

SIMATIC NET

PC software DK-16xx PN IO Porting Instructions and Layer 2 Interface




Programming Manual

Quick Start	1
Preparing RTAI and the Linux kernel	2
Description of driver porting	3
Description of porting the IO base library	4
Description of porting the Layer 2 library	5
Description of the "cp16xxtest" program	6
L2 - Quick start with the Layer -2 interface	7
L2 - Overview of the Layer 2 interface	8
L2 - Description of the Layer 2 functions and data types	9
L2 - Creating a Linux Ethernet driver	10

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.

Table of contents

1	Quick Start.....	7
1.1	Architecture of DK-16xx PN IO software	7
1.2	Installation on Linux	8
2	Preparing RTAI and the Linux kernel.....	11
2.1	Basic procedure for generating, installing and testing the real-time extension RTAI	11
2.1.1	Stage 1: Downloading source files from the Internet.....	11
2.1.2	Stage 2: Extracting Source Files	12
2.1.3	Stage 3: Configuring and generating the Linux kernel	13
2.1.4	Stage 4: Installing the generated Linux kernel	14
2.1.5	Stage 5: Configuring and generating the RTAI real-time extension	15
2.1.6	Stage 6: Checking whether the real-time extension works.....	16
2.2	Basic procedure for installing the DK-16xx PN IO on Linux	17
3	Description of driver porting.....	21
3.1	Requirements for the target operating system	21
3.2	How the driver works	21
3.3	Basic communication between the library and the driver	23
3.3.1	Directory structure and files	24
3.3.2	Non operating system-specific functions	26
3.3.3	Functions dependent on the operating system.....	28
3.4	Porting the driver step-by-step.....	29
3.4.1	Stage 1: Porting the macros of the "os_linux.h" file	30
3.4.2	Stage 2: Initialization and deinitialization	32
3.4.3	Stage 3: Finding the CP and including the resources of the CP in the operating system.....	32
3.4.4	Stage 4: Defining the driver interface	33
3.4.5	Stage 5: Porting the connection establishment and termination from the IO Base library to the driver	35
3.4.6	Stage 6: Porting send functionality from the IO Base library to the firmware	36
3.4.7	Stage 7: Porting the receive functionality from the firmware to the IO Base library	36
3.4.8	Stage 8: Porting memory mapping to the user address space.....	37
3.4.9	Stage 9: Porting additional IO controls for the "cp16xptest" driver test application.....	38
3.5	Driver debug support	38
3.6	Testing the driver	39
4	Description of porting the IO base library.....	41
4.1	Requirements for the target operating system	41
4.2	How the IO Base library works	41
4.2.1	Directory structure and files	43
4.2.2	Functions dependent on the operating system.....	44
4.3	Porting the IO Base library step-by-step.....	45

4.3.1	Stage 1: Porting the trace module	45
4.3.2	Stage 2: Porting the IO Base library link for the driver	45
4.4	IO-Base library debug support	45
4.5	Testing the IO-Base library	47
5	Description of porting the Layer 2 library	49
5.1	Requirements for the target operating system.....	49
5.2	How the Layer 2 library works.....	49
5.3	Directory structure and files	50
5.4	Porting the Layer 2 library step-by-step	51
5.5	Testing the Layer 2 library	51
6	Description of the "cp16xxtest" program.....	53
6.1	Directory structure and files	53
6.2	Porting the "cp16xxtest" program	53
6.3	Testing the "cp16xxtest" Program.....	54
7	L2 - Quick start with the Layer -2 interface	55
8	L2 - Overview of the Layer 2 interface.....	57
8.1	How a typical Layer 2 user programming interface is used	57
8.2	Software architecture	57
8.3	How a typical Layer 2 user program runs	59
8.3.1	Initialization phase	59
8.3.2	Send data.....	60
8.3.3	Receive data	61
8.3.4	Completion phase	62
8.4	Callback mechanism.....	63
9	L2 - Description of the Layer 2 functions and data types.....	65
9.1	l2eth_open (register with Layer 2 interface).....	65
9.2	l2eth_set_mode (set operating mode)	66
9.3	L2ETH_CBF_MODE_COMPL (signal operating mode)	67
9.4	L2ETH_CBF_STATUS_IND (signal status)	68
9.5	l2eth_get_information (query parameters)	69
9.6	l2eth_set_information (set parameters)	70
9.7	l2eth_allocate_packet (reserve send job)	71
9.8	l2eth_send (send data)	72
9.9	L2ETH_CBF_SEND_COMPL (signal send result).....	73
9.10	l2eth_free_packet (release send job).....	74
9.11	L2ETH_CBF_RECEIVE_IND (signal receipt of data)	75
9.12	l2eth_return_packet (return receive job)	76

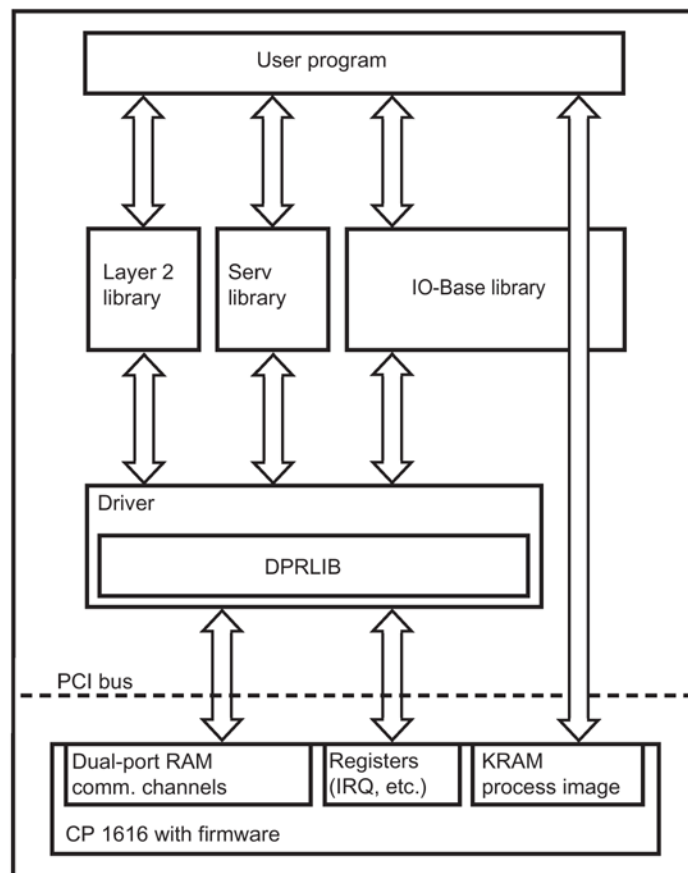
9.13	l2eth_close (deregister from Layer 2 interface)	77
9.14	Data types	78
9.14.1	Basic data types	78
9.14.2	Error codes	78
9.14.3	L2ETH_MAC_ADDR (type for MAC address)	79
9.14.4	L2ETH_MODE (type for operating mode).....	80
9.14.5	L2ETH_PACKET (job type for sending and receiving)	80
9.14.6	L2ETH_QUERY (job type for status query and parameter assignment)	81
9.14.7	L2ETH_PORT_STATUS (type for port status)	82
9.14.8	L2ETH_OID (type for object identifier).....	83
10	L2 - Creating a Linux Ethernet driver	85
10.1	Basics of developing a Linux Ethernet driver based on the Layer 2 functions	85
10.1.1	Interfaces of the Linux Ethernet driver to Linux	86
10.1.2	Interfaces of the Linux Ethernet driver to the Layer 2 interface.....	88
10.1.3	Point to note when compiling	88

Quick Start

1.1 Architecture of DK-16xx PN IO software

Description

The following graphic shows the software layers and communication paths of the DK 16xx PN IO software, the following table explains the terminology used in the graphic.



Picture element	Description
IO Base library	The IO Base library provides the IO Base user programming interface. The functions required for driver communication and the trace must be ported from the IO Base library.
Serv library	The Serv library makes the Serv user programming interface available. When porting the Serv library, the driver communications, synchronization and file access functions defined in the "os_linux.h" file must be adapted to the new operating system.
Layer 2 library	The Layer 2 library makes the Layer 2 Ethernet user programming interface available. The functions required for driver communication and the trace must be ported from the Layer 2 library.
Driver	The driver is responsible for the communication between the following software components: <ul style="list-style-type: none"> • IO Base library – Serv library – firmware • Layer 2 interface – firmware • Integration of the hardware resources in the operating system All driver functions must be ported to the target operating system.
DPRLIB library	The DPRLIB library is used by the driver and makes all non platform-dependent functions required for communication available to the firmware via the dual-port RAM.
Dual-port RAM	The dual-port RAM is the memory area of the CP 1616 that is used for handling communication between the firmware and host. This memory area is divided into independent communication channels.
Register	Register is the memory area in which the registers of the CP 1616 are stored.
KRAM	KRAM is the memory in which the process data is stored.
Arrows	Arrows represent the independent communication channels.

1.2 Installation on Linux

Introduction

The development kit provides you with source files in Linux for the sample applications, the driver, the IO Base library, Serv library and Layer 2 library. These source files can be ported to other operating systems.

To install the driver, the IO Base library, Serv library and Layer 2 library, you require Linux with kernel source files installed and a development environment, for example GNU-C-Compiler.

To use isochronous real time (IRT), we recommend the installation of the real-time extension RTAI, since without these extensions, Linux takes up to 1 ms to report an interrupt to the application.

Administrator privileges

To install the driver of the PROFINET IO library and the Layer 2 library, you require administrator privileges.

Linux system requirements

The table below contains the recommended versions of the required software components.

System partner	Version
Linux system	Suse Linux 10.1, 10.2, 11.2
Kernel	As of version 2.6.10; version 2.6.32.2 is used in the example
GNU C compiler (GCC)	As of version 3.3.5
Kernel source files	Appropriate for kernel
Real-Time Application Interface RTAI available at RTAI (www.rtai.org)	To suit the kernel, version 3.8 is used in the example

Note

You will find the latest information on Linux system requirements on the Internet at RTAI (www.rtai.org).

Hardware requirements

The table below lists the system resources required by the driver and IO Base library.

System parameter	Values
Hard disk space	Approximately 1 MB for source files, maximum 3 MB for the firmware and S7 configuration backup files when using the Serv library.
Processor	Intel 386 or higher
RAM	Min. 800 KB RAM for driver and library
DMA-compliant memory	64 Kbytes if IRT mode is required or when using the Layer 2 interface
Interrupts	<ul style="list-style-type: none"> • In IRT mode: one non-shared interrupt • For operation without IRT: one interrupt, either shared or non-shared

Preparing RTAI and the Linux kernel

This chapter explains how to prepare the Linux kernel and install real-time capability with RTAI.

2.1 Basic procedure for generating, installing and testing the real-time extension RTAI

Description

The following procedure simply outlines the principles underlying installation. The actual installation could change at any time. You should therefore always read the installation instructions supplied for the kernel and RTAI.

Follow the steps outlined below.

Note

Adapt the version numbers in the paths and commands.

2.1.1 Stage 1: Downloading source files from the Internet

Description

If you do not already have the required files, download them from the Internet as described below:

Step 1

Command: Program of your choice

Description: Download the current RTAI version from RTAI (<http://www.rtai.org>).

Note

If the files are downloaded when using a Windows operating system, it is possible that the file name changes.

You should therefore rename the files as they were before the download.

Step 2

Command: Program of your choice
Description: Change to the user "root" with the Switch User command.
Extract the downloaded RTAI files to the directory "/usr/src/RTAI-3.8". If you use an RTAI version other than 3.8, change the version number or RTAI accordingly in the path name.
Change to the folder "RTAI-3.8/base/arch/x86/patches". This folder contains real-time patches for the supported Linux kernel versions.

Step 3

Command: Program of your choice
Description: Select one of the supported Linux kernels and download the kernel sources from Kernel (<http://www.kernel.org>) to the directory /usr/src.

2.1.2 Stage 2: Extracting Source Files

Description

After you have downloaded the files from the Web, they are still compressed. Follow the steps outlined below to extract the files.

Step 1

Command: su
Description: Change to the user "root" with the Switch User command.

Step 2

Command: cd /usr/src
Description: Change to the "/usr/src/" directory.

Step 3

Command: bunzip2 linux-2.6.32.2.tar.bz2
tar -xf linux-2.6.32.2.tar
Description: Extract the Linux kernel source code. Adapt the version number of the Linux kernel accordingly, here 2.6.32.2.

2.1.3 Stage 3: Configuring and generating the Linux kernel

Description

The section below describes configuration and generation of a Linux kernel with real-time capability.

Step 1

Command: su

Description: Change to the user "root" with the Switch User command.

Step 2

Command: cd /usr/src/linux-2.6.32.2

Description: Change to the "/usr/src/linux-2.6.32.2" directory.

Step 3

Command: patch -p1 -i ../rtai-3.8/base/arch/x86/patches/hal-linux-2.6.32.2-x86-2.5-00.patch

Description: Add the patch to the Linux source code.

Step 4

Command: cat /proc/config.gz | gunzip > .config
make oldconfig

Description: Accept the kernel configuration from the running kernel and extend the configuration if any options are undefined. Accept or the default is by repeatedly pressing the "ENTER" key.

Step 5

Command: make menuconfig or make xconfig

Description: Reconfigure the kernel.

