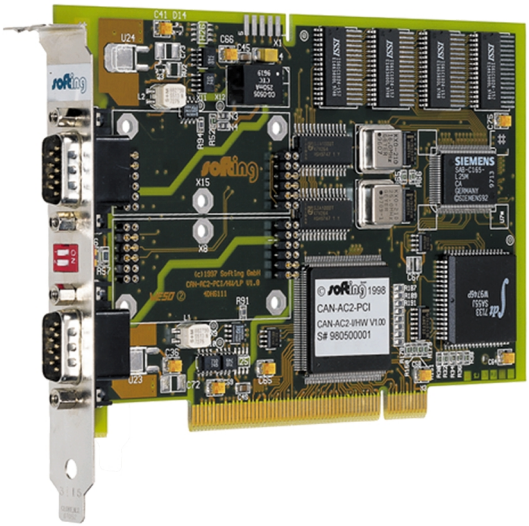
CAN-AC2-PCI
User Manual

Version 4.07 March 2002



SOFTING AG

Richard-Reitzner-Allee 6
D-85540 Haar, Germany
Telephone (++49) 89/4 56 56-0
Telefax (++49) 89/4 56 56-399

Printed in Germany 2002

HCN03E0203

Contents

List of figures

List of tables

Engineering notes:

Preface

About this manual

This user manual is written for users operating **CAN-AC2-PCI** (**C**ontroller **A**rea **N**etwork **A**pplication **C**ontroller **2** for **P**eripheral **C**omponent **I**nterconnection) together with the operating systems Windows 9x/ME, Windows 2000/XP and Windows NT 4.0.

It includes the following topics:

- Chapter 1 gives a common introduction about the product CAN-AC2-PCI and its application.
- In Chapter 2 the installation procedure of software and hardware are described as well as the installed components. Helpful notes support the uncomplicated installation. A 'Quick start' is included.
- Chapter 3 describes the CAN-AC2-PCI hardware. The main function is explained, bus termination and I/O connectors are defined.
- Chapter 4 helps to program the CAN access with the API (Application Programming Interface). The API function definition is included as well as a description of the main operational modes and the programming. Furthermore, some helpful implementation notes are made.
- Useful information about the test and example program 'Can_test.exe' can be found in chapter 5.
- Chapter 6 reports the error codes which may occur during board initialization.

In addition to this user manual, always observe the *Release Notes* contained in file README.TXT. This file resides on the disk along with the setup program. The notes contain up-to-date information concerning the present software version.

Engineering notes:

1 Introduction

1.1 About the CAN-PCI interface

High performance hardware and software computer interfaces are necessary to connect devices and components to **CAN** (**C**ontroller **A**rea **N**etwork). CAN data streams must be preprocessed and buffered at the CAN interface due to the high real-time requirements of the CAN message traffic.

More and more, **PCI** (**P**ersonal **C**omputer **I**nterconnection) is replacing the ISA bus due to its ability of high data throughput and the advantage of auto configuration.

The CAN-AC2-PCI is an intelligent CAN interface board for IBM and compatible computers with PCI bus. It can be connected to two independent CAN network connections that can be operated in parallel. Data exchange with the PC takes place through a Dual-Port RAM (DPRAM). Together with the supplied driver libraries PC-based applications can easily be integrated into CAN networks.

The CAN-AC2-PCI interface:

- Offers an application interface to two independent, electrical isolated CAN networks.
- Provides a physical layer according to ISO 11898 CAN High Speed
- Can be optionally used with another physical interface.
- Relieves applications of real-time-sensitive tasks while receiving and transmitting CAN messages by means of buffering and filtering.
- Executes parts of an application directly on its own processor, communicating with the PC via its dual-ported RAM.
- Provides bit rates up to 1Mbit
- API supports WIN32 interrupts and cyclic transmission

- Can be used with additional CAN standard software and operating system drivers.

1.2 Scope of application

1.2.1 PC interface

Each CAN-AC2-PCI is supplied with loadable onboard firmware and a driver library to implement a PC application interface. The library can be linked with the application programs and thus allows the application access to the CAN networks. The driver library supports:

- Initialization of the CAN chips
- Transmission of data frames and remote frames with time stamped confirmation (may use interrupts)
- Event-driven reception of time stamped data frames and remote frames (may use interrupts)

The CAN connection is implemented by two separate CAN channels with the SJA1000 CAN controller from Philips according to CAN specification 2.0B (11bit and 29bit identifier).

The physical interface consists of two electrically isolated CAN High Speed interfaces according to ISO 11898. Customer-specific connections can be implemented as piggyback.

Connection to the CAN bus system is made through the two 9-pin D-Sub connectors on the slot bracket. The pin-out conforms to the **CiA** standard (User Group: "**C**AN **i**n **A**utomation").

1.2.2 CANalyzer

In addition to its use as a PC interface the CAN-AC2-PCI serves as a platform for the CANalyzer software package. This emulation and analysis system for CAN networks provides analytical and simulation options that go beyond the basic functions used to transmit and receive CAN messages. For example, it is possible to detect and transmit individual CAN error frames and accurately measure the bus load of the CAN network.

Beside the CANalyzer software a special license is required for CANalyzer operation. The interfaces including CANalyzer functionality are often referred to as CAN-AC2-PCI/ANA.

1.3 Supported systems

The CAN-AC2-PCI V4.07 API functions are integrated in a 32bit DLL according to standard call convention. Thus, it is the basis to support all compilers, measurement tools and visualization systems which are able to provide access to 32bit Windows DLLs, e.g.:

- Microsoft Visual C/C++ 4.0 (32bit) upwards[1)]
- Borland C/C++ 4.5 (32bit) upwards[1)]
- Borland C++ Builder 1.0 upwards[1)]
- Watcom C/C++ version 1.1 (Powersoft)[1)]
- Visual Basic 5.0 upwards (Microsoft)[1)]
- Delphi 2.0 upwards (Borland)[1)]
- LabVIEW 5.0 (National Instruments)[1)]
- LabWindows CVI 3.1 (National Instruments)[1)]
- HPVEE 4.01 (Hewlett Packard)[1)]
- Testpoint 3.3 (Keithley)[1)]
- WIZCON (PCSOFT)[1)]
- VISUA (SSS)[1)]

For examples and more information about the supported systems please visit our homepage http://www.softing.com or contact the technical support hotline (++49) 89/4 56 56-337.

Due to special driver implementations Softings CAN interfaces are further supported by the following systems:

- DASYLAB (DATALOG)[1)]
- DIADEM (GfS)[1)]

Scope of delivery

[1)] All products mentioned are trademarks of their respective companies.

Before you begin to install the CAN-AC2-PCI you should make sure that all of the parts listed below are at hand.

The CAN-AC2-PCI is delivered with the following components:

- CAN-AC2-PCI board
- CD with installation software and documentation

2 How to install CAN-AC2-PCI

2.1 System requirements

To run the CAN-AC2-PCI your PC must meet the following requirements:

- 100% IBM-compatible
- At least one available PCI slot according to PCI spec. 2.1
- Windows 9x/ME, Windows NT 4.0 or Windows 2000/XP running
- at least 2 MByte free on hard disk

2.2 Quick start

1. Install the software by running ‘CAN-AC2-PCI V4.07N Setup.EXE’ from installation disk.
2. Switch off the PC and plug in the CAN-AC2-PCI into a free PCI slot.
3. Boot the PC.
4. Start ‘Can_test.exe’ from the command line and choose any operational mode

If the test program states ‘Chips are running’ the installation was successful and the PCI card works properly. Quit the test by pressing ‘q’ or proceed further tests (see chap. 6).

If the test program returns any error code please refer to Chapter 6.

2.3 How to install the API driver software

2.3.1 General

The API driver software for the CAN-AC2-PCI is installed from the installation CD by executing 'CAN-AC2-PCI V4.07N Setup.EXE'.

The Windows based installation program 'CAN-AC2-PCI V4.07N Setup.EXE' automatically detects the operating system, copies the related API driver files to hard disk and prepares the system registry if necessary. The installation path can be adjusted to customer's choice. Status messages inform about the success of the installation. For system specific notes see sections 1.3.2 and 1.3.3.

2.3.2 Windows 98/ME, Windows 2000/XP

In Windows 98/ME and Windows 2000/XP the setup program copies the files to the hard disk as described in Table 2-1 and Table 2.2 (default installation path presumed). '%MAINDIR%' is the representation of the installation path of the CAN-AC2-PCI API software chosen by the customer.

Furthermore the setup program sets the registry keys for enabling event logging as described in Table 2.3.

After successful finish of the setup program the PC is prepared for the PnP installation of the hardware.

NOTE:
Due to the PnP mechanism of Windows 98/ME and Windows 2000/XP the software installation should be processed before installation of the hardware. If the CAN-AC2-PCI is plugged in first the appearing installation assistant should be closed by clicking the 'Cancel' button.

Table 2-1: Files installed for CAN-AC2-PCI in Windows 98/ME and Windows 2000/XP (WDM))

File	Description
Windows\System\vcanpciw.sys	WDM device driver for hardware access
Windows\System\vcanpcid.dll	Device driver DLL Windows 98/ME only
Windows\System32\vcanpcid.dll	Device driver DLL Windows 2000/XP only
Windows\Inf\vcanpciw.inf	Installation script file
%MAINDIR%\readme.txt	'Readme' file with installation and application information
%MAINDIR%\Install.log	Logging file for uninstallation
%MAINDIR%\Unwise.exe	Uninstall program
%MAINDIR%\win32\canacpci.dll	CAN-AC2-PCI API DLL
%MAINDIR%\win32\canacpci.lib	C import library for 'canacpci.dll' (Microsoft standard)
%MAINDIR%\win32\canacpci.def	Module definition file for linking with Borland applications
%MAINDIR%\Include\Canlay2.h %MAINDIR%\Include\Can_def.h	API header files with necessary definitions and declarations

Table 2-2: Example files for CAN-AC2-PCI in Windows 98/ME and Windows 2000/XP (WDM)

File	Description
%MAINDIR%\win32\Can_test.exe	Test and example program
%MAINDIR%\Source\Can_test.c %MAINDIR%\Source\Dynobuf.c %MAINDIR%\Source\Statobuf.c %MAINDIR%\Source\Fifo.c %MAINDIR%\Source\Intexmpl.c	"C" Example source code of 'Can_test.exe'
%MAINDIR%\bas\Can_Object.frm %MAINDIR%\bas\CAN_TEST.FRM %MAINDIR%\bas\CAN_TEST.frx %MAINDIR%\bas\CAN_TEST_PCI.vbp %MAINDIR%\bas\can_values.frm %MAINDIR%\bas\CANACPCI.BAS %MAINDIR%\bas\COMMON.BAS %MAINDIR%\bas\test_specific.bas	Visual Basic 6.0 Example source code
%MAINDIR%\bas\CANACPCI.DLL	API DLL
%MAINDIR%\bas\readme.txt	Readme for Visual Basic Example

Table 2-3: Registry keys for CAN-AC2-PCI in Windows 2000/XP at <Service Key>\Eventlog\System\CanACxPci [1]

Registry key	Type	Value
TypesSupported	REG_DWORD	7
EventMessageFile	REG_SZ	Path to WDM device driver for hardware access (see Table 2.1)

[1] <Service Key> = HKLM\System\CurrentControlSet\Services

2.3.3 Windows NT 4.0

The setup program 'CAN-AC2-PCI V4.07N Setup.EXE' installs the files shown in Table 2-6 and Table 2.7 on the hard disk. Furthermore, all necessary Windows NT 4.0 registry keys are set in the <Service Key> [1]in the registry as described in Table 2-4 and Table 2-5. Additional registry entries are made by the system after succeeding hardware installation.

NOTE:
After the software installation the system has to be restarted to put the new registered driver in effect.

Table 2-4: Registry keys for CAN-AC2-PCI in Windows NT 4.0 at <Service Key>\vcanpcid[1]

Registry key	Type	Value
ErrorControl	REG_DWORD	1
Start	REG_DWORD	2 (Automatic)
Type	REG_DWORD	1
Group	REG_SZ	"Extended base"
DisplayName	REG_SZ	"CAN-AC-PCI (Softing)"

[1] <Service Key> = HKLM\System\CurrentControlSet\Services

Table 2-5: Registry keys for CAN-AC2-PCI in Windows NT 4.0 at <Service Key>\EventLog\System\vcanpcid[1]

Registry key	Type	Value
TypesSupported	REG_DWORD	7
EventMessageFile	REG_SZ	Path to device driver for hardware access (see Table 2.6)

NOTE:
The registry entries should not be changed by the customer since this may result in faulty driver functionality.

[1] <Service Key> = HKLM\System\CurrentControlSet\Services

Table 2-6: Files installed for CAN-AC2-PCI in Windows NT 4.0

File	Description
Windows\System\vcanpcid.sys	Device driver for hardware access
Windows\System\vcanpcid.dll	Device driver DLL
%MAINDIR%\readme.txt	'Readme' file with installation and application information
%MAINDIR%\Install.log	Logging file of installation
%MAINDIR%\Unwise.exe	Uninstall program
%MAINDIR%\win32\canacpci.dll	CAN-AC2-PCI API DLL
%MAINDIR%\win32\canacpci.lib	C import library for 'canacpci.dll' (Microsoft standard)
%MAINDIR%\win32\canacpci.def	Module definition file for linking with Borland applications
%MAINDIR%\Include\Canlay2.h %MAINDIR%\Include\Can_def.h	API header files with necessary definitions and declarations

Table 2-7: Example files for CAN-AC2-PCI in Windows NT 4.0

File	Description
%MAINDIR%\win32\Can_test.exe	Test and example program
%MAINDIR%\Source\Can_test.c %MAINDIR%\Source\Dynobuf.c %MAINDIR%\Source\Statobuf.c %MAINDIR%\Source\Fifo.c %MAINDIR%\Source\Intexmpl.c	"C" Example source code of 'Can_test.exe'
%MAINDIR%\bas\Can_Object.frm %MAINDIR%\bas\CAN_TEST.FRM %MAINDIR%\bas\CAN_TEST.frx %MAINDIR%\bas\CAN_TEST_PCI.vbp %MAINDIR%\bas\can_values.frm %MAINDIR%\bas\CANACPCI.BAS %MAINDIR%\bas\COMMON.BAS %MAINDIR%\bas\test_specific.bas	Visual Basic 6.0 Example source code
%MAINDIR%\bas\CANACPCI.DLL	API DLL
%MAINDIR%\bas\readme.txt	Readme for Visual Basic Example

2.3.4 Windows 95

In Windows 95 the setup program copies the files to the hard disk as described in Table 2-8 and Table 2.9 (default installation path presumed). '%MAINDIR%' is the representation of the installation path of the CAN-AC2-PCI API software chosen by the customer.

