SIEMENS

SIMATIC NET

PC software
DK-HN-IE PN IO Porting instructions

Programming Manual

09/2016

C79000-G8976-C439-01

Legal information

Warning notice system

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

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

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

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

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

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

Qualified Personnel

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

Proper use of Siemens products

Note the following:

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

Trademarks

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

Disclaimer of Liability

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

Siemens AG
Division Process Industries and Drives
Postfach 48 48
90026 NÜRNBERG
GERMANY

Document order number: C79000-G8976-C439-01
℗ 08/2016 Subject to change

Table of contents

Introduction

1

Trademarks

The following and possibly other names not identified by the registered trademark sign ® are registered trademarks of Siemens AG:

SIMATIC NET, HARDNET, SOFTNET, CP 1612, CP 1613, CP 5612, CP 5613, CP 5614, CP 5622

Industry Online Support

In addition to the product documentation, the comprehensive online information platform of Siemens Industry Online Support at the following Internet address:
Link: (https://support.industry.siemens.com/cs/de/en/)

Apart from news, there you will also find:

- Project information: Manuals, FAQs, downloads, application examples etc.
- Contacts, Technical Forum
- The option submitting a support query:
  Link: (https://support.industry.siemens.com/My/ww/en/requests)
- Our service offer:

  Right across our products and systems, we provide numerous services that support you in every phase of the life of your machine or system - from planning and implementation to commissioning, through to maintenance and modernization.

You will find contact data on the Internet at the following address:
Link: (http://www.automation.siemens.com/aspa_app/?ci=yes&lang=en)

SITRAIN - Training for Industry

The training offer includes more than 300 courses on basic topics, extended knowledge and special knowledge as well as advanced training for individual sectors - available at more than 130 locations. Courses can also be organized individually and held locally at your location.

You will find detailed information on the training curriculum and how to contact our customer consultants at the following Internet address:

Link: (http://sitrain.automation.siemens.com/DE/sitrain/default.aspx?AppLang=en)

Security information

Siemens provides products and solutions with industrial security functions that support the secure operation of plants, systems, machines and networks.

In order to protect plants, systems, machines and networks against cyber threats, it is necessary to implement – and continuously maintain – a holistic, state-of-the-art industrial security concept. Siemens' products and solutions only form one element of such a concept.

Customer is responsible to prevent unauthorized access to its plants, systems, machines and networks. Systems, machines and components should only be connected to the enterprise network or the internet if and to the extent necessary and with appropriate security measures (e.g. use of firewalls and network segmentation) in place.

Additionally, Siemens' guidance on appropriate security measures should be taken into account. For more information about industrial security, please visit
Link: (http://www.siemens.com/industrialsecurity)

Siemens' products and solutions undergo continuous development to make them more secure. Siemens strongly recommends to apply product updates as soon as available and to always use the latest product versions. Use of product versions that are no longer supported, and failure to apply latest updates may increase customer's exposure to cyber threats.

To stay informed about product updates, subscribe to the Siemens Industrial Security RSS Feed under
Link: (https://support.industry.siemens.com/cs/ww/en/)

1.1 Note on the SIMATIC NET glossary - DVD + Internet

SIMATIC NET glossary

Explanations of many of the specialist terms used in this documentation can be found in the SIMATIC NET glossary.

You will find the SIMATIC NET glossary here:

- SIMATIC NET Manual Collection or product DVD

  The DVD ships with certain SIMATIC NET products.

- On the Internet under the following address:

  38652101 (https://support.industry.siemens.com/cs/ww/en/view/38652101)