Make sure that the following options are set correctly:

Options	Value
Enable Loadable module support -> Module versioning support	OFF
Processor type and features -> Subarchitecture type	PC-compatible
Processor type and features -> Processor family	Select the processor family closest to your processor. (Pentium Classic normally works with newer Intel processors) If you have a multicore processor, do not select a processor family that does not support TSC!
Processor type and features -> Generic x86 support	OFF
Processor type and features -> Symmetric multi-processing support	For single core systems: Off For multicore systems: On
lpipe support -> lpipe support or Processor type and features -> Interrupt pipeline	ON
Kernel hacking -> Compile the kernel with frame pointers	OFF

Save the configuration before exiting by answering the prompt "Save the new kernel configuration?" with "Yes".

Step 6

Command: make clean all
Description: Compile the kernel.

2.1.4 Stage 4: Installing the generated Linux kernel

Description

Once you have generated the kernel, this must be installed so that it can be loaded the next time you restart the computer. Follow the steps outlined below:

Step 1

Command: su
Description: Change to the user "root" with the Switch User command.

Step 2

Command: cd /usr/src/linux-2.6.32.2
Description: Change to the "/usr/src/linux-2.6.32.2" directory.

Step 3

Command: make modules_install
Description: Install the kernel modules.

Step 4

Command: make install
Description: Install the kernel.

Step 5

Command: reboot
Description: Restart your PC and select the entry for the kernel you have just installed in the Boot menu.

2.1.5 Stage 5: Configuring and generating the RTAI real-time extension

Description

After installing the kernel, the modules for the real-time extension for RTAI must be configured and generated. Proceed as follows:

Step 1

Commands: su
cd /usr/src/RTAI-3.8
Description: Change to the user "root" with the Switch User command, and then change to the "/usr/src/RTAI-3.8" directory.

Step 2

Commands: make menuconfig

Description: Configure RTAI.

Match the RTAI options with those of your Linux kernel.

Note the following points:

- If your Linux kernel is set to SMP, RTAI must also be set to this.
- The path to the Linux source code must also be correctly set.
- For SMP, the number of processors in the kernel must match the setting in RTAI.
- If you use a hyperthreading CPU and hyperthreading is enabled in the BIOS, the SMP option must be selected for the kernel and for RTAI (a processor with hyperthreading behaves like two processors).
- Set the number of CPUs you are using in "Machine(x86)->Number of CPU's". You will find the number in the /proc/cpuinfo file.

Step 3

Commands: make install

Description: Compile and install RTAI.

2.1.6 Stage 6: Checking whether the real-time extension works

Description

Checking whether the real-time extension integrated in the Linux kernel actually works is based on latency measurements of the sample program supplied with RTAI.

Running the test

Command: su

Description: Change to the user "root" with the Switch User command.

Start the test programs that ship with RTAI:

- "/usr/realtime/testsuite/user/latency./run" or
- "/usr/realtime/testsuite/kern/latency/run"

The test programs measure the delay times (latency measurement) and display them continuously on the screen.

These times must only change slightly when you increase system load. This can, for example, occur when the mouse is moved quickly, when you type quickly on the keyboard or when the hard disk or other peripheral devices are accessed.

Note

The changes in latency have decisive effects on the functionality of your user program. The latency should only be a fraction of the cycle time. If the latency is too long, an overrun can occur. This should be avoided.

Procedure following an unsatisfactory test

If the latency changes considerably, your system configuration is only suitable for real-time applications with certain restrictions or is not suitable at all. This also applies to isochronous real time (IRT).

In this situation, you should try to change the options for the kernel and RTAI, for example:

- Disable support of ACPI.
- Disable support of APIC and APM.
- Disable support of SMP or hyperthreading.
- Disable "Legacy Support for USB" in the BIOS.
- Disable the 3D acceleration for your X windows (graphic user interface).
- Disable the graphics mode, for example with the command line command "init 3"; then repeat the latency measurements.

To increase the load, you can, for example, switch over from one console to another with the shortcuts Ctrl + Alt + F1 to Ctrl + Alt + F7 and start further programs.

If the latency measurement is successful, this means that you will need to change the graphics card driver. Tip: The VESA frame buffer driver has often proved to be a suitable alternative.

- Disable support of all unnecessary options (USB, sound card, modem etc.).

If these suggestions do not help, you can obtain further help on the Web site of the manufacturer and:

- RTAI (<http://www.rtai.org>)

2.2 Basic procedure for installing the DK-16xx PN IO on Linux

The table below describes the actions to be carried out when installing the driver and the IO Base library and Layer 2 library from a shell (command line). To do this, you may have to make a number of platform-specific modifications.

Step 1

Command: su
Description: Open a "root shell".

Step 2

Command: `mount -t iso9660 /dev/cdrom /media/cdrom`
Description: Mounting the CD

Step 3

Command: `cp /media/cdrom/linux-sw/host-xxx.tar.gz`
Description: Copy the files.

Step 4

Command: `tar -xzf host-xxx.tar.gz`
("xxx" is a placeholder)
Description: Extract the files.

Step 5

Command: `cd host_linx`
Description: Change to the installation directory.

Step 6

Command: `export RTAI=y`
Description: If the real-time extension is used.

Step 7

Command: `make`
Description: Generate the driver, the IO Base library, the Serv library and the Layer 2 library.
This is only possible if the real-time extensions were successfully installed, see section "Basic procedure for generating, installing and testing the real-time extension RTAI (Page 11)".

Step 8

Command: make install

Description: Install the driver, IO Base library, Serv library, Layer 2 library and header files.

The PROFINET IO, Serv and Layer 2 libraries are copied to the "/usr/lib" directory and the driver to the "./lib/modules/[kernel version]/misc" directory.

The header files of the IO-Base library "pniobase.h", "pnioursd.h", "pnioursrt.h", "pnioursrx.h", "pniourrx.h" and the header files of the Serv library "servusrx.h" are copied to the "/usr/include" library.

The header files of the Layer 2 library

"l2eth_user.h", "l2eth_errs.h" and "l2eth_rqb.h" are copied to the "/usr/include" directory.

Step 9

Command: make load

Description: Load and start the driver.

Note

Note that you must start the driver again manually each time the computer is restarted. You can have the driver start automatically by configuring the file "/etc/rc.d" manually; for an example, refer to makefile under the maketarget "autoload".

Installing a sample program

The table below describes which actions must be carried out to install the sample programs in a shell (command line). To do this, you may have to make a number of platform-specific modifications.

Command: make test

Description: Generate the test programs.

Testing after installation

The table below shows you how to test the driver, the IO Base library and the Layer 2 library following installation:

Action	Description
Testing the driver.	Call "make load". No error message should be displayed if the CP 1616 has been installed correctly in the PC.
Testing the firmware on the CP 1616.	Take the configuration of the sample application "pnieoeasy" and use STEP 7 or NCM to download from a configuring station to the CP 1616.
Testing the dual-port RAM.	Call the application "pnioping". This must output "success". This is only possible as with testing firmware when a configuration has been successfully downloaded to the CP 1616.
Test the IO Base library.	Install the sample application "pnieoeasy". This test application implements an IO controller. The required configuration is enclosed with the example.
Test the Layer 2 library.	Using the sample application program "l2eth_ping", try to reach a computer in your test network.

Description of driver porting

This chapter explains the functionality of the Linux driver.

You will also learn step-by-step how to port the driver to your target operating system.

3.1 Requirements for the target operating system

Description

The driver requires the following operating system functionality:

- Threads
- Mutexes
- Semaphores
- Memory mapping from the kernel address space to the user address space if the address areas differ.
- Guaranteed reaction times to interrupts in isochronous real-time mode.
If the reaction times to interrupts are extremely high, IRT can only be operated with long cycle times.

3.2 How the driver works

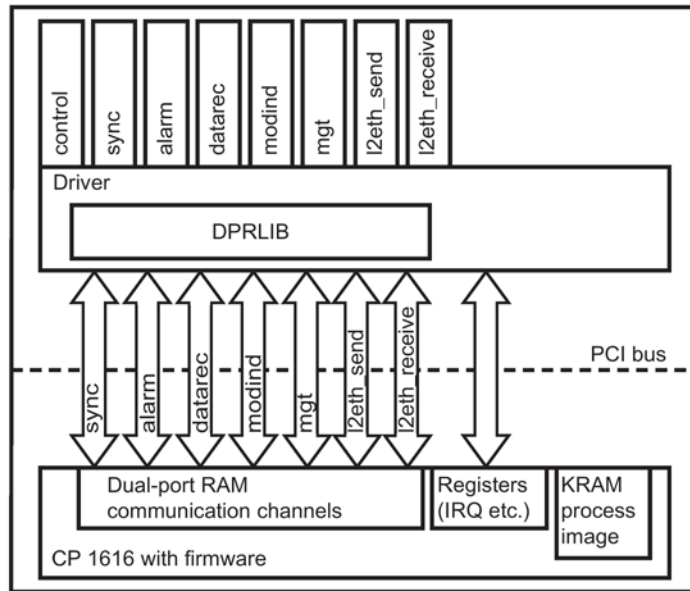
Overview

The driver is used to activate the CP 1616 and to integrate the memory windows and IRQs of the CP 1616 in the operating system. It:

- processes interrupts.
- maps the process image on the CP for the IO Base library.
- handles jobs between the IO Base library and the firmware on the CP.

The driver also contains a watchdog function that monitors the firmware on the CP to allow the CP 1616 to be reset. In turn, the driver is monitored by the firmware and must report to the firmware that it is operational at defined intervals.

The following schematic shows the basic structure of the driver and the CP 1616. The arrows indicate the communications channels of the driver to the hardware and firmware. Communication channels are memory areas on the CP 1616 that contain two ring buffers (one ring buffer for jobs from the driver to the firmware and one ring buffer for jobs from the firmware to the driver). The boxes above the driver represent the device files. On Linux, device files are driver access points via which applications communicate with the driver.



Description of the makefile

To load the driver, call the supplied makefile with "make load" in the "cp16xx" directory. When it is activated, it creates the "cp16xx1" entry and the "cp16xx1/control" subentry in the device tree ("/dev"). The script for the communication channels in the dual-port RAM also creates the following device files:

Device files /dev/cp16xx1/...	Supported file operations	Communication channel for ...
sync	open, read, write, ioctl	... synchronous jobs
alarm	open, read, write, ioctl	... asynchronous alarm jobs
modind	open, read, write, ioctl	... asynchronous changes in protocol state
datarec	open, read, write, ioctl	... data set transfer
mgt	open, read, write, ioctl	... management of application watchdog function
Control	open, read, write, ioctl, mmap	... the instance management of the driver - This communication channel has no equivalent in the dual-port RAM.
l2eth_send	open, read, write, ioctl	... Ethernet send jobs
l2eth_receive	open, read, write, ioctl	... Ethernet receive jobs

These device files are used by the IO Base, Serv and Layer 2 libraries to communicate with the module. For the precise sequence and the required script commands, refer to the "/driver/cp16xxloader" script file.

Description of driver startup

The driver allocates all of the PCI resources required for the dual-port RAM, register, IRQ and process image memory (KRAM). The driver then triggers an interrupt on the CP so that the firmware initializes the communications channels. The firmware uses an interrupt and a status value in the configuration structure to inform the driver that initialization was successful and that it is ready for communication.

During initialization, the driver registers with the firmware for time monitoring.

Description of the driver in the productive phase

The driver stores jobs coming from the IO Base library in the communication channel required by the IO Base library and triggers an interrupt in the firmware. Once the firmware has processed these jobs, it places an acknowledgment for the jobs on the communication channels and indicates this by sending an interrupt to the driver. The driver then transfers the acknowledgment to the IO Base library.

Once the firmware has written jobs for the IO Base library in the communication channels, it signals this with an interrupt to the driver. The driver then transfers these jobs to the IO Base library. As soon as the IO Base library has processed these jobs, it in turn sends an acknowledgment. An acknowledgment is sent in the same way as a job is sent to the firmware.

Signaling IRT interrupts in the productive phase

IRT uses two interrupts: STARTOP and OPFAULT.

The STARTOP interrupt signals that the IRT data was transferred to the host memory and that the application can start to process data. Completion of processing of IRT data must be signaled to the IO Base library by the application. The IO-Base library then starts a DMA transfer in which the new IRT data is transferred to the communications processor.

If this transfer is made too late, the communications processor cannot transfer the IRT data over the network and triggers the OPFAULT interrupt to report this to the application.

3.3 Basic communication between the library and the driver

Interface between driver and library

The IO Base library communicates with the driver using file operations, IO controls and memory mapping functions.

Registering an IO Base library instance with the driver

The IO Base library uses four communications channels in the dual-port RAM. The IO Base library opens a device file for each channel. The IO Base library also requires the "/dev/cp16xx1/control" device file for the IRT interrupt and DMA functionality as well as for managing instances. To allow the driver to distribute the jobs and acknowledgments from the firmware to several applications, when the applications register for the communications channels, they inform the driver of the device file handles that they received when opening the "/dev/cp16xx1/control" device file.

The IO Base library is registered with the driver in four steps:

1. The IO Base library opens the "/dev/cp16xx1/control" device file.
2. The IO Base library sends the IO control CP16XX_IOC_OAPP (register application) with the file handle for the "/dev/cp16xx1/control" device file to obtain an application handle from the driver.
3. Open a device file for each dual-port RAM communications channel.
4. The opened device files are linked to the application with the application handle.

Sending the job packets from the IO Base library to the firmware

The IO Base library can send job packets to the firmware via the driver. This takes place using the file operations "read", "write" and "ioctl".

