsaufruf: IO-Stack > Applikation, synchron</th></tr>
<tr><td colspan="4">Mit dieser Funktion wird der Applikation der Soll-Ausbau der Module/Submodule aus der Projektierung mitgeteilt. Dazu wird eine Datenstruktur vom Typ "PNIO_EXP" übergeben. Diese enthält eine Liste aller konfigurierten Submodule. Für jedes Submodul ist darin jeweils enthalten: der Steckplatz (API/Slot/Subslot), die Modul- und Submodul-ID, die IO-Eigenschaften (in, out, in-out) sowie die Offsets für die Input- bzw. Outputdaten des Submoduls, und deren Provider- und Consumerstates.
Übernahme der Ownership
Die Submodule der Sollkonfiguration müssen vor dem Datenaustausch der AR zugeordnet werden, andernfalls können über die AR keine gültigen Daten mit diesem Submodul ausgetauscht werden.
Wenn die Device-Applikation die AR-Ownership für ein Submodul NICHT übernehmen möchte (d. h. keine gültigen IO-Daten für dieses Submodul austauschen möchte), so muss sie das Element OwnSessionKey = 0 setzen. Andernfalls (also wenn die Applikation gültige Daten für dieses Submodul austauschen möchte) muss OwnSessionKey unverändert bleiben.
Jeder Subslot kann maximal nur einer AR zugeordnet sein, d. h. bei mehreren ARs kann nur eine AR die Ownership für einen Subslot übernehmen. Für weitere ARs wird dieses Submodul dann als „superordinated locked“ gekennzeichnet.
Prüfung der Soll/Ist-Konfiguration durch die Applikation
Desweiteren muss die Applikation prüfen, ob die Sollkonfiguration mit der Ist-Konfiguration übereinstimmt. Ist in einem Slot/Subslot ein falsches Submodul gesteckt (d. h. ein inkompatibles Submodul, welches daher nicht in den Datenaustausch gehen kann), so muss in der Struktur "PNIO_EXP_SUB" das Element "IsWrongSubmod" = "PNIO_TRUE" gesetzt werden. Wenn hingegen das korrekte Submodul gesteckt wurde oder ein kompatibles Ersatzmodul, so bleibt der Wert von "IsWrongSubmod" unverändert (die Default-Voreinstellung ist bereits "PNIO_FALSE").</td></tr>
<tr><td rowspan="3">Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td>PNIO_UINT32</td><td>ArNum</td><td>AR-Nummer (1...N)</td></tr>
<tr><td>PNIO_EXP*</td><td>pOwnSub</td><td>Pointer auf Datenstruktur, diese enthält eine Liste aller konfigurierten Submodule und deren Eigenschaften (Steckplatz, Datentyp, Modul/Submodul-ID, Offsets der Daten und IoxS in der IOCR)</td></tr>
<tr><td rowspan="2">Output</td><td>PNIO_VOID</td><td>pOwnSub ->
Sub[i].
OwnSessionKey</td><td>Session Key unverändert: Ownership für Submodul wird übernommen
SessionKey = 0: Ownership wird abgelehnt</td></tr>
<tr><td>PNIO_VOID</td><td>pOwnSub ->
Sub[i].
IsWrongSubmod</td><td>= PNIO_TRUE, wenn ein inkompatibles Submodul gesteckt wurde und damit kein Datenaustausch möglich ist
Unverändert (PNIO_FALSE) lassen, wenn das korrekte (Modul/Submodul-ID identisch) oder ein kompatibles Ersatzmodul gesteckt wurde, so dass Datenaustausch möglich ist.</td></tr>
</table>

4.1.11.3 PNIO_cbf_ar_indata_ind

<table>
<tr><th colspan="3">PNIO_cbf_ar_indata_ind()</th><th>Funktionsaufruf: IO-Stack > Applikation, synchron</th></tr>
<tr><td colspan="4">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.</td></tr>
<tr><td rowspan="3">Input</td><td>PNIO_UINT32</td><td>DevHndl</td><td>Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.</td></tr>
<tr><td>PNIO_UINT16</td><td>ArNum</td><td>Spezifiziert die AR-Nummer, diese wurde mit "PNIO_cbf_ar_info_ind()" als Element der Struktur "PNIO_AR_TYPE" übergeben.</td></tr>
<tr><td>PNIO_UINT16</td><td>SessionKey</td><td>Session Key</td></tr>
<tr><td>Output</td><td>-</td><td>-</td><td>-</td></tr>
</table>

4.1.11.4 PNIO_cbf_ar_disconn_ind

PNIO_cbf_ar_disconn_ind()			**Funktionsaufruf: IO-Stack > Applikation, synchron**
Der Stack teilt der Applikation mit, dass ein Disconnect-Event aufgetreten ist. Ein Disconnect-Event tritt auf, wenn eine bestehende Verbindung unterbrochen wurde oder explizit vom IO-Controller oder Device abgebaut wurde.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT16	ArNum	Spezifiziert die AR-Nummer, diese wurde mit "PNIO_cbf_ar_info_ind()" als Element der Struktur "PNIO_AR_TYPE" übergeben.
	PNIO_UINT16	SessionKey	Session Key
	PNIO_AR_REASON	ReasonCode	Beschreibt den Grund für den Verbindungsabbruch, mögliche Reason-Codes für einen Abbruch sind in Datei "iodapi.h" in der enum-Typdefinition "PNIO_AR_REASON{..}" hinterlegt.
Output	-	-	-