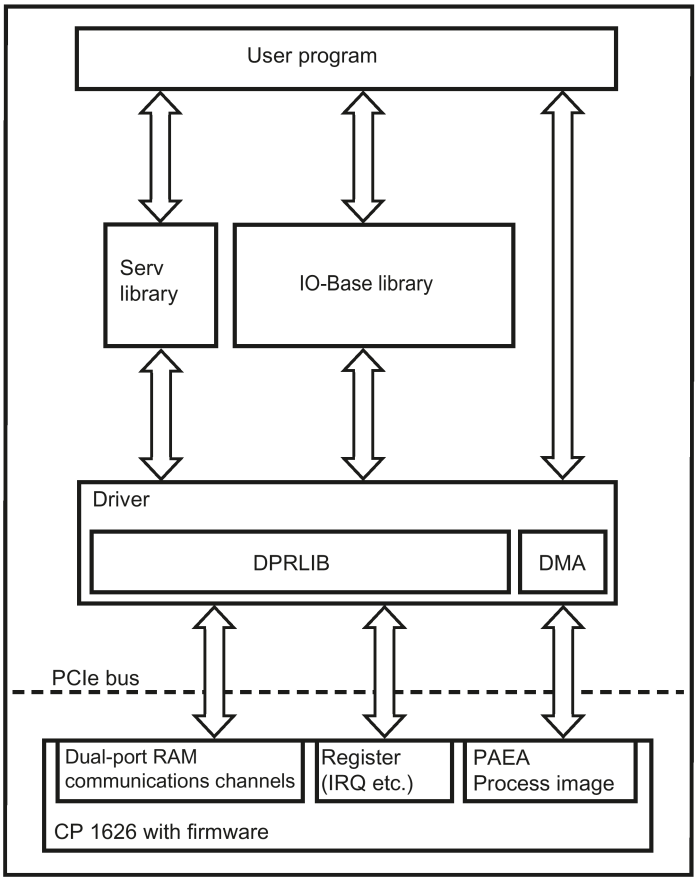
Quick Start

2

2.1 Architecture of the DK HN-IE PN IO software

Description

This document applies to the module CP 1626. The following graphic shows the software layers and communication paths of the DK HN-IE PN IO software. The following table explains the terminology used in the graphic.



Picture element	Description
IO-Base library	The IO-Base library makes the IO-Base user programming interface available. 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.
Driver	The driver performs the following functions: • Communication between the software components IO-Base library – Serv library – firmware. • Integration of the hardware resources in the operating system. All driver functions must be ported to the target operating system.
DPRLIB	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 1626 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 1626 are located.
PAEA	PAEA-RAM is the memory containing the process data of the CP 1626.
Arrows	Arrows are independent communication channels.

2.2 Installation in Linux

Introduction

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

To install the driver, the IO-Base library and the Serv 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 longer to report an interrupt to the application.

Administrator privileges

To install the driver of the IO-Base interface, 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 11.2
Kernel	As of version 2.6.10, in the example version 2.6.32.2 is used
GNU C compiler (GCC)	As of version 3.3.5
Kernel source files	Suitable for the kernel
Real-time extension RTAI, available at RTAI (www.rtai.org)	Suitable for the kernel, in the example version 3.8 is used

In terms of the size of the swap partition it is practical to adopt the specifications of the Linux installation program.

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 computer	Values
Hard disk space	At least 100 MB, with extensive configurations also more.
Processor	At least Intel Pentium 4 or higher.
RAM	At least 8 MB.

System computer	Values
Memory with DMA capability	At least 4 MB.
Interrupts	• In IRT operation: one non-shared interrupt • In operation without IRT: one interrupt, either shared or non-shared

Preparing RTAI and the Linux kernel 3

3.1 Preparing the system

So that general operation in real time is possible, the BIOS settings must first be checked and, if necessary, adapted. This applies especially to the option that allows the processor clock rate to be adapted. This setting is known as "Speedstep", "Enhanced Idle Power State", "P-States", "IST", "EIST", "Cool'n'Quite" or "PowerNow" and it must be disabled.

3.2 Generating, installing and testing real-time extension RTAI

Description

The following procedure outlines the principles underlying installation. The installation can change so you should therefore always read the installation instructions supplied for the kernel and RTAI. Follow the steps described in the following subsections.

Note

Adapt the version numbers in the paths and cmmands.

3.2.1 Downloading source files from the Internet

Description

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

Step 1

Download version 3.8 of RTAI from the Internet.

Link: (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: su
Description: Change to the user "root" with the substitute user command.