Receiving job packets from the firmware

The IO Base library can receive job packets from the firmware via the driver. This is achieved with the "read" file operation.

Memory access functionality for reading process data

To allow the IO Base library to make the process data available for the IO Base user program, the driver provides the IO Base library with a service with which the memory for the process image can be mapped to the address space of the application. Mapping the process image memory to the address space of the user has the following advantages:

- Fast, direct data access for the application
- No interrupts for data access

3.3.1 Directory structure and files

Description

The "driver" directory contains the files that are not specific to the operating system.

The "driver\linux" directory contains the files required by the driver for functions with the Linux operating system. During porting, these files must be adapted to a different operating system.

The files supplied with the development kit are listed in the tables below. The header files of the Linux kernel are also required to allow

generation in Linux. If you want to port to an alternative operating system, you will need the header files of the target operating system.

Function of the platform-specific files

The table below shows the files that are platform specific and have to be adapted for porting.

Driver files	Purpose of the individual files
os.h os_linux.h	Contain macro templates that must be filled with operating system functions, e.g. mutexes, events, semaphores, event signaling.
cp16xx_linux.c	Contain the operating system adapter, the driver registration and deregistration, device detection and communication mechanisms between kernel and operating system.
cp16xx_linux_irq.c	Contains the operating system-specific functions for handling interrupts.
cp16xx_linux_irq_rtai.c	Contains the operating system-specific functions for handling interrupts when using RTAI.
cp16xx_linux_net.c	Contains the sample implementation of the L2 driver.

Function of the non platform-specific files

The table below shows the files that are platform independent and must not be modified. These files can be modified at any time by means of an update or error correction. They form the non platform-specific library "DPRLIB".

Driver files	Purpose of the individual files
cp16xx.h	IO control - Definitions of the driver
cp16xx_base.c cp16xx_base.h	Non operating system-specific driver functions containing user management, parameter passing, watchdog functions and the access to the registers of the module.
dprlibhost.c dprlib.h dpr_msg.h wd_dpr_msg.h mgt_dpr_msg.h	Contain driver-internal, non platform-dependent functions which handle communication to the firmware via the dual-port RAM.
dprintern.c dpintern.h	Contain driver-internal and non platform-dependent functions which handle communication to the firmware via the dual-port RAM. These files are also used by the firmware.
driver_ver.h fw_vers_bldnum.h fw1616dk_vers.h	These files are used for versioning.

3.3.2 Non operating system-specific functions

Table with user management functions and structures

Function/structures	Description
struct cp16xx_app_data	Application management structure
cp16xx_app_free()	Releases a management structure.
cp16xx_app_new()	Sets up a management structure.
cp16xx_app_search()	Searches for the management structure for a particular application.

Table with device management functions and structures

Function/structures	Description
struct cp16xx_card_data	Management structure for a CP - This is set up when the driver is loaded and is released again when the driver is unloaded.

Description of the DPRLIB functions

The non platform-dependent functions which are responsible for data transmission to the firmware are grouped together in the "DPRLIB" library. These functions are used by the driver. To make the driver source files clearer to understand, they are explained in the following table. Communication takes place via channels in the dual-port RAM which are configured as ring buffers.

Note

These source files must not be modified since they can be changed at any time during updates and bug fixes and because the DPRLIB must match the firmware of the CP.

Function/structure	Description
CpData	<p>The function pointers listed below must be entered in this structure by the driver. This structure contains all the dependencies between the DPRLIB and the driver and must also be passed on to the DPRLIB with each call.</p> <p>trigger_irq Function pointer to the function used to trigger an interrupt to the CP 1616.</p> <p>wakeup_daemon Function pointer to a function which is called as soon as the DPRLIB disconnects the link to the firmware.</p> <p>parent Points to the structure "cp16xx_card_data" which is filled during the hardware binding.</p>
DPRLIB_start()	Start connection to the firmware - This function initializes the dual-port RAM.

Function/structure	Description
DPRLIB_stop()	Stop connection to the firmware - This function resets the dual-port RAM.
DPRLIB_channel_write_message()	Writes a job packet to a communication channel in the dual-port RAM.
DPRLIB_channel_read_message()	Reads a job packet from a communication channel in the dual-port RAM.
DPRLIB_channel_register_cbf()	Registers a callback which is called when a job packet is received from the dual-port RAM. The CP communication channel and the job packet size are transferred to this callback as parameters. This callback allocates the required memory and calls DPRLIB_channel_read_message() to obtain the job packet.

Functions called by the operating system

The entry function is called by the operating system as soon as ...
cp16xx_base_ioctl()	... the IO Base library calls an "ioctl" for a device file.
cp16xx_base_ioctl_1()	... the IO Base library calls an "ioctl" for a "control" device file.
cp16xx_base_ioctl_2()	... the IO Base library calls an "ioctl" for a device file other than "control".
cp16xx_os_driver_cleanup()	... the driver is unloaded.
cp16xx_base_ioctl()	... the IO Base library calls "ioctl" for a device file.
cp16xx_base_read()	... the IO Base library calls "read" for a device file.
cp16xx_base_release()	... the IO Base library calls "fclose" for a device file.
cp16xx_base_write()	... the IO Base library calls "write" for a device file.

Standardized functions

Entry functions	Description
cp16xx_card_init	Initializes the management structures of the driver.
cp16xx_card_uninit	Deinitializes the management structures of the driver.
cp16xx_dma_init	Allocates DMA memory in the operating system.
cp16xx_dma_uninit	Releases allocated DMA memory.
cp16xx_pci_init()	Enters IO areas of the module in the address area of the driver
cp16xx_pci_uninit()	Removes IO areas of the module from the address area of the driver.

3.3.3 Functions dependent on the operating system

Description of the functions

The following functions described in the form of tables contain parts specific to the operating system and need to be ported to use another system.

Channel management functions and structures

Function/macro/structures	Description
struct cp16xx_channel	Management structure for the communication channels
DPR_CHANNEL_INIT_OS()	Internal function for setting up a management structure in the locked state.
DPR_CHANNEL_UNINIT_OS	Function for deinitializing a management structure.
DPR_CHANNEL_LOCK()	Locks a management structure.
DPR_CHANNEL_UNLOCK()	Unlocks a management structure.
DPR_CHANNEL_WAKEUP()	Sets up a synchronization object.
DPR_CHANNEL_WAIT_FOR_WAKEUP()	Waits for signaling of a synchronization object.

Functions specific to the operating system that are called by standardized functions (see Section 3.3.2 Non operating system-specific functions)

Entry function	Description
cp16xx_irq_shared_cbf	Interrupt service routine
cp16xx_os_driver_cleanup()	Is called by the operating system as soon as the driver is unloaded.
cp16xx_os_driver_cleanup()	Called by Linux as soon as the driver is unloaded.
cp16xx_os_driver_init()	Called by Linux as soon as the driver is loaded.
cp16xx_os_init_irq()	Internal function used to set up the interrupt service routine.
cp16xx_os_ioctl()	Is called as soon as the IO Base library calls an "ioctl" for a device file.
cp16xx_os_irq_init()	Registers the interrupt service routine with the operating system.
cp16xx_os_irq_uninit()	Internal function used to remove the interrupt service routine.
cp16xx_os_mmap()	Is called as soon as the IO Base library calls a "mmap" for a device file.
cp16xx_os_mmap_dma_remap()	Used internally by "cp16xx_control_mmap()" for DMA memory; maps the kernel address space to the user address space.
cp16xx_os_open()	Is called as soon as the IO Base library calls an "fopen" for a device file.

Entry function	Description
cp16xx_os_pci_init_resources()	Internal function used to set up the PCI resources of the CP.
cp16xx_os_pci_probe()	Called by Linux as soon as a CP is found. This function generates a module instance and registers the found module with the operating system.
cp16xx_os_pci_remove()	Called by the operating system as soon as the driver is unloaded as long as a CP exists.
cp16xx_os_pci_uninit_resources()	Internal function used to release the PCI resources of the CP.
cp16xx_os_read()	Is called as soon as the IO Base library calls a "read" for a device file.
cp16xx_os_release()	Is called as soon as the IO Base library calls a "fclose" for a device file.
cp16xx_os_reset()	Is called by the driver before the module reset and allows additional functions to be performed when necessary.
cp16xx_os_write()	Is called as soon as the IO Base library calls a "write" for a device file.
cp16xx_trigger_irq()	Trigger interrupt to CP 1616.
down_timeout()	Auxiliary function used implement a semaphore with timeout.

3.4 Porting the driver step-by-step

General

Porting requires an empty skeleton driver. This skeleton is filled with functions during the course of these porting instructions. If you wish, you can copy a number of structures and functions from the Linux driver files supplied. The porting instructions are divided into 9 steps:

Step	Description
1	Preparation: Porting the macros in the "os_linux.h" file
2	Initialization and deinitialization
3	Locating the CP and integrating the CP resources in the operating system. Creating management structures for the CP resources.
4	Defining the driver interface
5	Ports the connection establishment and termination from the IO Base library to the driver.
6	Ports the send functionality from the IO Base library to the firmware via the driver.
7	Ports the receive functionality from the firmware to the IO Base library.
8	Ports the memory mapping in the user address space.
9	Porting the low-level test application and debug IO controls.

Note

For simple porting, temporarily disable the firmware time monitoring of the driver by setting the "watchdog_cycle" variable in the "cp16xx_base.c" file to 0.

Syntax

<pre>watchdog_cycle = 0;</pre>	<pre>/* watchdog_cycle; watchdog temporarily deactivated*/</pre>
--------------------------------	----------------------------------------------------------------------

Optimizing watchdog functions

An incorrect implementation of the watchdog functions can prevent connections being established to the communications partners. Enable the firmware watchdog for the driver again only when you have implemented and tested the relevant functions thoroughly:

- PNIO_CP_set_appl_watchdog()
- PNIO_CP_trigger_watchdog()
- PNIO_CBF_APPL_WATCHDOG()

3.4.1 Stage 1: Porting the macros of the "os_linux.h" file

Overview

This file contains all the macro templates that the driver needs. The driver encapsulates all function calls to the operating system using basic macros. You must port this file so that you can then simply copy parts of the Linux driver in the following steps.

Creating a new operating-system-specific header file

If you have an operating system that is not supported, your task is to create a new operating system define, for example, "_MYOS" and to port all macros in the "os_linux.h" file. You then save the file under a different name, for example, "os_myos.h".

Integrating the new header file

The last job is to make sure that the previously ported file is included when your driver source files include the "os.h" file. You do this by defining the operating system define mentioned above (for example "_MYOS") and in your make file and inserting the following lines in "os.h":

```
#ifdef _MYOS
```

```
#include "os_myos.h"

#endif
```

Porting the macros

The following macros are defined for the driver:

Macro	Functionality
DPR_THREAD_HANDLE	Type which represents thread handles.
DPR_THREAD_CREATE (tid, name, prio, c, d, func, arg)	Generates a thread and returns 1; returns 0 if an error occurs. tid:Reference of a variable in which the thread handle is stored. name :Name of the thread prio:Priority of the thread c:Thread option (not used) d:Stack size func:Pointer to the thread function arg:Pointer to memory that the thread function receives as an argument.
DPR_THREAD_DELETE (hThread)	Releases a thread handle. hThread: Handle to be released.
DPR_SEMAPHORE	Type that represents semaphores (depending on the operating system).
DPR_SEM_CREATE (semObj)	Generates a counting semaphore. semObj:Reference to the variable in which the semaphore is stored.
DPR_SEM_WAIT (semObj)	Waits for semaphores. semObj:Reference to the variable in which the semaphore is stored.
DPR_SEM_WAIT_TIME (semObj, msecs)	As above, but with "timeout" in ms.
DPR_SEM_POST (semObj)	Sets the semaphore. semObj:Reference to the variable in which the semaphore is stored.
DPR_SEM_DESTROY (semObj)	Deletes the semaphore. semObj:Reference to the variable in which the semaphore is stored.
DPR_TASK_DELAY (msecs)	Time delay msecs:Time delay in milliseconds
DPR(_INTERPROCESS)_MUTEX	Type that represents a (cross-process) Mutex (specific to operating system).
DPR(_INTERPROCESS)_MUTEX_CREATE_UNLOCKED	Generates a (cross-process) Mutex.
DPR_INTERPROCESS_MUTE_LOCK	Occupies a (cross-process) Mutex.
DPR_INTERPROCESS_MUTE_UNLOCK	Releases a (cross-process) Mutex.

Macro	Functionality
DPR_INTERPROCESS_MUTE_DESTROY	Deletes a (cross-process) Mutex.
DPR_MEMCPY_TO_USER	Copies data to the user. If you are using an operating system without kernel address separation, a "memcpy" will always suffice here.
DPR_MEMCPY_FROM_USER	Copies data from the user. If you are using an operating system without kernel address separation, a "memcpy" will always suffice here.
DPRLIBERRMSG (fmt, args...)	Output in the event of an error; arguments as with "printf".
DPRLIBLOGMSG (fmt, args...)	Debug output if DEBUG is defined; arguments as for "printf".
DPR_ASSERT(x)	Assert macro to define a defined stoppage if errors occur, for example to implement a memory dump or emergency stop.

3.4.2 Stage 2: Initialization and deinitialization

Stage 2: Initialization and deinitialization

In this step, you can test porting of the file "os_linux.h".