4.1.11.5 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 ApplicationReady-Meldung an den IO-Controller übertragen. In diesem Fall darf "PNIO_async_appl_rdy()" **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_async_appl_rdy()" aufrufen. **"PNIO_async_appl_rdy()" darf und muss nur für diejenigen Submodule aufgerufen werden, bei denen die Funktion "PNIO_cbf_param_end_ind()" mit dem Returnwert "PNIO_SUBMOD_STATE_APPL_RDY_FOLLOWS" abgeschlossen wurde.**			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT16	ArNum	Spezifiziert die AR-Nummer, diese wurde mit "PNIO_cbf_ar_info_ind()" als Element der Struktur "PNIO_AR_TYPE" übergeben.
	PNIO_UINT16	SessionKey	Session Key
	PNIO_UINT32	Api	API-Nummer (nur gültig, wenn SubslotNum ungleich 0)
	PNIO_UINT16	SlotNum	Slot-Nummer (nur gültig, wenn SubslotNum ungleich 0)
	PNIO_UINT16	SubslotNum	• == 0: ParamEnd für alle Module, • <> 0: ParamEnd nur für das spezifizierte Modul
	PNIO_BOOL	MoreFollows	• "PNIO_TRUE": Weitere "PNIO_cbf_param_end_ind()"-Aufrufe für weitere Subslots folgen • "PNIO_FALSE": Dies ist der letzte aller "PNIO_cbf_param_end_ind()"-Aufrufe

PNIO_cbf_param_end_ind()			**Funktionsaufruf: IO-Stack > Applikation, synchron**
Output	PNIO_SUBMOD_ STATE	return	• "PNIO_SUBMOD_STATE_RUN", wenn das Submodul gültige Daten liefern kann • "PNIO_SUBMOD_STATE_STOP", wenn das Submodul keine gültigen Daten liefern kann • "PNIO_SUBMOD_STATE_APPL_RDY_FOLLOWS", wenn der Submodulanlauf noch nicht abgeschlossen ist und diese Info später mittels "PNIO_async_appl_rdy()" an den Stack übergeben wird.

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". Die Funktion ist einmalig im Anlauf aufzurufen, um das Device in den Zustand "OPERATE" zu setzen.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	DevState	Neuer Status des Device. Mögliche Werte sind • "PNIO_DEVSTAT_OPERATE" • "PNIO_DEVSTAT_CLEAR"
Output	PNIO_UINT32	return	• "PNIO_OK": Auftrag wurde ordnungsgemäß ausgeführt • "PNIO_NOT_OK": Fehler bei Auftragsausführung aufgetreten

4.1.13 Hardware-Komparatoren für taktsynchronen Betrieb

4.1.13.1 Hardware-Komparatoren für taktsynchronen Betrieb

Für den taktsynchronen Betrieb (IRT), speziell für die Behandlung taktsynchroner Applikationen, wurden Funktionen implementiert, mit denen zu bestimmten Ereignissen während des IO-Zyklus GPIO-Signale angesteuert werden können oder Callback-Funktionen aufgerufen werden.

Solche Ereignisse können sein:

- eine vorgegebene Verzögerung zum Zyklusanfang
- IO-Datentransfer am Bus beendet (TRANS_END)

Die Konfiguration dafür ist in sogenannten Iso-Objekten hinterlegt. Ein Iso-Objekt beinhaltet die Art des Events (Zeitmarke/TRANS_END), notwendige Parameter wie Delay-Zeit oder Callback-Funktionspointer, sowie die für das Objekt benötigten ERTEC-internen Ressourcen (Timer, Multiplexer, ISRs, GPIOs).

Insgesamt können zur gleichen Zeit sechs verschiedene Zeitmarken definiert werden sowie ein TRANS_END-Event. Abhängig von der Konfiguration der Zeiten, ISRs und GPIOs können durch Mehrfachnutzung ERTEC-interner Ressourcen in Summe auch mehr als sieben Iso-Objekte aktiviert werden. Die absolute Anzahl ist abhängig von der Konfiguration.

Ablauf-Kontext der Anwender-Callback-Funktion

Die Callback-Funktionen der EVMA-Events laufen auf der Interrupt- oder Systemebene des Betriebssystems ab. Diese wird in der Systemanpassung in Datei "ecos_bspadapt_ertec.c" eingestellt. Dort sind für eCos die Funktionen "**Bsp_EVMA_ISR()**" und "**Bsp_EVMA_DSR()**" implementiert, die als eCos-ISR- bzw. DSR-Handler für die entsprechenden EVMA-Events installiert wurden. Bei einem auftretenden EVMA-Interrupt-Event wird zunächst vom Betriebssystem die Funktion "Bsp_EVMA_ISR()" aufgerufen. Falls die ISR mit Return (CYC_ISR_HANDLED | CYG_ISR_CALL_DSR) aufgerufen wird, so wird auch "Bsp_EVMA_DSR()" ausgeführt, andernfalls nicht.

Die Einstellung, ob die Anwender-Callback-Funktion im ISR- oder DSR-Kontext ausgeführt wird, ist dabei einfach konfigurierbar mittels

```
#define USER_CONTEXT_ISR        1        oder
#define USER_CONTEXT_ISR        0
```

Weitere Änderungen an "Bsp_EVMA_DSR" und "Bsp_EVMA_ISR" durch den Anwender sind normalerweise nicht notwendig.

Hinweis

Die Callback-Funktionen sollten aufgrund der hohen Ablaufebene so kurz wie möglich sein. Beachten Sie die zulässigen Randbedingungen des Betriebssystems, wenn Sie Anwendercode im ISR- oder DSR-State ausführen. Im DSR-Zustand sind beispielsweise keine wartenden Betriebssystem-Service-Aufrufe zulässig, im ISR-State überhaupt keine Betriebssystem-Service-Aufrufe.

Die EVMA-Funktionen sind in erster Linie für taktsynchrone IRT-Funktionen gedacht, können aber bei Bedarf auch im RT-Mode verwendet werden.