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

Select one of the supported Linux kernels (for RTAI V3.8 for example V2.6.32.2 can be recommended) and download the kernel sources from the Internet.

Link: (http://www.kernel.org)

Save the kernel sources in the /usr/src directory.

3.2.2 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 substitute user command.

Step 2

Command: cd /usr/src
Description: Go 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.
Command: bunzip2 rtai-3.5-cv.tar.bz2
tar –xf rtai-3.5-cv.tar
Description: Extract the source code of the real-time extension RTAI.

3.2.3 Configuring and generating the Linux kernel

Description

Below the configuration and generation of a Linux kernel with real time capability is described.

Step 1

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

Step 2

Command: cd /usr/src/linux-2.6.32.2
Description: Go 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 RTAI patch to the Linux source code.

Step 4

Command: cat /proc/config.gz | gunzip > .config
make oldconfig
Description: Adopt the kernel configuration from the running kernel and extend the configuration if any options are undefined. Accept all the defaults 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

Options	Value
Processor type and features → Processor family	Select the processor family most similar to your processor. (Pentium-Classic normally works with newer Intel processors) If you have a multicore processor, then only select a processor family that supports 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
Processor type and features → Support sparse irq numbering	OFF
Processor type and features → IBM Calgary IOMMU support	OFF
Processor type and features → AMD IOMMU support	OFF
Processor type and features → Allow for memory hot-add	OFF
Ipipe support → Ipipe support or Processor type and features → Interrupt pipeline	ON
Kernel hacking → Compile the kernel with frame pointers	OFF
Bus options → Support for Interrupt Remapping	OFF

Save the configuration by answering the question "Save new kernel configuration?" with "Yes" before you exit.

Step 6

Command: make clean all
Description: Compile the kernel.

3.2.4 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 PC. 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: Go 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 with for the kernel you have just installed in the Boot menu.

3.2.5 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. Follow the steps outlined below:

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 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 being used in "Machine(x86)->Number of CPU's". You will find the number in the file /proc/cpuinfo.

Step 3

Commands: make install

Description: Compile and install RTAI.

3.2.6 Checking whether the real-time extension RTAI 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 be significantly shorter than the configured cycle time if you increase the system load. This can, for example, occur when the PC mouse is moved quickly, when you type quickly on the keyboard or when other peripheral devices or the hard disk 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 for example on the Web site of the manufacturer.

Link: (http://www.rtai.org)

3.3 Basic procedure for installing the DK HN-IE PN IO software in Linux.

The section below describes the actions to be carried out when installing the driver, the IO-Base 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: Mount 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: Generation of the driver, the IO-Base and the Serv library.

This is only possible if the real-time extensions were successfully installed, see Section "Generating, installing and testing real-time extension RTAI (Page 11)".

Step 8

Command: make install

Description: Installation of the driver, the IO-Base and the Serv library and header files.

The PROFINET IO and the Serv library 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", "pniousrd.h", "pniousrt.h", "pniousrx.h", "pnioerrx.h" and the header files of the Serv library "servusrx.h" are copied to the "/usr/include" library.

Step 9

Command: make load

Description: Load and start the driver.

Note

Note that you must start the driver again manually each time the PC 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 section below describes the actions to be carried out when installing 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 and the IO Base library following installation:

Action	Description
Testing the driver	Call "make load". No error message should be displayed if the CP 1626 has been installed correctly in the PC.
Testing the firmware on the CP 1626	Take the configuration of the sample application "ctrl_rw_digital_io" and use TIA Portal STEP 7 or NCM to download from a configuration station to the CP 1626.
Testing the dual-port RAM	Call the application "serv_get_fw_info". The configuration of the CP must output this to you. If the host does not receive an answer, a "PNIO_ERR_NO_FW_COMMUNICATION" is returned.
Testing the IO-Base library	Put the sample application "ctrl_rw_digital_io" into operation. This test application implements an IO controller. The required configuration is enclosed with the example.

Description of driver porting

4

This chapter explains the functionality of the Linux driver. You will learn step-by-step how to port the driver to your target operating system.