Perform the following steps to test the debug macro templates:

Step	Description
1	Add the following line to the initialization routine of your driver: <code>DPRLIBERRMSG("start %s\n",cp16xx_driver_version);</code> In your initialization routine, create a semaphore with the previously ported macros and start a thread that immediately waits for the semaphore and sets a global variable "gThreadStopped" to 1 before thread closes.
2	Copy the variable "cp16xx_driver_version" from the file "cp16xx_base.c" to your driver file.
3	In your deinitialization routine, set the semaphore so that the thread can finish. Wait until the variable "gThreadStopped" changes to 1. Add the following line to the end of your deinitialization routine of the driver: <code>DPRLIBERRMSG("stop %s\n",cp16xx_driver_version);</code>
4	Compile the driver and test the initialization or deinitialization.

3.4.3 Stage 3: Finding the CP and including the resources of the CP in the operating system

Description

When the resources are included, five PCI memory areas are mapped to the operating system and an interrupt service routine is integrated. The "CpData" structure is then filled. This structure is required by the non platform-dependent DPRLIB and contains all the callbacks that the driver must make available to the DPRLIB library.

To port the hardware detection and resource integration to the operating system, you must port the operating system-specific functions listed in section "Stage 2: Initialization and deinitialization (Page 32)".

The following functions also need to be ported:

Function	Description
cp16xx_irq_reset_mask_xxx()	This function disables RT/IRT interrupts

3.4.4 Stage 4: Defining the driver interface

Stage 4: Defining the driver interface

In this step, you define the interface between the application (in this case, the IO Base library) and the driver. The firmware of the CP 1616 communicates with the IO Base library over several communication channels in the dual-port RAM that are set up as ring buffers.

Linux driver

The Linux driver creates a device file for each communication channel to be able to pass on these communication channels transparently as far as the IO Base library. This allows the driver or IO Base library to implement a simple Read/Write interface and avoids the need for additional packaging of the send or receive jobs. The driver also creates an additional "dev/cp16xx1/control" device file with which additional services of the IO Base library can be made available.

The approach incorporating several access points was selected so that the Linux driver does not need a multiplexer or a request block interface. As a result, all communication channels can be activated independently (i.e. without disabling each other). Another advantage is that each "access point" has a small interface.

Services of the driver

The tables below list all of the required driver services. You must use these tables to create a suitable equivalent interface to your driver. In the following steps, it is always assumed that the interface implemented in Linux is used.

Services of the access point "/dev/cp16xx1/control" access point

The access point "control" supports the following interface:

Control	Description
fopen	Function for obtaining an operating system file handle for "/dev/cp16xx1/control".
fclose	Function for closing a "/dev/cp16xx1/control" device file.
mmap	Service for mapping PCI resources of the CP 1616 to the user address space

Control	Description
IO control: CP16XX_IOC_CAPP	Service for deleting an application instance handle
IO control: CP16XX_IOC_GET_L2_DMA_RANGE	Service for obtaining the DMA-compliant memory reserved for the Layer 2 interface.
IO control: CP16XX_IOC_IRTCBF	Service for registering the use of interrupt notifications
IO control: CP16XX_IOC_OAPP	Service for creating an application instance handle
IO control: CP16XX_IOC_SET_DMA_RANGE	Service for setting parameters for the DMA transfer
IO control: CP16XX_IOCRESET	Service for hardware reset of the CP 1616
IO control: CP16XX_IOCshutdown	Service used to shutdown a communication link in an emergency, i.e. an application is deregistered from the driver and all data packets in the dual-port RAM are removed.
read	Service for blocking the reading of certain interrupts - The interrupt is selected using the transferred length parameter.

Services for the remaining access points

The accesses "sync", "alarm", "modind" "datarec", "l2eth_send", "l2eth_receive".

Control	Description
fopen	Function for obtaining an operating system file handle for the device file that corresponds to a communication channel in the dual-port RAM.
fclose	Function used to close a device file that corresponds to a communication channel in the dual-port RAM.
Write	Service used to send a job packet to the firmware.
read	Service for receiving a job package from the firmware.
IO control: CP16XX_IOC_BIND	Service to bind the device file handle to the application instance handle.
IO control: CP16XX_IOC_UNBIND	Service to unbind the device file handle from the application instance handle.

Procedure with only single access to the driver

If your operating system allows "access" to a driver only once, you will have to implement a type of request block interface and a multiplexer. The request block could have the following structure:

```

struct driver_request
{
    unsigned long opcode,
    unsigned long channel,
    unsigned long dataSize,
    unsigned char* ptrToBuffer
}
// read,write,ioctl...
// which channel to use
// net length of message
// pointer to message
    
```

If you require a request block interface and your operating system also distinguishes between the kernel and user address space, you must always map the data pointer to the kernel address space first!

Request block interface without mapping

If you have an interface that does not provide mapping, you can select the following request block interface:

<pre>struct driver_request { unsigned long opcode, unsigned long channel, unsigned long dataSize, unsigned char Buffer[4096] }</pre>	<pre>// read,write,ioctl... // which channel to use // net length of message // Buffer for message</pre>
------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------

Constraints for drivers with only one access point

If you only need to implement one access point for the driver, make sure that the communication channels to the firmware can be operated independent of each other. This means that the IO Base library must always be capable of reading and writing from within different threads from different communication channels.

You can achieve this if your driver supports reentry into the driver or if your driver works without blocking jobs.

3.4.5 Stage 5: Porting the connection establishment and termination from the IO Base library to the driver

Description

The Linux driver features a two-stage registration mechanism. It can be reduced to a single-stage system if only one "access point" to the driver exists and only one multiplexer is implemented.

In the first stage, the IO-Base library calls an "fopen" for all access points to obtain a non operating system-dependent file handle.

In the second stage, the IO Base library fetches an application handle by sending the CP16XX_IOC_OAPP IO control to the "Control" file handle.

The IO-Base library now calls the CP16XX_IOC_BIND IO control for the remaining file handles specifying the application handle.

From this time onward, the driver knows which file handles together form a communication unit to the firmware.

When the connection is released, the IO control call CP16XX_IOC_UNBIND is first called for all file handles, with the exception of the control file handle. The IO control CP16XX_IOC_CAPP is then sent to the control file handle.

3.4 Porting the driver step-by-step

Finally, an "fclose" is issued to all file handles.

The following functions for connection establishment and termination must be ported from the IO Base library to the driver:

Functions	Description
DPR_CHANNEL_INIT_OS()	Internal function for setting up a management structure in the locked state.
DPR_CHANNEL_LOCK()	Locks a management structure.
DPR_CHANNEL_UNINIT_OS()	Function for deinitializing a management structure.
DPR_CHANNEL_UNLOCK()	Unlocks a management structure.
DPR_CHANNEL_WAIT_FOR_WAKEUP()	Waits for signaling of a synchronization object.
DPR_CHANNEL_WAKEUP()	Sets up a synchronization object.

3.4.6 Stage 6: Porting send functionality from the IO Base library to the firmware

Description

Data is sent by the IO Base library via the firmware to the driver using a "write" call. This call transfers a pointer to the job packet for the firmware and the length of the job packet. The driver takes the pointer and length from the parameters transferred with "write" and calls the "DPRLIB_channel_write_message()" function. This function belongs to the non platform-dependent dual-port library and writes the job packet to the dual-port RAM.

To port this functionality, you only need the following function:

Function	Description
cp16xx_os__write()	This function is called by Linux as soon as the application issues a "write". This function determines the channel and calls the function "DPRLIB_channel_write_message()".

3.4.7 Stage 7: Porting the receive functionality from the firmware to the IO Base library

Description

Receiving job packets from the firmware involves five steps:

Step	Description
1	The IO Base library calls the "read" function from within a thread. If job packets from the firmware are already available, the "read" call is immediately canceled. Otherwise the thread blocks with this call until a job package is received from the firmware.
2	Using an interrupt, the firmware signals that it has written a job packet to the dual-port RAM.

Step	Description
3	<p>The interrupt service routine ISR "cp16xx_irq_shared_cbf()" registered in Section "Stage 3: Finding the CP and including the resources of the CP in the operating system (Page 32)" is called due to the interrupt in Step 2.</p> <p>This ISR calls the DPRLIB library-internal "dprlib_int_callback()" stored in the "comming_irq" field in the "CpData" structure in Section "Stage 3: Finding the CP and including the resources of the CP in the operating system (Page 32)".</p> <p>The ISR "cp16xx_irq_shared_cbf()" then acknowledges the interrupt and terminates.</p>
4	<p>The callback from Stage 3 sets a DPRLIB library-internal semaphore to wake up a DPRLIB library-internal worker thread.</p>
5	<p>The DPRLIB library-internal worker thread determines the communication channel in which a job packet is located, and calls the function "cp16xx_channel_cbf()" that was registered by the driver.</p> <p>This function occupies a block memory and calls the DPRLIB_channel_read_message() function to copy the data to the block memory.</p> <p>The block memory is then inserted in the chained block list of the corresponding channel.</p> <p>Finally, the IO Base library thread in the blocked "read" is woken up so that it can return the job packet to the IO Base library.</p>

Note

The mechanism described above ensures that the driver spends as little time as possible within the interrupt context.

This is important so that the start of other interrupt service routines, for example of the operating system or other hardware, is delayed as little as possible.

To port this functionality, you need the following function:

Function	Description
cp16xx_os_read()	This function is called as soon as the IO Base library calls a "read". This call blocks until data can be read.

3.4.8 Stage 8: Porting memory mapping to the user address space

Description

The "/dev/cp16xx1/control" device file supports the IO control for mapping memory areas of the CP 1616 to the user address space.

Linux provides only one "mmap" interface which contains only the parameters "Offset" and "Length". For this reason, the information indicating the PCI bar from which the memory area is to be mapped must be specified by means of an OR operation of a constant for the offset of the area. The constants are listed in the file "cp16xx.h".

Example:

3.5 Driver debug support

To map 100 bytes starting at offset 200 from the PCI bar (specified with the MMAP_OFFSET_IRTE), the length must be set to 100 and the offset to MMAP_OFFSET_IRTE + 200 when mmap is called.

To port this functionality, you only need the following function:

Function	Description
cp16xx_os_mmap()	This function is called by Linux as soon as the IO Base library sends "mmap" for the "/dev/cp16xx1/control" device file.

3.4.9 Stage 9: Porting additional IO controls for the "cp16xptest" driver test application

Description

If you test the driver with the "cp16xptest" low-level test application and want to use a memory dump of the CP 1616 for a support query, you must adapt the application to your driver interface and port the following additional IO controls for the "/dev/cp16xx1/control" device file:

IO control	Description
CP16XX_IOCREGW	Service used to write to a register of the CP 1616.
CP16XX_IOCREGR	Service used to read from a register of the CP 1616.
CP16XX_IOC DPRAMW	Service used to write to an address of the dual-port RAM.
CP16XX_IOC DPRAMR	Service used to read from an address of the dual-port RAM.
CP16XX_IOC WIRQ	Service used to wait for an interrupt.

3.5 Driver debug support

Creating a memory dump of the CP 1616

If you have ported the IO controls for the low-level test applications, you can create two important memory dumps using "cp16xptest". These are helpful when requesting support.

You can use the call "cp16xptest -s mr 0 0 x800000 > memdump" to create a memory dump from the dual-port RAM.

You can use the call "cp16xptest -s fr 0 0 x200000 > regdump" to create a memory dump from the registers.

Saving diagnostic information

If you encounter problems, save the diagnostic information of the firmware with the "cp16xptest trace -f filename" call.

This diagnostics information is helpful when requesting support.

Listing parameters

You can use the call "cp16xxtest" to obtain a complete list of parameters.

Resetting firmware

The "cp16xxtest reset" call resets the firmware and restarts it.

3.6 Testing the driver

Description

When porting has been completed, the new driver must be tested on the target operating system.

Procedure

The following table contains a suggestion for the test order when testing the driver on your target platform:

Step	Description
1	Starting and stopping the driver, finding the hardware Call "make load" and "make unload".
2	Triggering and receiving an interrupt 1. Call "testapps/cp16xxtest ir" in the first shell. 2. Call "testapps/cp16xxtest fw 17450 1" in a second shell. This register write access to offset 17450 causes the CP 1616 to trigger an interrupt. 3. If your interrupt binding is functioning correctly, the program will terminate in the first shell.
3	Data exchange 1. Call "testapps/pnioping" in a shell. 2. The user test program sends a special job packet to the firmware and expects the firmware to return this packet with a return value. If "pnioping" reports "success", you have successfully implemented the dual-port RAM and interrupt binding in your driver.

Description of porting the IO base library

This chapter explains the functionality of the IO-Base interface and how to port it to your target operating system.