After successful finish of the setup program the PC is prepared for the PnP installation of the hardware.

NOTE:
Due to the PnP mechanism of Windows the software installation should be processed before installation of the hardware. If the CAN-AC2-PCI is plugged in first the appearing installation assistant should be closed by clicking the 'Cancel' button.

Table 2-8: Files installed for CAN-AC2-PCI in Windows 95

File	Description
Windows\System\vcanpcid.vxd	Virtual device driver for hardware access
Windows\System\vcanpcid.dll	Device driver DLL
Windows\Inf\vcanpcid.inf	Installation script file
%MAINDIR%\readme.txt	'Readme' file with installation and application information
%MAINDIR%\Install.log	Logging file for uninstallation
%MAINDIR%\Unwise.exe	Uninstall program
%MAINDIR%\win32\canacpci.dll	CAN-AC2-PCI API DLL
%MAINDIR%\win32\canacpci.lib	C import library for 'canacpci.dll' (Microsoft standard)
%MAINDIR%\win32\canacpci.def	Module definition file for linking with Borland applications
%MAINDIR%\Include\Canlay2.h %MAINDIR%\Include\Can_def.h	API header files with necessary definitions and declarations

Table 2-9: Example files for CAN-AC2-PCI in Windows 95

File	Description
%MAINDIR%\win32\Can_test.exe	Test and example program
%MAINDIR%\Source\Can_test.c %MAINDIR%\Source\Dynobuf.c %MAINDIR%\Source\Statobuf.c %MAINDIR%\Source\Fifo.c %MAINDIR%\Source\Intexmpl.c	"C" Example source code of 'Can_test.exe'
%MAINDIR%\bas\Can_Object.frm %MAINDIR%\bas\CAN_TEST.FRM %MAINDIR%\bas\CAN_TEST.frx %MAINDIR%\bas\CAN_TEST_PCI.vbp %MAINDIR%\bas\can_values.frm %MAINDIR%\bas\CANACPCI.BAS %MAINDIR%\bas\COMMON.BAS %MAINDIR%\bas\test_specific.bas	Visual Basic 6.0 Example source code
%MAINDIR%\bas\CANACPCI.DLL	API DLL
%MAINDIR%\bas\readme.txt	Readme for Visual Basic Example

2.3.5 Uninstall support

After successful installation the uninstall support window of the ‘software’ entry in the ‘system control’ includes the ‘CAN-AC2-PCI V4.07’ entry. Double clicking this entry the ‘Unwise.exe’ in the installation root of the CAN-AC2-PCI software is entered and all steps of the preceding installation are undone.

NOTE:
The system changes made by the Windows 9x/ME or Windows 2000/XP PnP system must be undone by additionally deinstallation of the ‘Remove CAN-AC2-PCI driver’ entry in the ‘software’ panel of the the Windows 9x/ME or Windows 2000/XP ‘system control’.

NOTE:
After uninstalling the software with ‘Unwise.exe’ in Windows NT 4.0 some registry entries are left which were made by the system. These entries can be cleaned up manually by deleting

HKLM\SYSTEM\CurrentControlSet\Services\vcanpcid.

2.4 How to install the hardware

Install the CAN-AC2-PCI board into your PC by following steps:

1. First switch OFF your PC.
2. Make sure that any peripheral devices are powered down (Monitor, etc.).
3. Remove the housing cover (refer to your PC manual if necessary).
4. Select an available slot for the CAN-AC2-PCI board and remove the slot cover
5. Plug the CAN-AC2-PCI board into the socket of the motherboard applying light pressure until the corner of the slot cover sits snugly to the housing.
6. Fasten the slot cover of the CAN-AC2-PCI board with the screw that you removed before.
7. Reassemble the housing cover.
8. Turn ON the PC and applicable peripheral devices.

NOTE:

(1) Improper handling of the CAN-AC2-PCI board can lead to its destruction by electrostatic discharges. Therefore, you should discharge yourself on a grounded object such as the metal housing of the PC before each and every contact with the board.

(2) If you run Windows 9x/ME or Windows 2000/XP on your PC please be sure that the software is already installed before you plug in the board for the first time.

2.5 How to test the installation

After installation of hardware and software the test program ‘Can_test.exe’ in ‘Win32’ directory of the installed software can be executed from the command line to test the installation:

1. Run ‘Can_test.exe’
2. After successful loading of the firmware the program states version of hard- and software, chip types and serial number of the board. Errors while initializing the PCI board are stated with error number and text.
3. Input ‘i’ for interrupt mode and an operational mode of your choice.
4. The program acknowledges success of interrupt initialization and of the operational mode.

NOTE:
When the program prints ‘Chip is running’ the hardware was successfully initialized. Thus, the installation works properly.

5. Quit with ‘q’ or step to further tests (see Chap. 6).

3 Hardware description

3.1 Environmental conditions

For proper operation of the CAN-AC2 the following environmental conditions have to be observed:

- Operating temperature 0...+55°C
- max. temperature drift 3°/min
- Non operating temperature -20...+70°C
  (transport and storage)
- Relative humidity (non condensing) 5... 90%
- Range of air pressure 860...1060hPa (mbar)
- Power supplied by the PCI-Bus +5V (±5%) max. 500mA

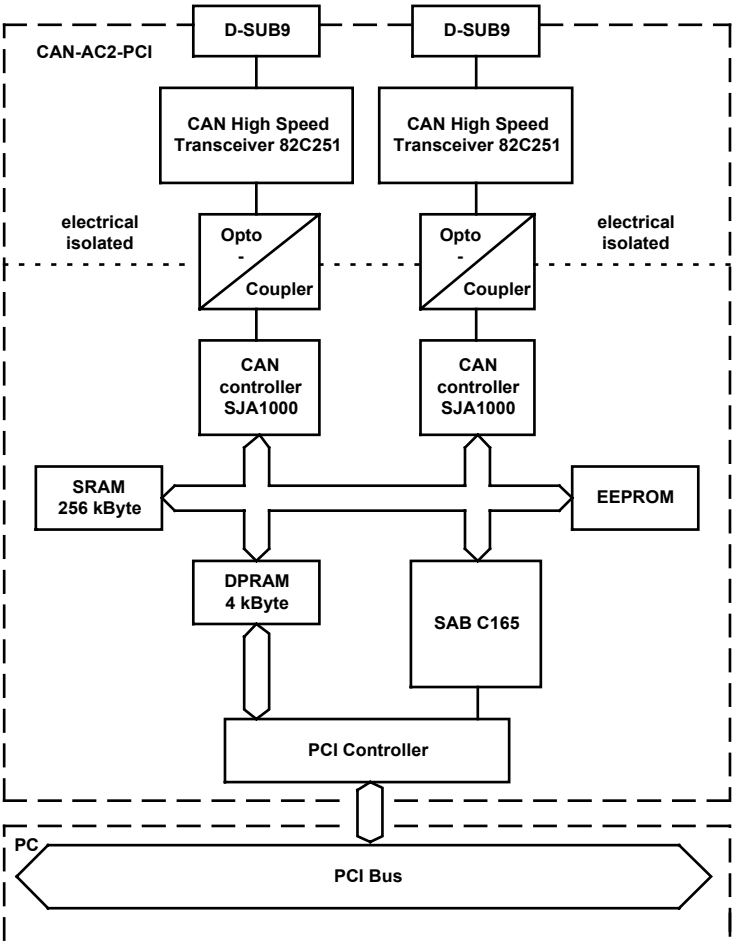


Fig. 3-1: Structure of the CAN-AC2-PCI

3.2 General description

The CAN-AC2-PCI provides an interface to two independent CAN networks (Version 2.0B) for IBM compatible PC systems via PCI bus. It is designed according to the PCI Local Bus Specification Rev. 2.1.

Fig. 3-1 shows the main structure of the hardware. In Fig. 3-2 the layout reveals the positions of the hardware parts on the PC board.

Main features

- Onboard micro controller SAB C165
- 256 kbytes RAM onboard
- PC to board communication via a dual ported RAM of 4kbyte (DPRAM)
- CAN controller Philips SJA1000
- Transceiver PCA82C251 from Philips according to CAN High Speed Specification ISO11898 (optional customer specific transceivers on piggyback)
- Electrically isolated by optocoupler

The PC application communicates with the onboard firmware via a DPRAM. The micro controller operates the CAN controllers and manages the handling of transmit and receive jobs. Thus, the PC is freed of executing these time critical tasks.

The connection of the CAN transmit and receive lines to the physical bus is realized by the transceiver device PCA82C251 from Philips according to the CAN High Speed specification (ISO11898). It converts the Tx0 and Rx0 lines of the CAN controller to CAN_H and CAN_L signals which are connected directly to the CAN bus.

Optionally, the CAN-AC2-PCI can be equipped with different transceiver chips on piggybacks (P1 and P2) for different physical specifications (see section 3.3).

Optocouplers are used to electrically isolate the CAN bus from the CAN controller. Each of the two channels is supplied by the PC via an isolated DC/DC converter. The ground potential is connected to PC ground via an RC element of 1MΩ resistance and 100nF capacity.

The CAN-AC2-PCI is connected to the CAN network via D-SUB 9 connectors. They are described in section 3.4.

Optionally, the user is enabled to set the proper CAN bus termination onboard with DIP switches (S1). Please refer to section 3.5.

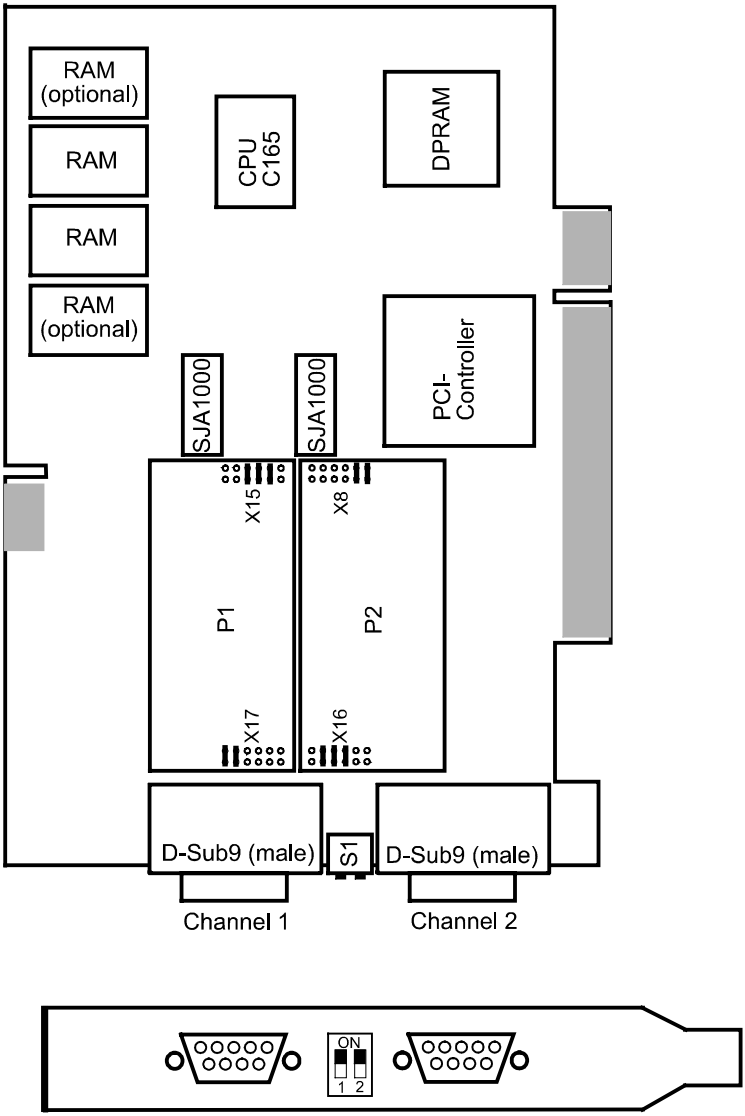


Fig. 3-2: CAN-AC2-PCI layout scheme

3.3 Physical bus interface

The standard version of the CAN-AC2-PCI board is equipped with PCA82C251 from Philips according to the CAN High Speed specification (ISO11898). In this case jumpers are plugged in the piggyback connectors to supply the CAN controller signals Tx and Rx to the default transceiver chips. This default jumper setting is shown in Fig. 3-3 and defined in Table 3-1.

NOTE:
Don't change the jumper settings if you run the interface in the default CAN High Speed environment. Changes may lead to malfunctions or destruction of the board.

If you need to implement the CAN interface in a CAN system with a different physical specification, the required transceiver circuits can be plugged in as piggybacks P1 and P2 . In this case the jumpers in Table 3-1 have to be removed.

Softing offers a CAN Low Speed Interface as piggyback which connects the CAN-AC2-PCI with CAN networks compliant to CAN Low Speed specification ISO DIS 11519-1. Please contact Softing sales department for more specific information about signaling and application of the transceiver piggybacks if needed.

