SIEMENS

SIMATIC

PROFINET
How to Port PN Driver V1.1

Parameter Manual

11/2015

A5E36330816-AA

Legal information

Warning notice system

This manual contains notices you have to observe in order to ensure your personal safety, as well as to prevent damage to property. The notices referring to your personal safety are highlighted in the manual by a safety alert symbol, notices referring only to property damage have no safety alert symbol. These notices shown below are graded according to the degree of danger.

⚠ DANGER

indicates that death or severe personal injury **will** result if proper precautions are not taken.

⚠ WARNING

indicates that death or severe personal injury **may** result if proper precautions are not taken.

⚠ CAUTION

indicates that minor personal injury can result if proper precautions are not taken.

NOTICE

indicates that property damage can result if proper precautions are not taken.

If more than one degree of danger is present, the warning notice representing the highest degree of danger will be used. A notice warning of injury to persons with a safety alert symbol may also include a warning relating to property damage.

Qualified Personnel

The product/system described in this documentation may be operated only by **personnel qualified** for the specific task in accordance with the relevant documentation, in particular its warning notices and safety instructions. Qualified personnel are those who, based on their training and experience, are capable of identifying risks and avoiding potential hazards when working with these products/systems.

Proper use of Siemens products

Note the following:

⚠ WARNING

Siemens products may only be used for the applications described in the catalog and in the relevant technical documentation. If products and components from other manufacturers are used, these must be recommended or approved by Siemens. Proper transport, storage, installation, assembly, commissioning, operation and maintenance are required to ensure that the products operate safely and without any problems. The permissible ambient conditions must be complied with. The information in the relevant documentation must be observed.

Trademarks

All names identified by ® are registered trademarks of Siemens AG. The remaining trademarks in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owner.

Disclaimer of Liability

We have reviewed the contents of this publication to ensure consistency with the hardware and software described. Since variance cannot be precluded entirely, we cannot guarantee full consistency. However, the information in this publication is reviewed regularly and any necessary corrections are included in subsequent editions.

Siemens AG
Division Digital Factory
Postfach 48 48
90026 NÜRNBERG
GERMANY

A5E36330816-AA
℗ 10/2015 Subject to change

Table of contents

Introduction 1

1.1 Scope

The scope of this document is to give a list of all system-dependent interfaces of PROFINET Driver for Controller (referred to below as "PN Driver") which are provided to make PN Driver portable to various operating systems.

PN Driver V1.1 is delivered entirely in source code and with a sample implementation for Microsoft Windows as well as a sample implementation for the Linux derivate Linux Debian 7.6 with Kernel V3.2 with a preemptive patch applied.

The PN Driver uses the Embedded PROFINET System (referred to below as “EPS”) to abstract the implementation of the IO-Base API from the underlying hardware, operating system and network adapters.

The sample implementation for Microsoft Windows uses WinPcap to make use of any Ethernet adapter installed on a Windows machine that is supported by WinPcap.

The sample implementation for Linux Debian uses an Intel lower layer implementation to make use of the Intel Ethernet adapters Intel I210 (referred to below as “Springville”) and Intel 82574L (referred to below as “Hartwell”).

1.2 Additional support

Additional user documentation

Besides this manual there are additional manuals for the PROFINET Driver for Controller:

- PROFINET IO-Base user programming interface manual
- PROFINET Driver for Controller Engineering Interface programming and operating manual
- Quick Start PN Driver V1.1 getting started manual

Additional support

If you have questions regarding the described PROFINET Driver for Controller that are not addressed in the documentation, please contact your local representative at the Siemens office nearest to you.

Please send questions, comments and suggestions regarding this manual in writing to the specified e-mail address.

In addition, you will find general information, current product information, FAQs and downloads that can be useful on the Internet (http://www.siemens.com/comdec).

Technical contact for Germany/worldwide

Siemens AG	Phone: +49 911 750 2080
ComDeC (http://www.siemens.com/comdec)	Phone: +49 911 750 4384
	Phone: +49 911 750 2078
	Fax: +49 911 750 2100
	E-mail: (mailto:ComDeC@siemens.com)
Office address:	Postal address:
Würzburger Str. 121	P.O. Box 2355
90766 Fürth, Germany	90713 Fürth, Germany

Technical contact for the USA

PROFI Interface Center (http://www.profiinterfacecenter.com)	Phone: +1 (423) 262-2576
	Fax: +1 (678) 297-7289
One Internet Plaza	E-mail: (mailto:PIC.industry@siemens.com)
Johnson City, TN 37604	

Technical contact for China

PROFI Interface Center China	Fax: +86-10-6476 4725
7, Wangjing Zhonghuan Nanlu	E-mail: (mailto:Profinet.cn@siemens.com)
100102 Beijing	

1.3 Security information

Siemens provides products and solutions with industrial security functions that support the secure operation of plants, solutions, machines, equipment and/or networks. They are important components in a holistic industrial security concept. With this in mind, Siemens' products and solutions undergo continuous development. Siemens recommends strongly that you regularly check for product updates.

For the secure operation of Siemens products and solutions, it is necessary to take suitable preventive action (e.g. cell protection concept) and integrate each component into a holistic, state-of-the-art industrial security concept. Third-party products that may be in use should also be considered. You can find more information about industrial security on the Internet (http://www.siemens.com/industrialsecurity).

To stay informed about product updates as they occur, sign up for a product-specific newsletter. You can find more information on the Internet (http://support.automation.siemens.com).

1.4 Overview

The PN Driver implements the IO-Base API specified in PROFINET IO-Base user programming interface manual.

Internally, the PN Driver uses the EPS to integrate the Siemens implementation of the PROFINET IO stack.

Note

The PROFINET stack must be compiled as a 32-bit library or 32-bit application.

PN Driver test application running on a Debian Linux derivative

The following figure shows the PN Driver test application running on a Debian Linux derivative. The EPS contains the PROFINET stack. The network adapters Springville and Hartwell can be used to establish a PROFINET network. The Ethernet device driver for standard adapters (EDDS) uses the Intel lower layer implementation provided by the PROFINET stack to send and receive frames on Springville and Hartwell boards.

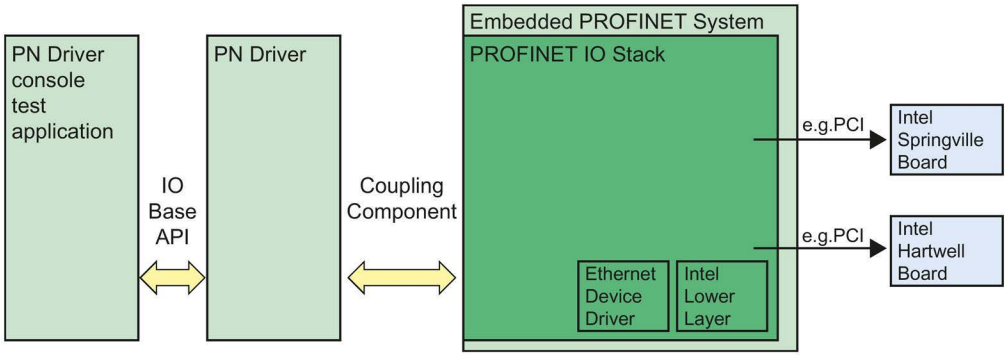


Figure 1-1 PN Driver test application - Debian Linux

PN Driver test application running on a Windows 7 PC

The following figure shows the PN Driver test application running on a Windows 7 PC. The EDDS uses the Packet32 lower layer provided by the PROFINET stack. This lower layer uses WinPcap to establish a network connection on a driver supported by WinPcap.
PN Driver supports all Ethernet adapters that are supported by WinPcap.

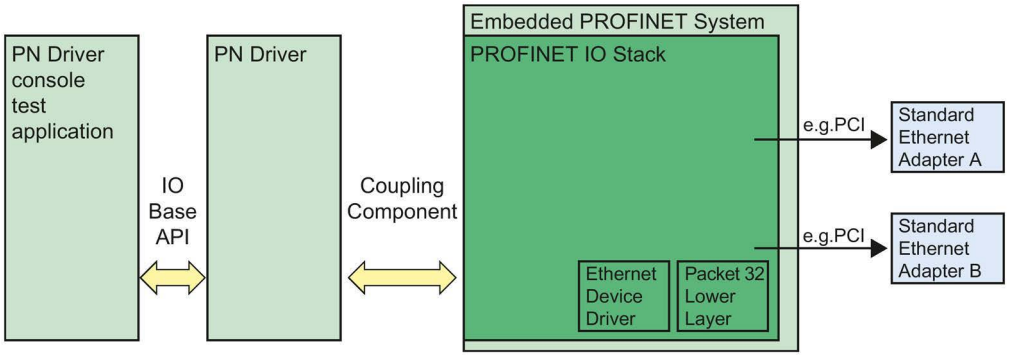


Figure 1-2 PN Driver test application - Windows 7

Overview of the porting of PN Driver

A systems integrator may want to port the PN Driver to other operating systems such as VxWorks or to an embedded system with a proprietary operating system. As it is not feasible to describe the porting to each and every operating system, the following paragraphs will provide an overview of where the PN Driver has to be adapted.

The architecture of the EPS is portable in order to fulfill these aims. Since the PN Driver uses the EPS to make calls to the operating system or the hardware, the PN Driver can be ported to other systems by adapting the EPS.

The architecture of the EPS is described in the following sections. Also, hints are provided for the systems integrator for porting the PN Driver/EPS to another system or adding additional Ethernet adapters to the EPS.

Depending on the target platform, operating system and used Ethernet adapter, the number of porting steps may vary.

Overview of how to port PN Driver to other operating systems

PN Driver interfaces with the EPS which itself is based on the POSIX API; see section POSIX API/Macro IF (OS abstraction) (Page 11).

- If the desired operating system includes an implementation of the POSIX API, you will be able to adapt PN Driver to that operating system by following the description in this manual.
- If you want to use an operating system other than Windows 7 or Linux Debian, the native POSIX API of that operating system should be used. Its native implementation must be linked to the EPS POSIX API/Macro IF. The porting instructions in section Porting instructions (Page 13) must be followed.
- If the desired operating system does not include an implementation of the POSIX API, you may use any appropriate implementation to adapt PN Driver to that operating system. The implementation must be linked to the EPS POSIX API/Macro IF.

  See, as an example, the implementation for the Windows operating system that the PN Driver offers. Please observe the Windows sample in section POSIX functions (Page 11) as well as the porting instructions in section Porting instructions (Page 13).

Overview of how to port PN Driver to other platforms

- If you want to use an embedded platform other than a personal computer, e. g. an embedded system, the platform abstraction of the EPS must be adapted. Please refer to the porting instructions in section Porting instructions (Page 23).
- These adaptations being done, changes to the configuration of the EPS are also necessary. The changes are described in section EPS APP (application abstraction) (Page 24).

Overview of how to use other Intel Ethernet adapters

You can use PN Driver on Windows 7 with any Ethernet adapter supported by WinPcap, and on Linux Debian having RT patch with Intel Springville or Intel Hartwell Ethernet adapters. If you choose to use the PN Driver Linux implementation as a base for your porting, and want to use Intel Ethernet adapters **other** than Intel Springville or Intel Hartwell, you have to implement an EPS PN board driver.