4.1 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 when operating in isochronous real-time mode. If the reaction times to interrupts are extremely high, IRT can only be operated with long cycle times.

4.2 How the driver basically works

Overview

The driver is used to activate the CP 1626 and to integrate the memory windows and IRQs of the CP 1626 in the operating system. It performs the following functions:

- Processing interrupts
- Referencing the process image of the CP for the IO-Base library.
- Processing jobs between thwe IO-Base library and the firmware on the CP.

The driver also contains a watchdog function that monitors the firmware on the CP. This can be recognized when the firmware no longer functions correctly.

The following figure shows the basic structure of the driver and the CP 1626. The arrows indicate the communications channels of the driver to the hardware and firmware. Communication channels are memory areas on the CP 1626 that contain 2 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. In Linux, device files are driver access points via which applications communicate with the driver.

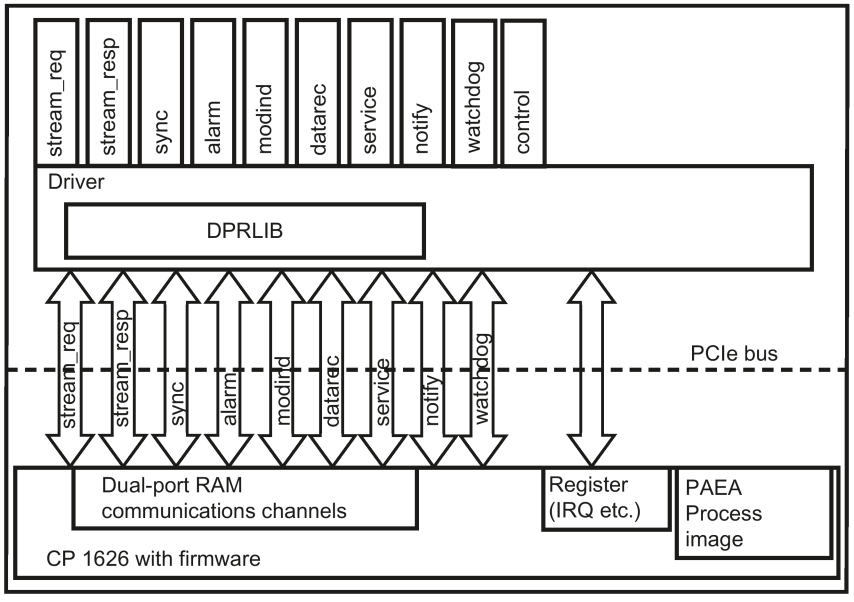


Figure 4-1 Function diagram of the driver and firmware with the available communications channels.

Description of the "Makefile"

To load the driver, call the supplied "Makefile" with "make load" in the "cp1626"" directory. When it is activated, it creates the "cp16xx1" entry and the "cp1626_1/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	Communications channel for ...
stream_req	open, read, write, ioctl	...sending data streams
stream_resp	open, read, write, ioctl	...receiving data streams
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 record transfer
service	open, read, write, ioctl	.. the management of the CP
notify	open, read, write, ioctl	... feedback when monitoring with SERV_CP_info
watchdog	open, read, write, ioctl	... the monitoring of application and firmware
control	open, read, write, ioctl, mmap	... the instance management of the driver - This communications channel has no equivalent in the dual-port RAM.

These device files are used by the IO-Base and the Serv library to communicate with the CP. 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 PCIe resources required for the dual-port RAM, register, IRQ. 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. After this, the firmware registers with the driver for time monitoring.

Description of the driver in the productive phase

The driver stores jobs coming from the IO-Base library in the communications 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 on the communications channels and indicates this by sending an interrupt to the driver. The driver then transfers the acknowledgments to the IO-Base library.

Once the firmware has written jobs for the IO-Base library in the communications 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 events in the productive phase

The IRT functionality uses the following interrupt-based events:

- APPL_START
- APPL_FAULT
- BUS_CYCLE
- TIME_SLICE_VIOLATION

The event APPL_START signals the completion of the IRT-IN data transfer to the host memory. The application can now start to process the IO data (IN/OUT data). Once processing is complete this must be signaled by the application to the IO-Base library. The library acknowledges this with APPDONE so that the IRT data (OUT data) is transferred back to the module.