4.1.13.2 PNIO_IsoActivateIsrObj

PNIO_IsoActivateIsrObj()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Setzt einen Komparator, der eine Callback-Funktion zu einem vorgegebenen Zeitpunkt aufruft. Der Zeitpunkt ist eine parametrierbare Verzögerungszeit zum Zyklusanfang (NewCycle).			
Input	PNIO_VOID (PNIO_VOID*)	pIsrCbf	Callbackfunktion, die zum Zeitpunkt (NewCycle + DelayTim_ns) aufgerufen wird. Die Ablaufebene (ISR oder DSR) kann in der Systemanpassung festgelegt werden.
	PNIO_UINT32	DelayTim_ns	Vorgebbare Verzögerungszeit in ns zum Zyklusanfang (NewCycle). Zulässige Werte sind 0 ... Sendclock. Eine Reduction Ratio wird nicht berücksichtigt!
	PNIO_ISO_OBJ_HNDL*	pObjHnd	Auf die vorgegebene Adresse liefert die Funktion ein Handle zurück, welches beim Löschen des Objekts mittels "PNIO_IsoFreeObj()" anzugeben ist. Konnte kein Iso-Objekt angelegt werden, wird "Null" zurückgeliefert.
Output	PNIO_UINT32	return	• "PNIO_OK": Auftrag wurde ordnungsgemäß ausgeführt • "PNIO_NOT_OK": Fehler bei Auftragsausführung aufgetreten

4.1.13.3 PNIO_IsoActivateGpioObj

PNIO_IsoActivateGpioObj()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Setzt einen Komparator, der einen Puls zu einem vorgegebenen Zeitpunkt an einem vorgegebenen GPIO ausgibt. Der Zeitpunkt ist eine parametrierbare Verzögerungszeit zum Zyklusanfang (NewCycle). Der GPIO wird dabei in der Funktion auf Datenrichtung „Ausgang“ gesetzt sowie die entsprechende Alternate-Funktion (PNPLL_OUT) eingestellt.			
Input	PNIO_UINT32	Gpio	GPIO-Nummer am ERTEC. Zulässig sind GPIO 0...7.
	PNIO_UINT32	DelayTim_ns	Vorgebbare Verzögerungszeit in ns zum Zyklusanfang (NewCycle). Zulässige Werte sind 0 ... Sendclock. Eine eventuell projektierte Reduction Ratio wird nicht berücksichtigt!
	PNIO_ISO_GPIO_LEVEL_TYPE	GpioLevelType	Legt fest, ob der Ausgang high- oder low-aktiv ist. Zulässig sind "ISO_GPIO_LOW_ACTIVE", "ISO_GPIO_HIGH_ACTIVE".
	PNIO_ISO_OBJ_HNDL*	pObjHnd	Auf die vorgegebene Adresse liefert die Funktion ein Handle zurück, welches beim Löschen des Objekts mittels "PNIO_IsoFreeObj()" anzugeben ist. Konnte kein Iso-Objekt angelegt werden, wird "Null" zurückgeliefert.
Output	PNIO_UINT32	return	• "PNIO_OK": Auftrag wurde ordnungsgemäß ausgeführt • "PNIO_NOT_OK": Fehler bei Auftragsausführung aufgetreten

4.1.13.4 PNIO_IsoActivateTransEndObj

PNIO_IsoActivateTransEndObj()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Setzt einen Komparator, der eine Callback-Funktion nach Bendigung des Übertragungszyklusses (TRANS_END-Event) am Bus aufruft. Der Zeitpunkt ist eine parametrierbare Verzögerungszeit zum Zyklusanfang (NewCycle).			
Input	PNIO_VOID (PNIO_VOID*)	pIsrCbf	Callbackfunktion, die zum Zeitpunkt (NewCycle + DelayTim_ns) aufgerufen wird. Die Ablaufebene (ISR oder DSR) kann in der Systemanpassung festgelegt werden.
	PNIO_ISO_OBJ_HNDL*	pObjHnd	Auf die vorgegebene Adresse liefert die Funktion ein Handle zurück, welches beim Löschen des Objekts mittels "PNIO_IsoFreeObj()" anzugeben ist. Konnte kein Iso-Objekt angelegt werden, wird "Null" zurückgeliefert.
Output	PNIO_UINT32	return	• "PNIO_OK": Auftrag wurde ordnungsgemäß ausgeführt • "PNIO_NOT_OK": Fehler bei Auftragsausführung aufgetreten

4.1.13.5 PNIO_IsoFreeObj

PNIO_IsoFreeObj()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Löschen eines vorher generierten Iso-Objektes. **Hinweis**: Ein mit "PNIO_IsoActivateGpioObj()" konfigurierter GPIO wird nicht neu konfiguriert. Soll dieser GPIO anschließend mit einer anderen Funktion belegt werden (z. B. als GPIO-Input), so muss die Applikation diese Umstellung mittels "Bsp_SetGpioMode" selber durchführen.			
Input	PNIO_ISO_OBJ_HNDL	ObjHnd	Objekt-Handle, welches zuvor mit "PNIO_IsoActivatexxxxx()" erzeugt wurde.
Output	PNIO_UINT32	return	• "PNIO_OK": Auftrag wurde ordnungsgemäß ausgeführt • "PNIO_NOT_OK": Fehler bei Auftragsausführung aufgetreten

4.1.14 Fehlerbehandlung

4.1.14.1 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	PNIO_ERR_ENUM	return	Die möglichen Werte sind in der enum-Typdefinition "PNIO_ERR_ENUM" in Datei "pnioerrx.h" hinterlegt: • "PNIO_OK" • "PNIO_ERR_xxxxxx"

Die Definition "PNIO_ERR_ENUM" ist in der Datei "pnioerrx.h" hinterlegt.

4.1.14.2 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 die Source-Datei (Package ID, Module ID) und die Zeilennummer im Code. Die Applikation kann z. B. abhängig von der Fehlerklasse Fehlerbehandlungen anstoßen.			
Input	PNIO_UINT32	DevHndl	Device-Handle, welches vom Stack mit "PNIO_device_open()" erzeugt wurde.
	PNIO_UINT32	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"

PNIO_Log()			Funktionsaufruf: IO-Stack > Applikation, synchron
	PNIO_UINT32	PackID	Kennzeichnet das Paket, in dem der Fehler aufgetreten ist. Definierte Werte sind: • "PNIO_PACKID_ACP" • "PNIO_PACKID_BSPADAPT" • "PNIO_PACKID_CLRPC" • "PNIO_PACKID_CM" • "PNIO_PACKID_DCP" • "PNIO_PACKID_EDD" • "PNIO_PACKID_GSY" • "PNIO_PACKID_LLDP" • "PNIO_PACKID_LSAS" • "PNIO_PACKID_MRP" • "PNIO_PACKID_NARE" • "PNIO_PACKID_OHA" • "PNIO_PACKID_OS" • "PNIO_PACKID_PNPB" • "PNIO_PACKID_PNDV" • "PNIO_PACKID_POF" • "PNIO_PACKID_SOCK" • "PNIO_PACKID_TSKMA" • "PNIO_PACKID_OTHERS"
	PNIO_UINT32	ModID	Kennzeichnet zusammen mit PackID das Modul, in dem der Fehler aufgetreten ist. Die Modul-ID ist eindeutig innerhalb eines Paketes.
	PNIO_UINT32	LineNum	Nummer der Zeile, in der der Fehler aufgetreten ist.
Output	-	-	-