- Please observe the PN device driver sample implementation "eps_pndevdrv", see section Integration of the Intel lower layer implementation into a PN device driver (Page 46).
- You will find additional information as well as a description of the driver concept in section EPS driver concept (Ethernet adapters abstraction) (Page 32).
- A summary is given in section Porting instructions (Page 54).
- These implementations being done, changes to the configuration of the EPS are also necessary. The changes are described in section EPS APP (application abstraction) (Page 24).

Overview of the Embedded PROFINET System (EPS)

The following figure shows an overview of the EPS.

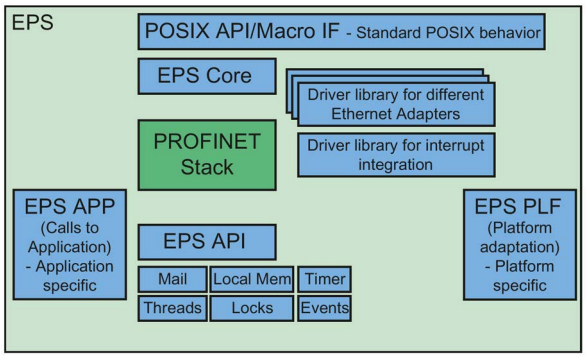


Figure 1-3 Structure of the Embedded PROFINET System

The EPS contains these parts (see figure above):

- A core with an API to start up/shut down the PROFINET stack
- An implementation and API for
  - Locks
  - Threads
  - Mailbox
  - Events
  - Timer
  - Local memory management
- The PROFINET stack components
  - Logical Device (LD) PNIO components
  - Hardware Device (HD) PNIO components
  - The standard integration of these components into PROFINET stack interface (PSI)
- An abstraction to the POSIX implementation of the operating system (see section POSIX API/Macro IF (OS abstraction) (Page 11)) as well as the platform plus OS itself (see section EPS PLF (hardware abstraction) (Page 14))
- An abstraction to the application (see section EPS APP (application abstraction) (Page 24))
- An API to integrate pluggable drivers (see section EPS driver concept (Ethernet adapters abstraction) (Page 32))
- A library of sample drivers for different network adapters (see section PN device driver (Page 34))
- A trace system for internal diagnostics (see section Trace (Page 55))

POSIX API/Macro IF (OS abstraction) 2

2.1 POSIX functions

POSIX functions and POSIX macros

PN Driver uses POSIX macros to call POSIX functions. A detailed description of the intended behavior of these POSIX functions can be found on the Internet (http://www.unix.org/single_unix_specification).

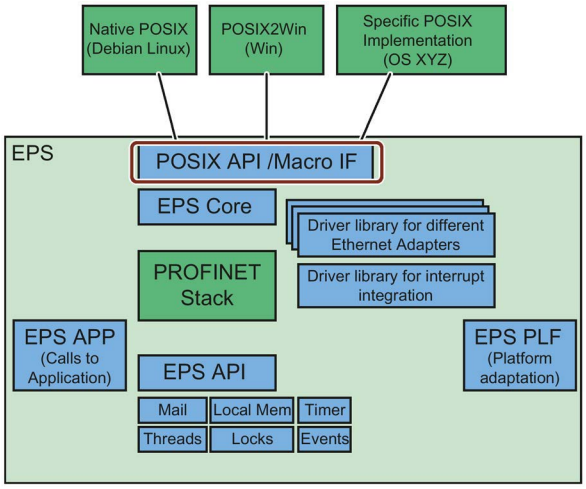


Figure 2-1 EPS POSIX API/Macro IF overview

Since there is more than one version of the POSIX API, some of the referenced functions may not be implemented in the POSIX version that the target operating system provides. The Windows operating system does not natively provide a POSIX API. You can implement the POSIX functionality as in the **Windows** sample below. Moreover, the macros can be linked directly to a system function, see the **Linux** sample below.

Table 2- 1 Mapping of EPS output functions

EPS output function	Default mapping (POSIX implementation)
EPS_POSIX_PTHREAD_ATTR_INIT	int pthread_attr_init(pthread_attr_t *attr);
EPS_POSIX_PTHREAD_ATTR_SETINHERITSCHED	int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
EPS_POSIX_PTHREAD_ATTR_SETSTACK	int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize)
EPS_POSIX_PTHREAD_ATTR_SETSTACKSIZE	int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)

EPS output function	Default mapping (POSIX implementation)
EPS_POSIX_PTHREAD_ATTR_SETSCHEDPARAM	int pthread_attr_setschedparam(pthread_attr_t * attr,const struct sched_param *param);
EPS_POSIX_PTHREAD_ATTR_SETSCHEDPOLICY	int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
EPS_POSIX_PTHREAD_ATTR_SETCPUAFFINITY	No default implementation available. See sample in \src\source\pnboards\eps\epssys\etc\windows\posix2win.c
EPS_POSIX_PTHREAD_CREATE	int pthread_create(pthread_t * thread,const pthread_attr_t * attr,void*(*start_routine)(void*), void * arg);
EPS_POSIX_PTHREAD_JOIN	int pthread_join(pthread_t thread, void** th_ret);
EPS_POSIX_PTHREAD_KILL	int pthread_kill(pthread_t thread, int sig);
EPS_POSIX_PTHREAD_MUTEX_DESTROY	int pthread_mutex_destroy(pthread_mutex_t *mutex);
EPS_POSIX_PTHREAD_MUTEX_INIT	int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *attr);
EPS_POSIX_PTHREAD_MUTEX_LOCK	int pthread_mutex_lock(pthread_mutex_t *mutex);
EPS_POSIX_PTHREAD_MUTEX_UNLOCK	int pthread_mutex_unlock(pthread_mutex_t *mutex);
EPS_POSIX_TIMER_CREATE	int timer_create(clockid_t clockid, struct sigevent * evp,timer_t * timerid);
EPS_POSIX_TIMER_SETTIME	int timer_settime(timer_t timerid, int flags, const struct itimerspec * value,struct itimerspec * ovalue);
EPS_POSIX_SEM_INIT	int sem_init(sem_t *sem, int pshared, unsigned value);
EPS_POSIX_SEM_DESTROY	int sem_destroy(sem_t *sem);
EPS_POSIX_SEM_WAIT	int sem_wait(sem_t *sem);
EPS_POSIX_SEM_POST	int sem_post(sem_t *sem);
EPS_POSIX_CLOCK_GETTIME	int clock_gettime(clockid_t clock_id, struct timespec *tp);
EPS_POSIX_GETTIMEOFDAY	int gettimeofday(struct timeval * tp, void * tzp);
EPS_POSIX_NANOSLEEP	int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);

Thread priorities

These defines describe thread priorities:

Table 2- 2 Thread priorities

EPS_POSIX_THREAD_PRIORITY_NORMAL	see samples
EPS_POSIX_THREAD_PRIORITY_ABOVE_NORMAL	see samples
EPS_POSIX_THREAD_PRIORITY_HIGH	see samples
EPS_POSIX_THREAD_PRIORITY_HIGH_PERFORMANCE	see samples

Samples

Linux sample

\src\source\pnboards\eps\epssys\etc\posix\eps_posix_cfg_linux.h.

Windows sample

\src\source\pnboards\eps\epssys\etc\posix\eps_posix_cfg_windows.h.

\src\source\pnboards\eps\epssys\etc\windows\posix2win.c

2.2 Porting instructions

Mandatory action

1. Set the preprocessor macro EPS_RTOS_CFG_INCLUDE_H=<posix_macro_header.h>.

Optional

1. Create a new file <posix_macro_header.h> with the macro definitions.
   You may use the sample files "eps_posix_cfg_linux.h" or "eps_posix_cfg_windows.h".

EPS PLF (hardware abstraction) 3

3.1 EPS PLF overview

Platform independence of PN Driver

The PN Driver may run on different platforms. With PN Driver V1.1, the supported sample platforms are Windows 7 32/64 bit and Linux Debian on an x86 architecture.

The platform independence of the PN Driver implementation is realized by the EPS platform abstraction (EPS PLF). Platform-specific calls are performed through the EPS by calling EPS_PLF_XXX output functions. The function declarations are done in "\src\source\pnboards\eps\epssys\src\eps_plf.h".

Sample implementations

The following figure shows the sample implementations for PN Driver V1.1. You may add another implementation, shown in the "Custom (embedded) system" box.

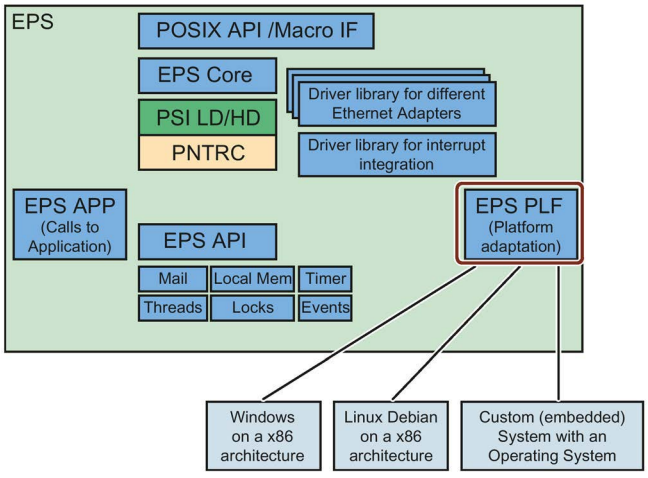


Figure 3-1 EPS PLF overview

3.2 Platform preprocessor defines

Defines

The symbol EPS_PLF has to be set to a value defined in "\src\source\pnboards\eps\epssys\src\eps_plf_types.h".

Note

These defines are reserved and must not be changed.

```
#define EPS_PLF_WINDOWS_X86 6 // Windows on IntelX86
#define EPS_PLF_LINUX_X86 7 // Linux on IntelX86 (PN Driver)
```

- The configuration settings have to be done in this file:

  \src\source\pnd\src\cfg\<config_file.h>

- To make usage of this file, set the name of the file as precompile setting:

  EPS_PSI_CFG_PLATFORM_H=<config_file.h>

Sample Linux Debian

\src\source\pnd\src\cfg\pnd_psi_cfg_plf_linux_intel_interniche.h

EPS_PSI_CFG_PLATFORM_H=pnd_psi_cfg_plf_linux_intel_interniche.h

Sample Windows 7

\src\source\pnd\src\cfg\pnd_psi_cfg_plf_windows_wpcap_interniche.h

EPS_PSI_CFG_PLATFORM_H=pnd_psi_cfg_plf_windows_wpcap_interniche.h

Toolchain settings

TOOL_CHAIN_MICROSOFT selects structure packing implementation for Microsoft compilers.
See \src\source\pnd\src\cfg\pnd_psi_cfg_plf_windows_wpcap_interniche.h

3.3 EPS_PLF_REGISTER_SHM_IF

Function syntax

LSA_VOID
EPS_PLF_REGISTER_SHM_IF
(LSA_VOID* pShmIf)

Description

- This is a function prototype. This function is optional.
- This function is called at "eps_init()".
- Registers the shared memory interface in the EPS. This is required if the EPS runs in a distributed architecture.