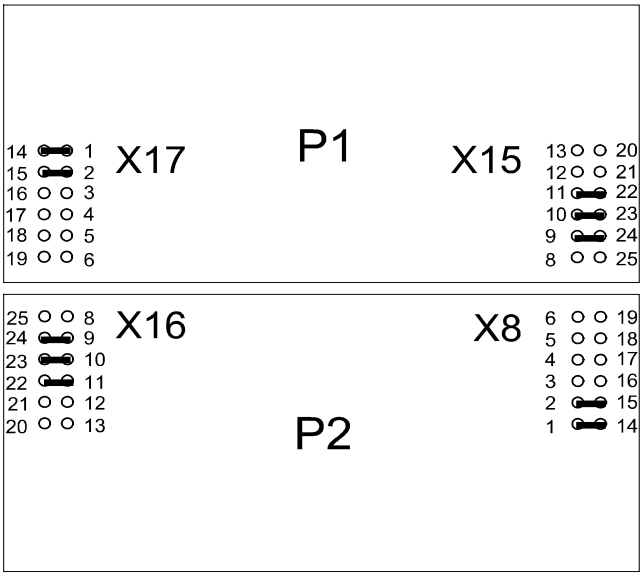


Fig. 3-3: Default jumper setting of piggyback connectors

Table 3-1: Jumper setting for CAN High Speed (default)

Jumper	**Signal**
X8.1-14	CAN_H channel 2
X8.2-15	CAN_L channel 2
X16.11-22	Rx0 channel 2
X16.10-23	Rx1 channel 2
X16.9-24	Tx0 channel 2
X17.1-14	CAN_H channel 1
X17.2-15	CAN_L channel 1
X15.11-22	Rx0 channel 1
X15.10-23	Rx1 channel 1
X15.9-24	Tx0 channel 1

3.4 I/O connector

The pining of the D-Sub connectors are defined according to the CiA recommendation for the CAN High Speed Bus.

The shield is connected to the earth via PC housing. To prevent high compensation currents due to earth loops the cable shielding can be connected to pin 5 (Drain) instead of the D-SUB 9 shield. This potential is connected to PC ground via a RC element of 1MΩ resistance and 100nF capacity to PC ground.

Fig. 3-4 and Table 3-3 show the pining of the D-Sub 9 connector which is valid for both of the CAN channels.

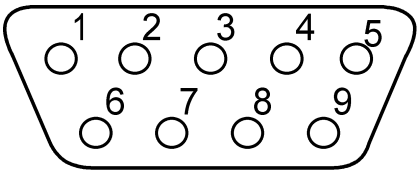


Fig. 3-4: Pinning of the 9-pin D-sub connector

Table 3-2: 9-pin D-Sub connector acc. to CIA recommendation

Pin	Signal
1	N.C.
2	CAN_L
3	GND (DCDC)
4	N.C.
5	Drain (1M/100n to PC GND)
6	GND (DCDC)
7	CAN_H
8	N.C.
9	N.C.

3.5 Bus termination

The CAN High Speed bus should be terminated with 124 Ohm between CAN_H and CAN_L on each end of the network (see Fig. 3-5).

This termination resistance should be realized either in the cable or on the CAN-AC2-PCI directly. To put the onboard termination into operation the DIP switch on the front panel (S1 in Fig. 3-2) is to be switched to 'ON' which is the default setting.

NOTE
Invalid bus termination can cause communication errors.

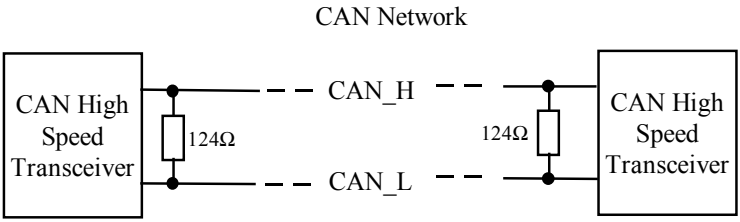


Fig. 3-5: Bus termination of the CAN connection

Engineering notes:

4 Software description

4.1 About the CAN-AC2 API

The CAN-AC2-PCI **API** (**A**pplication **P**rogramming **I**nterface) is realized as a 32bit-DLL for Windows.

Different operational modes of the interface can be configured: FIFO and object buffer mode. Thus, the programmer is enabled to adapt it to the communication task in the most suitable way.

The CAN-AC2-PCI API is designed to be conform to the API's of Softings other CAN interfaces (PCMCIA, ISA, PC/104). It provides the following functionality:

- Initialization of CAN parameters, e.g. bit rate, output control a. o.
- Transmission and reception of data and remote frames
- Message filtering
- Acknowledgment on successful transmission (optional)
- Automatic response to remote frames (optional)
- Error state detection
- Bus state detection
- Interrupt support
- Cyclic transmission

This chapter describes the basic operational modes, functions and program sequences of the API.

4.2 API driver concept

The API functions to program the interface for CAN access are supplied in a 32bit Windows DLL ‘Canacpci.dll’. This library accesses the DPRAM on the CAN-AC2-PCI via the 32bit driver DLL ‘Vcanpcid.dll’ and the hardware driver.

The hardware driver in Windows 95 is a ring 0 virtual device driver ‘Vcanpcid.vxd’. In Windows NT the kernel mode driver ‘Vcanpcid.sys’ needs to be started before accessing the CAN interface. The hardware driver in Windows 98/ME and Windows 2000/XP is a WDM (Windows Driver Model) device driver.

Driver and driver DLL are placed in the system directory of the OS. We recommend to copy the API DLL ‘Canacpci.dll’ to the local directory of the application to prevent access errors due to existence of API DLLs of different versions.

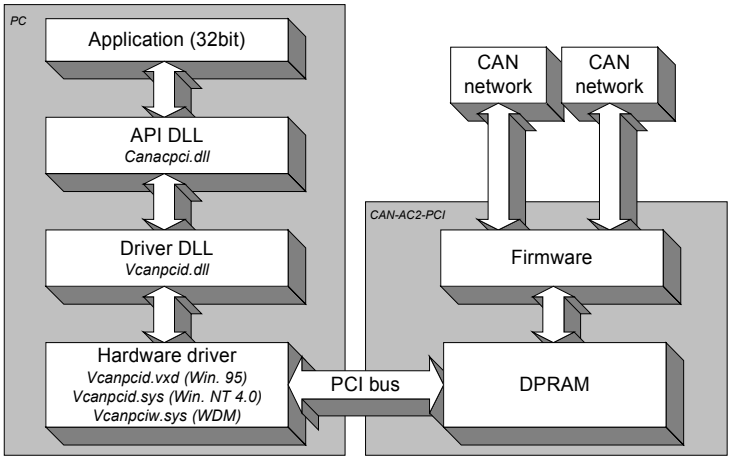


Fig. 4-1: Access structure of the API software

4.3 Operational modes of the interface

The CAN-AC2-PCI together with its driver library offers two alternative operating modes handling CAN messages: FIFO operation and CAN object buffer. Furthermore, the object buffer can be defined as static or dynamic.

4.3.1 FIFO mode

The communication between the CAN bus and the PC application through the dual ported RAM is processed sequentially using FIFOs (Fig. 4-2). The message that is entered first into the FIFO (First In First Out), is the next to be processed further. Each of the FIFOs can bear a maximum of 255 entries.

FIFO mode is chosen calling *CANPC_enable_fifo* (see Fig. 4-5)

4.3.1.1 Transmission request

The 'Transmit FIFO' handles all transmit requests of the application entered by *CANPC_send_data*[2].

If the Transmit FIFO gets full new transmit requests are denied and the application is informed by the error return code.

4.3.1.2 Receive events and transmit acknowledges

Received messages, bus events and transmit acknowledges on successful transmission are transferred to the application through the 'Receive FIFO'. They can be read out of the FIFO using *CANPC_read_ac*.

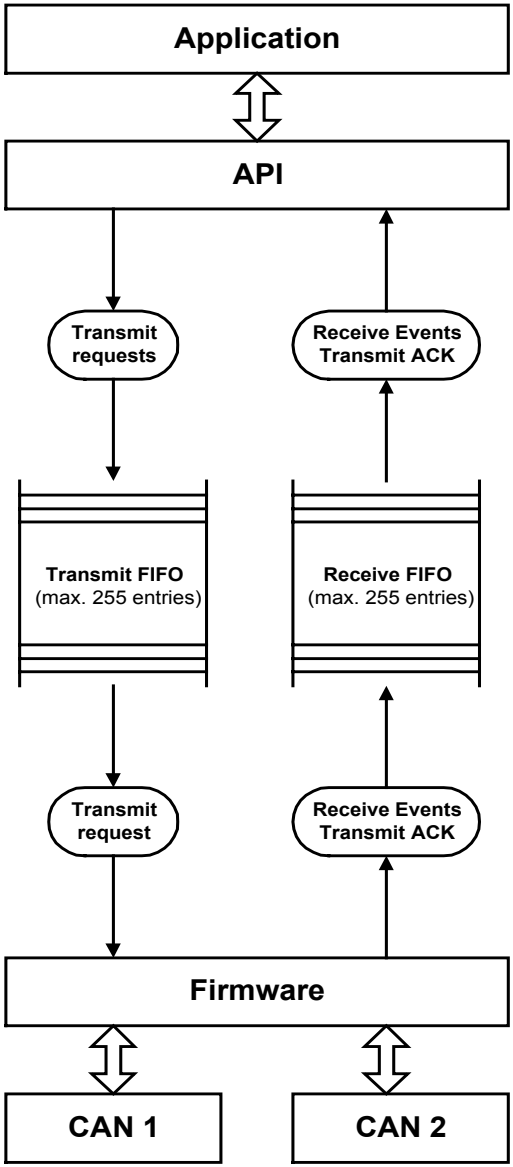


Fig. 4-2: FIFO mode structure

4.3.2 Dynamic object buffer mode

The dynamic object buffer mode is chosen calling *CANPC_enable_dyn_obj_* (see Fig. 4-6). In this operational mode the CAN messages and their data are stored in 4 object lists, i.e. transmission and reception lists for each CAN channel (Fig. 4-3). Each list can bear a maximum of 200 objects.

The entries of the lists, i.e. CAN messages of interest, have to be defined by the application using *CANPC_define_object* in the initialization routine. An object includes identifier and data of a CAN message. The API handles the objects with their object number which is returned by *CANPC_define_object* to the application program.

It is possible at any time to read or write the data of a defined object. Thus, the application always has a consistent representation of a defined "CAN database".

The handling of transmission requests, received messages, transmit acknowledges and remote frames are individually switched on or off for each object by definition (*ReceiveIntEnable*, *AutoRemoteEnable*, *TransmitAckEnable*). The interface offers two main handling mechanisms for these interaction tasks, FIFO or polling. They can be configured using *CANPC_initialize_interface*.

4.3.2.1 Transmission requests

A transmit request is commanded by *CANPC_send_object* or *CANPC_write_object*.

If *TransmitReqFifoEnable* is set in *CANPC_initialize_inter-face*, the transmit request for an object is transferred to the CAN controller through a FIFO. Otherwise, the transmit object lists are polled for objects to be sent.

The FIFO has a maximum of 255 entries. An overrun of the FIFO is recognized and reported to the application.

Polling is processed from low to high object numbers.

4.3.2.2 Transmit acknowledges

On successful transmission of an object a corresponding acknowledge can inform the application using *CANPC_read_ac*.

The acknowledges can be switched on or off for either all objects (*TransmitAckEnableAll*) or for each transmit object by definition (*TransmitAckEnable*).

If the transmit acknowledge FIFO is configured (*TransmitAckFifoEnable*), the transmit acknowledges are transferred through a FIFO to the application. Otherwise, the transmit object lists are polled for acknowledged objects.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost transmit acknowledge messages are recognized, counted and reported to the application.

Polling is processed from low to high object numbers and CAN1 before CAN2.

4.3.2.3 Receive events

Calling *CANPC_read_ac,* the application is informed about reception of objects and other bus events.

The report of a received object and generating an interrupt to the application can be switched on/off by definition (*ReceiveIntEnable*) for filter functionality. The data of the received object are entered into the receive object list in any case.

If a receive FIFO is configured (*ReceiveFifoEnable*), the received objects and status messages are transferred through a FIFO to the application. Otherwise, the receive object lists are polled for received objects.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost messages are recognized, counted and reported to the application.

Polling is processed from low to high object numbers and CAN1 before CAN2.

4.3.2.4 Remote frames

If automatic transmission on reception of remote frames is configured by definition for an object (*AutoRemoteEnable*), the interface sends automatically a data frame with the same identifier. Otherwise, the remote frame is inserted into the object list and should be replied by the application.

If FIFO for auto remote transmission is configured (*TransmitRemoteFifoEnableAll*), the incoming remote frames are passed on for auto transmission through a FIFO. Otherwise, the remote request is stored in the transmit object lists, which are polled for transmission of data frames.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost remote transmit requests are recognized, counted and reported to the application.

Polling is processed from low to high object numbers and CAN1 before CAN2.

NOTE:
The remote frame is only answered automatically after the first call of *CANPC_supply_object_data* or *CANPC_write_object* for the related object. This assures that no non-initialized data are transmitted. If a remote frame is received before the first call of *CANPC_supply_object_data* or *CANPC_write_object* an error is reported to the application.