4.1.14.3 PNIO_set_iops

PNIO_set_iops()			Funktionsaufruf: Applikation > IO-Stack, synchron
Mit dieser Funktion kann die Applikation optional den Providerstatus (IOPS) für Submodule setzen, die keine IO-Daten besitzen und für die daher eine IOPS-Aktualisierung im Rahmen des zyklischen Datenaustauschs nicht möglich ist. Dies gilt z. B. für das Ethernet Interface und die Ethernet Ports, die über Subslotnummern >=0x8000 modelliert werden. Der gesetzte Status wird mit dem nächsten Aufruf von "PNIO_initiate_data_write" zum Controller übertragen und bleibt erhalten, bis er durch einen erneuten Aufruf von "PNIO_set_iops" überschrieben wird.			
Input	PNIO_UINT32	Api	API-Nummer
	PNIO_UINT32	SlotNum	Slot-Nummer
	PNIO_UINT32	SubNum	Subslot-Nummer
	PNIO_UINT8	Iops	Zu setzender IOPS-Wert ("PNIO_S_GOOD", "PNIO_S_BAD")
Output	PNIO_UINT32	return	"PNIO_OK," "PNIO_NOT_OK"

4.1.15 Sonstige Funktionen

4.1.15.1 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	PNIO_INT8*	fmt	Formatstring, wie bei "printf"
	...	x, y, ...	Variable Argumentenliste wie bei "printf()"
Output	-	-	-

4.1.15.2 PNIO_TrcPrintf

PNIO_TrcPrintf()			**Funktionsaufruf: IO-Stack > Applikation, synchron / Applikation > Applikation, synchron**
Zentrale Ausgabe von Meldungen auf "PNIO_print()" oder zur späteren Auswertung als ASCII-Zeichen in 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 weitergeleitet. Der Anwender kann den Umlaufpuffer im Debugger auswerten oder mittels von ihm zu implementierenden Funktionen nach Belieben exportieren (z. B. über TCP/IP o.ä.).			
Input	PNIO_INT8*	fmt	Formatstring, wie bei "printf"
	...	x, y, ...	Variable Argumentenliste wie bei "printf()"
Output	-	-	-

4.1.15.3 PNIO_get_version

PNIO_get_version()			**Funktionsaufruf: Applikation > IO-Stack, synchron**
Auslesen der Versionskennung des Development Package.			
Input	PNIO_DK_VERSION*	pVersion	Zeiger auf die Versionskennung vom Typ "PNIO_VERSION". Die Funktion kopiert die Version an die vorgegebene Adresse.
Output	PNIO_UINT32	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 PNIO-Stack im Anlauf aufgerufen, bevor die Tasks des PNIO-Stacks gestartet wurden. Hier sind eventuell notwendige Initialisierungen für das BspAdapt-Interface einzutragen.			
Input	-	-	-
Output	PNIO_UINT32	return	"PNIO_OK"

4.2.1.2 Bsp_GetMacAddr

Bsp_GetMacAddr()			**Funktionsaufruf: IO-Stack > BSP, synchron**
Auslesen der lokalen Geräte-MAC-Adresse vom IO-Controller			
Input	PNIO_UINT8*	pDevMacAddr	Puffer für die 6 byte lange MAC-Adresse. Die Funktion schreibt anschließend die MAC-Adresse in den vorgegebenen Puffer.
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.2.1.3 Bsp_GetPortMacAddr

Bsp_GetPortMacAddr()			**Funktionsaufruf: IO-Stack > BSP, synchron**
Auslesen der lokalen Port-MAC-Adresse vom IO-Controller. Neben der Geräte-MAC-Adresse muss es bei PROFINET für jeden Port eine weitere MAC-Adresse geben.			
Input	PNIO_UINT8*	pPortMacAddr	Puffer für die 6 byte lange MAC-Adresse. Die Funktion schreibt anschließend die MAC-Adresse in den vorgegebenen Puffer.
	PNIO_UINT32	PortNum	Portnummer (1…N, N = Portanzahl), für die die Port-MAC-Adresse gelesen werden soll.
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.2.1.4 Bsp_EbSetLed (Implementierung optional)

Bsp_EbSetLed()			**Funktionsaufruf: IO-Stack > BSP, synchron**
Optionale Funktion zum Ein-/Ausschalten diverser LEDs auf dem Evaluation Board. Die Funktion wird nicht im PNIO-Stack selbst, sondern lediglich in der Beispielapplikation verwendet, um Betriebsbereitschaft und zyklischen Datenaustausch anzuzeigen.			
Input	PNIO_LEDTYPE	Led	Spezifiziert die LED-Nummer, siehe enum "PNIO_LEDTYPE" in Datei "bspadapt.h".
	PNIO_UINT32	Val	1: LED einschalten, 0: LED ausschalten
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.2.2 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 Address)
- Records des Physical Device (PDEV), die bei Verbindungsaufbau vom PNIO-Controller geschrieben wurden
- I&M 1...4 (Identifikations- und Maintenance-Daten, siehe /1/)

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.