Parameter data type	Parameter name	In/Out	Meaning
LSA_VOID*	pShmIf	In	Pointer to IF configuration

3.4 EPS_PLF_DISABLE_INTERRUPTS

Function syntax

LSA_VOID
EPS_PLF_DISABLE_INTERRUPTS
(LSA_VOID)

Description

- This is a function prototype. This function is mandatory.
- Disables all interrupts, including the scheduler interrupt.
- The EPS uses this function to implement an interrupt lock.

3.5 EPS_PLF_ENABLE_INTERRUPTS

Function syntax

LSA_VOID
EPS_PLF_ENABLE_INTERRUPTS
(LSA_VOID)

Description

- This is a function prototype. This function is mandatory.
- Enables all interrupts, including the scheduler interrupt.
- The EPS uses this function to implement an interrupt lock.

3.6 EPS_PLF_EXCHANGE_LONG

Function syntax

long
EPS_PLF_EXCHANGE_LONG
(long volatile *pAddr, long lVal)

Description

- This is a function prototype. This function is mandatory.
- This function is used to set a 32-bit variable to the specified value as an atomic operation.

Parameter data type	Parameter name	In/Out	Meaning
long volatile	pAddr	In/Out	A pointer to the value to be exchanged. The function sets this variable to lVal and returns its prior value.
long	lVal	In/Out	The value to be exchanged with the value pointed to by pAddr. In response the function returns the initial value of the pAddr parameter.
Return value	long		Swapped value

3.7 EPS_PLF_PCI_MAP_MEMORY

Function syntax

LSA_BOOL
EPS_PLF_PCI_MAP_MEMORY
(LSA_UINT8** pBase, LSA_UINT32 uBase, LSA_UINT32 uSize)

Description

- This function maps memory from the PCI tree to the UserSpace.

Parameter data type	Parameter name	In/Out	Meaning
LSA_UINT8**	pBase	Out	Pointer to address where the mapping will take place
LSA_UINT32	uBase	In	LocalAddress to memory which will be mapped
LSA_UINT32	uSize	In	Size of mapped memory
Return value	LSA_BOOL		LSA_TRUE or LSA_FALSE if no memory was mapped.

3.8 EPS_PLF_PCI_GET_DEVICES

Function syntax

LSA_VOID

EPS_PLF_PCI_GET_DEVICES

(LSA_UINT16 uVendorId, LSA_UINT16 uDeviceId, EPS_PLF_PCI_CONFIG_SPACE_PTR_TYPE pConfig-SpaceOut, LSA_UINT16 uMaxDevice, LSA_UINT16 *pFoundDevices)

Description

- This function is used to search for all PCI network adapters that have the given PCI vendor ID and PCI device ID.

Parameter data type	Parameter name	In/Out	Meaning
LSA_UINT16	uVendorId	In	PCI vendor ID
LSA_UINT16	uDeviceId	In	PCI device ID LocalAddress to memory, which will be mapped
EPS_PLF_PCI_CONFIG_SPACE_PTR_TYPE	pConfigSpaceOut	In/Out	Array of config space parameters with size of uMaxDevice. Only the following values need to be filled:
LSA_UINT16	uVendorId	In	Vendor ID
LSA_UINT16	uDeviceId	In	Device ID
LSA_UINT16	uBusNr	In	PCI location - bus
LSA_UINT16	uDeviceNr	In	PCI location – device number
LSA_UINT16	uFunctionNr	In	PCI location – function number
LSA_UINT8*	pBase0	In	Value of BAR0. Lowest four bits must be 0.
LSA_UINT32	uSize0	In	Size of BAR0
struct info			
LSA_BOOL	bValid	Out	LSA_TRUE – Config is valid
LSA_BOOL	bMSISupport	Out	LSA_TRUE – Message-Signaled Interrupts (MSI) supported, see PCI specification
LSA_BOOL	bMSIXSupport	Out	LSA_TRUE – MSI-X supported, see PCI specification
LSA_UINT32	Status	Out	Status from configspace
LSA_UINT32	Command	Out	Command to trigger an interrupt in the Advanced Programmable Interrupt Controller (APIC)
LSA_UINT32	IOApicLine	Out	Line in the APIC

Parameter data type	Parameter name	In/Out	Meaning
struct Bar			
LSA_UINT32	Bar	Out	Base Address Register ID
LSA_UINT32	Barsize	Out	Size of the BAR
LSA_UINT32	BarMaskedVal	Out	Masked value of the BAR
LSA_UINT16	uMaxDevice	In	Number of found devices

3.9 EPS_PLF_PCI_ENA_INTERRUPT

Function syntax

LSA_VOID
EPS_PLF_PCI_ENA_INTERRUPT
(EPS_PLF_PCI_CONFIG_SPACE_PTR_TYPE pConfig-
Space, EPS_PLF_PCI_ISR_CBF_PTR_TYPE pCbf);

Description

- This function is used to enable the interrupt of the given PCI device and to register the interrupt service routine.

Parameter data type	Parameter name	In/Out	Meaning
EPS_PLF_PCI_CONFIG_SPACE_PTR_TYPE	pConfigSpace	In	See description in EPS_PLF_PCI_GET_DEVICES
EPS_PLF_PCI_ISR_CBF_PTR_TYPE	pCbf	In	Container structure for pCbf, uParam and pArgs
EPS_PLF_PCI_ISR_CBF	pCbf	In	Callback function that is called when an interrupt occurs.
LSA_UINT32	uParam	In	Parameter passed to the callback function
LSA_VOID*	pArgs	In	Parameter passed to the callback function

3.10 EPS_PLF_PCI_DIA_INTERRUPT

Function syntax

LSA_VOID
EPS_PLF_PCI_DIA_INTERRUPT
(EPS_PLF_PCI_CONFIG_SPACE_PTR_TYPE pConfig-
Space, EPS_PLF_PCI_ISR_CBF_PTR_TYPE pCbf);

Description

- This function is used to disable the interrupt of the given PCI device.

Parameter data type	Parameter name	In/Out	Meaning
EPS_PLF_PCI_CONFIG_SPACE_PTR_TYPE	pConfigSpace	In	See description in EPS_PLF_PCI_GET_DEVICES (Page 19).

3.11 EPS_PLF_PCI_READ_BYTE

Function syntax

LSA_VOID
EPS_PLF_PCI_READ_BYTE
(LSA_UINT32 uBusNum, LSA_UINT32 uDeviceNum,
LSA_UINT32 uFuncNum, LSA_UINT32 uOffset, LSA_UINT8*
uVal);

Description

- This is a function prototype. This function is optional for PN Driver.
- This function is used to read a byte from the PCI address space.

Parameter data type	Parameter name	In/Out	Meaning
LSA_UINT32	uBusNum	In	PCI location - bus
LSA_UINT32	uDeviceNum	In	PCI location – device number
LSA_UINT32	uFuncNum	In	PCI location – function number
LSA_UINT32	uOffset	In	Offset from the base address of the PCI device
LSA_UINT8*	uVal	Out	Value from this address

3.12 EPS_PLF_PCI_READ_DOUBLE_WORD

Function syntax

```
LSA_VOID
EPS_PLF_PCI_READ_DOUBLE_WORD
(LSA_UINT32 uBusNum, LSA_UINT32 uDeviceNum,
LSA_UINT32 uFuncNum, LSA_UINT32 uOffset,
LSA_UINT32* uVal);
```

Description

- This is a function prototype. This function is optional for PN Driver.
- This function is used to read a double word from the PCI config space of a device.

Parameter data type	Parameter name	In/Out	Meaning
LSA_UINT32	uBusNum	In	PCI location - bus
LSA_UINT32	uDeviceNum	In	PCI location – device number
LSA_UINT32	uFuncNum	In	PCI location – function number
LSA_UINT32	uOffset	In	Offset from the base address of the PCI device
LSA_UINT32*	uVal	Out	Value from this address

3.13 EPS_PLF_PCI_TRANSLATE_PCI_TO_LOCAL_ADDR

Function syntax

```
LSA_UINT8*
EPS_PLF_PCI_TRANSLATE_PCI_TO_LOCAL_ADDR
(LSA_UINT8* uPciAddress);
```

Description

- This is a function prototype. This function is optional.
- This function is used to read a double word from the PCI config space of a device.

Parameter data type	Parameter name	In/Out	Meaning
LSA_UINT8*	uPciAddress	In	Address within the PCI address space
Return value	Translated address (user space address)		

3.14 Porting instructions

You may want to port the PN Driver to an embedded system. In order to do that, the EPS PLF output functions listed in this section have to be implemented according to the target operating system/target platform.

Linux @ x86 sample

\src\source\pnboards\eps\epssys\etc\plf\eps_plf_linux.c

Windows @ x86 sample

\src\source\pnboards\eps\epssys\etc\plf\eps_plf_windows.c

EPS APP (application abstraction) 4

4.1 EPS APP overview

Implementation of functions

The EPS requires the application to implement some functions since it cannot fully decide how the system is structured.

The following figure shows the PN Driver configuration for EPS APP.

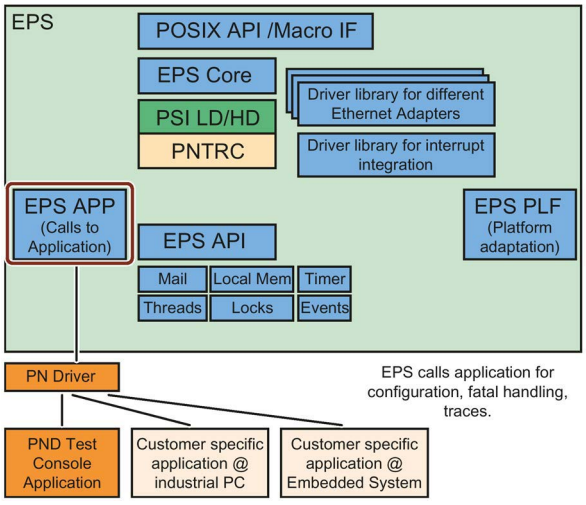


Figure 4-1 EPS APP - application abstraction sample

The orange boxes show the PN Driver and the PN Driver test console application. The light boxes show two sample configurations, one being a customer-specific application on an industrial PC, the other one a configuration for an embedded system.

You can change the behavior of the PN Driver by changing the output functions described in this section.

Note

The PN Driver interacts with the EPS APP interface, the customer-specific application may not interact directly with the EPS.

4.2 Function EPS_APP_INIT

4.2.1 EPS_APP_INIT

Function syntax

LSA_VOID
EPS_APP_INIT
(LSA_VOID_PTR_TYPE hSys, EPS_HW_INFO_PTR_TYPE pEpsHw)

Description

- This function is called by the EPS at "eps_init".
- The following has to be done in this function:
  - Provide local memory
  - Provide local fast memory
  - Install "eps_shm_if" driver
  - Install additional drivers (e. g. for interrupting at real time operating system, exception handling)
  - Setting initial log levels for tracing
- The following "eps_shm_if" drivers are available:
  - "eps_noshm" (Windows, Linux)

In order to install a driver, call the XXX_install function, e. g. "eps_pn_soc1_drv_install".