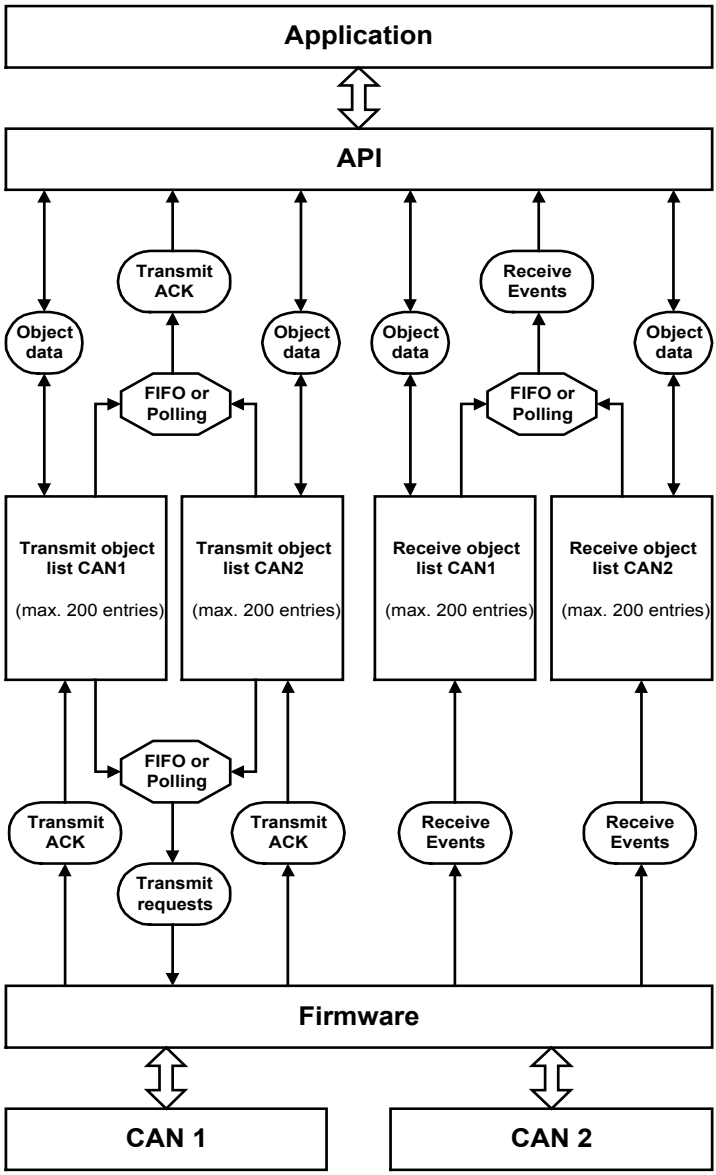


Fig. 4-3: Dynamic object buffer mode

4.3.3 Static object buffer mode (only for 11-bit identifiers)

The static object buffer mode is automatically chosen if none of the other operating modes is enabled (Fig. 4-7). In that mode the CAN messages and their data are stored in 2 object lists for CAN 1, one for transmission and one for reception (see Fig. 4-4). CAN 2 is automatically operating in FIFO mode (see section 4.3.1).

In opposition to the dynamic object buffer, the object lists holds all 2048 standard CAN identifiers (11 bit format according to CAN 2.0A spec.). The objects of these lists can be optionally defined by the application using *CANPC_define_object*. Hence, an individual configuration of the handling for each object is possible.

It is possible to access the object data at any time. Thus, the application always has a consistent representation of the complete "CAN database" of CAN 1 (only for CAN 2.0A spec.).

The handling of transmission requests, received messages, transmit acknowledges and remote frames on CAN1 can be configured individually by the application using *CANPC_initialize_interface*. The interface offers two main mechanisms for these interaction tasks, FIFO or polling.

4.3.3.1 Transmission request

CAN1:

A transmit request is commanded by *CANPC_send_object* or *CANPC_write_object*.

If the transmit FIFO is configured (*TransmitReqFifoEnable*), the transmit request for an object on CAN 1 is transferred through a FIFO to the CAN 1. Otherwise, the transmit object list is polled for objects to be sent. This polling can be limited to those transmit objects defined using *CANPC_define_object*. Otherwise, all transmit objects are polled (*TransmitPollAll*).

The FIFO has a maximum of 255 entries. An overrun of the FIFO is recognized and reported to the application.

Polling is processed from low to high identifiers.

CAN2:

The Transmit FIFO of CAN2 handles all transmit requests of the application entered by *CANPC_send_data2*. If the Transmit FIFO gets full new transmit requests are denied and the application is informed.

4.3.3.2 Transmit acknowledges

CAN1:

On successful transmission of an object a corresponding acknowledge can inform the application by using *CANPC_read_ac*.

The acknowledges can be switched on or off for either all objects (*TransmitAckEnableAll*) or for each transmit object by definition (*TransmitAckEnable*).

If the transmit acknowledge FIFO is configured (*TransmitAckFifoEnable*), the transmit acknowledges of an object are transferred through a FIFO to the application. Otherwise, the transmit object list is polled for acknowledged objects.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost transmit acknowledge messages are recognized, counted and reported to the application.

Polling is processed from low to high object numbers.

CAN2:

Transmit acknowledges on successful transmission are transferred to the application through the 'Receive FIFO'.

4.3.3.3 Receive events

CAN1:

By calling *CANPC_read_ac* the application is informed about reception of objects and other bus events.

If *ReceiveEnableAll* is set, all data and remote frames are received by the interface. Otherwise, the user can define the objects to be received (*CANPC_define_object*).

Furthermore, the report of a received object to the application and generation of an interrupt can be switched on/off either globally (*ReceiveIntEnableAll*) or individually by definition (*ReceiveIntEnable*). The data of the received object are entered into the receive object list in any case.

If FIFO mode is configured (*ReceiveFifoEnable*), the received objects and status messages are transferred through a FIFO to the application. Otherwise, the receive object list is polled for received messages.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost messages are recognized, counted and reported to the application.

Polling is processed from low to high identifiers and can be limited to those receive objects defined using *CANPC_define_object*. Then the objects are polled in succession of their definition. Otherwise, all receive objects are polled (*ReceivePollAll*).

CAN2:

Received messages and bus events are transferred to the application through the 'Receive FIFO'.

4.3.3.4 Remote frames

CAN1:

If automatic transmission on a reception of a remote frame is configured by definition for an object (*AutoRemoteEnable*) or globally (*AutoRemoteEnableAll*), the interface sends automatically a data frame with the same identifier. Otherwise, the remote request is stored in the transmit object list, which is polled for transmission of data frames.

If the FIFO for auto remote transmission is configured (*TransmitRemoteFifoEnableAll*), the incoming remote frames are passed on for auto transmission through a FIFO. Otherwise, they are stored in the object list, which is polled for transmission of data frames.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost remote transmit requests are recognized, counted and reported to the application.

CAN2:

Remote frames are passed through the receive FIFO to the application.

NOTE:
The remote frame is only answered automatically after the first call of *CANPC_supply_object_data* or *CANPC_write_object* for the related object. This assures that no non-initialized data are transmitted. If a remote frame is received before the first call of *CANPC_supply_object_data* or *CANPC_write_object* an error is reported to the application.

NOTE:
Please note that the objects defined first are also polled first, and in this way a higher priority and a lower polling time is maintained relative to the objects that follow. It is sensible to define objects in the sequence of their identifiers in order to make prioritization of objects with low identifiers the same as on the CAN bus. This is true for static as well as for dynamic object buffer mode.

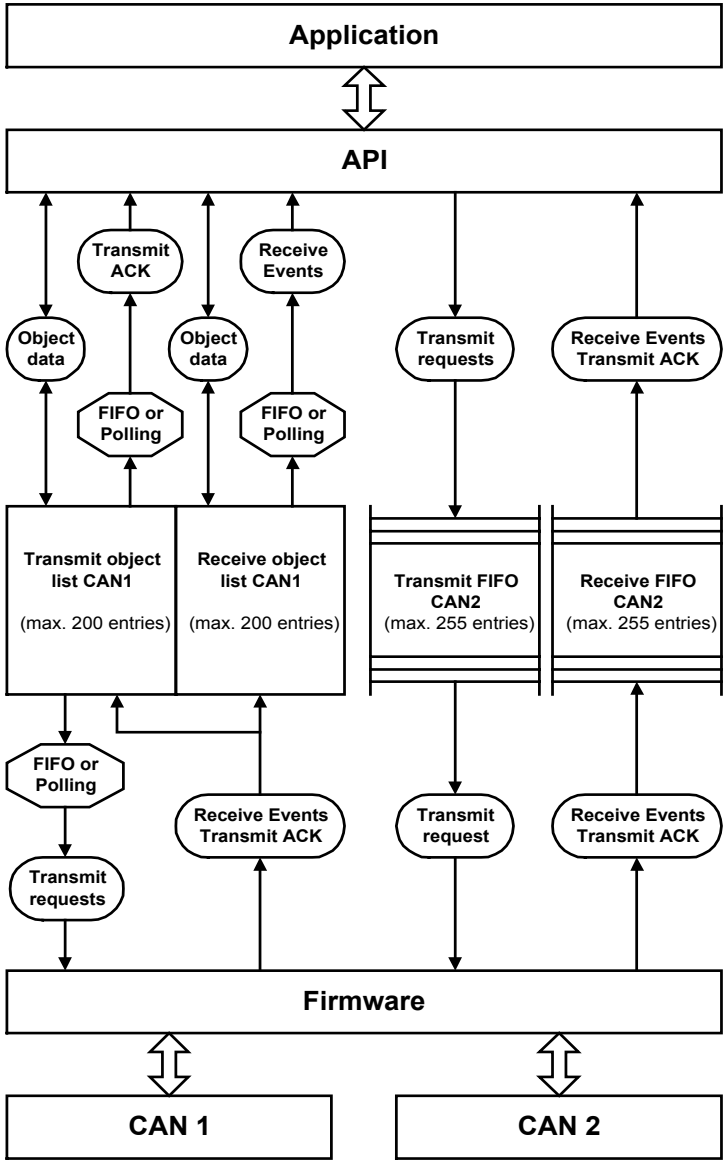


Fig. 4-4: Static object buffer mode

4.3.4 Comparison FIFO to object buffer mode

The advantage of object buffer compared to FIFO operation is that the **last** received data of an object are always available to the application in all cases. Even though, if older receptions still have not been processed. No data are lost if an overrun of the object received message FIFO occurs.

When the transmit request FIFO is full the data can be buffered, and the application is freed of this task. Hence, the transmit request is denied but the data are buffered anyway.

It is possible at any time to read out or write in the receive and transmit objects. Thus, the application always has access to the provided CAN database.

An additional advantage of the object buffer is that data of objects are available to the application very quickly after they are received from the bus, even if the application still has not processed older messages. Accordingly, messages can be transmitted before lower priority messages, even if the lower priority messages were requested first. This is true if the object buffer is operated in polling mode.

FIFO operation offers the advantage that data of an object or an identifier are not overwritten by other received data of the same object until they are evaluated by the application (overrun). Therefore, when transmitting, a sequence of data of an object can be buffered and transmitted.

Furthermore, FIFO provides full access to all identifiers possible on CAN1 and CAN2, even for extended identifier. No relation between identifier and defined object number has to be processed by the application.

4.4 Implementation

The CAN-AC2-PCI board has to be programmed in a specific sequence of instructions for proper operation.

NOTE:
For proper spelling of the library functions 'CANPC_' has to be added to the instruction syntax in Fig. 4-5, 4-6 and 4-7.

4.4.1 Board initialization

After program start the CAN interface must be initialized by *INIPC_initialize_board*. Second *CANPC_reset_board* has to be called to load the firmware.

4.4.2 CAN initialization

The CAN chips are placed into reset status using *CANPC_reset_chip*. Then, CAN specific parameters are initialized using the *CANPC_initialize_chip* for bit timing, *CANPC_set_acceptance* for filtering CAN messages and *CANPC_set_output_control* for the physical signal specification.

After the CAN specific initialization it must be decided whether FIFO, static or dynamic object buffer are to be used (see chap. 5.1).

4.4.3 FIFO mode

The function *CANPC_enable_fifo* must be called to enable the FIFO mode (Fig. 4-5). As an option, CAN channel 1 or 2 can be initialized for confirmation of successful transmissions using *CANPC_enable_fifo_transmit_ack*.

Furthermore the error frame detection can be initialized by *CANPC_enable_error_frame_detection.* If an interrupt is used, it is configured by *CANPC_set_interrupt_event*.

The function *CANPC_start_chip* ends the initialization and places the CAN controller in operating status. From this point onwards transmit jobs can be issued and incoming data can be monitored.

To monitor the bus events *CANPC_read_ac* should be polled or has to be implemented in an interrupt thread.

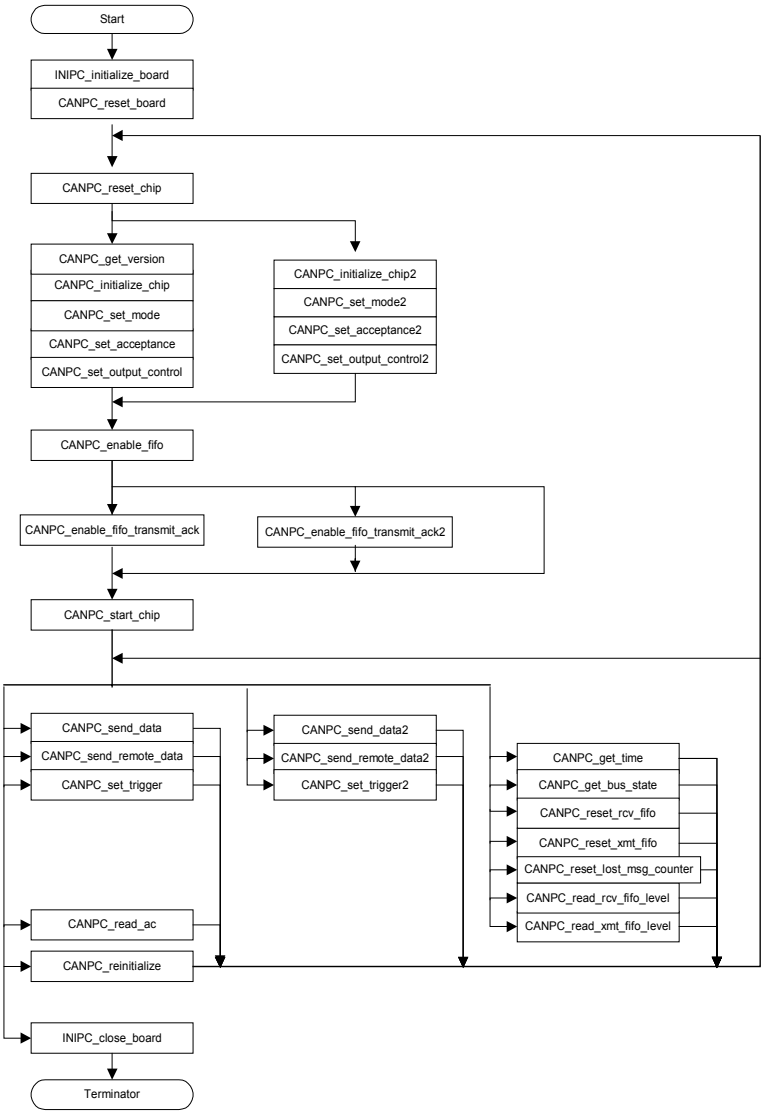


Fig. 4-5: Flow chart programming FIFO mode

4.4.4 Object buffer mode

The operating modes of object buffer are enabled using *CANPC_initialize_interface*. Beforehand the object buffer can be switched to dynamic object buffer (Fig. 4-6) by calling *CANPC_enable_dyn_obj_buf*. Otherwise the static object buffer is chosen by default (Fig. 4-7).