4.1 Requirements for the target operating system

Required operating system functionality

The IO Base library requires the following operating system functionality:

- Threads
- Mutexes
- Semaphores
- Standard C/C++ libraries

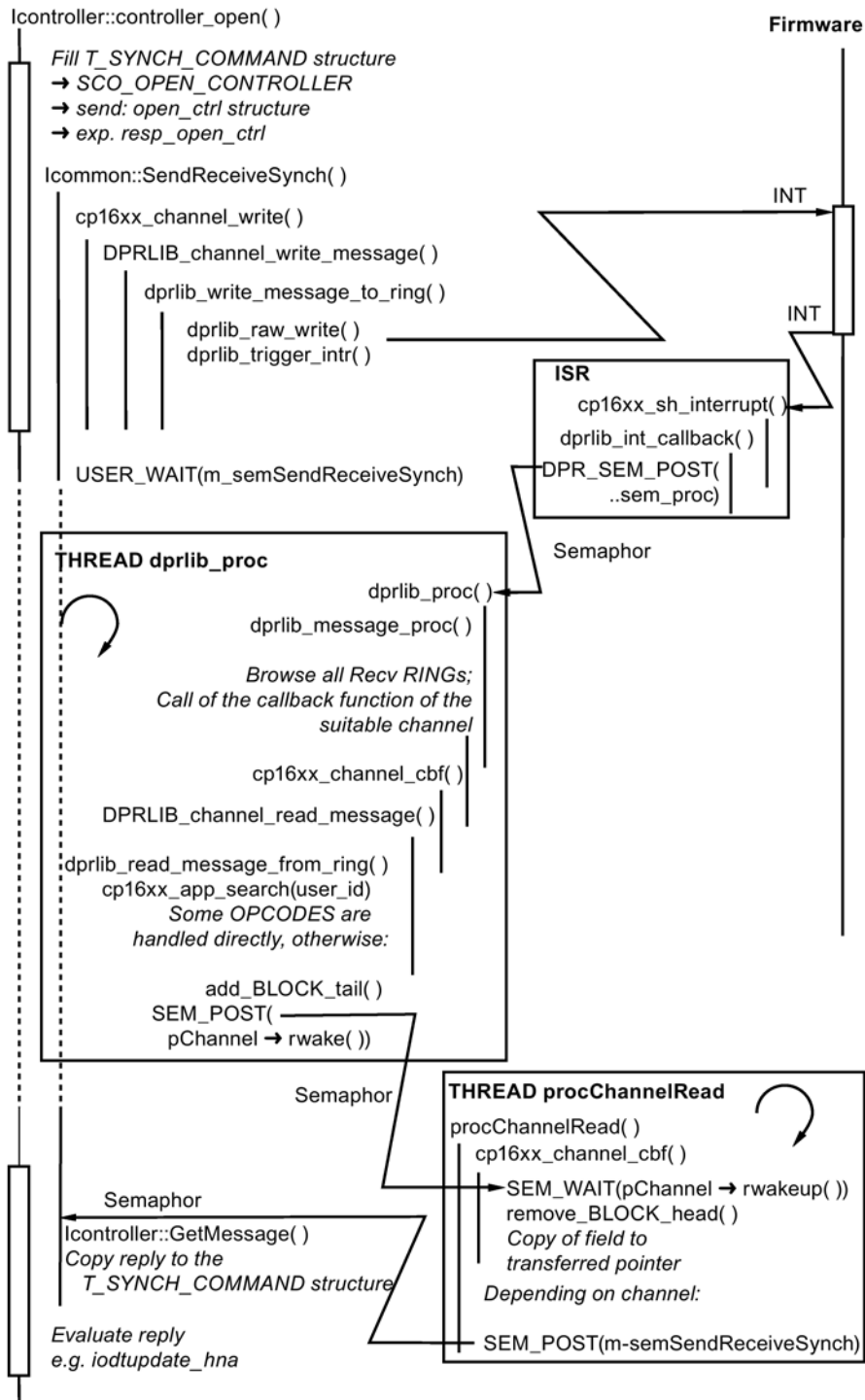
4.2 How the IO Base library works

Overview

The IO Base library provides the application with PROFINET IO functionality in the form of the IO Base interface. Your main task is to port the functions responsible for communication with the driver.

The following schematic shows an overview of the functional relationship between the IO Base interface and the firmware.

4.2 How the IO Base library works



4.2.1 Directory structure and files

Description

The source files and headers for the IO Base library can be found in the "pniolib" directory.

The table below lists the modules of the "pniolib" directory and explains their function.

Module name	Function
iodataupdate	Responsible for process image access.
iobase	Locates the data within the process image.
kramiotlb	Contains the PROFINET IO logic.
tracelib	Contains the trace functionality.
version	Contains the version header files

Description of the module directory content

The following table described the directories of the modules and their content.

Note

The number of modules may differ depending on the supplied software version.

Directory	Contents
csd	Make files
src	Source files
inc	Header files

Files to be ported

The table below shows the files that are platform specific and have to be adapted for porting. The IO-Base library was implemented in C++ and uses the standard C/C++ libraries.

Files	Purpose of the individual files
os.h os_linux.h	Contain macro templates that must be filled with operating system functions, e.g. creation of mutexes, events, semaphores, signaling of events.
trace_os.c traceout.cpp	Contain functions for the trace mechanism. The functions only have to be ported if you use a target platform without a file system or file mapping.

4.2.2 Functions dependent on the operating system

Functions for binding the IO-Base library to the driver

Function	Description
ICommon::InitCp()	Initializes the communication channels to the driver.
ICommon::UninitCp()	Deinitializes the communication channels to the driver.
ICommon::SendReceiveSynch()	Sends a synchronous job packet to the firmware via the driver and reads the acknowledgment from the firmware.
ProcChannelRead()	Thread function for reading out job packets and acknowledgments from the firmware.
ICommon::OpenDprChannel()	Opens a communication channel to the driver.
ICommon::CloseDprChannel()	This function closes a communication channel to the driver.
ICommon::Send()	Sends a job packet to the firmware via the driver.

Trace functions

Function	Description
TRC_GetCurrentThreadId()	Supplies the thread ID.
TRC_GetCurrentProcessId()	Supplies the process ID.
TRC_GetFormattedLocalTime()	Supplies the date and time as a string.
TRC_OutputDebugString()	Writes a trace entry to the console.
TRC_ExtractBegin()	Opens the trace configuration file for the trace and maps it to the memory to provide faster direct access for the function "TRC_ExtractKey()". If your target platform does not support file mapping, you simply have to read in the complete file. If your target platform does not have a file system, you can leave this function empty.
TRC_ExtractKey()	Reads an entry from the trace configuration file. If your target system does not have a file system, you must permanently encode the values for the entries.
TRC_ExtractEnd()	Removes the trace configuration file for the trace from the memory.

4.3 Porting the IO Base library step-by-step

General

Porting requires a C/C++ development environment with the standard C/C++ libraries. You perform porting in two steps:

Step	Description
1	Port the trace module.
2	Port the IO-Base library link for the driver.

4.3.1 Stage 1: Porting the trace module

Description

The file "traceout.cpp" only has to be ported if your target system does not contain a file system. In this case, you must convert the file accesses to, for example, memory accesses.

The file "trace_os.c" contains the logic required to read and evaluate the trace configuration file. The individual functions that must be ported are listed in the table in the section on trace functions in Section "Functions dependent on the operating system (Page 44)".

4.3.2 Stage 2: Porting the IO Base library link for the driver

Description

The file "fct_common.cpp" contains the code for communication with the driver. This is the only file of the IO-Base library that needs to be ported. Only the calls "fopen", "fclose", "write", "read" and "ioctl" have to be adapted to the target system. The individual functions to be ported are listed in the table in Section "Functions dependent on the operating system (Page 44)".

4.4 IO-Base library debug support

Description

Debug support is available in the form of a trace file mechanism. The trace quality is configured in the file "pnioTRACE.conf".

Description of the trace configuration file

The trace configuration file "pniotrace.conf" has the following entries:

Entry	Description
TRACE_TIME	0:No trace 1:Trace ON
TRACE_DEST	0:No trace 1:Create a new file for the trace 2:Append the trace to an existing trace 3:Trace to console
TRACE_DEPTH	Trace depth - Value of 3 means: Enable traces for value 1-3. 0:None 1:Trace with error 2:Trace with warnings 3:Trace with information 4-9:Trace depth level 1 to 6 0x0FFFFFFF:All traces
TRACE_FILE_ENTRIES	Maximum number of trace entries
TRACE_FILE_FAST	Trace file access type 0:Slow, in other words, the trace file is opened for every entry and closed again after the trace entry has been output. 1:Fast, in other words, the trace file is opened once and closed only after the application is exited.
TRACE_FILE_NAME	Trace file name
TRACE_GROUP	Submodule to be traced - See "tracesub.h" for the permitted values. The values can be ORed so that several submodules can be traced at the same time.
TRACE_MAX_BACK_FILES	Maximum number of created trace files - If this value is 2, this means the following: <ul style="list-style-type: none"> the current trace file has reached the maximum number of entries the trace file is renamed a new trace file is created If the new current trace file reaches the maximum number of entries, the first file is deleted and the second file becomes the first file.
TRACE_LEVEL_MODE	Not evaluated at present.
TRACE_SHOW_LINES	Not evaluated at present.
TRACE_APPLICATION	Not evaluated at present.
TRACE_DestHelp_NOTRACE	Not evaluated at present.
TRACE_DestHelp_NEWFILE	Not evaluated at present.
TRACE_DestHelp_SAMEFILE	Not evaluated at present.
TRACE_DestHelp_DEBUGOUT	Not evaluated at present.

Note

If you use IRT mode, deactivate the trace otherwise your real-time capability will be impaired.

If, however, you also require the trace functionality in IRT mode, you may need to improve the performance of the trace module; for example: by replacing output operations (file or console) with memory operations.

4.5 Testing the IO-Base library

Description

When porting has been completed, the IO-Base library must be tested on the target operating system.

Procedure

Test the individual blocks of functions of the IO-Base library in the specified order:

Step	Description
1	Test the controller functionality For this purpose, install the "pniocasy" demo application and the associated configuration. To do this, you will need the structure specified in the configuration.
2	Test the device functionality For this purpose, install the "pniodevice" demo application. To do this, you will need an additional IO controller.

Description of porting the Layer 2 library

This chapter explains the functionality of the Layer 2 interface and how to port it to your target operating system.

5.1 Requirements for the target operating system

Required operating system functionality

The Layer 2 library requires the following operating system functionality:

- Threads
- Mutexes
- Semaphores
- Standard C/C++ libraries

5.2 How the Layer 2 library works

Overview

The Layer 2 library provides the application with send and receive functionality for Ethernet packets.

To port the Layer 2 library, the functions responsible for communication between the Layer 2 interface and the driver must be adapted.

5.3 Directory structure and files

Description

The source files and header for the Layer 2 library are in the "l2lib" directory and have the following structure:



Description of the module directory content

The table below lists the directories and explains their content.

Directory	Contents
csd	Make files
src	Source files
inc	Header files

Files to be ported

The table below shows the files that are platform specific and have to be adapted for porting. The Layer 2 library was implemented in C++ and uses the standard C/C++ libraries.

Library files	Purpose of the individual files
os.h os_linux.h	These files contain macro templates that must be filled with operating system functions, for example creation of mutexes, events and semaphores, signaling of events.
l2eth_base.cpp	Communication with the driver

5.4 Porting the Layer 2 library step-by-step

General

Porting requires a C/C++ development environment with the standard C/C++ libraries. You perform porting in two steps:

Step	Description
1	Port the trace module if porting of the IO Base library was skipped. This module was ported during porting of the IO-Base library. If you have not yet ported the IO-Base library, refer to Section "Stage 1: Porting the trace module (Page 45)".
2	Port the Layer 2 library link to the driver. The file "l2eth_base.cpp" contains the code for communication with the driver. This is the only file of the Layer 2 library that needs to be ported. Only the calls "fopen", "fclose", "write", "read" and "ioctl" need to be ported to the target system.

5.5 Testing the Layer 2 library

Description

When porting has been completed, the Layer 2 library must be tested on the target operating system.

Procedure

Test the Layer 2 library with the "l2eth_ping" sample user program:

Step	Description
1	Connect the communications processor to a test network.
2	Activate the Ping service.
3	Attempt to reach a network partner with the "l2eth_ping" call. "l2eth_ping" requires both the IP number as well as the MAC address of a network partner. The call for the test program is as follows: l2eth_ping -i <IP number> -m <MAC address> <IP number>:IP address of the network partner in the format xxx.xxx.xxx.xxx <MAC address>:MAC address of the network partner in the format yy:yy:yy:yy:yy:yy x stands for a decimal digit (example: 192.168.10.1) y stands for a hexadecimal digit (example: 00:0A:CF:01:02:D2)

Description of the "cp16xxtest" program

Overview

This chapter describes the functionality of the "cp16xxtest" program and explains how to port it to a target operating system.

The program allows the firmware to be reset and restarted and diagnostic information to be saved.

When porting, the functions for communication between the program and the driver need to be adapted.

Note

The "cp16xxtest" program supports commissioning and troubleshooting of the module and must therefore always be ported to the target system.

6.1 Directory structure and files

Description

The source files and headers for the "cp16xxtest" program can be found in the "Examples\testapps" directory.

Files to be exported

The table below shows the files that are platform-specific and have to be adapted for porting.

Library files	Purpose of the individual files
<ul style="list-style-type: none"> • driver\inc\os.h • driver\inc\cp16xx.h • traceinfo.h 	These files contain macro templates that must be filled with operating system functions, for example: creation of mutexes, events and semaphores, signaling of events.
cp16xxtest.c	Communication with the driver

6.2 Porting the "cp16xxtest" program

Porting the "cp16xxtest" program requires a C/C++ development environment with the standard C/C++ libraries.

The program consists of the file "cp16xxtest.c".

The file executes the calls of the "cp16xx" driver and the standard IO calls with:

- fopen
- fclose
- read
- write
- ioctl

Note

If the standard IO calls do not exist in the operating system, they will need to be implemented.

6.3 Testing the "cp16xxtest" Program

Description

After porting the "cp16xxtest" program, test it with the target operating system.

Procedure

Follow the steps outlined below:

Step	Description
1	Check that the CP 1616 has the current firmware.
2	Call the "cp16xxtest" program without specifying arguments. Reaction :The program displays all available arguments.
3	Make sure that no user applications is active. Call the "cp16xxtest" program with the "re-set" argument. All LEDs on the Ethernet ports of the module light up permanently for several seconds. The LEDs then display the link or activation status (normal mode) again.