Parameter data type	Parameter name	In/Out	Meaning
LSA_VOID_PTR_TYPE	**hSys**	**In**	
EPS_HW_INFO_PTR_TYPE	**pEpsHw**	**Out**	
EPS_MEM_REGION_TYPE	LocalMemPool	Out	Local memory, used for malloc operations. **Note** that the PROFINET stack as well as the PN Driver uses this memory for malloc/free operations. Communication memory for acyclical communication (so-called Non-Real-Time (NRT) memory) as well as the Process Image (PI) memory for cyclical communication are provided by the PN device driver, see section EPS_PNDEV_OPEN_FCT (Page 36).
EPS_MEM_REGION_TYPE	FastMemPool	Out	Local fast memory, used for IO data.

4.2.2 How the EPS calls EPS_APP_INIT()

Interaction between EPS and PN Driver

At the startup of the PN Driver, the PN Driver calls the EPS at "SERV_CP_init".

The EPS performs a callback to the PN Driver by calling "EPS_APP_INIT()".

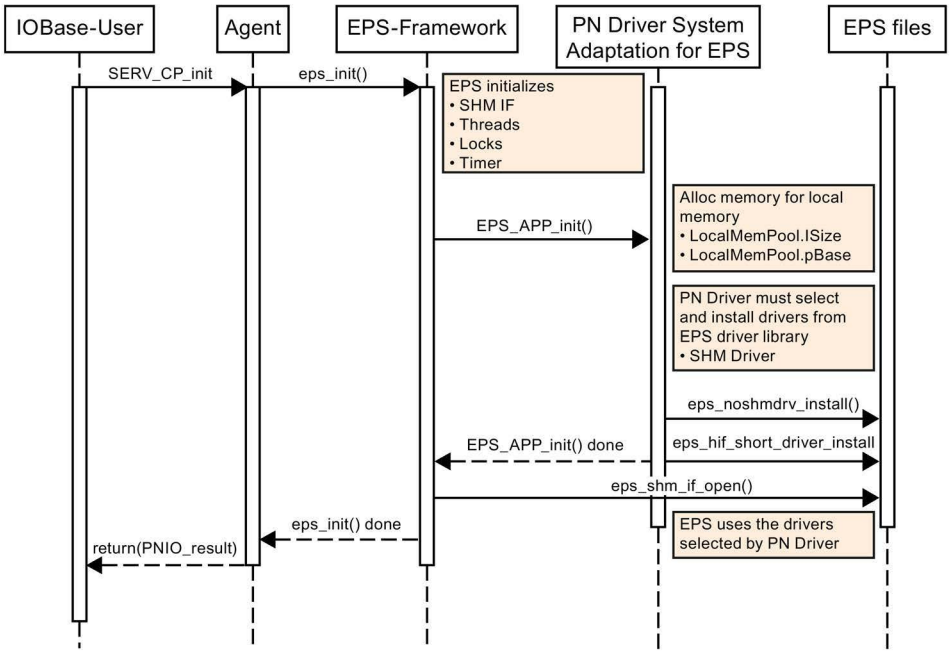


Figure 4-2 Interaction between EPS and PN Driver at "SERV_CP_init"

4.2.3 Porting instructions

Using another EPS driver

1. If you want to use another driver provided by the EPS, replace the call of the eps_XXX_install() function by another driver listed in the description of "EPS_APP_INIT()".

Using a custom driver

1. If you want to use a custom driver that is not included in the PN Driver delivery, this driver has to be implemented and registered in "EPS_APP_INIT()" using the eps_XXX_install() functions.
   See section EPS driver concept (Ethernet adapters abstraction) (Page 32) for more detailed information.

4.3 EPS_APP_UNDO_INIT

Function syntax

LSA_VOID
EPS_APP_UNDO_INIT
(LSA_VOID_PTR_TYPE hSys, EPS_HW_INFO_PTR_TYPE pEpsHw)

Description

- This function is called by the EPS at "eps_undo_init".
- Here, the initialization (e. g. allocating memory for LocalMemPool) can be undone.

Parameter data type	Parameter name	In/Out	Meaning
LSA_VOID_PTR_TYPE	**hSys**	**In**	
EPS_HW_INFO_PTR_TYPE	**pEpsHw**	**In**	
EPS_MEM_REGION_TYPE	LocalMemPool	Out	Local memory, used for malloc operations
EPS_MEM_REGION_TYPE	FastMemPool	Out	Local fast memory, used for IO data.

4.4 Function EPS_APP_INSTALL_DRV_OPEN

4.4.1 EPS_APP_INSTALL_DRV_OPEN

Function syntax

LSA_VOID
EPS_APP_INSTALL_DRV_OPEN
(PSI_LD_RUNS_ON_TYPE ldRunsOnType)

Description

- The user application has to implement this function.
- The following has to be done in this function:
  - Install "eps_pn_dev" drivers
- The following "eps_pn_dev" drivers are available:
  - "eps_pndevdrv" (Windows, Linux)
  - "eps_winpcapdrv" (Windows only)

  In order to install a driver, call the XXX_install function, e. g. "eps_pndevdrv_install".

Parameter data type	Parameter name	In/Out	Meaning
PSI_LD_RUNS_ON_TYPE	**ldRunsOnType**	**In**	This parameter was passed to the EPS at "eps_open". The EPS passes this parameter back to the application. In this way, the application can install different drivers only when necessary.

4.4.2 How the EPS calls EPS_APP_INSTALL_DRV_OPEN()

Interaction between EPS and PN Driver

At the startup of the PN Driver, the PN Driver calls the EPS at "SERV_CP_startup".

The EPS performs a callback to the PN Driver by calling "EPS_APP_INSTALL_DRV_OPEN()".

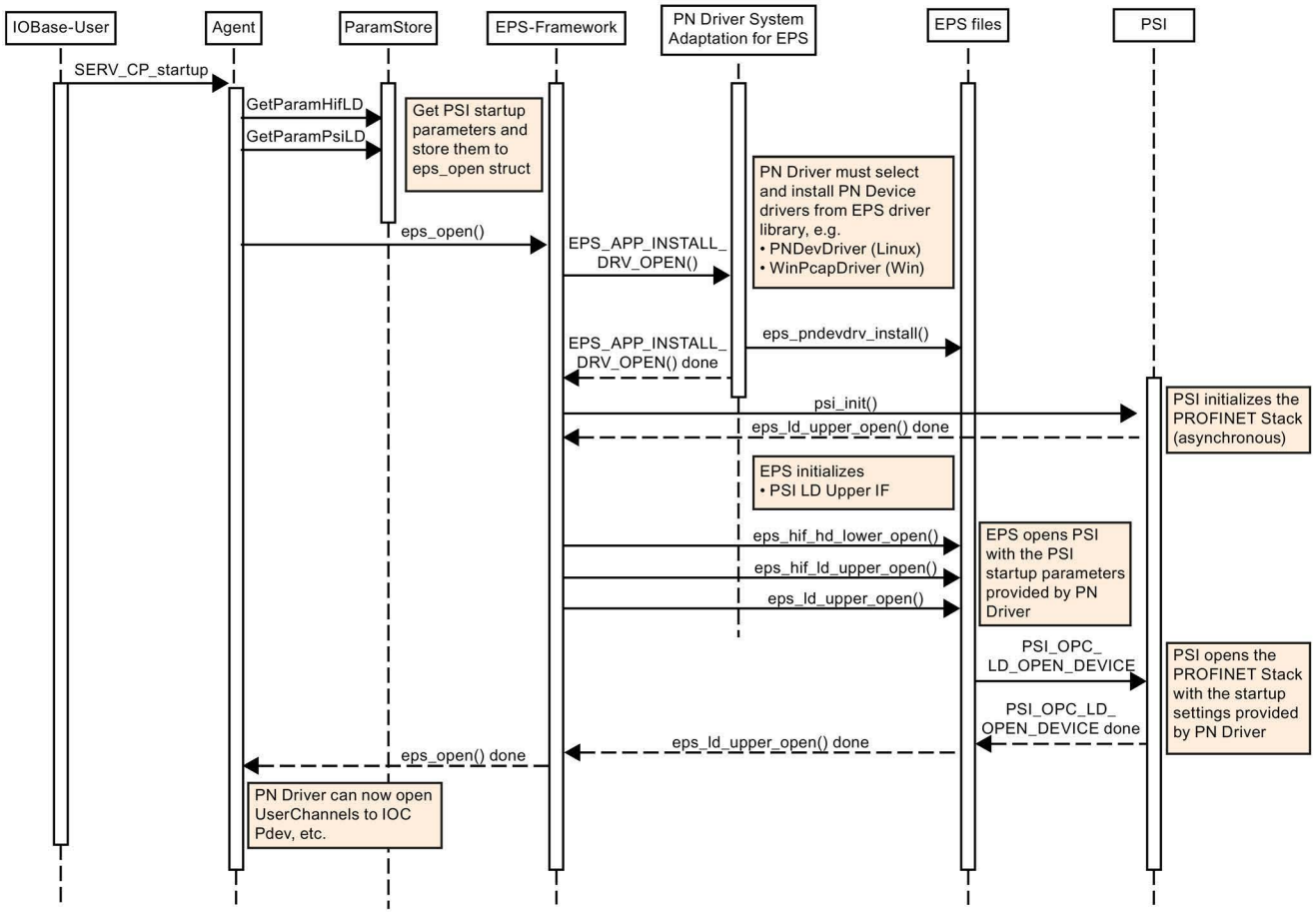


Figure 4-3 Interaction of SERV_CP_startup and EPS_APP_INSTALL_DRV_OPEN

After the call of "EPS_APP_INSTALL_DRV_OPEN()", the EPS uses the installed PN device drivers in order to access Ethernet adapters. Each registered PN device driver is asked by the EPS whether the Ethernet adapter at the given PCI location/MAC address can be opened or not. The first driver that supports the adapter is used, i. e. the order in which the drivers are installed does matter. See section PN device driver (Page 34).

Example

"eps_winpcapdrv" and CustomDriverXYZ are installed within "EPS_APP_INSTALL_DRV_OPEN". Both can open the Ethernet adapter with the MAC address <mac_address>. EPS asks "eps_winpcapdrv" to open the board, which succeeds. Therefore, CustomDriverXYZ will not be asked to open the board.

4.4.3 Porting instructions

Using another EPS driver

1. If you want to use another driver provided by the EPS, replace the call of the eps_XXX_install() function by another driver listed in the description of "EPS_APP_INSTALL_DRV_OPEN()".

Using a custom driver

1. If you want to use a custom driver that is not included in the PN Driver delivery, this driver has to be implemented and registered in "EPS_APP_INSTALL_DRV_OPEN()" using the eps_XXX_install() functions.
   See section PN device driver (Page 34) for more detailed information.