Object specific settings can be made by calling *CANPC_define_object*. The definition is necessary in dynamic object buffer mode but optionally in static object buffer mode.

The function *CANPC_start_chip* ends the initialization and puts the CAN-AC2-PCI in operating status. From this point onward transmit jobs can be issued and incoming data can be monitored.

To monitor the bus events *CANPC_read_ac* or *CANPC_read_rcv_data* are polled by the application.

NOTE:
The interrupt is only supported in FIFO mode.

Using it in object buffer mode can cause false function return codes of certain API functions due to interruption of their handshake communication with the firmware.

Since the static object buffer is used only for CAN channel 1, the second channel can be accessed in FIFO mode (send_data2, send_remote2).

The dynamic object buffer interfaces both CAN channels via the object lists.

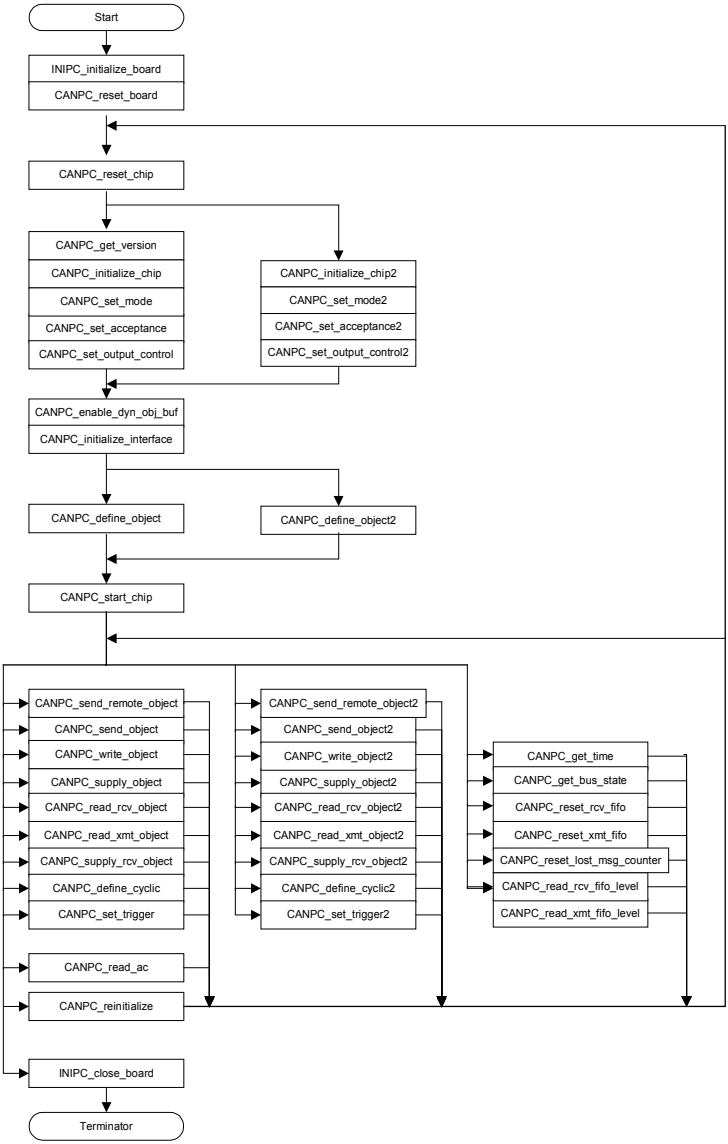


Fig. 4-6: Flow chart programming dynamic object buffer mode

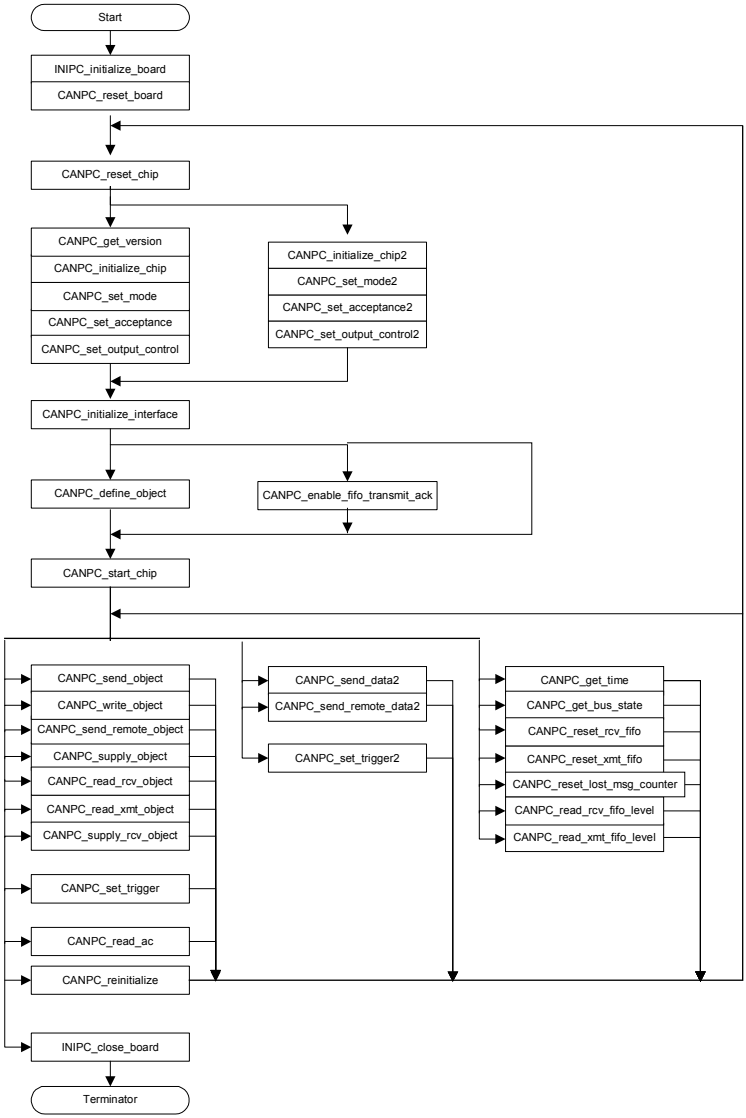


Fig. 4-7: Flow chart programming static object buffer mode

4.4.5 Exit board

The application should be finished after calling *INIPC_close_board*. This function releases the system resources locked for the application by *INIPC_initialize_board.*

Otherwise, the application may have problems to get the handle to the DPRAM a second time without system exit (e.g. applications with LABView a.o.).

NOTE:
If *INIPC_close_board* is not called at the end of operation the handle to the DPRAM may remain locked for the surrounding process. Thus, it can't be accessed by a succeeding initialization without system exit (e.g. applications with LABView).

This concerns all default exits of the application as well as program termination by errors which occur after successful call to *INIPC_initialize_board*.

4.5 Interrupt processing

4.5.1 Interrupt events

For many applications it is useful to be informed by interrupt about occurrence of CAN events. Otherwise, the CAN-AC2-PCI must be polled for new events which requires more PC processor time.

The firmware triggers a hardware interrupt to the PC on the following CAN events:

- Reception of data, remote and error frames
- Acknowledge on successful transmissions if enabled
- Change of bus state
- The hardware interrupt number is automatically assigned to the CAN-AC2-PCI board by the operating system. The application is informed about the used interrupt number by the parameter *Interrupt* in the resource structure of *INIPC_initialize_board.*

NOTE:
The interrupt is only supported in FIFO mode.

Using it in object buffer mode can cause false function return codes of certain API functions due to interruption of their handshake communication with the firmware.

4.5.2 WIN32 interrupt programming

If the CAN-AC2-PCI driver detects an interrupt it triggers a WIN32 event which can be evaluated by the application to control a WIN32 process or thread. Thus, an application or thread can be created which is only processed in case of the interrupt.

As a prerequisite the interrupt event must be created by the application. The hardware driver must be supplied with the handle of this WIN32 event by API function *CANPC_set_interrupt_event*. Furthermore a thread must be created and started which gets into WAIT status until the interrupt event is triggered by the driver. Then, the necessary interrupt activities can be processed and the thread gets back into WAIT status.

Before termination of the WIN32 process the created resources should be released for proper operation.

The application of the WIN32 interrupt is exemplary implemented in the test program 'Can_test.exe'. The interrupt relevant functions are sampled in 'Intexmpl.c' in 'Source' directory of the installed software. This C source code provides macro functions for initialization and termination of the interrupt handling as well as an interrupt service thread which may be linked to a customer application.

4.6 Description of the API functions

4.6.1 INIPC_initialize_board

int INIPC_initialize_board(

CANPC_RESSOURCES cp_resources)

Function Parameters:

Table 4-1: Elements of structure CANPC_RESSOURCES:

Type/Name	Description
unsigned short uSocket	Not used (only for compatibility to CANcard API)
unsigned short uInterrupt	Returns interrupt line enabled for the CAN-AC2-PCI (driver information)
unsigned long uIDPRAMemBase	Returns DPRAM base address enabled for the CAN-AC2-PCI (driver information)
unsigned long uIDPRMemSize	Returns DPRAM size enabled for the CAN-AC2-PCI (driver information)
ChipType uChip	Not used (only for compatibility to CANcard API)
unsigned short uIOAdress	Not used (only for compatibility to CANcard API)
unsigned short uRegisterBase	Not used (only for compatibility to CANcard API)

INIPC_initialize_board enables the memory access to the DPRAM of the CAN-AC2-PCI. Thus, it is necessarily called before any other API function.

If the DPRAM access is denied the function returns an error code which corresponds to the error cause. The error codes of *INIPC_initialize_board* are documented in Chapter 6.

The parameters of the resource structure are not to be set by the application. Some of the variables are returned with the related values of the driver setup (see Table 4-1).

NOTE:
If *CANPC_initialize_board* fails, other API functions should not be called since the non-initialized memory handle may cause an access violation.

Function Return Codes:

0:	Initialization successful
others:	See error codes in Chapter 6

4.6.2 CANPC_reset_board

int CANPC_reset_board(void)

CANPC_reset_board loads and resets the firmware on the interface. The firmware is included in the Windows DLL.

If the firmware download fails the function returns an error code which corresponds to the error cause. The error codes of *CANPC_reset_board* are documented in Chapter 6.

Function Return Codes:

0:	Loading and reset successful
Others:	see Chapter 6

4.6.3 CANPC_reset_chip

int CANPC_reset_chip(void)

This function terminates a possible bus operation and places the CAN chips into reset status.

After reset the bit timing, acceptance register and output control register have to be defined before the CAN controller are started by the related API functions.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.4 CANPC_get_version

int CANPC_get_version(

int *sw_version,
int *fw_version,
int *hw_version,
int *license,
int *can_chip_type);

This function provides useful information the version numbers of hard-, soft- and firmware, license and CAN chip types of the CAN-AC2-PCI.

It can be called after the firmware is loaded by *CANPC_reset_board*.

Function Parameters:

- **sw_version:**

Pointer to the entry of the version number of driver software.

The number is encoded as **sw_version* / 100 as the main version number; **sw_version* % 100 refers to the subordinate part of the number.

- **fw_version:**

Pointer to the entry of the version number of the firmware.

The number is encoded as **fw_version* / 100 as the main version number; **fw_version* % 100 refers to the subordinate part of the number.

- **hw_version:**

Pointer to the entry of the version number of the hardware.

The number is encoded as **hw_version* % 0x100H as the main version number; **hw_version* / 0x100H refers to the subordinate part of the number.

- **licence:**

Pointer entry of the license type of the CAN-AC2-PCI

01H:	Licensed for operation with interface software
02H:	Licensed for operation with CANalyzer software

- **can_chip_type:**

Pointer to entry containing the last three digits of the CAN chip type.

can_chip_type[0]:	CAN 1
can_chip_type[1]:	CAN 2

5:	NEC72005	(CANcard)
1000:	SJA1000	(CANcard-SJA)
		(CAN-AC2/PCI)
527:	Intel 82527	(CAN-AC2/527)
200:	Philips 82C200	(CAN-AC2)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.5 CANPC_get_serial_number

CANPC_get_serial_number(unsigned long *SerialNumber)

This function returns the serial number of the CAN-AC2-PCI in **SerialNumber*.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.6 CANPC_initialize_chip[2]

```
int     CANPC_initialize_chip(
                int     presc,
                int     sjw,
                int     tseg1,
                int     tseg2,
                int     sam)
int     CANPC_initialize_chip2(
                int     presc,
                int     sjw,
                int     tseg1,
                int     tseg2,
                int     sam)
```

Function Parameters:

Table 4-2: Bit timing parameter

Name	Description	Range
presc:	CAN-Prescaler	[1..32]
sjw:	CAN-Synchronisation-Jump-Width	[1..4]
tseg1:	CAN-Time-Segment 1	[1..16]
tseg2:	CAN-Time-Segment 2	[1..8]
sam:	Number of samples	[0, 1]

The functions define the bit timing (baud rate) of the CAN chips. *CANPC_initialize_chip* initializes CAN chip 1, *CANPC_initialize_chip2* initializes CAN chip 2. Parameters *presc*, *sjw*, *tseg1* and *tseg2* represent logical values that are used to describe the bit timing. These values are converted and written to the bus timing register 1 and 2 of the Philips SJA1000.