L2 - Quick start with the Layer -2 interface

Introduction

This chapter presents a recommended step-by-step procedure for creating a user program in the C/C++ programming language based on the Layer 2 Base user programming interface.

Procedure

By following the steps below, you can create a Layer 2 user program quickly and effectively.

Step	Description
1	Familiarize yourself with the following files. You then know what support has been supplied and how you can use it. The subfolder contains the following: <ul style="list-style-type: none"> • Readme files with additional information and the latest modifications • C header files of the Layer 2 user programming interface as described in Section "Software architecture (Page 57)". • Sample program
2	Familiarize yourself with the basic characteristics of the Layer 2 interface. You can do this by reading Section "How a typical Layer 2 user program runs (Page 59)" in this manual.
3	Work through the source text of the sample program and check the meanings of the functions and data structures in Section "L2 - Description of the Layer 2 functions and data types (Page 65)".
4	Change the supplied sample program to suit your purposes. In this way, you will get to know important techniques and no longer need to develop them yourself. Compile and bind the modified sample program and then test it.
5	Now create your layer 2 user program covering the entire functionality.

L2 - Overview of the Layer 2 interface

This chapter explains the basic characteristics of the Layer 2 user programming interface to prepare you for creating your own Layer 2 user program.

Function calls and data access are described in detail in the Section "L2 - Description of the Layer 2 functions and data types (Page 65)".

The Layer 2 interface allows a user program direct access to the Data Link Layer of the CP 1616 or CP 1604 module. This makes it possible to configure the Layer 2 interface, to query its status and to send and receive Ethernet frames.

8.1 How a typical Layer 2 user programming interface is used

Description

A typical use of the Layer 2 interface is the binding of network protocols (for example TCP/IP) of the host system to the CP 1616 or CP 1604 module.

If your operating system has its own standard Layer 2 interface, it is possible to link the layer 2 interface described here to it. This allows the existing network protocols to be used without any further adaptation. For the NDIS interface in Windows, there is already a link in the Windows-specific part of the driver.

The implementation of your own network protocols is also a typical application.

The Layer 2 interface is suitable neither for RT nor IRT communication (PROFINET) nor for LLDP.

The Layer 2 interface is designed for optimum performance and guaranteed reaction time. When being operated at the same time as PROFINET IO, the reaction time of layer 2 functions can be impaired briefly when IO devices drop out and restart again.

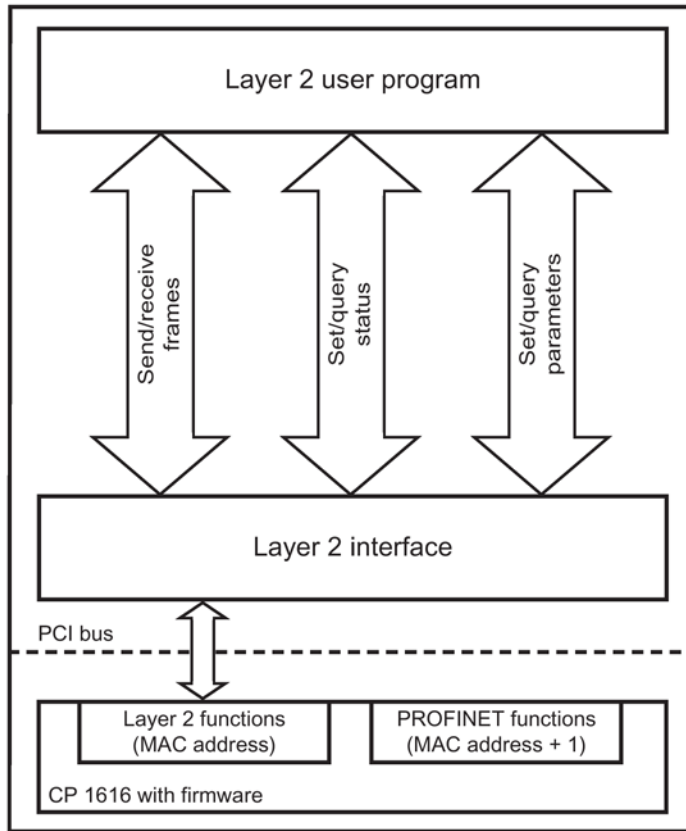
8.2 Software architecture

Layer 2 architecture along with CP 1616/CP 1604

With the functions of the Layer 2 interface, it is possible to configure the interface, to query its status and to send and receive Ethernet frames.

The CP 1616 or CP 1604 module has a separate MAC address for communication over the Layer 2 interface. This is different from the MAC address used by the PROFINET functions of the module. This ensures the logical separation of the Layer 2 interface from the PROFINET functions.

The Layer 2 interface uses the MAC address printed on the module, the PROFINET functions on the module use the next higher MAC address.



Header files and C modules

The Layer 2 user programming interface is a C programming interface consisting of several files (C module and header).

To use the Layer 2 user programming interface, you require the following files:

File type	File name	Purpose
Header file	l2eth_user.h	Declaration of the Layer 2 interface functions
Header file	l2eth_errs.h	Definition of the error codes
C module	l2eth_user.c	Implementation of the Layer 2 interface functions

8.3 How a typical Layer 2 user program runs

Overview

The typical sequence of a Layer 2 user program can be divided into 3 phases.

- Initialization phase
- Send and receive operation
- Completion phase

Some functions of the Layer -2 interface are implemented using callback functions that are registered during initialization (see Section "Callback mechanism" (Page 63)).

These are explained in detail below.

8.3.1 Initialization phase

Description

The initialization phase is divided into the following steps:

Step 1

- Action:** l2eth_open()
- Purpose:**
- Select CP.
 - Initialize interface.
 - Register callback functions for:
 - Receiving frames
 - Acknowledgment for completed sending of frames.
 - Status change
 - Change of operating mode

Step 2

- Action:** If necessary:
l2eth_get_information()
- Purpose:**
- Checks whether the interface has a link to the next station.
 - Gets maximum frame length.

Step 3

Action: If necessary:
l2eth_set_information()
Purpose: Makes settings for multicast addresses.

Step 4

Action: l2eth_set_mode()
Purpose: Sets ONLINE mode: "Mode=L2ETH_ONLINE"

Step 5

Action: Wait for:
L2ETH_CBF_MODE_COMPL()
Purpose: Waits until the L2ETH_ONLINE mode is set - From this point on, all other callback functions can be called; sending and receiving is possible.

8.3.2 Send data

Overview

The following steps are necessary for sending a frame:

Step 1

Action: l2eth_allocate_packet()
Purpose: Reserve send job buffer (L2ETH_PACKET structure) and send buffer in memory of the host system.

Step 2

Action: Set L2ETH_PACKET structure
Purpose: Set up send job buffer

Step 3

Action: Set send data
Purpose: Set up send frame

Step 4

Action: l2eth_send()
Purpose: Send packet.

Step 5

Action: Wait for L2ETH_CBF_SEND_COMPL()
Purpose: Check whether the frame was sent.

Step 6

Action: l2eth_free_packet()
Purpose: Release the send job buffer

... etc.

8.3.3 Receive data

Overview

The following steps are necessary for receiving a frame:

Step 1

Action: Wait for L2ETH_CBF_RECEIVE_IND()
Purpose: Detects that a frame was received.

Step 2

Action: Evaluate L2ETH_PACKET structure
Purpose: Evaluate receive job buffer.

Step 3

Action: Evaluate received data
Purpose: Copy or process received data.

Step 4

Action: If necessary: `l2eth_return_packet()`
Purpose: Depending on the return value of `L2ETH_CBF_RECEIVE_IND()`, return the receive job buffer to the interface.

8.3.4 Completion phase

Description

The completion phase consists of the following steps:

Step 1

Action: `l2eth_set_mode()`
Purpose: Sets OFFLINE mode: "Mode=L2ETH_OFFLINE"

Step 2

Action: Waiting for:
`L2ETH_CBF_MODE_COMPL()`
Purpose: Waits until the L2ETH_OFFLINE mode is set. From this point on, no further callback functions will be called, sending and receiving is deactivated.

Step 3

Action: If necessary `l2eth_return_packet()`
Purpose: Return any receive job buffers not yet returned.

Step 4

Action: If necessary `l2eth_free_packet()`
Purpose: Return send job buffers that are still reserved.

Step 5

Action: `l2eth_close()`
Purpose: Release internal resources, deregister user programs.

8.4 Callback mechanism

How it works

Callback functions are specified by the Layer 2 user program. A callback function can be given any name.

A callback function is called by the Layer 2 interface due to an asynchronous event. The sequence of the user program is interrupted and the callback function is started in a separate thread. This means that synchronization techniques are necessary.

Coordinating the sequence of callbacks

A callback function can interrupt the Layer 2 user program at any time. Callback functions can also interrupt each other. A callback function must therefore be designed for simultaneous, multiple execution (reentrant) since it can be called from different threads. In practical terms, this means that writing and reading of shared variables must be protected by synchronization mechanisms.

Avoid waiting in callback functions, particularly when entering critical sections. A renewed call for this callback function would be blocked. Instead, you should, where possible, use a separate database.

Note

Include multithreading standard libraries when you compile your user program

Note

All functions except the callback function itself can be called within a callback function. This means, for example, that the functions of the Layer 2 user programming interface described here can be called!

L2 - Description of the Layer 2 functions and data types

9

This section describes the individual functions of the Layer 2 user programming interface in detail and the data types used.

The chapter is primarily intended as a source of reference when you are writing your Layer - 2 user program.

9.1 l2eth_open (register with Layer 2 interface)

Description

This function registers the user program and its callback functions with the Layer 2 interface and initializes the interface. This function selects the CP via which communication will take place.

All callback functions must be specified; in other words none of the function pointers may be NULL.

If successful, a valid handle is returned. The supplied handle is required for all further function calls to identify the user programs.

On completion of this function, the Layer 2 interface is in the L2ETH_OFFLINE mode.

Syntax

```
L2ETH_UINT32          l2eth_open(  
L2ETH_UINT32          CpIndex,          /* in */  
L2ETH_CBF_RECEIVE_IND CbfReceiveInd,   /* in */  
L2ETH_CBF_SEND_COMPL CbfSendCompl,     /* in */  
L2ETH_CBF_STATUS_IND CbfStatusInd,     /* in */  
L2ETH_CBF_MODE_COMPL CbfModeCompl,     /* in */  
L2ETH_UINT32          /* out */  
);
```

Parameters

Name	Description
CpIndex	Number of the module via which communication will take place. Currently only the value 1 is accepted.
CbfReceiveInd	Pointer to a callback function that is called by the Layer 2 interface when a receive the job is reported to the application.

9.2 l2eth_set_mode (set operating mode)

Name	Description
CbfSendCompl	Pointer to a callback function that is called by the Layer 2 interface when a send job has been processed.
CbfStatusInd	Pointer to a callback function that is called by the Layer 2 interface when a status change is reported to the application.
CbfModeCompl	Pointer to a callback function called by the Layer 2 interface when the job for changing the mode has been processed.
pHandle	Pointer to a handle - The handle that is assigned to the registered communication channel is returned to the user program if the action was successful. This must be included in all further function calls.

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_INTERNAL
- L2ETH_ERR_MAX_REACHED
- L2ETH_ERR_NO_FW_COMMUNICATION
- L2ETH_ERR_NO_RESOURCE
- L2ETH_ERR_PRM_CP_ID
- L2ETH_ERR_PRM_PCBF

9.2 l2eth_set_mode (set operating mode)

Description

This function activates (L2ETH_ONLINE) or deactivates (L2ETH_OFFLINE) receiving and sending over the Layer 2 interface. It must be called after the l2eth_open() call to activate the interface and before the l2eth_close() call to deactivate the interface.

If this function is completed successfully with the return value L2ETH_OK, the change to the new mode or any possible error when changing the mode are signaled by calling the L2ETH_CBF_MODE_COMPL() callback function.

If the function is not completed with the return value L2ETH_OK, the L2ETH_CBF_MODE_COMPL() callback function is not called.

This function may only be called again when a previous l2eth_set_mode() call was acknowledged by calling the L2ETH_CBF_MODE_COMPL() callback function.

If a mode is set that is already active, this is not an error and this mode is signaled by calling the L2ETH_CBF_MODE_COMPL() callback function.

Syntax

```
L2ETH_UINT32      l2eth_set_mode(  
    L2ETH_UINT32  Handle,                /* in */  
    L2ETH_MODE    Mode    /* in */  
);
```

Parameters

Name	Description
Handle	Handle from l2eth_open()
Mode	Operating mode L2ETH_ONLINE or L2ETH_OFFLINE

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_PRM_HND
- L2ETH_ERR_PRM_MODE
- L2ETH_ERR_NO_FW_COMMUNICATION
- L2ETH_ERR_INTERNAL

9.3 L2ETH_CBF_MODE_COMPL (signal operating mode)

Description

This function, that must be provided by the Layer 2 user program, is called by the Layer 2 interface when a job to change the mode of the Layer 2 interface has been processed. Such jobs result from calling the l2eth_set_mode() function.

If the new mode has then set successfully, the value L2ETH_OK is returned in the "Error" error code. The "Mode" parameter specifies the new mode.

If an error occurred when setting the new mode, an error code other than L2ETH_OK is returned in the "Error" error code. In this case, the "Mode" parameter contains the originally required mode that could not be set.