4.5 EPS_APP_KILL_EPS

Function syntax

LSA_VOID
EPS_APP_KILL_EPS
(LSA_VOID)

Description

- Stops EPS system because of a fatal error.

Note

The EPS and the PROFINET stack expect that this function call will **never return**. **The behavior is undefined if this function returns**. This function must either terminate the application or implement an endless loop.

Parameter data type	Parameter name	In/Out	Meaning
LSA_VOID	-	-	-

4.6 EPS_APP_BREAKPOINT

Function syntax

LSA_VOID
EPS_APP_BREAKPOINT
(LSA_VOID)

Description

- Stops at corresponding hardware breakpoint.

Parameter data type	Parameter name	In/Out	Meaning
LSA_VOID	-	-	-

EPS driver concept (Ethernet adapters abstraction) 5

5.1 EPS driver concept overview

Call of interfaces

The EPS is portable to different systems which can differ in their respective behavior. The architecture of the EPS is modular and configurable. Each driver encapsulates a platform- or hardware-specific functionality that the EPS can use by applying the requested interface functions. Thus, the EPS calls only interfaces, never the driver itself. There are three interfaces that the EPS uses:

- Upper IF drivers (EPS PN Driver interface, see section PN device driver (Page 34))
- Lower IF drivers (EPS SHM IF, for PN Driver only the NoSharedMemory driver)
- Platform-specific interrupt service routine drivers (ISR driver)

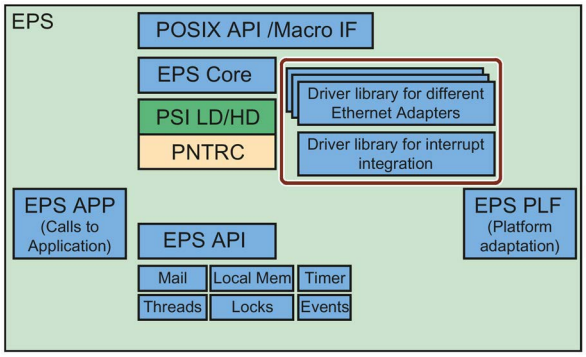


Figure 5-1 EPS driver concept

Each driver has to provide an "install()"- and "uninstall()"-function. Within this "install", the driver has to:

- Map the functions required by the interface to a real implementation
- Call the XXX_register() function provided by the EPS core

Sample

- The "eps_noshm" driver implements the "eps_shm_if". For the PN Driver test console application, it is installed and registered at "EPS_APP_INIT()".
- The sources for the driver are located in this path: \src\source\pnboards\eps\epssys\src\eps_noshmdrv.c
- The driver implements the install function "eps_noshmdrv_install()":

```
LSA_VOID eps_noshmdrv_install (EPS_NOSHMDRV_INSTALL_ARGS_PTR_TYPE pInstallArgs)
{
 EPS_SHM_IF_TYPE sNoShmDrvIf;
 g_pEpsNoShmDrv = &g_EpsNoShmDrv;
 eps_memset(g_pEpsNoShmDrv, 0, sizeof(*g_pEpsNoShmDrv));
 eps_noshmdrv_init_critical_section();
 g_pEpsNoShmDrv->sInstallArgs = *pInstallArgs;

//Note: Function pointer mapping takes place:
 g_pEpsNoShmDrv->sHw.AppReady = eps_noshmdrv_app_ready;
 g_pEpsNoShmDrv->sHw.AppShutdown = eps_noshmdrv_app_shutdown;
 g_pEpsNoShmDrv->sHw.FatalError = eps_noshmdrv_sig_fatal;
 g_pEpsNoShmDrv->sHw.Close = eps_noshmdrv_close;

#if ( PSI_CFG_USE_PNTRC == 1 )
 g_pEpsNoShmDrv->sHw.TraceBufferFull = eps_noshmdrv_trace_buffer_full_cbf;
 g_pEpsNoShmDrv->sHw.sPntrcParam = pInstallArgs->sPntrcParam;
#endif

 g_pEpsNoShmDrv->bInit = LSA_TRUE;

 sNoShmDrvIf.open = eps_noshmdrv_open;
 sNoShmDrvIf.close = eps_noshmdrv_close;
 sNoShmDrvIf.uninstall = eps_noshmdrv_uninstall;
 sNoShmDrvIf.update_device_list = eps_noshmdrv_update_device_list;

//Note: The driver is registered in the EPS by calling "eps_shm_if_register()":
 eps_shm_if_register(&sNoShmDrvIf);
}
```

5.2 PN device driver

5.2.1 PN device driver overview

PN device driver interfaces

The PN device driver interfaces provide access to the hardware registers and the memory of an Ethernet adapter. For PN Driver V1.1, there are two sample implementations as shown in the following figure.

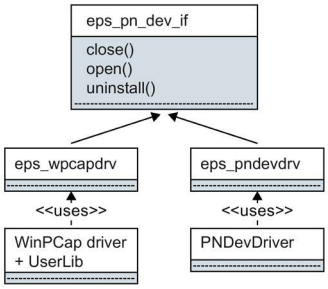


Figure 5-2 PN device driver interface overview

Sample implementations

Table 5- 1 Sample implementations for PN device driver interfaces

Name	Description	Path
eps_wpcapdrv	Windows only, wrapper for WinPcap	\src\source\pnboards\eps\epssys\ src\eps_wpcapdrv.c
eps_pndevdrv	Linux only, driver for Intel boards Springville and Hartwell with extended functionality. Comes with the kernel-mode driver PNDevDriver	\src\source\pnboards\eps\epssys\ src\eps_pndevdrv.c

To register the driver, the structure must be provided to "eps_pndev_if_register" as detailed in section EPS driver concept overview (Page 32).

5.2.2 eps_pndev_if_register_device

Function syntax

LSA_VOID
eps_pndev_if_register_device
(EPS_PNDEV_IF_DEV_INFO_PTR_TYPE pDev)

Description

- This function is not part of the PN device driver interface. It is an auxiliary function to register the PN Driver in the EPS and is listed here for that reason.
- Registers a PN board in the registered devices storage.
- A module that implements the PN device driver IF must call this function in order to register the driver in the EPS.

Parameter data type	Parameter name	In/Out	Meaning
EPS_PNDEV_IF_DEV_INFO_PTR_TYPE	pDev		Wrapper structure
EPS_PNDEV_OPEN_FCT	open	In	See section EPS_PNDEV_OPEN_FCT (Page 36)
EPS_PNDEV_CLOSE_FCT	close	In	See section EPS_PNDEV_CLOSE_FCT (Page 38)
EPS_PNDEV_UNINSTALL_FCT	uninstall	In	See section EPS_PNDEV_UNINSTALL_FCT (Page 39)

5.2.3 EPS_PNDEV_OPEN_FCT

Function syntax

LSA_UINT16
(*EPS_PNDEV_OPEN_FCT)
(EPS_PNDEV_LOCATION_PTR_TYPE pLocation,
EPS_PNDEV_OPEN_OPTION_PTR_TYPE pOption, struct
eps_pndev_hw_tag** ppHwInstOut, LSA_UINT16 hd_id);

Description

- This is a function prototype. This function is called at "eps_open".
- EPS calls this function to open an Ethernet board at the given location (PCI bus x, device y, function z). This is done during "SERV_CP_startup".
- If this function is called, the function pointers listed in the table below have to be linked to an implemention.

Parameter data type	Parameter name	In/Out	Meaning
EPS_PNDEV_LOCATION_PTR_TYPE	pLocation	In	Location specifier
EPS_PNDEV_OPEN_OPTION_PTR_TYPE	pOption	In	Open options
struct eps_pndev_hw_tag**	ppHwInstOut	Out	Wrapper structure
LSA_VOID*	hDevice	Out	Void pointer to driver-specific structures. This structure is passed every time the driver is called. Sample eps_pndevdrv: Here, a pointer to a struct eps_pndevdrv_board_tag is passed.
EPS_BOARD_INFO_TYPE			Wrapper structure
LSA_UINT16	edd_type		Reserved
LSA_UINT16	nr_of_ports		Number of physical ports
LSA_SYS_PTR_TYPE	hd_sys_handle		System handle of the hardware device
LSA_SYS_PTR_TYPE[]	if_sys_handle		System handle of the logical device
EPS_MAC_TYPE[]	if_mac		Interface MAC addresses
EPS_MAC_TYPE[]	port_mac		Port MAC addresses
EPS_PORT_MEDIA_TYPE[]	port_media_type		Enumeration of the media type
EPS_PORT_MAP_TYPE	port_map		Mapping from the physical port to the logical port
EPS_BOARD_MEM_TYPE	nrt_mem		Memory for the non-real-time memory used for acyclic communication

Parameter data type	Parameter name	In/Out	Meaning
EPS_BOARD_MEM_TYPE	pi_mem		Memory for the process image used for cyclic communication
EPS_BOARD_MEM_TYPE	hif_mem		Memory for serialization. Only used if the EPS runs in a distributed architecture.
EPS_BOARD_EDDI_TYPE	eddi		Reserved
EPS_BOARD_EDDP_TYPE	eddp		Reserved
EPS_BOARD_EDDS_TYPE	edds		Board information for boards with a standard Ethernet controller. PN Driver uses this structure. For more details, see section Integration of the Intel lower layer implementation into a PN device driver (Page 46).
LSA_BOOL	is_valid		Flag to determine if default values shall be used (FALSE) or if the values in this structure shall be used (TRUE)
LSA_VOID_PTR_TYPE (casted to INTEL_LL_HANDLE_TYPE)	ll_handle		LL handle (depends on LL adaptation). See src\source\edds\src\intel\intel_usr.h
LSA_VOID_PTR_TYPE (casted to EDDS_LL_TABLE_TYPE)	ll_function_table		LL function table (depends on LL adaptation) See src\source\edds\src\inc\llif.h. Note that the LL functions are set by the EPS using the Packet32 LL or Intel LL-Package provided by EDDS. See section Integration of the Intel lower layer implementation into a PN device driver (Page 46).
LSA_VOID (casted to INTEL_LL_PARAMETER_TYPE)	ll_params		LL params, see src\source\edds\src\intel\intel_usr.h. Note that the parameters pRegBaseAdd, DeviceID and Disable1000MBitSupport are set by EPS.
LSA_VOID_PTR_TYPE	pRegBaseAdd		Base pointer of BAR0.
LSA_UINT16	DeviceID		Device ID, e. g. for Springville or Hartwell.
LSA_BOOL	Disable1000MBitSupport		Reserved with TRUE.
EPS_PNDEV_ENABLE_ISR_FCT	EnableIsr		See descriptions in this section.
EPS_PNDEV_DISABLE_ISR_FCT	DisableIsr		
EPS_PNDEV_SET_GPIO	SetGpio		
EPS_PNDEV_CLEAR_GPIO	ClearGpio		

Parameter data type	Parameter name	In/Out	Meaning
EPS_PNDEV_TIMER_CTRL_START	TimerCtrlStart		
EPS_PNDEV_TIMER_CTRL_STOP	TimerCtrlStop		
EPS_PNDEV_READ_TRACE_DATA	ReadTraceData		
EPS_PNDEV_WRITE_TRACE_DATA	WriteTraceData		
EPS_PNDEV_SAVE_DUMP	SaveDump		
EPS_PNDEV_ASIC_TYPE	asic_type		Enumeration of the supported ASICs
EPS_PNDEV_BOARD_TYPE	board_type		Enumeration of the supported boards
Return value	**LSA_UINT16**		EPS_PNDEV_RET_ERR EPS_PNDEV_RET_OK EPS_PNDEV_RET_DEV_NOT_FOUND EPS_PNDEV_RET_DEV_OPEN_FAILED EPS_PNDEV_RET_UNSUPPORTED

Porting instruction

If a new ASIC or board type is supported, extend the enumerations "EPS_PNDEV_ASIC_TYPE" and "EPS_PNDEV_BOARD_TYPE".