The **baud rate** is calculated by the following formula, whereby certain limit conditions must be maintained:

$$\text{Baud rate} = \frac{f_{crystal}}{2 * presc * (1 + tseg1 + tseg2)}$$

The crystal frequency $f_{crystal}$ is 16 MHz.

The limitations of the bit timing of the used CAN controllers lead to following **conditions**:

$$8 \leq (1 + tseg1 + tseg2) \leq 25$$

$$tseg1 + tseg2 \geq 2 * sjw$$

$$tseg2 \geq sjw$$

The prescaler divides the crystal frequency by *presc* to build the clock cycle time Δt.

The parameter *sam* defines how many **samples** are taken to detect the bit level.

sam = 0 $\rightarrow$ 1 sample (high speed buses)

sam = 1 $\rightarrow$ 3 samples (low/medium speed buses)

The sampling point is defined at the edge between time segment 1 and time segment 2. It is recommended to place the sampling point between 50% and 80% of the bit time. At high baud rates the communication is more stable if the sample is taken in the last quarter of the bit time.

The synchronization jump width is used to compensate the time shifts between the different CAN nodes in the network. It defines the maximum number *sync* of clock cycles by which the time segment 1 may be lengthened and time segment 2 shorted during resynchronization.

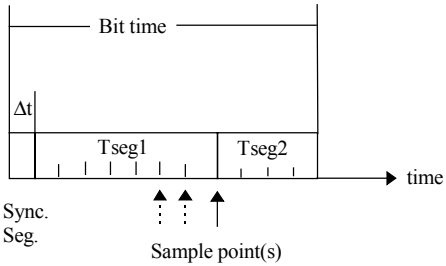


Fig. 4-8: Bit period

Table 4-3: Baud rate examples

baud rate	presc	sjw	tseg1	tseg2
1 Mbaud	1	1	4	3
800 kBaud	1	1	6	3
500 kBaud	1	1	8	7
250 kBaud	2	1	8	7
125 kBaud	4	1	8	7
100 kBaud	4	4	11	8
10 kBaud	32	4	16	8

Function Return Codes:

0:	Initialization successful
-1:	Parameter error
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.7 CANPC_set_mode[2]

```
int     CANPC_set_mode(
                int     SleepMode,
                int     SpeedMode)
int     CANPC_set_mode2(
                int     SleepMode,
                int     SpeedMode)
```

Function Parameters:

- SleepMode: default: 0
- SpeedMode: default: 0

CANPC_set_mode and *CANPC_set_mode2* are dummy function to ensure compatibility to other CAN interfaces with different CAN controllers.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.8 CANPC_set_output_control[2]

```
int     CANPC_set_output_control  (int     OutputControl)
int     CANPC_set_output_control2 (int     OutputControl)
```

Function Parameters:

- OutputControl:	Input/Output-Control-Register [0 to FF_{Hex} or -1]

This function defines the setting of the Output Control Register (OCR) of the CAN chip. This is used to adapt the CAN chip to the physical bus interface being used. *CANPC_set_output_control* initializes CAN1, and *CANPC_set_output_control2* initializes CAN2.

If the CAN-AC2-PCI with CAN controller Philips SJA1000 is used with the CAN High Speed interface (default) the output control register must be set to a value of FB_{Hex}. If you like to adapt the interface to a different bus physic consult the SJA data sheet for the required OCR setting.

Setting the OCR=03H switches off the transmission lines Tx0 and Tx1 of the CAN controller. Thus, the CAN-AC2-PCI can't send any data frame or any acknowledge bit on received messages. Thus, the interface can monitor the activities on the CAN network without influencing it.

The OCR specification of the Philips SJA1000 is described in Table 4-4 to 4-7.

The default values for CAN High Speed can also be chosen automatically by passing the default parameter -1 which assures compatibility with other using CAN High Speed Standard.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized:

Output control specification of Philips SJA1000:

Table 4-4: Output control Philips SJA1000

Bit	Function
7	OCTP1
6	OCTN1
5	OCPOL1
4	OCTP0
3	OCTN0
2	OCPOL0
1	OCMODE1
0	OCMODE0

Table 4-5: Output control mode of Philips SJA1000

OCMODE1	OCMODE0	Function
1	0	Normal Mode (TX0 and TX1 CAN Output)
1	1	Normal mode (Tx0 CAN Output, TX1 Bus Clock)
0	0	not implemented
0	1	not implemented

The voltage levels at the CAN outputs TX0 and TX1 depend on both the output configuration, which is determined by OCTPx and OCTNx, and the output polarity, which is determined by OCPOLx. Table 4-7 shows output status as a function of these settings for Philips SJA00.

Table 4-6: Configuration of CAN output pins TX0 and TX1

Operating Mode	OCTPx	OCTNx	OCPOLx	TXD	TPx	TNx	Level at TXx
FLOAT	0	0	0	0	off	off	high resistance
	0	0	0	1	off	off	high resistance
	0	0	1	0	off	off	high resistance
	0	0	1	1	off	off	high resistance
PULL DOWN	0	1	0	0	off	on	logic "0"
	0	1	0	1	off	off	high resistance
	0	1	1	0	off	off	high resistance
	0	1	1	1	off	on	logic "0"
PULL UP	1	0	0	0	off	off	high resistance
	1	0	0	1	on	off	logic "1"
	1	0	1	0	on	off	logic "1"
	1	0	1	1	off	off	high resistance
PUSH PULL	1	1	0	0	off	on	logic "0"
	1	1	0	1	on	off	logic "1"
	1	1	1	0	on	off	logic "1"
	1	1	1	1	off	on	logic "0"

TXx: Output pin x, x=0 for TX0, x=1 for TX1

TPx: Transistor that switches from supply voltage to TXx

TNx: Transistor that switches from TXx to ground

TXD: Data to be transmitted, 0=dominant, 1=recessive

4.6.9 CANPC_set_acceptance[2]

int CANPC_set_acceptance(

unsigned int AccCodeStd,
unsigned int AccMaskStd,
unsigned long AccCodeXtd,
unsigned long AccMaskXtd)

int CANPC_set_acceptance2(

unsigned int AccCodeStd,
unsigned int AccMaskStd,
unsigned long AccCodeXtd,
unsigned long AccMaskXtd)

Function Parameters:

Table 4-7: Filter parameters

Name	Description	Range
AccCodeStd:	Acceptance code for standard frames	[0 to $7FF_{Hex}$]
AccMaskStd:	Acceptance mask for standard frames	[0 to $7FF_{Hex}$]
AccCodeXtd:	Acceptance code for extended frames	[0 to $1FFFFFFF_{Hex}$]
AccMaskXtd:	Acceptance mask for extended frames	[0 to $1FFFFFFF_{Hex}$]

The function *CANPC_set_acceptance* initializes the acceptance filter of the CAN controller. *CANPC_set_acceptance* sets the acceptance registers of CAN1, *CANPC_set_acceptance2* sets the acceptance registers of CAN2.

The acceptance filter defines which identifiers should be passed into the receive buffer of the CAN controller.

To receive an identifier all bits of the identifier that were initialized as 1 in the acceptance mask must match the corresponding bit in the acceptance code. A "0" in the acceptance mask register means "Don't care" for the identifier bit at this position.

The parameters are converted and written into the acceptance code and mask registers of the Philips SJA1000.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.10 CANPC_enable_fifo

int CANPC_enable_fifo(void)

FIFO operation of the interface is activated calling this function (see Section 4.1.1 "FIFO Operation"). If this function is not used, then the CAN-AC2-PCI operates with an object buffer mode.

NOTE:
The static object buffer is not usable for 29bit identifiers.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.11 CANPC_enable_error_frame_detection

int CANPC_enable_error_frame_detection(void)

This function enables the detection of error frames by the application. Receiving an error frame sets the function return value of *CANPC_read_ac* to 15.

NOTE:
Error frame detection is only available in FIFO mode and for channel 2 in static object buffer.

Function Return Codes:

0:	Function successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.12 CANPC_enable_timestamps

int CANPC_enable_timestamps(void)

This is a dummy function which is only necessary to provide API compatibility to the CAN-AC2 ISA interface.

Function Return Codes:

0:	Function successful

4.6.13 CANPC_enable_fifo_transmit_ack[2]

int CANPC_enable_fifo_transmit_ack(void)
int CANPC_enable_fifo_transmit_ack2(void)

CANPC_enable_fifo_transmit_ack enables the report of successful transmit jobs in FIFO mode to the PC application.

If a transmission of a data or remote frame is acknowledged by another CAN device a related message for the application is entered into the receive FIFO and the interrupt is set. Reading the receive FIFO by *CANPC_read_ac* the acknowledges are reported by a special function return value (see section 4.3.26).

NOTE:
This function is only applicable in FIFO mode and for channel 2 in static object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.14 CANPC_enable_dyn_obj_buf

int CANPC_enable_dyn_obj_buf(void)

CANPC_enable_dyn_obj_buf configures the API to run in dynamic object buffer mode (see section 4.1.2). If this function is not used, then the CAN-AC2-PCI operates with the static object buffer or in the FIFO mode (if *CANPC_enable_fifo* has been called).

NOTE:
The static object buffer is usable for 29bit identifiers.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.15 CANPC_initialize_interface

int CANPC_initialize_interface(

int	ReceiveFifoEnable,
int	ReceivePollAll,
int	ReceiveEnableAll,
int	ReceiveIntEnableAll,
int	AutoRemoteEnableAll,
int	TransmitReqFifoEnable,
int	TransmitPollAll,
int	TransmitAckEnableAll,
int	TransmitAckFifoEnable,
int	TransmitRmtFifoEnable)

CANPC_initialize_interface configures properties and structure the object buffer (see sections 4.1.2 and 4.1.3). It may not be used in FIFO operation, i.e. after *CANPC_enable_fifo* has been called.

Function Parameters:

- **ReceiveFifoEnable:**

Type of receive message handling from firmware to PC application.

1: Receive messages of data frames or remote frames are transferred to the PC through the receive message FIFO (see section 4.1.1).

0: The PC ascertains receive messages of data frames or remote frames by polling the objects in the receive object lists using the function *CANPC_read_ac* (see 5.1.2 and 5.1.3). Under certain conditions this can cause a longer running time of the function *CANPC_read_ac*, and can therefore result in lower throughput rates.

- **ReceivePollAll:**

This flag is only meaningful for *ReceiveFifoEnable* = 0 with static object buffer (should be 0 with dynamic object buffer).

1: Polling of all receive objects when *CANPC_read_ac* is called (see section 4.1.3)

0: Polling of only those receive objects which have been defined using *CANPC_define_object* (see Section 4.1.2 and 4.1.3)

- **ReceiveEnableAll:**

This flag is only meaningful with static object buffer (must be 0 with dynamic object buffer).

1: All data frames and remote frames on CAN 1 with standard identifiers are received. No receive objects need to be defined (However: *CANPC_define_object* can be used nevertheless, in order to activate receive objects for polling by the application under the conditions *ReceivePollAll* = 0 and *ReceiveFifoEnable* = 0)

0: All receive objects that are passed to the PC must be defined beforehand using *CANPC_define_object*. Objects that are not defined using *CANPC_define_object* are not received by the (filter functionality).

- **ReceiveIntEnableAll:**

This flag is only meaningful while *ReceiveEnableAll* = 1 with static object buffer (should be 0 with dynamic object buffer).

1: When receiving an arbitrary object (declared using *CANPC_define_object* or if *ReceiveEnableAll* = 1) the receive message is passed to the PC application. Additionally, an interrupt is generated to the PC. The application program can read the object using *CANPC_read_ac*.

0: Receipt of an object is only reported to the PC (with interrupt) if the object has been declared in *CANPC_define_object* with *ReceiveIntEnable* = 1. Otherwise the data of the object will indeed be entered into object buffer (and they can be read using *CANPC_read_rcv_data*), but no information is generated for the application regarding receipt of the object (readable by *CANPC_read_ac*).

- **AutoRemoteEnableAll:**

This flag is only meaningful while *ReceiveEnableAll* = 1 with static object buffer (should be 0 with dynamic object buffer).

1: When receiving an arbitrary remote frame the interface independently transmits a data frame with the same identifier (see 5.1.3).

0: When receiving a remote frame the interface only transmits a data frame with the same identifier if the corresponding receive object has been declared in *CANPC_define_object* with *AutoRemoteEnable* = 1. Otherwise the remote frame is reported to the PC (calling *CANPC_read_ac* or *CANPC_read_rcv_data*). The PC must transmit an explicit response (data frame) then.

NOTE:
A data frame is transmitted after the first call of *CANPC_supply_object_data* or *CANPC_write_object* initialized the object data. A remote frame arriving before data initialization results in error report -6 in the function *CANPC_read_ac*.

- **TransmitReqFifoEnable:**

  1: Transmit jobs for data frames or remote frames are transferred to the CAN bus through the transmit job FIFO (see sections 4.1.2 and 4.1.3)

  0: Transmit jobs for data frames or remote frames are ascertained by the firmware by polling the objects in the transmit object lists (see sections 4.1.2 and 4.1.3).

- **TransmitPollAll:**

This flag is only meaningful for *TransmitReqFifoEnable* = 0 with static object buffer (should be 0 with dynamic object buffer).

1 Polling of all transmit objects (see section 4.1.3)

0: Polling of only those transmit objects that have been defined using *CANPC_define_object* (see section 4.1.2 and 4.1.3)

- **TransmitAckEnableAll:**

  1: The interface acknowledges (in conjunction with an interrupt to the PC) all data frames and remote frames after successful transmission on the bus. This acknowledgment can be read in *CANPC_read_ac* or *CANPC_read_xmt_data* (see section 4.1.2 and 4.1.3).