If the acknowledgement (APPDONE) is not triggered up to the start of the network transfer, the module ignores the OUT data. This is signaled to the application with the APPL-FAULT event.

If it is detected that the acknowledgement of the interrupt source APPL_START was missed, this is made up for by an error handling in the driver (TIME_SLICE_VIOLATION). The application and the IO-Base library are not involved in this

4.3 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 shared memory.

Registering an IO-Base library instance with the driver

The IO-Base library uses seven communications channels in the dual-port RAM. To do this, the IO-Base library opens a device file for every channel. The IO-Base library also requires the "/dev/cp1626_1/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 IO-Base instances, when these register with the communications channels they must inform the driver of their device file handle that they received when opening the "/dev/cp1626_1/control" device file.

The IO-Base library registers with the driver in four steps:

1. The IO-Base library opens the "/dev/cp1626_1/control" device file.
2. The IO-Base library sends the IO control CP16XX_IOC_OAPP (register application) with the file handle for the "/dev/cp1626_1/control" device file to obtain an application handle from the driver.
3. It now opens 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 (e.g. Controller open etc.) 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 referenced in the address space of the application. This referencing of the process image memory in the address space of the user has the following advantages:

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

4.3.1 Directory structure and files

Description

The "driver" directory contains the files that are independent of 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

Driver files	Purpose of the individual files
os.h os_linux.h	Contain initial macros 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.

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 module_macros.h pci_map.h ppa_hall.h ppa_host_drv.c ppa_host_drv.h ppa_if.h ppa_named_bitfields.h ppa_regs.h watchdog_rqb.h	Non operating system-specific driver functions containing user management, parameter passing, watchdog functions and the access to the register of the CP.

Driver files	Purpose of the individual files
dpr_channel.h dprlibhost.c dprlib.h dpr_msg.h host_dma_ram.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.
host_version.h build_nr.h prod.h siemens.h version.h	These files are used for versioning.

4.3.2 Non operating system-specific functions

Table with user management functions and structures

User management functions and structure	Description
CP16XX_APP_DATA_STRUCT	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

Device management function and structure	Description
CP16XX_CARD_DATA_STRUCTstruct	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 implemented 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.

DPRLIB functions and structure	Description	
CP_DATA_STRUCT	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.	
	trigger_irq	Function pointer to the function used to trigger an interrupt to the CP 1626.
	wakeup_daemon	Function pointer to a function which is called as soon as the DPRLIB disconnects the link to the firmware.
	Parent	Points to the structure "CP16XX_APP_DATA_STRUCT" which is filled out during the hardware connection.
DPRLIB_start()	Start connection to the firmware - This function initializes the dual-port RAM.	
DPRLIB_stop()	Stop connection to the firmware - This function resets the dual-port RAM.	
DPRLIB_channel_ write_message()	Writes a job packet to a communications channel in the dual-port RAM.	
DPRLIB_channel_ read_message()	Reads a job packet from a communication channel in the dual-port RAM. This function is not blocking and returns immediately. If a packet was read out successfully, it returns a DPR_OK.	
DPRLIB_channel_ register_cbf()	Registers a callback which is called when a job packet is received from the dual-port RAM. As parameters this callback has the CP communications channel and the job packet size. 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_1control_interface()	... the IO-Base library calls an "ioctl" for a "control" device file.
cp16xx_base_ioctl_common2()	... the IO-Base library calls an "ioctl" for a device file other than "control".

The entry function ...	is called by the operating system as soon as ...
cp16xx_os_driver_cleanup()	... the driver is unloaded.
cp16xx_base_check_read_permission()	... with a "read", the IO-Base library checks the registered application instance handle. The actual "read" takes place via cp16xx_os_read.
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 CP in the address area of the driver
cp16xx_pci_uninit()	Removes IO areas of the CP from the address area of the driver.

4.3.3 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 if necessary need to be ported.