5.2.4 EPS_PNDEV_CLOSE_FCT

Function syntax

LSA_UINT16
(*EPS_PNDEV_CLOSE_FCT)
(struct eps_pndev_hw_tag* pHwInstIn);

Description

- This is a function prototype. This function is called at "eps_open".
- EPS calls this function to close an Ethernet board at the given location (PCI bus x, device y, function z). This is done during "SERV_CP_shutdown".

Parameter data type	Parameter name	In/Out	Meaning
struct eps_pndev_hw_tag*	**pHwInstIn**	In	Wrapper structure, see description in section EPS_PNDEV_OPEN_FCT (Page 36).

5.2.5 EPS_PNDEV_UNINSTALL_FCT

Function syntax

```
LSA_VOID
(*EPS_PNDEV_UNINSTALL_FCT) (LSA_VOID);
```

Description

- This is a function prototype. This function is called at "eps_close".

5.2.6 EPS_PNDEV_ENABLE_ISR_FCT

Function syntax

```
LSA_UINT16
 (*EPS_PNDEV_ENABLE_ISR_FCT)
(struct eps_pndev_hw_tag* pHwInstIn,
EPS_PNDEV_INTERRUPT_DESC_PTR_TYPE pInterrupt,
EPS_PNDEV_CALLBACK_PTR_TYPE pCbf);
```

Description

- This is a function prototype.
- Enables the board interrupt or starts a polling thread to poll interrupt registers.

Parameter data type	Parameter name	In/Out	Meaning
struct eps_pndev_hw_tag*	**pHwInstIn**	In	Wrapper structure, see description in section EPS_PNDEV_OPEN_FCT (Page 36)
EPS_PNDEV_INTERRUPT_DESC_PTR_TYPE	pInterrupt	In	Enumeration of the interrupt type
EPS_PNDEV_CALLBACK_PTR_TYPE	pCbf	In	Wrapper structure
EPS_PNDEV_ISR_CBF	pCbf	In	Callback function that is called when an interrupt occurs or the polling timer expires.
LSA_UINT32	uParam	In	Parameter passed to the callback function

Parameter data type	Parameter name	In/Out	Meaning
LSA_VOID*	pArgs	In	Parameter passed to the callback function
Return value	**LSA_UINT16**		EPS_PNDEV_RET_ERR EPS_PNDEV_RET_OK EPS_PNDEV_RET_DEV_NOT_FOUND EPS_PNDEV_RET_DEV_OPEN_FAILED EPS_PNDEV_RET_UNSUPPORTED

5.2.7 EPS_PNDEV_DISABLE_ISR_FCT

Function syntax

LSA_UINT16
(*EPS_PNDEV_DISABLE_ISR_FCT)
(struct eps_pndev_hw_tag* pHwInstIn,
EPS_PNDEV_INTERRUPT_DESC_PTR_TYPE pInterrupt);

Description

- This is a function prototype.
- Disables the board interrupt or stops a polling thread to poll interrupt registers.

Parameter data type	Parameter name	In/Out	Meaning
struct eps_pndev_hw_tag*	**pHwInstIn**	In	Wrapper structure, see description in section EPS_PNDEV_OPEN_FCT (Page 36)
EPS_PNDEV_INTERRUPT_DESC_PTR_TYPE	pInterrupt	In	Enumeration of the interrupt type
Return value	**LSA_UINT16**		EPS_PNDEV_RET_ERR EPS_PNDEV_RET_OK EPS_PNDEV_RET_DEV_NOT_FOUND EPS_PNDEV_RET_DEV_OPEN_FAILED EPS_PNDEV_RET_UNSUPPORTED

5.2.8 EPS_PNDEV_TIMER_CTRL_START

Function syntax

LSA_UINT16
(*EPS_PNDEV_TIMER_CTRL_START)
(struct eps_pndev_hw_tag* pHwInstIn,
EPS_PNDEV_CALLBACK_PTR_TYPE pCbf);

Description

- This is a function prototype.
- Starts a HW timer of timer-controlled polling thread.

Parameter data type	Parameter name	In/Out	Meaning
struct eps_pndev_hw_tag*	**pHwInstIn**	In	Wrapper structure, see description in section EPS_PNDEV_OPEN_FCT (Page 36)
EPS_PNDEV_CALLBACK_PTR_TYPE	pCbf	In	Wrapper structure
EPS_PNDEV_ISR_CBF	pCbf	In	Callback function that is called when an interrupt occurs or the polling timer expires.
LSA_UINT32	uParam	In	Parameter passed to the callback function
LSA_VOID*	pArgs	In	Parameter passed to the callback function
Return value	**LSA_UINT16**		EPS_PNDEV_RET_ERR EPS_PNDEV_RET_OK EPS_PNDEV_RET_DEV_NOT_FOUND EPS_PNDEV_RET_DEV_OPEN_FAILED EPS_PNDEV_RET_UNSUPPORTED

5.2.9 EPS_PNDEV_TIMER_CTRL_STOP

Function syntax

LSA_UINT16
(*EPS_PNDEV_TIMER_CTRL_STOP)
(struct eps_pndev_hw_tag* pHwInstIn,
EPS_PNDEV_CALLBACK_PTR_TYPE pCbf);

Description

- This is a function prototype. This function is optional.
- Stops a HW timer of timer-controlled polling thread.

Parameter data type	Parameter name	In/Out	Meaning
struct eps_pndev_hw_tag*	**pHwInstIn**	In	Wrapper structure, see description in section EPS_PNDEV_OPEN_FCT (Page 36)
Return value	**LSA_UINT16**		EPS_PNDEV_RET_ERR EPS_PNDEV_RET_OK EPS_PNDEV_RET_DEV_NOT_FOUND EPS_PNDEV_RET_DEV_OPEN_FAILED EPS_PNDEV_RET_UNSUPPORTED

5.2.10 EPS_PNDEV_READ_TRACE_DATA

Function syntax

LSA_UINT16
(*EPS_PNDEV_READ_TRACE_DATA)
(struct eps_pndev_hw_tag* pHwInstIn, LSA_UINT32 offset,
LSA_UINT8* ptr, LSA_UINT32 size);

Description

- This is a function prototype. This function is optional.
- Reads trace data from a shared memory. Used if the EPS runs in a distributed architecture.

Parameter data type	Parameter name	In/Out	Meaning
struct eps_pndev_hw_tag*	**pHwInstIn**	In	Wrapper structure, see description in section EPS_PNDEV_OPEN_FCT (Page 36)
LSA_UINT32	offset	In	Offset in the trace memory (source)
LSA_UINT8*	ptr	In	Pointer to the memory to which the trace data has to be written (destination)
LSA_UINT32	size	In	Length to copy
Return value	**LSA_UINT16**		EPS_PNDEV_RET_ERR EPS_PNDEV_RET_OK EPS_PNDEV_RET_DEV_NOT_FOUND EPS_PNDEV_RET_DEV_OPEN_FAILED EPS_PNDEV_RET_UNSUPPORTED

5.2.11 EPS_PNDEV_WRITE_TRACE_DATA

Function syntax

LSA_UINT16
(*EPS_PNDEV_WRITE_TRACE_DATA)
(struct eps_pndev_hw_tag* pHwInstIn, LSA_UINT32 offset,
LSA_UINT8* ptr, LSA_UINT32 size);

Description

- This is a function prototype. This function is optional.
- Writes configuration data for traces into a shared memory. Used if the EPS runs in a distributed architecture.

Parameter data type	Parameter name	In/Out	Meaning
struct eps_pndev_hw_tag*	**pHwInstIn**	In	Wrapper structure, see description in section EPS_PNDEV_OPEN_FCT (Page 36)
LSA_UINT32	offset	In	Offset in the trace memory (destination)
LSA_UINT8*	ptr	In	Pointer to the memory where the trace data has to be written to (source)
LSA_UINT32	size	In	Length to copy
Return value	**LSA_UINT16**		EPS_PNDEV_RET_ERR EPS_PNDEV_RET_OK EPS_PNDEV_RET_DEV_NOT_FOUND EPS_PNDEV_RET_DEV_OPEN_FAILED EPS_PNDEV_RET_UNSUPPORTED

5.2.12 EPS_PNDEV_SAVE_DUMP

Function syntax

LSA_UINT16
(*EPS_PNDEV_SAVE_DUMP)
(struct eps_pndev_hw_tag* pHwInstIn);

Description

- This is a function prototype. This function is optional.
- The EPS calls this function in case of a fatal error.
- Saves a dump of the hardware registers as well as the memory to a file system.

Parameter data type	Parameter name	In/Out	Meaning
struct eps_pndev_hw_tag*	**pHwInstIn**	In	Wrapper structure, see description in section EPS_PNDEV_OPEN_FCT (Page 36)
Return value	**LSA_UINT16**		EPS_PNDEV_RET_ERR EPS_PNDEV_RET_OK EPS_PNDEV_RET_DEV_NOT_FOUND EPS_PNDEV_RET_DEV_OPEN_FAILED EPS_PNDEV_RET_UNSUPPORTED

5.3 Integration of the Intel lower layer implementation into a PN device driver

EDDS component

The software component **E**thernet **D**evice **D**river for "**S**tandard MAC" (EDDS) is a set of so-called lower layer functions to access the Ethernet adapter. In combination with the hardware addresses provided in "EPS_PNDEV_OPEN_FCT", the EDDS is able to control the Ethernet controller ASIC.

In the PN Driver delivery, two sets of lower layer implementations are provided:

- Intel lower layer for Intel Ethernet controllers
- Packet32 lower layer for Ethernet controller supported by WinPcap

Note

The PN device driver must provide this set of lower layer functions in the structure "EPS_BOARD_EDDS_TYPE".