  0: All objects whose data frames and remote frames are to be acknowledged by the Interface after successful transmission, must have been declared with the parameter *TransmitAckEnable*=1 in *CANPC_define_object*. Transmission of all other objects is not reported to the application.

- TransmitAckFifoEnableAll:

  1: Acknowledgements of transmitted data frames or remote frames are transferred to the application through the transmit-acknowledge-FIFO (see sections 4.1.2 and 4.1.3).

  0: Acknowledgements of transmitted data frames or remote frames are ascertained by polling of the objects in the interface (see sections 4.1.2 and 4.1.3). Under certain conditions this can cause a longer running time of the function *CANPC_read_ac* and thus lead to lower throughput rates of the interface.

- **TransmitRmtFifoEnableAll:**

This parameter selects the handling mechanism for objects with Auto Remote Control configured (*AutoRemoteEnable* is set).

1: Incoming remote frames are buffered in a FIFO and are passed on for transmission of data frames (see sections 4.1.2 and 4.1.3").

0: Incoming remote frames are stored in object lists, which are polled for transmission of data frames (see section 4.1.3").

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-5:	DPRAM size conflict (stat. obj. buf. requires 64kByte)
-6	Parameter conflict
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.16 CANPC_define_object[2]

```
int CANPC_define_object(
        unsigned long   Ident,
        int             *ObjectNumber,
        int             Type,
        int             ReceiveIntEnable,
        int             AutoRemoteEnable,
        int             TransmitAckEnable)

int CANPC_define_object2(
        unsigned long   Ident,
        int             *ObjectNumber,
        int             Type,
        int             ReceiveIntEnable,
        int             AutoRemoteEnable,
        int             TransmitAckEnable)
```

The function *CANPC_define_object* defines and configures the communication objects of the transmit and receive object lists in object buffer mode. *CANPC_define_object2* for channel 2 can only be used in the dynamic object buffer mode.

In dynamic object buffer mode all used objects have to be defined, while in static object buffer mode the function can be used optionally for individual configuration of the object handling.

In static object buffer mode the returned object number equals the identifier. But in dynamic object buffer mode it corresponds to the succession of definition in the related object list.

NOTE:
The API functions handle the objects by their object number. Hence, the user is recommended to setup a table of relations between identifier and object number in dynamic object buffer mode.

Function Parameters:

- Ident:

Identifier

[0 to $7FF_{Hex}$] for standard objects
[0 to $1FFFFFFF_{Hex}$] for extended objects

- ObjectNumber:

In the mode dynamic object buffer the object number in the related object list is returned in this parameter. It is a handle for the online access to this object (*CANPC_send_object*, *CANPC_read_rcv_data*...).

The identifier itself will no longer be referenced. In the mode static object buffer the object number is equally to the identifier.

- Type:

Direction of transmission and type of identifier

0: Standard receive object: Data frames and remote frames with standard identifiers (11 bit) can be received.

1: Standard transmit object: Data frames and remote frames with standard identifiers (11 bit) can be transmitted.

2: Extended receive object: Data frames and remote frames with extended identifiers (29 bit) can be received.

3: Extended transmit object: Data frames and remote frames with extended identifiers (29 bit) can be transmitted.

- ReceiveIntEnable (only for receive objects):

1: When receiving an object with the identifier *Ident* the receive **message** is passed to the PC application. Additionally, an interrupt is generated to the PC. The application program can read the object using *CANPC_read_ac*.

0: After receipt of an object the object data are indeed entered into object buffer (and they can be read using *CANPC_read_rcv_data*), but no information is generated for the application regarding receipt of the object. No interrupt is generated to the PC.

- AutoRemoteEnable (only for receive objects):

1: When receiving a remote frame with the identifier *Ident* the CAN-AC2-PCI transmits a data frame with the same identifier independently form the PC (see sections 4.1.2 and 4.1.3).

0: When receiving a remote frame the remote frame is reported to the PC (can be read using *CANPC_read_ac* or *CANPC_read_rcv_data*). The PC must transmit an explicit response (data frame).

NOTE:
The remote frame is only answered automatically after the first call of *CANPC_supply_object_data* or *CANPC_write_object*. This assures that no non-initialized data are transmitted. A remote frame arriving before the first call of *CANPC_supply_object_data* or *CANPC_write_object* results in error report -6 in the function *CANPC_read_ac*. For the auto remote feature it is necessary to define a transmit object as well as a receive object with the same identifier.

- TransmitAckEnable (only for transmit objects):

1 A data frame or remote frame with the identifier *Ident* is acknowledged (in conjunction with an interrupt to the PC) after successful transmission. This acknowledgement can be read using *CANPC_read_ac* or *CANPC_read_xmt_data* (see 5.1.2 and 5.1.3).

0: A data frame or remote frame with the identifier *Ident* is not acknowledged to the application after successful transmission on the bus.

NOTE:
Please note that the objects defined first are also polled first, and in this way a higher priority and a lower polling time is maintained relative to the objects that follow. It is sensible to define objects in the sequence of their identifiers in order to make prioritization of objects with low identifiers the same as on the CAN bus. This is true for static as well as for dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Parameter error
-2	Dyn. Obj. buffer mode not enabled
-3:	Error accessing DPRAM
-4:	Timeout firmware communication
-5:	DPRAM size conflict (stat. Obj. buf. requires 64kByte)
-6.	Parameter conflict
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.17 CANPC_optimize_rcv_speed

int CANPC_optimize_rcv_speed(void)

This is a dummy function (only for compatibility to the CAN-AC2 V4.0 API from Softing).

Function Return Codes:

0:	Function successful

4.6.18 CANPC_start_chip

int CANPC_start_chip(void)

The function *CANPC_start_chip* puts the CAN controllers of both CAN channels into operational mode. From now on transmit jobs can be issued and reception of messages is monitored.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.19 CANPC_define_cyclic[2]

```
int CANPC_define_cyclic(
        int             ObjectNumber,
        unsigned int    Rate,
        unsigned int    Cycles)

int CANPC_define_cyclic2(
        int             ObjectNumber,
        unsigned int    Rate,
        unsigned int    Cycles)
```

The function *CANPC_define_cyclic* defines cyclic transmission of a communication object for CAN channel1 previously defined by *CANPC_define_object*. (*CANPC_define_cyclic2* for channel 2)

The cyclic transmission is started and stopped by the value of *Rate.* The settings (transmission start/stop) are put into operation by the first call of *CANPC_send_object* or *CANPC_write_object* for the object after the definition call.

Alternatively, the cyclic transmission is stopped automatically if the defined number of cycles *Cycles* is reached.

NOTE:
If defined and started a cyclic object has to be stopped before any succeeding redefinition. Redefinition of the cycle rate while running the transmission results in faulty transmission.

The transmitted data contents are defined by *CANPC_supply_object* or *CANPC_write_object*. They can be modified during cyclic transmission as well.

NOTE:
This function can only be used in dynamic object buffer mode.

Function Parameters:

- ObjectNumber:

Object reference returned by *CANPC_define_object*.

- Cycles [0..65535]:

0:	Unlimited cyclic repetition
1..65535:	Number of cyclic repetitions

- Rate [0..65535]:

0:	Disable cyclic transmission (stop)
1..65535:	Transmission rate in ms

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.20 CANPC_send_remote_object

```
int CANPC_send_remote_object(
        int     ObjectNumber,
        int     DataLength)
int CANPC_send_remote_object2(
        int     ObjectNumber,
        int     DataLength)
```

Function Parameters:

- ObjectNumber: Object number
- DataLength: Number of data bytes

This function initiates transmission of a remote frame for a transmit object specified by the object number. The remote frame has a data length 0; however, the data length code is physically transmitted with the data length code *DataLength*.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by *CANPC_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 4.1.2 and 4.1.3).

CANPC_send_remote_object transmits a remote frame on CAN channel 1, *CANPC_send_remote_object2* transmits a remote frame on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Last request still pending
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.21 CANPC_supply_object_data[2]

```
int CANPC_supply_object_data(
        int     ObjectNumber,
        int     DataLength,
        byte    *pData)

int CANPC_supply_object_data2(
        int     ObjectNumber,
        int     DataLength,
        byte    *pData)
```

Function Parameters:

- ObjectNumber:	Object number
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be transmitted

This function enters current data into the object buffer of the transmit object specified by *ObjectNumber*.

The data are not transmitted directly onto the bus, but rather they are prepared for pickup by a remote frame (Auto Remote) or a later transmit job (later: *CANPC_send_object*).

ObjectNumber is the reference to the object returned by *CANPC_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 4.1.2 and 4.1.3).

CANPC_supply_object_data supplies transmit data for an object on CAN channel 1, *CANPC_supply_object_data2* supplies transmit data for an object on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Request overrun
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.22 CANPC_supply_rcv_object_data[2]

```
int CANPC_supply_rcv_object_data(
        int     ObjectNumber,
        int     DataLength,
        byte    *pData),

int CANPC_supply_rcv_object_data2(
        int     ObjectNumber,
        int     DataLength,
        byte    *pData)
```

Function Parameters:

- ObjectNumber:	ObjectNumber
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be written in the object buffer

This function enters new data into the object buffer of the specified receive object.

This function can be used for initialization of receive objects in order to get reasonable values even before the first reception of a respective data frame took place.

ObjectNumber is the reference to the object returned by *CANPC_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 4.1.2 and 4.1.3).

CANPC_supply_rcv_object_data supplies receive data for an object on CAN channel 1, *CANPC_supply_rcv_object_data2* supplies receive data for an object on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.23 CANPC_send_object[2]

```
int CANPC_send_object(
        int     ObjectNumber,
        int     DataLength),

int CANPC_send_object2(
        int     ObjectNumber,
        int     DataLength)
```

Function Parameters:

- ObjectNumber: ObjectNumber
- DataLength: Number of data bytes to be transmitted

This function transmits a data frame for the transmit object specified by *ObjectNumber*. The data frame has a length of *DataLength* bytes. The data transmitted are the last entered into transmit object buffer using *CANPC_supply_object_data* or *CANPC_write_object*.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO to be further processed. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by *CANPC_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 4.1.2 and 4.1.3). *CANPC_send_object* transmits a data frame on CAN channel 1, *CANPC_send_object2* transmits a data frame on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Request overrun
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.24 CANPC_write_object[2]

```
int CANPC_write_object(
        int     ObjectNumber,
        int     DataLength,
        byte    *pData),

int CANPC_write_object2(
        int     ObjectNumber,
        int     DataLength,
        byte    *pData)
```

Function Parameters:

- ObjectNumber:	ObjectNumber
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be transmitted

This function performs an update of the data in the object buffer of the transmit object specified by *ObjectNumber*. Then a data frame is transmitted with *DataLength* bytes.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO to be further processed. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by CANPC_define_object. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 4.1.2 and 4.1.3). *CANPC_write_object* transmits a data frame on CAN channel 1, *CANPC_write_object2* transmits a data frame on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Request overrun
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.25 CANPC_read_rcv_data[2]

```
int CANPC_read_rcv_data(
        int             ObjectNumber,
        byte            *pRCV_Data,
        unsigned long   *Time)

int CANPC_read_rcv_data2(
        int             ObjectNumber,
        byte            *pRCV_Data,
        unsigned long   *Time)
```

Function Parameters:

- ObjectNumber:	Object number
- pRCV_Data:	Pointer to the address field of data being received
- Time:	Pointer to a time stamp parameter

This function copies the data of the receive object specified by *ObjectNumber* to the address *pRCV_Data*. The data are read, even if no new data were received. 8 data bytes are always copied to *pRCV_Data*, independent of the length of the received data frame.

If data in the object buffer are overwritten before they were read by the application or a remote request is not read quickly enough an overrun is signaled to the application by the function return code (overrun in object buffer).

If a remote frame was received the user is informed by a specific return code.

Time returns the instant of the last received data with a resolution of 1 microsecond (time stamp is reset in *CANPC_start_chip*).

ObjectNumber is the reference to the object returned by *CANPC_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 4.1.2 and 4.1.3).

CANPC_read_rcv_data reads the data of an object of CAN channel 1. *CANPC_read_rcv_data2* reads the data of an object on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	No new data received
1:	Data frame received
2:	Remote frame received
-1:	Receive data frame overrun
-2:	Receive remote frame overrun
-3:	Object no active
-7:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.26 CANPC_read_xmt_data[2]

```
int CANPC_read_xmt_data(
        int     ObjectNumber,
        int     *pDataLength,
        byte    *pXMT_Data),

int CANPC_read_xmt_data2(
        int     ObjectNumber,
        int     *pDataLength,
        byte    *pXMT_Data)
```

Function Parameters:

- ObjectNumber:	ObjectNumber
- pDataLength:	Pointer to entry of number of transmitted data bytes
- pXMT_Data:	Pointer to the address field of data to be transmitted

This function reads the data and the initialized data length of the transmit object specified by *ObjectNumber*. Further, it checks whether a frame has been transmitted for this object.

If no transmission acknowledgments are returned by the object the function return code 1 indicates that the last transmit job was acknowledged by another CAN node. The return code -1 means that the last transmission acknowledgment has not been read by the application yet.

ObjectNumber is the reference to the object returned by *CANPC_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 4.1.2 and 4.1.3).

CANPC_read_xmt_data reads the data of an transmit object on CAN channel 1, *CANPC_read_xmt_data2* reads the data of an transmit object on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	No message was transmitted
1:	Message was transmitted
-1:	Transmit acknowledge overrun
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.27 CANPC_send_data[2]

```
int CANPC_send_data(
        unsigned long   Ident,
        int             Xtd,
        int             DataLength,
        byte            *pData)

int CANPC_send_data2(
        unsigned long   Ident,
        int             Xtd,
        int             DataLength,
        byte            *pData)
```

Function Parameters:

- Ident:	Identifier
- Xtd:	Identifier length 0: Standard Identifier 1: Extended Identifier
- DataLength:	Number of data bytes to be transmitted
- pData:	Pointer to the address field of the data

This function transmits a data frame with the passed parameters on CAN channel 1 (*CANPC_send_data*) or on CAN channel 2 (*CANPC_send_data2*).