Channel management functions and structures

Function / macro / structure	Description
CP16XX_CHANNEL_STRUCT	Management structure for the communications 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 the standardized 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_ioctrl()	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 an "mmap" for a device file.
cp16xx_os_mmap_dma_remap()	Used internally by "cp16xx_control_mmap( )" for DMA memory; references the kernel address space in the user address space.
cp16xx_os_open()	Is called as soon as the IO Base library calls an "fopen" for a device file.
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 CP instance and registers the found CP 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 CP reset and allows additional functions to be performed by the porting engineer when necessary.
cp16xx_os_write()	Is called as soon as the IO-Base library calls a "write" for a device file.
down_timeout()	Auxiliary function used implement a semaphore with timeout.

4.4 Porting the driver step-by-step

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 the following eight 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	Porting the connection establishment and termination from the IO-Base library to the driver.
6	Porting the send functionality from the IO-Base library to the firmware via the driver.
7	Porting the receive functionality from the firmware to the IO-Base library.
8	Porting the memory referencing in the user address space.

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

Overview

This file contains all the initial macros that the driver needs. The driver encapsulates all function calls to the operating system using initial macros. You need to 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 selected above, for example "_MYOS", 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 to 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 a parameter.
DPR_THREAD_DELETE (hThread)	Releases a thread handle. hThread: Handle to be released.
DPR_SEMAPHORE	Type that represents a semaphore (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)	Wait for a semaphore. 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: 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.
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.

4.4.2 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 initial macros:

Step	Description
1	Add the following line to the initialization routine of your driver: TRC_OUT(0, LV_ERR, ("start %s\n",cp16xx_driver_version); 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" 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: TRC_OUT(0, LV_ERR, stop %s\n",cp16xx_driver_version);
4	Compile the driver and test the initialization or deinitialization.

4.4.3 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 referenced in the operating system and an interrupt service routine is integrated. The "CpData" structure is then filled out. 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 1: Porting the macros of the "os_linux.h" file (Page 31)".

4.4.4 Stage 4: Defining the driver interface

General

In this step, you define the interface between the IO- Base library and the driver. The firmware of the CP 1626 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 communications channel to be able to pass on these communications channels transparently as far as the IO-Base library. This allows the driver or IO-Base library to implement a simple Read/Write interface. There is no need to implement encapsulation of the send or receive jobs. The driver also creates an additional "dev/cp1626_1/control" device file with which additional services of the IO-Base library can be made available.

Services of the driver

The following table lists all the mechanisms/services required by the IO-Base library. These must be replaced by suitable equivalent interfaces to your ported driver. In the following steps, it is always assumed that the interface implemented in Linux is used.

Services of the access point "/dev/cp1626_1/control"

The access point "control" supports the following interface:

Control	Description
fopen	Function for obtaining an operating system file handle for the "/dev/cp1626_1/control" device file.
fclose	Function for closing a "/dev/cp1626_1/control" device file.
mmap	Service for referencing PCIe resources of the CP 1626 to the user address space
IO control: CP16XX_IOC_CAPP	Service for deleting an application instance handle
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_IOCRESET	Service for hardware reset of the CP 1626
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 reading of incoming events of the IO channels. The events are distinguished by the transferred length parameter.

Services for the remaining access points

The following access points exist:

- stream_req
- steam_resp
- sync_alarm
- modind
- datarec
- service

- notify
- watchdog

Control	Description
fopen	Function for obtaining an operating system file handle for the device file that corresponds to a communications channel in the dual-port RAM.
fclose	Function used to close a device file that corresponds to a communications 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 does not allow multiple access to the driver from the application context at the same time, you need to implement a request block interface and a multiplexer. The request block could have the following structure:

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

Note

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!

4.4.5 Stage 5: Porting the connection establishment and termination from the IO-Base library to the driver.

Description

For every connection setup, the Linux driver has a two-stage registration mechanism that must be run through for every IO-Base instance.

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 handle belongs to which application and how the communication to the firmware is structured.

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.