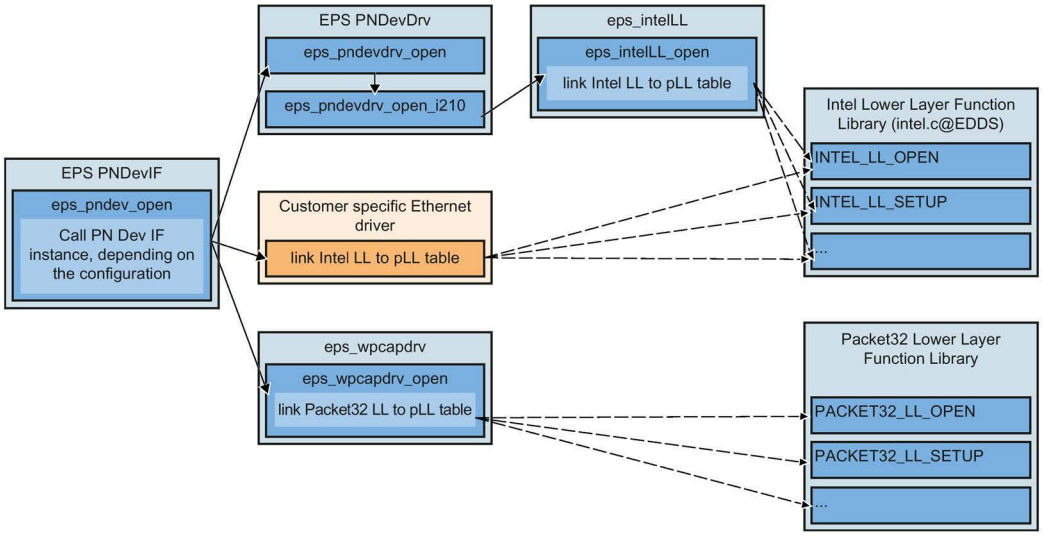


Figure 5-3 Integration of the lower layer tables to a PN device driver

Windows sample

The Windows sample of PN Driver uses WinPcap and the Packet32 lower layer as underlying driver to the network adapter.

In order to compile and run the enclosed Visual Studio project the include path for WinPcap header files and library files needs to be adapted to the local installation of WinPcap.

Linux Debian sample

The Linux Debian sample of PN Driver uses the PNDevDriver and the Intel lower layer as underlying driver to the network adapter.

Source files to be included

Depending on the variant the following source files must be included in the build.

Table 5- 2 Source files to be included in the build

PN device driver	Lower layer	Files
eps_wpcapdrv@PNDriver Windows	Packet32 LL	\src\source\pnboards\eps\epssys\src\eps_wpcapdrv.c \src\source\edds\packet32\packet32_edd.c \src\source\edds\packet32\packet32_inc.h \src\source\edds\packet32\packet32_usr.h Library files Packet.lib wpcap.lib Header files pcap.h
eps_pndevdrv@PNDriver Linux	Intel LL	\src\source\pnboards\eps\epssys\src\eps_pndevdrv.c \src\source\edds\intel\intel_edd.c \src\source\edds\intel\intel_cfg.h \src\source\edds\intel\intel_inc.h \src\source\edds\intel\intel_reg.h \src\source\edds\intel\intel_usr.h

5.4 PN device driver interrupt/event mechanism

5.4.1 Interrupt/event mechanism overview

Handling of interrupts

The software component **E**thernet **D**evice **D**river for "**S**tandard MAC" (EDDS) is responsible for handling send and receive interrupts from the Ethernet adapter. To handle these interrupts, two event modes will be supported:

- Interrupt driven mode
  The EDDS is called every time an interrupt occurs. The interrupt triggers the event to call the EDDS.
- Polling mode
  The EDDS is called in a cyclic manner. A polling thread triggers the event to call the EDDS.

Note

The cyclic communication of the PN Driver relies on this event. If the event does not occur, the EDDS cannot send any NRT or RT frames!

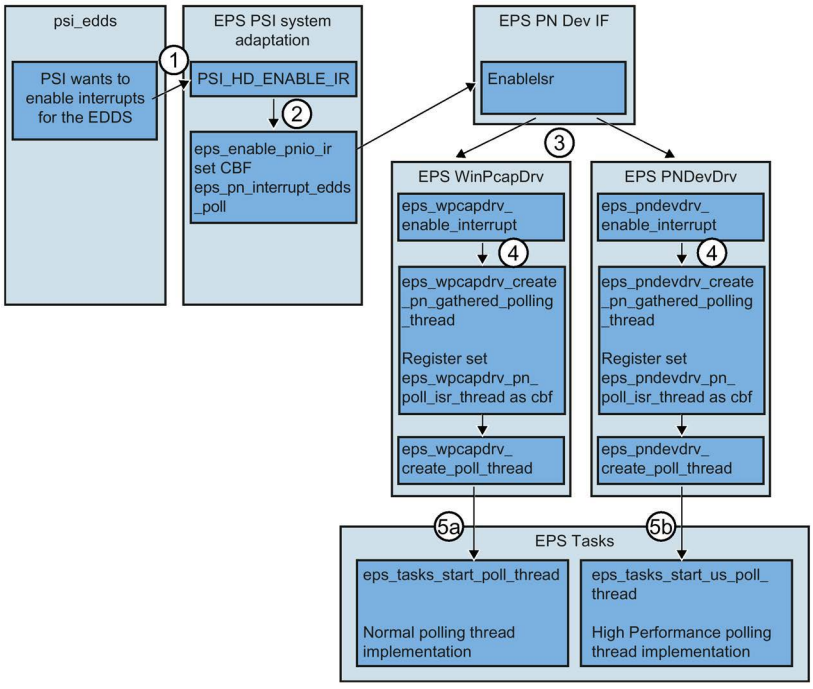
Polling mode

The sample implementation for the PN Driver currently supports only polling mode. For polling mode, the system must fulfill the requirements described in section Polling mode (Page 50).

5.4.2 How the EPS enables the event mechanism

Enabling EDDS polling thread

The following figure shows how the EPS enables the event mechanism.



① PROFINET stack interface (PSI) calls the OUT macro "PSI_HD_ENABLE_IR".

② The OUT macro is implemented by the EPS function "eps_enable_pnio_ir". This function calls the EPS PN Dev IF function EnableIsr.

③ Depending on the EPS PN Dev IF implementation, the "eps_wpcapdrv" or the "eps_pndevdrv" is called.

④ The PN device driver creates a polling thread and sets the callback function to this thread.

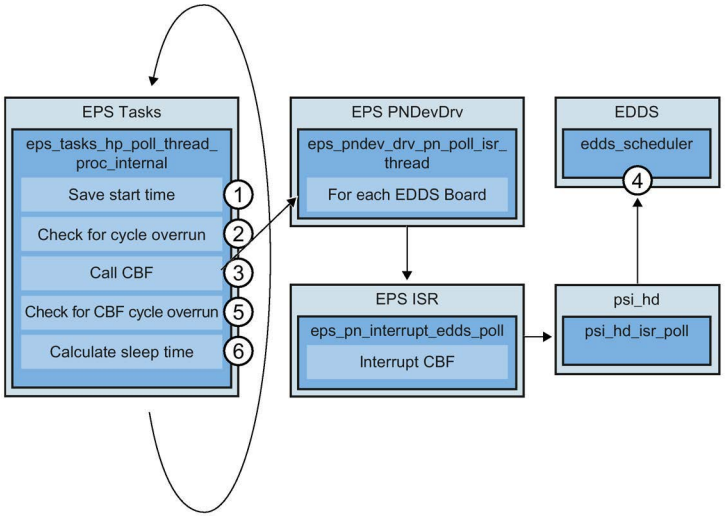
⑤ Creating a polling thread using "eps_tasks.c".

Figure 5-4 Enable EDDS polling thread

5.4.3 Polling mode

Polling mode for EDDS

The following figure shows the sequence for the polling mode.



① Make a time stamp at the beginning of the cycle.
② Check whether a cycle overrun occurred. This happens if the high performance polling thread was not scheduled in time.
③ Call the callback function. The PN device driver is called, this calls EPS ISR which calls PSI. PSI calls EDDS.
④ edds_scheduler is called. The frames are sent/received.
⑤ EPS tasks checks whether the callback function took longer than the scheduling cycle.
⑥ The sleep time until the next cycle is calculated.

Figure 5-5 Polling mode for EDDS

Requirements for polling mode

The sample implementation for the PN Driver currently supports only polling mode. For polling mode, the system must fulfill the requirements described in this section.

A cyclic polling task/thread must be set up. This polling thread must have a **cycle time of 1000 µs** (RT variant) or **32000 µs** (Windows variant). The scheduling of the thread has to be set to **"Run to Completion"** (also called **FIFO scheduling**). "Run to Completion" means that no other thread disrupts the work of the polling thread. If no other interrupts place a load on the system for longer periods, this ensures that the thread is triggered every 1 ms or every 32 ms respectively and that the thread finishes its work before the next trigger occurs. For a Linux system, the so-called preemptive patch must be applied to the kernel to achieve such behavior.

Note

The implementation of the polling mode heavily depends on the scheduling setting of the operating system. Most Linux systems have a default scheduling cycle of 1 millisecond or higher, which makes implementation of the polling mode impossible. The scheduling cycle can be changed in the configuration of the operating system.

Adaptive wait mechanism

To optimize the polling mode, the thread should implement an **adaptive wait mechanism** as well.

For the functionality see the sample in "\src\source\pnboards\eps\epssys\src\eps_tasks.c".

The functions "eps_tasks_start_poll_thread()" or "eps_tasks_start_us_poll_thread()" set up a normal or a high performance (HP) polling thread.

Note

The PN Device Driver "eps_pndevdrv" applies the high performance polling thread, see "\src\source\pnboards\eps\epssys\src\eps_pndevdrv.c".

The priority of the thread determines whether a normal polling thread or a high performance polling thread with adaptive wait mechanism is started.

If the priority "EPS_POSIX_THREAD_PRIORITY_HIGH_PERFORMANCE" is used, then the thread will run as a high performance (HP) polling thread. In all other cases a normal polling thread will run.

All threads use the same callback function "eps_tasks_thread_proc_internal()". This callback function uses a normal polling thread mechanism (e. g. for normal, reoccuring tasks without specific requirements) or a high performance polling thread mechanism described in this section.

The HP polling thread with adaptive wait mechanism is completely implemented in the function "eps_tasks_hp_poll_thread_proc_internal()".

The goal of the adaptive wait mechanism is to call the HP callback function (here "edds_scheduler()") once in each cycle. This cycle should be exactly equidistant (for Linux variant, 1 ms). For the remainder of the cycle "nanosleep()" is called with a sleep time T(sleep). To obtain an equidistant cycle goal, the sleep time is computed at each call adaptively.

Algorithm

1. At the start of HP polling thread CBF "T(Start)" is measured (time stamp).
2. Now it is computed whether the start of the callback function is too early “T(PreDelay)” or too late “T(PostDelay)”.
3. Then the main functionality of HP polling thread runs ("edds_scheduler").
4. At the end of CBF the time "T(Stop)" is measured (time stamp).
5. Now the runtime of the CBF can be computed "T(CBF)".
6. "T(Diff)" is the difference to the start of the last cycle "T(LastRef)".
7. – If the function is still within the cycle, then "T(Sleep)" is computed.
   – In case of cycle overflow a new cycle is started beginning with "T(Stop)".
8. The function calls "nanosleep" ("T(sleep)") to sleep exactly until the next cycle.