4.2.2.1 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	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.2.2.2 Bsp_nv_data_store

Bsp_nv_data_store()		**Funktionsaufruf: IO-Stack > BSP, synchron**	
Die Funktion wird vom PNIO-Stack (im Rahmen der I&M-Daten) oder der Kundenapplikation 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 werden 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	PNIO_NVDATA_TYPE	NvDataType	Spezifiert den NV-Datentyp, siehe enum "PNIO_NVDATA_TYPE"
	PNIO_VOID*	pMem	Zeiger auf die zu speichernden Daten
	PNIO_UINT32	MemSize	Länge der zu speichernden Daten
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.2.2.3 Bsp_nv_data_restore

Bsp_nv_data_restore()		**Funktionsaufruf: IO-Stack > BSP, synchron**	
Die Funktion wird vom PNIO-Stack (im Rahmen der I&M-Daten) oder der Kundenapplikation im Anlauf aufgerufen, um die nichtflüchtigen Daten wie Gerätename, IP-Suite, I&M 1...4 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	PNIO_NVDATA_TYPE	NvDataType	Spezifiert den NV-Datentyp, siehe enum "PNIO_NVDATA_TYPE"
	PNIO_VOID**	ppMem	Zeiger, wohin die Daten von "Bsp_nv_data_restore" kopiert wurden. Mit jedem Aufruf von "Bsp_nv_data_restore" wird dafür automatisch vom Stack ein Speicher allokiert, der nach Gebrauch **vom aufrufenden Programm** mittels "Bsp_nv_data_memfree()" wieder freigegeben werden muss.
	PNIO_UINT32*	pMemSize	Länge der zu speichernden Daten
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.2.2.4 Bsp_nv_data_memfree

Bsp_nv_data_memfree()			**Funktionsaufruf: IO-Stack > 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	PNIO_VOID*	pMem	Zeiger auf die von "Bsp_nv_data_restore" erhaltenen Daten
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.2.3 Anbindung der ERTEC Switch Interrupts (nur für ERTEC-Plattformen)

Der im ERTEC 200P integrierte Switch besitzt je einen hochprioren und einen niederprioren Interrupt, die an den PROFINET IO-Stack angebunden werden müssen. Die Interrupts müssen über Betriebssystem-Funktionen an die Interrupthandler adaptiert werden. Dazu ruft der Stack die Funkton "Bsp_ErtecSwiIntConnect()" auf

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 200P. Die Funktion wird vom PROFINET IO-Stack aufgerufen. **Die Funktion sollte unverändert übernommen werden.**			
Input	PNIO_CBF_ERTEC_SWI_INT_H	pErtecSwiIntH	Adresse des Interrupthandlers für den hochprioren Switch Interrupt. Der Zeiger ist vom Typ "PNIO_CBF_ERTEC_SWI_INT_H"
	PNIO_CBF_ERTEC_SWI_INT_L	pErtecSwiIntL	Adresse des Interrupthandlers für den niederprioren Switch Interrupt. Der Zeiger ist vom Typ "PNIO_CBF_ERTEC_SWI_INT_L"
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.2.4 GPIO-Anbindung (nur für ERTEC-Plattformen)

4.2.4.1 Bsp_ReadGPIOin_0_to_31 (Implementierung optional)

Bsp_ReadGPIOin_0_to_31()			**Funktionsaufruf: Applikation > Bsp, synchron**
Liest die Werte der GPIOs 0 ... 31 aus und stellt diese im 32-bit-Returnwert zur Verfügung.			
Input	-	-	-
	-	-	-
Output	PNIO_UINT32	return	Werte der GPIOs 0 ... 31, bitweise kodiert (GPIO0 in Bit 0, GPIO1 in Bit 1, usw.).

4.2.4.2 Bsp_SetGPIOout_0_to_31 (Implementierung optional)

Bsp_SetGPIOout_0_to_31()			**Funktionsaufruf: Applikation > BSP, synchron**
Setzt die Werte der GPIOs 0 ... 31 auf "1". Über eine Bitmaske kann dabei ausgewählt werden, welche GPIOs gesetzt werden sollen und welche unverändert bleiben sollen.			
Input	PNIO_UINT32	OutMsk	Bitweise kodierte Maske für die GPIOs 0 ... 31. Damit wird für jeden GPIO festgelegt, ob dieser gesetzt werden soll oder unverändert bleibt. • Bit = 1: GPIO wird gesetzt • Bit = 0: GPIO bleibt unverändert
Output	-	-	-

4.2.4.3 Bsp_ClearGPIOout_0_to_31 (Implementierung optional)

Bsp_ClearGPIOout_0_to_31()			**Funktionsaufruf: Applikation > BSP, synchron**
Setzt die Werte der GPIOs 0 ... 31 auf "0". Über eine Bitmaske kann dabei ausgewählt werden, welche GPIOs rückgesetzt werden sollen und welche unverändert bleiben sollen.			
Input	PNIO_UINT32	OutMsk	Bitweise kodierte Maske für die GPIOs 0 ... 31. Damit wird für jeden GPIO festgelegt, ob dieser rückgesetzt werden soll oder unverändert bleibt. • Bit = 1: GPIO wird rückgesetzt • Bit = 0: GPIO bleibt unverändert
Output	-	-	-

4.3 Schnittstelle zum Betriebssystem

4.3.1 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 Betriebssystem-Abstraktionsfunktionen sind im Modul "xxx_OS.C" implementiert. Alle notwendigen Schnittstellendefinitionen und Defines dazu sind in "OS.H" enthalten.

4.3.2 Verwaltung von Ressourcen

Die Verwaltung und 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 IDs 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.

4.3.3 Beschreibung der zu portierenden OS-Funktionen

4.3.3.1 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	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.3.3.2 OsAllocFX()

OsAllocFX()			**Funktionsaufruf: IO-Stack > Betriebssystem**
Allokierfunktion für dynamischen Speicher. Der Speicher wird nicht vorbelegt.			
Input	PNIO_VOID**	ppMem	Zeiger, wohin die Adresse des allokierten Speichers zu schreiben ist
	PNIO_UINT32	Length	Länge des Speichers in byte
	PNIO_UINT32	PoolID	Zur Laufzeitoptimierung können (optional) 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. "MEMPOOL_UNCACHED" und "MEMPOOL_RX_TX_BUF" werden derzeit nicht verwendet.
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.3.3.3 OsFreeX()

OsFreeX()		**Funktionsaufruf: IO-Stack > Betriebssystem**	
Rückgabe eines 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	PNIO_VOID*	pMem	Adresse des allokierten Speichers
	PNIO_UINT32	PoolID	Pool-ID, die beim Allokieren verwendet wurde
Output	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