It is guaranteed that all other callback functions are called only after the L2ETH_ONLINE mode has been reached.

It is also guaranteed that no further callback functions will be called after reaching the L2ETH_OFFLINE mode.

9.4 L2ETH_CBF_STATUS_IND (signal status)

Syntax

<pre>typedef void (* L2ETH_CBF_MODE_COMPL)(L2ETH_UINT32 Handle L L2ETH_MODE Mode, L2ETH_UINT32 Error);</pre>	<pre>/* in */ /* in */ /* in */</pre>
-------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------

Parameters

Name	Description
Handle	Handle from l2eth_open()
Mode	Required mode when calling l2eth_set_mode()
Error	Return values as in Section "Error codes (Page 78)"; the following are possible: <ul style="list-style-type: none"> • L2ETH_OK • L2ETH_ERR_NO_RESOURCE • L2ETH_ERR_NO_FW_COMMUNICATION • L2ETH_ERR_INTERNAL

Return values

None

9.4 L2ETH_CBF_STATUS_IND (signal status)

Description

This function, that must be provided by the Layer 2 user program, is called by the Layer 2 interface when the status of the Layer 2 interface has changed (indicated by "Oid"). Currently, this callback function is called only when the link of a port has changed (Oid=L2ETH_OID_MEDIA_CONNECT_STATUS).

The current value of this status can be queried with l2eth_get_information().

This callback function is called only when the Layer 2 interface is in the L2ETH_ONLINE mode.

Syntax

<pre>typedef void (* L2ETH_CBF_STATUS_IND)(L2ETH_UINT32 Handle, L2ETH_OID Oid);</pre>	<pre>/* in */ /* in */</pre>
---------------------------------------------------------------------------------------------------------	------------------------------

Parameters

Name	Description
Handle	Handle from l2eth_open()
Oid	Object ID to indicate the changed status

Return values

None

9.5 l2eth_get_information (query parameters)

Description

This function can be used to query settings and statuses of the Layer 2 interface. All settings and statuses can be read that are described in the Section "L2ETH_OID (type for object identifier) (Page 83)" with the "Read" type of access.

In particular, this function can be used to query a changed status that was signaled with the L2ETH_CBF_STATUS_IND() callback function.

The returned buffer in the job structure only contains value data when the return value of the function is "L2ETH_OK".

If the buffer in the job structure is too small to store the requested object data, the return value L2ETH_ERR_NO_RESOURCE is returned. In this case, the "BytesNeeded" value in the structure contains the required buffer length. The job can be repeated with a suitably large buffer.

This function can be called in every operating mode of the Layer 2 interface (L2ETH_ONLINE or L2ETH_OFFLINE).

Syntax

L2ETH_UINT32	l2eth_get_information(
L2ETH_UINT32	Handle,	/* in */
L2ETH_QUERY	*pQuery	/* out */
);	

Parameters

Name	Description
Handle	Handle from l2eth_open()
pQuery	Pointer to the job structure

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_INTERNAL
- L2ETH_ERR_NO_FW_COMMUNICATION
- L2ETH_ERR_NO_RESOURCE
- L2ETH_ERR_PRM_BUF
- L2ETH_ERR_PRM_HND
- L2ETH_ERR_PRM_LEN
- L2ETH_ERR_PRM_OID
- L2ETH_ERR_PRM_QUERY
- L2ETH_ERR_SEQUENCE

9.6 l2eth_set_information (set parameters)

Description

This function can be used to set parameters of the Layer 2 interface. All the parameters described in Section "L2ETH_OID (type for object identifier) (Page 83)" with the "Write" type of access can be written.

This function can be called in every operating mode of the Layer 2 interface (L2ETH_ONLINE or L2ETH_OFFLINE).

Syntax

<pre>L2ETH_UINT32 l2eth_set_information(L2ETH_UINT32 Handle, L2ETH_QUERY *pQuery);</pre>	<pre>/* in */ /* in */</pre>
------------------------------------------------------------------------------------------------------	------------------------------

Parameters

Name	Description
Handle	Handle from l2eth_open()
pQuery	Pointer to the job structure

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_INTERNAL
- L2ETH_ERR_NO_FW_COMMUNICATION
- L2ETH_ERR_NO_RESOURCE
- L2ETH_ERR_OID_READONLY
- L2ETH_ERR_PRM_BUF
- L2ETH_ERR_PRM_HND
- L2ETH_ERR_PRM_LEN
- L2ETH_ERR_PRM_OID
- L2ETH_ERR_PRM_QUERY
- L2ETH_ERR_SEQUENCE

9.7 l2eth_allocate_packet (reserve send job)

Description

This function reserves a send job buffer. The user fills this buffer and transfers it to the l2eth_send() function.

This function can be called in every operating mode of the Layer 2 interface (L2ETH_ONLINE or L2ETH_OFFLINE).

In concrete terms, this means that the "L2ETH_PACKET" structure and the buffer for the Ethernet frame is reserved with the maximum frame length. The "pBuffer" pointer in the L2ETH_PACKET structure that points to the Ethernet frame is set accordingly.

The user sets the "Context" value, the length of the Ethernet frame "Data Length" (without FCS) and the Ethernet frame itself and transfers the send job buffer to the "l2eth_send()" function.

Syntax

<pre>L2ETH_UINT32 l2eth_allocate_packet(L2ETH_UINT32 Handle, L2ETH_PACKET * *ppPacket);</pre>	<pre>/* in */ /* out */</pre>
--------------------------------------------------------------------------------------------------------	-------------------------------

Parameters

Name	Description
Handle	Handle from l2eth_open()
ppPacket	Pointer to the pointer of the reserved send job buffer; see structure in Section "L2ETH_PACKET (job type for sending and receiving) (Page 80)".

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_INTERNAL
- L2ETH_ERR_NO_RESOURCE
- L2ETH_ERR_PRM_HND
- L2ETH_ERR_PRM_PKT

9.8 l2eth_send (send data)

Description

This function sends frames.

When this function is completed successfully with the return value L2ETH_OK, the result of the send job is signaled by calling the L2ETH_CBF_SEND_COMPL() callback function.

The send job buffer remains occupied until it is returned again with the L2ETH_CBF_SEND_COMPL() callback function. Until then, the user program must not modify the buffer.

If the function is **not** completed with the return value L2ETH_OK, the L2ETH_CBF_SEND_COMPL() callback function is not called.

In this case, the send job buffer is not occupied.

The "Context" value in the "pPacket" send job buffer can be set to any value.

It is returned unchanged in L2ETH_CBF_SEND_COMPL(). This allows a reference to the send job to be established.

This function can be called several times in sequence without waiting for the L2ETH_CBF_SEND_COMPL() callback function to be called.

This function l2eth_send() can be called several times consecutively. Make sure, however, that at least one packet reserved with "l2eth_allocate_packet()" is always available. A maximum of 39 packets can be reserved at the same time.

This function can be called only when the Layer 2 interface is in the L2ETH_ONLINE mode.

Syntax

L2ETH_UINT32	l2eth_send(
L2ETH_UINT32	Handle,	/* in */
L2ETH_PACKET	*pPacket	/* in */
);	

Name	Description
Handle	Handle from l2eth_open()
pPacket	Pointer to the send job buffer; see structure in Section "L2ETH_PACKET (job type for sending and receiving) (Page 80)".

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_INTERNAL
- L2ETH_ERR_LLDP_FRAME
- L2ETH_ERR_NO_FW_COMMUNICATION
- L2ETH_ERR_NO_RESOURCE
- L2ETH_ERR_PRM_BUF
- L2ETH_ERR_PRM_HND
- L2ETH_ERR_PRM_LEN
- L2ETH_ERR_PRM_PKT
- L2ETH_ERR_SEQUENCE

9.9 L2ETH_CBF_SEND_COMPL (signal send result)

Description

This function, that must be provided by the Layer 2 user program, is called by the Layer 2 interface when the sending of a frame is completed.

The reference to the "pPacket" send job can be established by the "Context" value in the send job buffer. The "Context" value is returned unchanged; see structure in Section "L2ETH_PACKET (job type for sending and receiving) (Page 80)".

The result of the send job is signaled using the "Error" error code.

This callback function is called only when the Layer 2 interface is in the L2ETH_ONLINE mode.

9.10 l2eth_free_packet (release send job)

Syntax

<pre>typedef void (*L2ETH_CBF_SEND_COMPL)(L2ETH_UINT32 Handle, L2ETH_PACKET *pPacket, L2ETH_UINT32 Error);</pre>	<pre>/* in */ /* in */ /* in */</pre>
--------------------------------------------------------------------------------------------------------------------------------	---------------------------------------

Parameters

Name	Description
Handle	Handle from l2eth_open()
pPacket	Pointer to the send job buffer; see structure in Section "L2ETH_PACKET (job type for sending and receiving) (Page 80)".
Error	Error code of the send job as listed in Section "Error codes (Page 78)"; the following are possible: <ul style="list-style-type: none"> • L2ETH_OK • L2ETH_ERR_NO_RESOURCE • L2ETH_ERR_NO_FW_COMMUNICATION • L2ETH_ERR_INTERNAL • L2ETH_ERR_LLDP_FRAME

Return values

None.

9.10 l2eth_free_packet (release send job)

Description

This function releases a send job buffer again that was reserved with l2eth_allocate_packet().

It can be called in every operating mode of the Layer 2 interface (L2ETH_ONLINE or L2ETH_OFFLINE).

Syntax

<pre>L2ETH_UINT32l2eth_free_packet(L2ETH_UINT32Handle, L2ETH_PACKET*pPacket);</pre>	<pre>/* in */ /* in */</pre>
-----------------------------------------------------------------------------------------------	------------------------------

Parameters

Name	Description
Handle	Handle from l2eth_open()
pPacket	Pointer to the send job buffer; see structure in Section "L2ETH_PACKET (job type for sending and receiving) (Page 80)".

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_PRM_HND
- L2ETH_ERR_PRM_PKT
- L2ETH_ERR_PRM_BUF
- L2ETH_ERR_INTERNAL

9.11 L2ETH_CBF_RECEIVE_IND (signal receipt of data)

Description

This function, that must be provided by the Layer 2 user program, is called by the Layer 2 interface when a frame was received.

The Layer 2 user program now has two options with which to react to receipt of the frame:

- The received packet is processed immediately and returned to the Layer 2 interface when the callback function is exited.
- The received packet is retained by the callback function of the Layer 2 user program for later processing.

As soon as the processing of this packet is completed by the Layer 2 user program, the package must be returned explicitly to the Layer 2 interface with the l2eth_return_packet() function.

The Layer 2 interface is informed which of these two options is selected by the Layer 2 user program in the return value of the callback function; refer to the description of the return value below.

The "Context" value in the "pPacket" receive job buffer is reserved and must not be modified by the user program; see structure in Section "L2ETH_PACKET (job type for sending and receiving) (Page 80)".

The L2ETH_CBF_RECEIVE_IND callback function is called only when the Layer 2 interface is in the L2ETH_ONLINE mode.

9.12 l2eth_return_packet (return receive job)

Syntax

typedef L2ETH_UINT32 (* L2ETH_CBF_RECEIVE_IND)(L2ETH_UINT32Handle, L2ETH_PACKET*pPacket);	/* in */ /* in */
---------------------------------------------------------------------------------------------------------	----------------------

Parameters

Name	Description
Handle	Handle from l2eth_open()
pPacket	Pointer to the send job buffer; see structure in Section "L2ETH_PACKET (job type for sending and receiving) (Page 80)".

Return values

The Layer 2 user program must set the return value when it exits this function.

The return value specifies the number of following l2eth_return_packet() calls by the Layer 2 user program. Currently only the following return values are supported:

Return Value	Description
0	The receive job buffer "pPacket" is no longer required by the Layer 2 user program and there is therefore no "l2eth_return_packet()" call. The receive job buffer is therefore no longer available for the Layer 2 user program.
1	The receive job buffer "pPacket" is processed by the Layer 2 user program and is returned later with the l2eth_return_packet() call. Until then, the receive job buffer is reserved for the Layer 2 user program.

9.12 l2eth_return_packet (return receive job)

Description

This function returns a receive job buffer when the data of the received frame has been processed or copied. The call must be made dependent on the return value of the L2ETH_CBF_RECEIVE_IND() callback function.

This function can be called only when the Layer 2 interface is in the L2ETH_ONLINE mode.

Syntax

L2ETH_UINT32l2eth_return_packet(L2ETH_UINT32Handle, L2ETH_PACKET*pPacket);	/* in */ /* in */
---------------------------------------------------------------------------------------	----------------------

Parameters

Name	Description
Handle	Handle from l2eth_open()
pPacket	Pointer to the send job buffer; see structure in Section "L2ETH_PACKET (job type for sending and receiving) (Page 80)".

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_PRM_HND
- L2ETH_ERR_PRM_PKT
- L2ETH_ERR_PRM_BUF
- L2ETH_ERR_SEQUENCE
- L2ETH_ERR_INTERNAL

9.13 l2eth_close (deregister from Layer 2 interface)**Description**

This function deregisters the user program from the Layer 2 interface. All internal resources are released.

All callback functions are deregistered.

After processing this function, the handle is invalid and must not be used any longer.

This function can be called only when the Layer 2 interface is in the L2ETH_OFFLINE mode.