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()	Locking of the management structure for communication to the IO-Base library.
DPR_CHANNEL_UNINIT_OS()	Function for deinitializing a management structure.
DPR_CHANNEL_UNLOCK()	Unlocks a management structure.
DPR_CHANNEL_WAIT_FOR_ WAKEUP()	Wait for a signal of an incoming message using a synchronization object.
DPR_CHANNEL_WAKEUP()	Signal with a synchronization object that a message is pending.

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

Description

Messages are 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()".

4.4.7 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-ase library calls the "read" function from within a thread. If job packets from the firmware are already available in the dual-port RAM, the "read" call returns immediately. 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.
3	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 33)" is called due to the interrupt in Step 2. The ISR of the driver must call the non platform dependent function "cp16xx_process_dpr_irq" with the parameter of the "CP_16XX_CARD_DATA_STRUCT" structure contained. The ISR "cp16xx_irq_shared_cbf( )" then acknowledges the interrupt and terminates.
4	The function from stage 3 calls the DPRAM function dprlib_int_callback. Here an internal semaphore is set to wake a DPRLIB internal worker thread.
5	The DPRLIB library-internal worker thread determines the communications channel in which a job packet is located, and calls the function "cp16xx_base_read_cbf()" that was registered by the driver. This function occupies a block memory and calls the "DPRLIB_channel_read_message()" function to copy the data to the memory block. The memory block is then inserted in the chained block list of the corresponding channel. 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.

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 in the driver as soon as the IO-Base library has sent the blocking "read" for the communication from the driver to the IO-Base library. This is blocked until new data is present and can be read.

4.4.8 Stage 8: Porting memory referencing in the user address space

Description

The "/dev/cp1626_1/control" device file supports the IO control and as well as the registration/deregistration is also used among other things for including memory areas of the CP 1626 in the user address space for the IO-Base library.

Example:

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 issues "mmap" for the "/dev/cp1626_1/control" device file.

Description of porting the IO base library

5

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

5.1 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

5.2 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. The main task of the user is to port the functions responsible for communication with the driver.