4.3.3.4 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	PNIO_UINT16*	timer_id_ptr	Adresse, ab der die Timer-ID als Rückgabeparameter abgelegt wird
	PNIO_UINT16	timer_type	Timer-Typ (zyklischer oder One Shot-Timer). Mögliche Werte sind: • "LSA_TIMER_TYPE_ONE_SHOT" • "LSA_TIMER_TYPE_CYCLIC"
	PNIO_UINT16	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"
	PNIO_VOID*	callback_timeout	Callback-Funktion, welche bei Ablauf des Timers aufgerufen wird. Übergabeparameter sind dabei Timer-ID und User-ID. Die Timer-ID wird vom Stack bei "OsAllocTimer()" vergeben, die User-ID ist vom Anwender frei wählbar.
Output	PNIO_UINT32	return	"LSA_OK", "LSA_RET_ERR_PARAM"

4.3.3.5 OsStartTimer()

OsStartTimer()			Funktionsaufruf: IO-Stack > Betriebssystem
Input	PNIO_UINT16	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.
	PNIO_UINT32	user_id	Frei vom Anwender vergebbare User-ID, ist ebenfalls Übergabeparameter der Callback-Funktion
	PNIO_UINT16	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 50 ms aufgerufen.
Output	PNIO_UINT32	return	• "LSA_OK": OK, Timer wurde gestartet • "LSA_RET_ERR_PARAM": Parametrierfehler

4.3.3.6 OsStopTimer()

OsStopTimer()			Funktionsaufruf: IO-Stack > Betriebssystem
Anhalten eines laufenden Timers.			
Input	PNIO_UINT16	timer_id	Referenz des Timers
Output	PNIO_UINT32	return	• "LSA_OK": OK, Timer wurde angehalten • "LSA_RET_ERR_PARAM": Parametrierfehler

4.3.3.7 OsFreeTimer()

OsFreeTimer()			Funktionsaufruf: IO-Stack > Betriebssystem
Einen mit "OsAllocTimer" belegten Timer wieder freigeben.			
Input	PNIO_UINT16	timer_id	Referenz des Timers
Output	PNIO_UINT32	return	"LSA_OK", "LSA_RET_ERR_TIMER_IS_RUNNING"

4.3.3.8 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	PNIO_UINT32	MutexId	Identifier des Mutex. Mögliche Werte sind: • 0…. (MAXNUM_OF_NAMED_MUTEXES - 1)
Output		-	-

4.3.3.9 OsExitX

OsExit()			Funktionsaufruf: IO-Stack > Betriebssystem
Freigeben eines belegten Mutex.			
Input	PNIO_UINT32	MutexId	Identifier des Mutex. Mögliche Werte sind: • 0…. (MAXNUM_OF_NAMED_MUTEXES - 1)
Output	-	-	-

4.3.3.10 OsEnterShort

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

4.3.3.11 OsExitShort

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

4.3.3.12 OsAllocSemB

OsAllocSemB()			Funktionsaufruf: IO-Stack > Betriebssystem
Erzeugt ein binäres Semaphore. Das Semaphore muss im Initialzustand leer sein.			
Input	PNIO_UINT32*	pSemId	Zeiger, ab dem die Semaphore-ID zurückgeliefert wird.
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.3.3.13 OsFreeSemB

OsFreeSemB()			**Funktionsaufruf: IO-Stack > Betriebssystem**
Löscht ein binäres Semaphore, welches zuvor mit "OsAllocSemB()" erzeugt wurde.			
Input	PNIO_UINT32	SemId	Semaphore-ID, die von "OsAllocSemB()" erzeugt wurde.
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.3.3.14 OsTakeSemB

OsTakeSemB()			**Funktionsaufruf: IO-Stack > Betriebssystem**
Belegt ein binäres Semaphore. Die Funktion blockiert, wenn das Semaphore zuvor bereits anderweitig belegt wurde.			
Input	PNIO_UINT32	SemId	Semaphore-ID, die von "OsAllocSemB()" erzeugt wurde.
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.3.3.15 OsGiveSemB

OsGiveSemB()			**Funktionsaufruf: IO-Stack > Betriebssystem**
Gibt ein mit "OsTakeSemB()" belegtes binäres Semaphore wieder frei.			
Input	PNIO_UINT32	SemId	Semaphore-ID, die von "OsAllocSemB()" erzeugt wurde.
Output	PNIO_UINT32	return	"PNIO_OK", "PNIO_NOT_OK"

4.3.3.16 OsSetThreadPrio

OsSetThreadPrio()			**Funktionsaufruf: IO-Stack > Betriebssystem**
Hinweis:Wird nicht auf ERTEC-Plattformen verwendet, nur beim Soft_EDD vorhanden. Ändern der Taskpriorität. Im IO-Stack wird damit die Thread-Priorität 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 Ausgangsmakros "EDD_ENTER_HIGH" und "EDD_EXIT_HIGH". Dort wird die Priorität der Task vorübergehend auf "TASK_PRIO_HIGHEST" angehoben.			
Input	PNIO_UINT32	ThreadId	ID des Thread, dessen Priorität verändert werden soll.
	PNIO_UINT32	NewThreadPrio	Neu einzustellender Prioritätswert
Output	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

Hinweis

Eine Synchronisation von EDD-High Thread 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.