Syntax

L2ETH_UINT32l2eth_close(L2ETH_UINT32Handle);	//in
------------------------------------------------------	------

9.14 Data types

Parameters

Name	Description
Handle	Handle from l2eth_open()

Return values

Return values as in Section "Error codes (Page 78)"; the following are possible:

- L2ETH_OK
- L2ETH_ERR_INTERNAL
- L2ETH_ERR_NO_FW_COMMUNICATION
- L2ETH_ERR_PRM_HND

9.14 Data types

Description

The following data types are used in the Layer 2 interface.

9.14.1 Basic data types

Description

The following table describes the basic data types used by the Layer 2 interface.

Action	Description
L2ETH_UINT8	unsigned char (8 bits)
L2ETH_UINT32	unsigned long (32 bits)

9.14.2 Error codes

Description

The following table describes all the possible error codes of the layer 2 interface functions. The values are defined in the "l2eth_errs.h" include file.

The error codes are of the basic data type L2ETH_UINT32.

Error codes

Name	Description
L2ETH_ERR_INTERNAL	Internal error
L2ETH_ERR_LLDP_FRAME	The Ethernet frame is an LLDP frame. LLDP frames are not supported.
L2ETH_ERR_MAX_REACHED	Maximum number of Layer 2 channels opened.
L2ETH_ERR_NO_FW_COMMUNICATION	No connection to the firmware of the CP.
L2ETH_ERR_NO_RESOURCE	No more resources available.
L2ETH_ERR_OID_READONLY	Only reading allowed for this "Oid".
L2ETH_ERR_PRM_BUF	The "pBuffer" parameter in "pPacket" or "pQuery" is invalid.
L2ETH_ERR_PRM_CP_ID	"CpIndex" parameter is invalid.
L2ETH_ERR_PRM_HND	"Handle" parameter is invalid.
L2ETH_ERR_PRM_LEN	The length parameter in "pPacket" is invalid.
L2ETH_ERR_PRM_MODE	"Mode" parameter is invalid.
L2ETH_ERR_PRM_OID	"Oid" parameter in "pQuery" is invalid.
L2ETH_ERR_PRM_PCBF	The pointer to a callback function is missing.
L2ETH_ERR_PRM_PKT	"pPacket" parameter is invalid.
L2ETH_ERR_PRM_QUERY	"pQuery" parameter is invalid.
L2ETH_ERR_SEQUENCE	Calling this function is not permitted in the current mode.
L2ETH_OK	Function executed successfully.

9.14.3 L2ETH_MAC_ADDR (type for MAC address)**Description**

The L2ETH_MAC_ADDR data type describes a MAC address.

Syntax

```
typedef L2ETH_UINT8 L2ETH_MAC_ADDR[6];
```

Elements

Name	Description
L2ETH_MAC_ADDR	MAC address in network format (big endian)

9.14.4 L2ETH_MODE (type for operating mode)

Description

The L2ETH_MODE data type contains the IDs for the supported modes.

Syntax

```
typedef enum
{
    L2ETH_OFFLINE,
    L2ETH_ONLINE
} L2ETH_MODE;
```

Elements

Name	Description
L2ETH_OFFLINE	"Offline" mode - sending and receiving are deactivated. No callback functions are called.
L2ETH_ONLINE	"Online" mode - Sending and receiving are activated. Callback functions can be called.

9.14.5 L2ETH_PACKET (job type for sending and receiving)

Description

The L2ETH_PACKET data structure is required for send and receive jobs.

Syntax

```
typedef struct
{
    L2ETH_UINT32    DataLength;
    L2ETH_UINT8    *    pBuffer;
    L2ETH_UINT32    Context;
} L2ETH_PACKET;
```


Elements

Name	Description
DataLength	Length of the Ethernet frame (without FCS) - The value must be between 60 and 1518.
pBuffer	Pointer to an Ethernet frame complying with IEEE 802.3 - Starting with the target MAC address up to the user data.
Context	User-specific value for identifying the job - Is not modified by the Layer 2 interface.

9.14.6 L2ETH_QUERY (job type for status query and parameter assignment)**Description**

The L2ETH_QUERY data structure is required for setting and querying statuses and parameters.

Syntax

```
typedef struct
{
    L2ETH_OID      Oid;
    L2ETH_UINT8    *   pBuffer;
    L2ETH_UINT32   BufferLength;
    L2ETH_UINT32   BytesTransferred;
    L2ETH_UINT32   BytesNeeded;
} L2ETH_QUERY;
```

Elements

Name	Description
Oid	Object identifier - see Section "L2ETH_OID (type for object identifier) (Page 83)"
pBuffer	Points to the buffer containing the input or output data.
BufferLength	Number of bytes in "pBuffer"

Name	Description
BytesTransferred	When setting parameters Number of bytes in "pBuffer" that were used by the Layer 2 interface to set the parameters. When querying parameters or statuses Number of bytes in "pBuffer" returned by the Layer 2 interface to the Layer 2 user program.
BytesNeeded	This value is returned only when "BufferLength" is less than required by the transferred "Oid". In this error situation, the number of bytes required in "pBuffer" to be able to handle the request successfully is specified.

9.14.7 L2ETH_PORT_STATUS (type for port status)

Description

The L2ETH_PORT_STATUS data structure is required to query the status of a port.

Syntax

```
typedef struct
{
    L2ETH_UINT32    Link;
    L2ETH_UINT32    Bitrate;
    L2ETH_UINT32    Duplex;
} L2ETH_PORT_STATUS;
```

Elements

Name	Description	
Link	Status of the port: <ul style="list-style-type: none"> L2ETH_LINK_UP L2ETH_LINK_DOWN 	Connection to partner station exists. No connection to partner station.
Bitrate	Data transmission rate of the port(only known when Link Up): <ul style="list-style-type: none"> L2ETH_LINK_SPEED_10 L2ETH_LINK_SPEED_100 L2ETH_LINK_UNKNOWN 	10 Mbps 100 Mbps Unknown (link down)
Duplex	Duplex mode of the port (only known when Link Up): <ul style="list-style-type: none"> L2ETH_LINK_SPEED L2ETH_LINK_SPEED_100 L2ETH_LINK_UNKNOWN 	Half duplex Full duplex Unknown (link down)

9.14.8 L2ETH_OID (type for object identifier)

Description

The L2ETH_OID contains the IDs for the supported objects.

Syntax

```
typedef enum
{
    L2ETH_OID_PERMANENT_ADDRESS,
    L2ETH_OID_MAXIMUM_LIST_SIZE,
    L2ETH_OID_MULTICAST_LIST,
    L2ETH_OID_MAXIMUM_FRAME_SIZE,
    L2ETH_OID_MEDIA_CONNECT_STATUS
} L2ETH_OID;
```

L2ETH_OID_PERMANENT_ADDRES

Description: Reads the local MAC address
Access: Read only
BufferLength: 6 bytes
pBuffer: L2ETH_MAC_ADDR

L2ETH_OID_MAXIMUM_LIST_SIZE

Description: Reads the maximum number of multicast addresses
Access: Read only
BufferLength: 4 bytes
pBuffer: L2ETH_UINT32

L2ETH_OID_MULTICAST_LIST

Description: Reads and writes the multicast addresses
Access: Read and write
BufferLength: n x 6 bytes
pBuffer: Field from L2ETH_MAC_ADDR

L2ETH_OID_MAXIMUM_FRAME_SIZE

Description: Reads and sets the maximum length of the Ethernet frames (without FCS)
Access: Read and write
BufferLength: 4 bytes
pBuffer: L2ETH_UINT32

L2ETH_OID_MEDIA_CONNECT_STATUS

Description: Reads the link status of all ports
Access: Read only
BufferLength: 12 bytes for each port
pBuffer: Field from L2ETH_PORT_STATUS
(field index + 1 corresponds to the port number on the module)

L2 - Creating a Linux Ethernet driver

This chapter explains the basics of developing a Linux network driver when using the Layer 2 interface.

10.1 Basics of developing a Linux Ethernet driver based on the Layer 2 functions

Basic task

To create a Linux Ethernet driver as a dynamically loadable kernel module. It must register in the Ethernet interface of Linux.

Properties of the Linux Ethernet driver

The Linux Ethernet driver must have an interface defined by Linux.

If you use the Layer 2 functions, development only involves the development of the functions:

- Map the Linux interface to the Layer 2 interface
- Call kernel functions when status changes are received and receive Ethernet packets with callbacks of the layer 2 interface

Interfaces of the Linux Ethernet driver to Linux

The Linux Ethernet driver must provide the following interfaces:

- Module interface
- Ethernet port

Interfaces of the Linux Ethernet driver to the Layer 2 interface

Interfaces of the Linux Ethernet driver to the Layer 2 interface are the callback functions that must be specified when the `l2eth_open()` function is called:

`L2ETH_CBF_MODE_COMPL()`

`L2ETH_CBF_STATUS_IND()`

`L2ETH_CBF_SEND_COMPL()`

`L2ETH_CBF_RECEIVE_IND()`

10.1.1 Interfaces of the Linux Ethernet driver to Linux

Module interface

The module interface is used to initialize and are registered the Linux Ethernet driver. Implement the following functions in your Linux Ethernet driver; they are called by the Linux kernel: The module interface is used to initialize and are registered the Linux Ethernet driver. Implement the following functions in your Linux Ethernet driver; they are called by the Linux kernel:

Module Function	Description
<code>int module_init(void)</code>	The "module_init()" is called when the Linux Ethernet driver is loaded. Implement the following functionality in it: <ul style="list-style-type: none">• Register with Linux as an Ethernet module with the "alloc_netdev()" and "register_netdev()" functions.• Register with the Layer 2 interface with the "l2eth_open()" function.• Query the MAC address and the maximum number of supported multicast addresses on the Layer 2 interface with the "l2eth_get_information()" function.
<code>int module_exit(void)</code>	The "module_exit()" function is called when the Linux Ethernet driver is removed. Implement the following functionality in it: <ul style="list-style-type: none">• Terminate or connections with the "l2eth_close()" function.• Deregister from the kernel in Linux with the "unregister_netdev()" and "free_netdev()" functions.

Ethernet port

The Ethernet interface for sending and receiving Ethernet frames. Implement the following functions in your Linux Ethernet driver; they are called by the Linux kernel:

Ethernet functions	Description
<pre>int (*open)(struct net_device *dev)</pre>	<p>This function is called when the Ethernet interface is activated. Implement the following functionality in it:</p> <ul style="list-style-type: none"> • Activate receive mode on the Layer 2 interface with the <code>l2eth_set_mode(L2ETH_ONLINE)</code> function. • If successful, use the <code>netif_start_queue()</code> function to indicate that packets can be sent. • Read out the link status of the ports on the Layer 2 interface with the <code>l2eth_get_information()</code> function and register it with Linux with <code>netif_carrier_on()</code> or <code>netif_carrier_off()</code>.
<pre>int (*stop)(struct net_device *dev)</pre>	<p>This function is called when the Ethernet interface is deactivated. Implement the following functionality in it:</p> <ul style="list-style-type: none"> • Deactivate receive mode on the Layer 2 interface with the <code>l2eth_set_mode(L2ETH_OFFLINE)</code> function. • Signal the deactivated status to the kernel with the <code>netif_stop_queue()</code> function.
<pre>int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev)</pre>	<p>This function is called when an Ethernet packet needs to be sent. Implement the following functionality in it:</p> <ul style="list-style-type: none"> • For the Layer 2 interface, reserve a send buffer with the <code>l2eth_allocate_packet()</code> function and copy the send data from <code>skb</code>. • Send the send buffer with <code>l2eth_send()</code>. • Release the send buffer in <code>skb</code> with <code>dev_kfree_skb()</code>.
<pre>int (change_mtu)(struct net_device *dev, int new_mtu)</pre>	<p>This function is called when the MTU size needs to be changed. Use the <code>l2eth_set_information()</code> function for this purpose.</p>
<pre>void (*set_multicast_list)(struct net_device *dev)</pre>	<p>This function is called when the multicast list needs to be changed. Use the <code>l2eth_set_information()</code> function for this purpose.</p>

10.1.2 Interfaces of the Linux Ethernet driver to the Layer 2 interface

Description

To allow the Layer 2 interface to signal status changes, transfer received Ethernet frames and return the send buffer, the Ethernet driver must provide the following callback functions:

Callback functions	Description
L2ETH_CBF_MODE_COMPL()	Returns the result for the previously called "l2eth_set_mode()" function.
L2ETH_CBF_STATUS_IND()	This function is called when the status of the Layer 2 interface changes. Implement the following functionality in it: <ul style="list-style-type: none"> • When the link status is "L2ETH_LINK_UP", call the "netif_carrier_on()" function. • When the link status is "L2ETH_LINK_DOWN", call the "netif_carrier_off()" function.
L2ETH_CBF_SEND_COMPL()	This function is called when the Layer 2 interface returns a send buffer. Implement the following functionality in it: <ul style="list-style-type: none"> • Update the send statistics • Release the send buffer with the "l2eth_free_packet()" function.
L2ETH_CBF_RECEIVE_IND()	This function is called when the Layer 2 interface transfers a received Ethernet packet. Implement the following functionality in it: <ol style="list-style-type: none"> 1. Reserve a channel receive buffer in the kernel with the "dev_alloc_skb()" function. 2. Copy the received data to the kernel buffer. 3. Transfer the Linux kernel to the receive buffer with the "netif_rx()" function. 4. Return the Layer 2 receive buffer to the Layer 2 interface with the "l2eth_return_paket()" function.

10.1.3 Point to note when compiling

Description

Compile the Layer 2 library with the "__KERNEL__" compiler flag and bind it statically to your Ethernet driver.