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

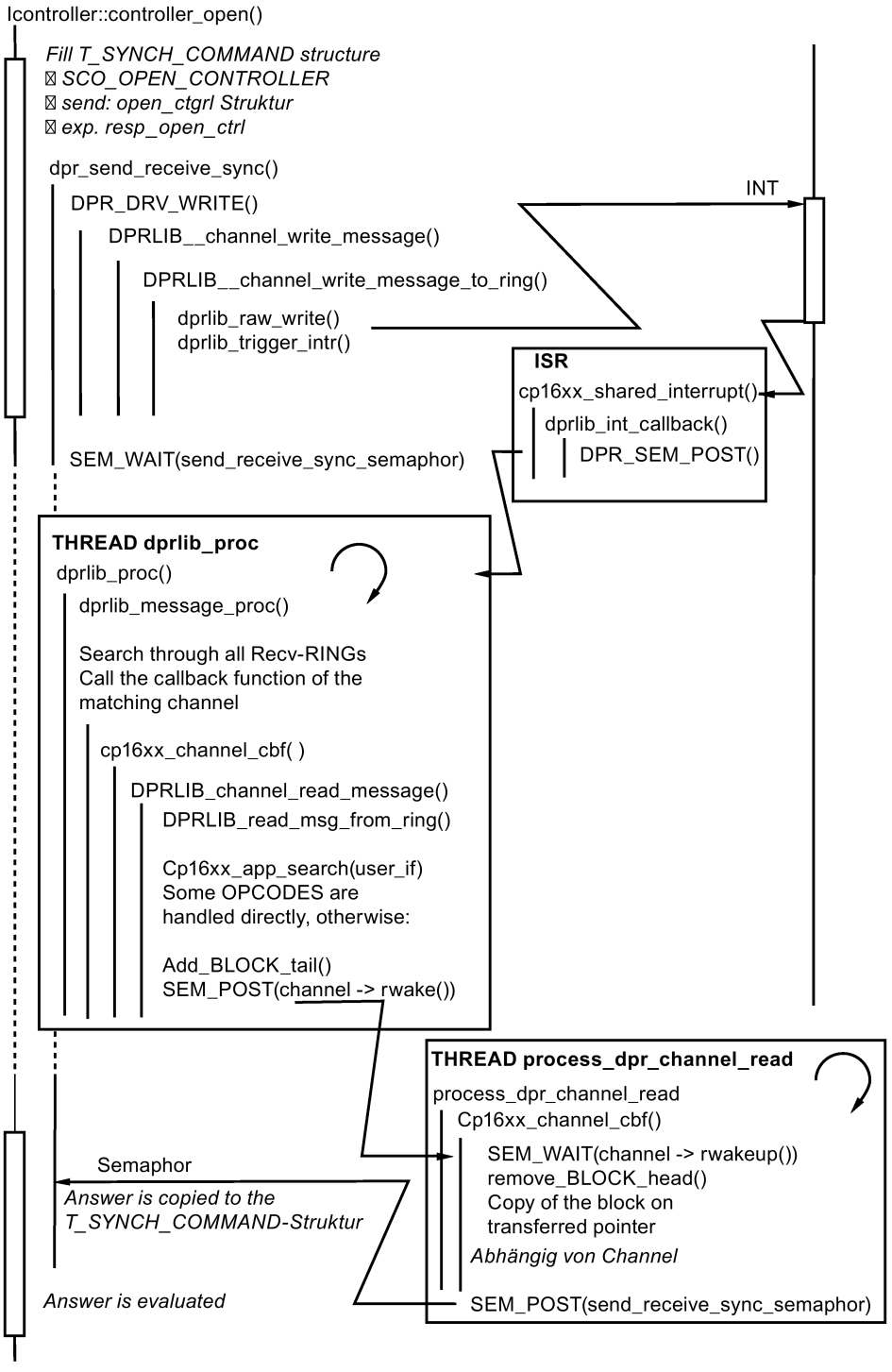


Figure 5-1 Communication of the application and driver with the firmware via the dual-port RAM.

5.3 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
dpram	DPRAM communication
iodataupdate	Access to the process image
iobase	Localizes the data within the process image
kramiotlb	Contains the PROFINET IO logic
Ppa	Transfer of IO data
tracelib	Contains the trace functionality
version	Contains the version header files
Watchdog	Watchdog

Description of the module directory content

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

Note

The number of modules differs depending on the supplied software version.

Directories	Content
csd	Make files
src	Source files
inc	Header files

Files that need 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 initial macros 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.

5.4 Functions dependent on the operating system

Functions for binding the IO-Base library to the driver

Function	Description
ICommon::InitCp()	Initializes the communications channels to the driver.
ICommon::UninitCp()	Deinitializes the communications channels to the driver.
dpr_send_receive_sync()	Sends a synchronous job packet to the firmware via the driver and then waits for the response from the firmware.
process_channel_read()	Thread function for reading out job packets and acknowledgments from the firmware.
dpr_open_channel()	Opens a communications channel to the driver.
dpr_close_channel()	This function closes a communications channel to the driver.
dpr_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.

5.5 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	Porting the trace module
2	Port the IO-Base library link for the driver.

5.5.1 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 in 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 42)".

5.5.2 Stage 2: Porting the IO-Base library link to the driver

Description

The file "dpr_adapter.cpp" contains the code for communication with the firmware via the driver. If the access functions there differ from those in your operating system you will need to adapt them there. The individual functions to be ported are listed in the table in Section "Functions dependent on the operating system (Page 42)".

5.6 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 - 8: Trace depths 1 to 8 (meaning see pniotrace.conf) 9: Trace for time-critical functions (IRT etc.) 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.

Entry	Description
TRACE_MAX_BACK_FILES	Maximum number of created trace files - If this value is 2, this means the following: • 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.

Example: By replacing output operations (file or console) with memory operations.

5.7 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, put the "ctrl_rw_digital_io" demo application and the associated configuration into operation. To do this, you will need the structure specified in the configuration.
2	**Test the device functionality** For this purpose, put the "dev_rw_digital_io" demo application into operation. To do this, you will need an additional IO controller.