4.3.3.17 OsCreateThread

OsCreateThread()			**Funktionsaufruf: IO-Stack > Betriebssystem**
Erzeugen eines Thread (Task). Alle Tasks laufen im selben Adressraum.			
Input	PNIO_VOID*	pThreadEntry	Einsprungadresse für die Task.
	PNIO_UINT8*	pThreadName	Angabe eines Namens für den Thread. Dies ist nur für Debugzwecke gedacht und kann entfallen, wenn das Betriebssystem diese Möglichkeit nicht unterstützt.
	PNIO_UINT32	ThreadPrio	Priorität der Task. **Hinweis:** Die Prioritäten der Tasks des IO-Stacks werden in der Datei "os_cfg.h" eingestellt.
	PNIO_UINT32*	pThreadId	Hier wird die Adresse der zurückgelieferten Thread-ID vorgegeben.
Output	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

4.3.3.18 OsStartThread

OsStartThread()			**Funktionsaufruf: IO-Stack > Betriebssystem**
Starten des mit "OsCreateThread()" erzeugten Thread. **Hinweis:** Bei manchen Betriebssystemen wird mit dem Create-Aufruf der Thread automatisch gestartet, bei anderen ist ein zusätzlicher Startaufruf 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	PNIO_UINT32	ThreadId	Thread-ID, wurde bei "OsCreateThread()" als Rückgabeparameter übergeben
Output	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

4.3.3.19 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	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

4.3.3.20 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 ISRs aufgerufen werden kann, wurde von OS für diesen Fall keine Thread-ID vergeben, da Systemtasks oder ISRs natürlich nicht mit "OsCreateThread" erzeugt wurden. Das bedeutet für den Anwender lediglich, dass er als Dummy-Thread-ID diejenige vom Post-Thread ("TskIdPost") zurückliefern muss.			
Input	-	-	-
Output	PNIO_UINT32	return	Thread-ID der laufenden Task. Bei unbekannten Threads, d. h. solchen, die nicht mit "OsCreateThread()" erzeugt wurden, muss "TskIdPost" zurückgeliefert werden.

4.3.3.21 OsCreateMsgQueue()

OsCreateMsgQueue()			Funktionsaufruf: IO-Stack > Betriebssystem
Erzeugen einer Message Queue. Eine Message Queue ist hier immer fest einem Thread zugeordnet. Einem Thread darf höchstens 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 "OsCreateMsgQueue" 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 Thread-Kontext "Thread_1" aufzurufen, wird "OsSendMessageX (Thread_1, fx, pdata, OS_MBX_PRIO_NORM)" aufgerufen.			
Input	PNIO_UINT32	ThreadId	ID des Thread, der die Message Queue zugeordnet werden soll.
Output	PNIO_UINT32	return	• "LSA_OK": Message Queue konnte zugeordnet werden • "LSA_NOT_OK": Fehler ist aufgetreten.

4.3.3.22 OsWait_ms()

OsWait_ms()			Funktionsaufruf: IO-Stack > Betriebssystem
Warten einer vorgegebenen Zeit.			
Input	PNIO_UINT32	PauseTime_ms	Wartezeit in Millisekunden.
Output	-	-	-

4.3.3.23 OsGetTime_us()

OsGetTime_us()			Funktionsaufruf: IO-Stack > Betriebssystem
Auslesen der aktuellen Zeit in Mikrosekunden seit dem letzten Systemstart.			
Input	-	-	-
Output	PNIO_UINT32	return	Aktuelle Zeit in Mikrosekunden.

4.3.3.24 OsGetUnixTime()

OsGetUnixTime()			Funktionsaufruf: IO-Stack > Betriebssystem
Auslesen der aktuellen Zeit in Sekunden. Hier wird die Anzahl Sekunden seit dem Systemstart angegeben.			
Input	-	-	-
Output	PNIO_UINT32	return	Aktuelle Zeit in Sekunden.

4.3.3.25 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. **Hinweis:** Ab Version 3.0.0 des Development Kit wurde die stack-interne zeitkritische Kommunikation von 8 byte auf 4 byte lange Messages umgestellt, um auch auf solche Betriebssysteme zeitoptimiert aufsetzen zu können, die nur 4 byte statt 8 byte lange Messages unterstützen.			
Input	PNIO_VOID**	ppMessage	PtrPtr auf Message
	PNIO_UINT32	ThreadId	ID des empfangenden (und damit die ID des eigenen) Thread
Output	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

4.3.3.26 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	PNIO_VOID**	ppMessage1	PtrPtr auf Message1
	PNIO_VOID**	ppMessage2	PtrPtr auf Message2
	PNIO_UINT32	ThreadId	ID des empfangenden (und damit die ID des eigenen) Thread
Output	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

4.3.3.27 OsSendMessage()

OsSendMessage()		**Funktionsaufruf: IO-Stack > Betriebssystem**	
Senden einer Message mit nur einem statt zwei Pointern. Empfangsseitig wird die Message mit "OsReadMessageBlocked()" empfangen.			
Input	PNIO_UINT32	ThreadId	ID des Empfänger-Thread
	PNIO_VOID*	pMessage	Zeiger auf die eigentliche Message
	PNIO_UINT32	MsgPrio	Der Message kann eine Priorität zugeordnet werden. Innerhalb der Systemimplementierung des PNIO-Thread haben alle Messages die gleiche Priorität. Der Parameter kann aber genutzt werden, wenn dies im Rahmen einer speziellen Portierung des IO-Stacks von Vorteil ist.
Output	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

4.3.3.28 OsSendMessageX()

OsSendMessageX()		**Funktionsaufruf: IO-Stack > Betriebssystem**	
Extended "OsSendMessage"-Funktion: Senden einer Message mit 2 Pointern. Empfangsseitig wird die Message mit "OsReadMessageBlockedX()" empfangen. Damit können 2 getrennte Messages mit einem "OsSendMessageX()" Aufruf übertragen werden. Begründung siehe Beschreibung der Funktion "OsCreateMsgQueue()".			
Input	PNIO_UINT32	ThreadId	ID des Empfänger-Thread
	PNIO_VOID*	pMessage1	Zeiger auf die eigentliche Message 1
	PNIO_VOID*	pMessage2	Zeiger auf die eigentliche Message 2
	PNIO_UINT32	MsgPrio	Der Message kann eine Priorität zugeordnet werden. Innerhalb der Systemimplementierung des PNIO-Thread haben alle Messages die gleiche Priorität. Der Parameter kann aber genutzt werden, wenn dies im Rahmen einer speziellen Portierung des IO-Stacks von Vorteil ist.
Output	PNIO_UINT32	return	"LSA_OK", "LSA_NOT_OK"

4.3.3.29 __InterlockedDecrement()

__InterlockedDecrement()		**Funktionsaufruf: IO-Stack > Betriebssystem**	
Dekrementieren eines Long-Wertes unter Interruptsperre.			
Input	PNIO_INT32	pVal	Adresse des zu dekrementierenden Wertes
Output	PNIO_INT32*	pVal	Der Wert ab der vorgegebenen Adresse wird dekrementiert.
	PNIO_INT32	return	Zusätzlich wird das Ergebnis ("*pVal") auch als Rückgabewert übertragen.