The figures below show how the adaptive wait mechanism works. The references in the figures show the variables which realize the times in code.

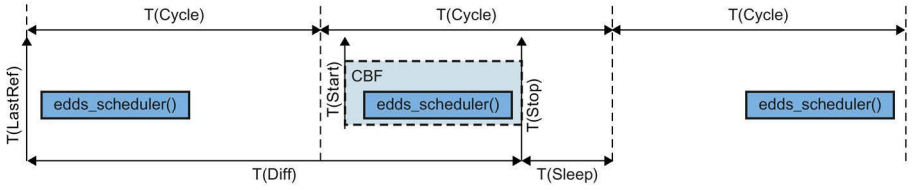
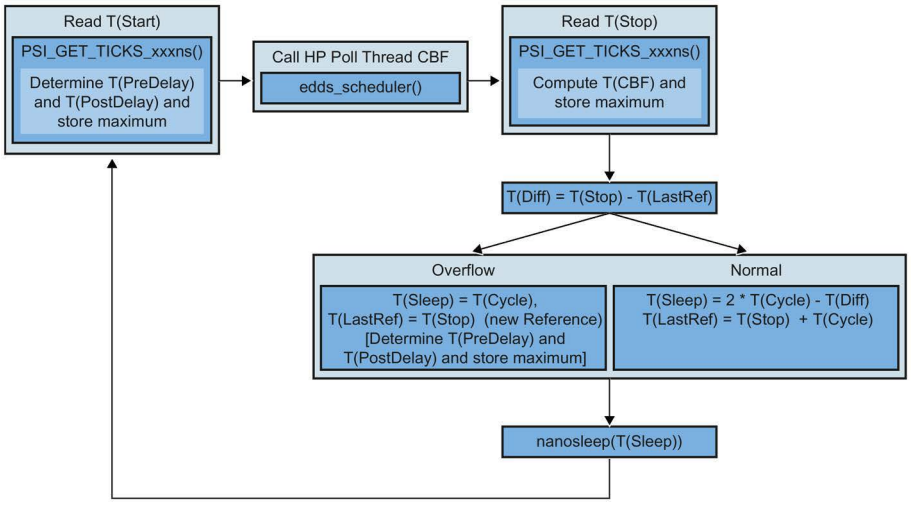


Figure 5-6 Callback function of the polling thread - "eps_tasks_thread_proc_internal()"

Adaptive cyclic high performance polling thread

- The thread "EPS_PNDEVGPISR" will be started as "poll_thread" with highest priority in the Linux OS (for supported Intel boards) and for WPCAP_POLL_THREAD (32 ms).
- The priority of the two timer groups [TIMER_TGROUP0] and [TIMER_TGROUP1] (second highest in Linux OS) will be decreased.
- Timer resolution for Linux must be set to 10 µs.
- The cyclic call of the HP polling thread callback function ("edds_scheduler") will be time-measured and the following sleep time will be adapted to an equidistant cycle.
- All other polling threads remain unchanged.



T(Start)	StartTimeNs
T(Stop)	StopTimeNs
T(Diff)	DiffTimeNs
T(Sleep)	SleepTimeNs
T(LastRef)	LastReferenceTimeNs
T(Cycle)	EPS_CFG_HP_POLL_CYCLE_TIME_NS
T(CBF)	CbfRunTimeNs
T(PreDelay)	StartPreDelayTimeNs
T(PostDelay)	StartPostDelayTimeNs

Figure 5-7 Adaptive cyclic high performance polling thread - "eps_tasks_hp_poll_thread_proc_internal()"

5.5 Porting instructions

Implementing a new driver

1. Set either EDDS_CFG_HW_PACKET32 or EDDS_CFG_INTEL.
2. Implement a PN device driver with all the functions described above in this section.
3. Link the Packet32 lower layer or the Intel lower layer table in the structure EPS_BOARD_EDDS_TYPE.
4. Register/install the driver when EPS_APP_INSTALL_DRV_OPEN() is called.
5. Set LL parameters "pRegBaseAdd", "DeviceID" and "Disable1000MbitSupport" (see section EPS_PNDEV_OPEN_FCT (Page 36)) as well as the lower layer table using Intel LL or Packet32 LL.

Sample

See "eps_wpcapdrv@PNDriver Windows" and "eps_pndevdrv@PNDriver Linux" in section Integration of the Intel lower layer implementation into a PN device driver (Page 46).

Note

Ethernet adapters require **physically contiguous host memory in the host RAM. "**Physically contiguous" means that the memory is accessible from the Ethernet adapter by a direct memory access (DMA). Please refer to the specification/manual of the Ethernet adapter. The memory must be mapped to the user space since the PROFINET stack runs in user space.

On most operating systems like Linux or Windows, this kind of memory can be allocated only with prioritized access level, e. g. in kernel space. In the PN Driver Linux sample implementation, the EPS solves this problem by using the kernel-mode driver PNDevDriver. The kernel-mode driver is called in the PN device driver using IO control calls (IOCTLs). See \src\source\pnboards\eps\epssys\src\eps_pndevdrv.c.

On an embedded system where the systems integrator designs the memory layout, this NRT memory can be set to a static region in the RAM using a linker file and linker symbols.

1. Set EDDS_CFG_INTEL.
2. Implement the following EPS_PLF output functions:
   - EPS_PLF_PCI_MAP_MEMORY (see section EPS_PLF_PCI_MAP_MEMORY (Page 18))
   - EPS_PLF_PCI_ENA_INTERRUPT (see section EPS_PLF_PCI_ENA_INTERRUPT (Page 20))
   - EPS_PLF_PCI_DIA_INTERRUPT (see section EPS_PLF_PCI_DIA_INTERRUPT (Page 21))
3. Write a kernel-mode driver that is able to allocate contiguous NRT memory and map this memory to the user space.

Trace

6

Overview

PN Driver has a built-in trace system called PROFINET Trace (PNTRC) that needs to be adapted for different operating systems. However, this trace system is optional.

Traces are useful to investigate faulty behavior in the PNIO stack. Traces are written in a proprietary binary format. To convert these binary traces to a readable format, a trace database is required. Since the PN Driver delivery package does not provide tools to create this database, the customer cannot use the trace system for the diagnostics of the customer application. However, the binary traces may be useful when contacting the customer support team.

When not configured at "SERV_CP_init()", nothing needs to be done here. (For "SERV_CP_startup()", refer to the PROFINET IO-Base user programming interface manual).

Integration in different platforms and operating systems

The integration in different platforms and operating systems is done by the EPS. If the PN Driver wants to use the integrated trace system, a callback function to write the trace data has to be provided by the PN Driver. In the Windows and Linux sample, a callback function is linked at "EPS_APP_INIT()":

```
NoShmDrvArgs.pTraceBufferFullCbf = eps_plf_trace_buffer_full;
```

Function syntax "eps_plf_trace_buffer_full"

LSA_VOID

eps_plf_trace_buffer_full

(LSA_VOID_PTR_TYPE hSys, LSA_VOID* hTrace,
LSA_UINT32 uBufferId, LSA_UINT8* pBuffer, LSA_UINT32
uLength)

Description

- This is a function prototype. This function is called by PNTRC when the trace buffer is full and needs to be emptied.
- In the PN Driver this function calls the "PNIO_PNTRC_BUFFER_FULL" function, see below.

Parameter data type	Parameter name	In/Out	Meaning
LSA_VOID_PTR_TYPE	hSys	In	System handle
LSA_VOID*	hTrace	In	Trace handle
LSA_UINT32	uBufferId	In	Buffer handle
LSA_UINT8*	pBuffer	In	Pointer to buffer
LSA_UINT32	uLength	In	Length of the buffer

Function syntax "PNIO_PNTRC_BUFFER_FULL"

```
void
(*PNIO_PNTRC_BUFFER_FULL)
( PNIO_UINT8* pBuffer,
PNIO_UINT32 BufferSize );
```

Description

- This is a function prototype. This function is called by PN Driver when "eps_plf_trace_buffer_full" is called, see above.
- The function pointer must be set in the structure "PNIO_DEBUG_SETTINGS_TYPE" in the function pointer "CbfPntrcBufferFull" before calling "SERV_CP_init". If "CbfPntrcBufferFull" is set to "LSA_NULL", this function will not be called.

Parameter data type	Parameter name	In/Out	Meaning
PNIO_UINT8*	pBuffer	In	Pointer to a memory region with a trace buffer (SRC)
PNIO_UINT32	BufferSize	In	Size of the trace buffer

Sample

- pndriver\test\pnd_test.c
  The trace buffer is written into a file using fwrite and fflush.

PSI settings (configuration of the PROFINET stack) 7

Precompile settings

The components of the PROFINET stack are configured and connected by the component PSI.

PSI itself requires precompile settings.

Please note the following:

NOTICE
These settings were defined by the PN Driver in cooperation with the development team of the Siemens PNIO stack. These values are reserved, you must not change these values. Changes to these values may cause faulty behavior of the communication stack.

You may perform the following changes.

Configuring the EDDS to Packet32/Intel lower layer

- When using an Intel lower layer:
  #define EDDS_CFG_HW_INTEL in <config_file.h>.
- When using the Packet32 lower layer:
  Set EDDS_CFG_HW_PACKET32 in <config_file.h>.

See section Integration of the Intel lower layer implementation into a PN device driver (Page 46).

Appendix A

A.1 Abbreviations/Glossary of terms

API	**A**pplication **P**rogramming **I**nterface
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
BAR	**B**ase **A**ddress **R**egister in a PCI device
CBF	**C**all**B**ack **F**unction
DMA	**D**irect **M**emory **A**ccess
EDDS	**E**thernet **D**evice **D**river for **S**tandard Ethernet adapters
EPS	**E**mbedded **P**ROFINET **S**ystem
EPS APP	EPS Application abstraction
EPS PLF	EPS Platform abstraction
IF	**I**nter**F**ace
Intel LL	Intel **L**ower **L**ayer implementation for EDDS
IO	**I**nput/**O**utput, e. g. for process values
ISR	**I**nterrupt **S**ervice **R**outine
MSI	**M**essage-**S**ignaled **I**nterrupts
NRT	**N**on-**R**eal-**T**ime communication, generic term for all Ethernet frames unequal to ethertype 0x8892
OS	**O**perating **S**ystem
Packet32 LL	Packet32 **L**ower **L**ayer implementation for EDDS
PCI	**P**eripheral **C**omponent **I**nterconnect
PI	**P**rocess **I**mage, an image of the process data inside a RAM. This memory is located in the communication RAM and used for cyclical communication.
PN	**P**ROFI**N**ET
PN Driver	**P**ROFI**N**ET Driver IO Controller Development Kit
PNDevDriver	**P**ROFI**N**ET Device Driver
PNIO	**P**ROFI**N**ET **IO**
POSIX	**P**ortable **O**perating **S**ystem **I**nterface
PSI	**P**ROFINET **S**tack **I**nterface
RT	**R**eal **T**ime, generic term for acyclic and cyclic real-time
SHM	**Sh**ared **M**emory