4.3.3.30 __InterlockedIncrement()

__InterlockedIncrement()			**Funktionsaufruf: IO-Stack > Betriebssystem**
Inkrementieren eines Long-Wertes unter Interruptsperre.			
Input	PNIO_INT32	pVal	Adresse des zu inkrementierenden Wertes
Output	PNIO_INT32*	pVal	Der Wert ab der vorgegebenen Adresse wird inkrementiert.
	PNIO_INT32	return	Zusätzlich wird das Ergebnis ("*pVal") auch als Rückgabewert übertragen.

4.3.3.31 OsIntDisable()

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

4.3.3.32 OsIntEnable()

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

4.3.4 Kapselung von Funktionsaufrufen der Standard-Libraries

Um eine möglichst große Plattformunabhängigkeit zu erreichen, werden im PNIO-Stack keine Standard-Library-Funktionen direkt aufgerufen. Stattdessen werden diese ebenfalls über das OS-Abstraktionsinterface geleitet. Damit müssen die zugehörigen Standard-Header-Dateien auch nur in wenigen Modulen des Development Package eingebunden werden.

Die Aufrufe werden in der Regel unverändert an die Standard-Library 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.

4.3.5 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
- "OsKeyScan32" liest einen 32-bit-Zahlenwert von der Standardkonsole
- "OsReboot" führt einen Wiederanlauf des Systems durch

4.4 Wichtige Hinweise und Einschränkungen

4.4.1 Anzahl von IO-Devices

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

4.4.2 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.

Es können beliebige Slot- und Subslotnummern mit Lücken verwendet werden. Somit ist der volle in der PNIO-Spezifikation definierte Wertebereich der Slot- und Subslotnummern nutzbar, lediglich die maximale Anzahl von Submodulen ist limitiert.

4.4.3 Maximale Anzahl der Nutzdaten für ein Device

Die maximale Anzahl der Prozessdaten wird durch die maximale Telegrammgröße (netto 1440 byte IOCR-Daten) und die Anzahl der Subslots mit Input- bzw. Output-Daten (Parameter in der GSD-Datei) bestimmt. Die maximale Datenlänge in byte ist:

- Max. Anzahl Inputbytes = MaxInputLength - 4 - (Anzahl Input-Subslots) – (Anzahl Output-Subslots)
- Max. Anzahl Outputbytes = MaxOutputLength - 4 - (Anzahl Input-Subslots) – (Anzahl Output-Subslots)

Ein Submodul, welches sowohl Input- als auch Output-Daten enthält, ist dabei sowohl als Input-Subslot als auch als Output-Subslot zu zählen.

"MaxInputLength" und "MaxOutputLength" sind Attribute in der GSD-Datei. Für den ERTEC 200P sind beide Werte auf 1440 byte festgelegt.

4.4.4 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

Anhang A

A.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 in der GSD Datei
DBAI	**D**irect **B**uffer **A**ccess **I**nterface
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 200P	**E**valuation **B**oard für ERTEC 200P (Bestandteil des Evaluation Kit ERTEC 200P)
EDDI	**E**thernet **D**evice **D**river für **I**RTE switch im ERTEC 200/400 (frühere Versionen: EDD_ERTEC)
EDDP	**E**thernet **D**evice **D**river für **P**N switch im ERTEC 200P
EDDS	**E**thernet **D**evice **D**river für **S**tandard Ethernet Controller (frühere Versionen: EDD_soft)
EDD	**E**thernet **D**evice **D**river, Oberbegriff für EDDI, EDDP, EDDS
ELOG	**E**rror **Log**ging für Debug-Zwecke (Level 1 = nur Fehler ausgeben)
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 **I**O-Stacks
IOCR	**I**nput/**O**utput **C**ommunication **R**elationship
IOCS	**I**nput/**O**utput Object **C**onsumer **S**tatus
IOPS	**I**nput/**O**utputObject **P**rovider **S**tatus
IRT	**I**sochrone **R**eal**T**ime, Class 2 (**I**RT Class 2) oder Class 3 (**I**RT Class 3)
LLDP	**L**ink **L**ayer **D**iscovery **P**rotocol (**I**EEE 802.1AB, Allows stations to exchange chassis and port information)
LSA	**L**ayer **S**tructure **A**rchitecture
LW	Laufwerk
MIB	**M**anagement **I**nformation **B**ase. Datenbasis für SNMP Dienste

MRP	**M**edia **R**edundancy **P**rotocol
NARE	**N**ame **A**ddress **Re**solution
NRT	**N**on **R**eal**t**ime ist ein Oberbegriff für alle Nicht – Realtime Telegramme (nicht Typ 0x8892)
NV	**N**on **V**olatile
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	**R**eal**t**ime ist ein Oberbegriff für azyklische und zyklische Realtime
RT Class	**R**eal**t**ime-Klasse gemäß PROFINET IO-Spezifikation
SI	**S**tandard **I**nterface
SNMP	**S**imple **N**etwork **M**anagement **P**rotokoll
SOCK	UDP Socket Interface für PROFINET IO, bezeichnet eines der Software-Pakete des IO-Stacks
TAP	**T**est **A**ccess **P**ort
UDP	**U**ser **D**atagram **P**rotokol
UUID	**U**niversal **U**nique **Id**entifier

A.2 Literaturverzeichnis

/1a/
PROFINET IO Specification IEC 61158 – part5

PROFINET IO Application Layer Service Specification

(Downloadbar von der PNO Website (http://www.profibus.com))

/1b/
PROFINET IO Specification IEC 61158 – part6

PROFINET IO Application Layer Protocol Specification

(Downloadbar von der PNO Website (http://www.profibus.com))

/2/
GSDML Specification for PROFINET IO

Version 2.3, Order No: 2.352

PROFIBUS Nutzerorganisation e.V.

(Downloadbar von der PNO Website (http://www.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://www.profibus.com))