The transmit request is processed through the transmit FIFO. If the FIFO is full the application is informed by the return value.

NOTE:
The function *CANPC_send_data* can only be used in FIFO mode, not in object buffer mode. The function *CANPC_send_data2* can only be used in FIFO mode or static object buffer mode, not in dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.28 CANPC_send_remote[2]

```
int CANPC_send_remote(
        unsigned long   Ident,
        int             Xtd,
        int             DataLength)

int CANPC_send_remote2(
        unsigned long   Ident,
        int             Xtd,
        int             DataLength)
```

Function Parameters:

- Ident:	Identifier
- Xtd:	Identifier length 0: Standard Identifier 1: Extended Identifier
- DataLength:	Number of data bytes requested remote

This function transmits a remote frame with the Identifier *Ident* on CAN channel 1 (*CANPC_send_remote*) or on CAN channel 2 (*CANPC_send_remote2*). The remote frame has data length 0; however, the data length specified by the parameter *DataLength* is transmitted in the DLC field of the remote frame.

The transmit request is processed through the transmit FIFO. If the FIFO is full the application is informed by the return value.

NOTE:
The function *CANPC_send_remote* can only be used in FIFO mode, not in object buffer mode. The function *CANPC_send_remote2* can only be used in FIFO mode or static object buffer mode, not in dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.29 CANPC_read_ac

int CANPC_read_ac(param_struct *ac_param)

By calling this function the application is informed about data transmission and reception as well as about various error conditions and bus events.

Several different CAN events can be distinguished by evaluation of the function return code (see Table 4-8). Certain information and parameters of interest are transferred in the elements of the parameter structure *param_struct* .

Elements of structure param_struct:

NOTE:
RC1 through RC12 in brackets specify the function return codes of *CANPC_read_ac* for which the described parameter is valid. The application should not evaluate the parameter if it comes with a different function return code than stated below.

- **unsigned long Ident:**

Identifier (FIFO mode) or object number (object buffer mode) of the data or remote frame which was received or successfully transmitted.

(RC1, RC2, RC3, RC8, RC9, RC10, RC11, RC12)

- **int DataLength:**

Number of received (RC1, RC9) or transmitted (RC3, RC10) data bytes.

The *DataLength* of the received frame is only valid in FIFO mode and should not be used in object buffer mode. In object buffer mode the data length of the CAN messages should be predefined by the project.

- **int RecOverrun_flag:**

The last received data of object *Ident* were not read by the PC and were overwritten by the new data (RC1, RC2, RC9, RC12). Only valid in object buffer mode.

- **int RCV_fifo_lost_msg:**

Number of lost messages in receive FIFO (RC1, RC2, RC8, RC9, RC11, RC12). Only valid in FIFO mode.

- **byte RCV_data[8]:**

Data bytes of the received data frame (RC1, RC9).

- **int AckOverrunFlag:**

This flag is set if an unread transmit acknowledge for a transmit object is overwritten by a new one (RC3, RC10). Only valid in object buffer mode.

- **int XMT_ack_fifo_lost_acks**:

Number of lost acknowledges messages in transmit-acknowledge-FIFO in object buffer mode due to FIFO overrun(RC3, RC10).

Only valid in mode object buffer configured with *TransmitAckFifoEnable*=1.

- **int XMT_rmt_fifo_lost_remotes:**

Number of lost jobs in remote transmit FIFO (RC4). Only valid in object buffer mode initialized with *TransmitRmtFifoEnable*=1.

- **int Bus_state:**

Returns the new CAN bus status if a status change occurred (RC5).

0:	error active
1:	error passive
2:	bus off

- **int Error_state:**

Not used. Only for conformity to CANcard and CAN-AC2 (ISA) API.

- **int Can:**

Number of CAN channel (1 or 2) where the event occurred which is defined by the function return code.

(RC1, RC2, RC3, RC4, RC5, RC7, RC8, RC9, RC10, RC11, RC12,RC15)

- **unsigned long Time:**

Time stamp of signaled events with a resolution of 1µs. The timer is reset in *CANPC_start_chip*. (RC1, RC2, RC9, RC12, RC3, RC5, RC8, RC10, RC11, RC15)

Table 4-8: Function return codes of CANPC_read_ac

FRC	Explanation
0:	No new event
1:	Standard data frame received
2:	Standard remote frame received
3:	Transmission of a standard data frame is confirmed
4:	Overrun of the remote transmit FIFO. Only with object buffer and auto remote feature.
5:	Change of bus status
6:	not implemented
7:	Not used
8:	Transmission of a standard remote frame is confirmed.
9:	Extended data frame received
10:	Transmission of an extended data frame is confirmed
11:	Transmission of an extended remote frame is confirmed
12:	Extended remote frame received
13, 14	Not valid. Only useful with CANcard API
15:	Error frame detected
-1:	Function not successful
-3:	Error accessing DPRAM
-4:	Timeout firmware communication
-6:	access to an abject denied, because the object has not been initialized with data using CANPC_supply_object()
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.30 CANPC_set_trigger[2]

int CANPC_set_trigger(int level);

int CANPC_set_trigger2(int level);

This function is a dummy function for compatibility to the CANcard API. It has no effect on the CAN-AC2-PCI.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.31 CANPC_reinitialize

int CANPC_reinitialize(void);

CANPC_reinitialize reinitializes and restarts the firmware loaded on the interface by *CANPC_reset_board*.

After firmware reinitialization the CAN controller should be reset and restarted (see Fig. 4-5, 4-6, 4-7).

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.32 CANPC_get_time

int CANPC_get_time(uns_long_ptr time);

Function Parameters:

- time: Time (32bit) in µs

CANPC_get_time returns the 32bit time from the onboard timer of the CAN-AC2-PCI in the parameter *time*. The unit of *time* is µs.

The timer is reset by *CANPC_reset_chip*.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.33 CANPC_get_bus_state

int CANPC_get_bus_state(int Can);

Function Parameters:

- CAN:	CAN bus number
1:	CAN 1
2:	CAN 2

CANPC_get_bus_state returns the current bus status of the CAN controller of channel number *Can*.

If the CAN controller is in bus off state it must be reset and started again to enable further access to the bus.

Function Return Codes:

0:	Error active
1:	Error passive
2:	Bus off
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.34 CANPC_reset_lost_msg_counter

int CANPC_reset_lost_msg_counter(void);

CANPC_reset_lost_msg_counter resets the counter for the receive messages which were lost while the receive FIFO remained full in FIFO mode.

The lost message counter is supplied in the parameter structure of *CANPC_read_ac*.

NOTE
This function is not useful in dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.35 CANPC_read_rcv_fifo_level

int CANPC_read_rcv_fifo_level(void);

CANPC_read_rcv_fifo_level returns the number of events in the receive FIFO waiting to be read by *CANPC_read_ac*.

In FIFO mode the transmission requests of both channels are added. In static object buffer mode only channel 2 is considered.

The FIFO level can be reset to 0 by *CANPC_reset_rcv_fifo* which clears the FIFO.

NOTE
This function is not useful in dynamic object buffer mode.

Function Return Codes:

0 ... 255:	Messages in receive FIFO
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.36 CANPC_reset_rcv_fifo

int CANPC_reset_rcv_fifo(void);

CANPC_reset_rcv_fifo resets the receive fifo in FIFO mode.

NOTE
This function is not useful in dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-3:	Error accessing DPRAM
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.37 CANPC_read_xmt_fifo_level

int CANPC_read_xmt_fifo_level(void);

CANPC_read_xmt_fifo_level returns the number of transmit jobs in the transmit FIFO waiting to be transmitted by the interface.

In FIFO mode the transmission requests of both channels are added. In static object buffer mode only channel 2 is considered.

A pending transmission request which is already entered into the transmit buffer of the CAN controller is not counted.

The FIFO level can be reset to 0 by *CANPC_reset_xmt_fifo* which clears the FIFO.

NOTE
This function is not useful in dynamic object buffer mode.

Function Return Codes:

0 ... 255:	Jobs in transmit FIFO
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.38 CANPC_reset_xmt_fifo(void);

int CANPC_reset_xmt_fifo(void);

CANPC_reset_xmt_fifo resets the transmit FIFO in FIFO mode.

NOTE
This function is not useful in dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-3:	Error accessing DPRAM
-4:	Timeout firmware communication
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done

4.6.39 CANPC_set_path

int CANPC_set_path (FAR * path)

This is a dummy function which is only necessary to provide API compatibility to the CAN-AC2 ISA and CANcard (PCMCIA) interface.

Function Return Codes:

0:	Function successful

4.6.40 CANPC_set_interrupt_event

CANPC_set_interrupt_event(HANDLE InterruptEvent)

This function gives a HANDLE (pointer) of a WIN32 event to the driver which is set if an interrupt is signaled to the PC by the CAN-AC2-PCI.

The event must be created beforehand by the application with *CreateEvent* which is a function of the WIN32 API and returns the required HANDLE. This WIN32 event can be used to control the processing of a WIN32 process or thread.

For more detailed information about the interrupt handling refer to Chapter 4.3.

0:	Function successful
-1:	Function not successful

4.6.41 INIPC_close_board

int INIPC_close_board(void)

This function releases and unlocks the system resources which were allocated by *INIPC_initialize_board*.

The function call should be applied at any possible application exit after successful call to *INIPC_initialize_board*. Otherwise, the application may have problems to get the handle to the DPRAM a second time without system exit (e.g. applications with LABView a.o.).

Function Return Codes:

0:	Function successful

Engineering notes:

5 Test program

5.1 About the test program

Together with the API driver software a simple example and test program is provided called CAN_TEST.EXE. As a 32bit console application it realizes basically the flow charts of the operational modes described in section 4.3.

The program monitors the CAN messages on the bus, informs about various CAN events and is able to transmit messages on the bus by hot key.

5.2 Testing installation an communication

It is recommended to use the program also as an installation and operation test running the interface with an loop back cable which connects channel 1 and 2 of the CAN-AC2-PCI. There, it should be assured that a valid bus termination is applied. This bus termination is described in Chapter 3 and can be set onboard by DIP switches.

First the program displays some useful information about hardware, software and firmware version as well as about serial number and license of the board. Otherwise, the error number and cause is shown if the program fails.

NOTE:
Run the program from the command line window to get the exact error code in case of a failure.

As an option the user can decide whether the CAN is monitored by **interrupt** or by **polling**:

- 'i' = Interrupt
- 'p' = Polling

Further, the **operational mode** has to be chosen:

- 'f' = FIFO
- 'd' = Dynamic object buffer
- 's' = Static object buffer

The program initializes the interface to **1 Mbaud** and output control conforms to **CAN High Speed**. If different settings are required the source code has to be changed.

NOTE:
The display of 'Chip is running' states that any initialization routines have been executed successfully. Thus, the initialization is working correctly.

After the initialization phase the program awaits input from the keyboard and monitors the CAN bus. Incoming events (*CANPC_read_ac*) are interpreted and displayed on the screen (e.g. reception of messages). Transmit and control requests can be issued to the interface (e.g. transmission of messages) using hotkeys.

NOTE:
Press 'h' for HELP to get an overview about the possible hotkeys and actions.

5.3 Example code

The C source code of the program is sampled in the ‘source’ directory of the installed software. It shows exemplary the programming of the operational modes as well as it provides basics of the 32bit interrupt programming.

The main body, board initialization routine and receive routine are sampled in CAN_TEST.C.

Operation mode specific routines are arranged within FIFO.C, DYNOBUF.C and STATOBUF.C.

In INTEXMPL.C all administrative functions for the interrupt handling are sampled. They can be directly involved into a customer application just adding the interrupt service actions into the interrupt thread.

NOTE:
The example program CAN_TEST is dedicated to start up and test the proper operation of the CAN interface, preferably with connection of CAN1 to CAN2. It may also serve as a basis for customer programs, but it is not suitable to realize customer specific communication without any changes to the source code.

For more examples you can visit our homepage http://www.softing.com or contact the technical support hotline ++49 89 456 56 337.

6 Error codes

This chapter defines the detailed error return codes of INIPC_initialize_board (Table 6-1) and CANPC_reset_board (Table 6-2) due to the variety of possible error causes while initializing the CAN-AC2-PCI DPRAM or loading the firmware onto it.

The error codes of most API functions are fully described in Chapter 4.

All possible error codes are defined in the include file CANLAY2.H. This header file is unique for all hardware platforms. Thus, some of the error codes in the header are not dedicated to the CAN-AC2-PCI at all.

6.1 INIPC_initialize_board

Table 6-1: Error codes of INIPC_initialize_board

Function return code	Error cause
FE00	No CAN device found
FE01	Internal error
FE02	General error
FE03	Time out
FE04	Driver call pending
FE05	Driver call canceled
FE06	Illegal driver call
FE07	Driver call not supported
FE08	Wrong driver DLL version
FE09	Wrong driver version
FE0A	Driver not found
FE0B	Not enough memory
FE0C	Too many devices
FE0D	Unknown device
FE0E	Device already exists
FE0F	Device already open
FE10	Resource in use
FE11	Resource conflict
FE12	Resource access error
FE13	Invalid access to physical Memory
FE14	Too many I/O ports
FE15	Unknown resource

6.2 CANPC_reset_board

Table 6-2: Error codes of CANPC_reset_board

Function return code	Description
-6	Binary data format error
-7	Binary data check sum error
-16	No card present
-17	No physical memory
-18	Invalid IRQ number
-19	Error accessing DPRAM
-20	Bad response from card
-21	SRAM seems to be damaged
-22	Invalid program start address
-23	Invalid record type
-24	No response after program start
-25	Bad response after program start
-26	PCI chip not supported
-27	Error reading PCI parameter
-38	Error initializing chip
-39	No CAN-AC2-PCI plugged in

Engineering notes:

Glossary

OS

Operating System

AC

Application **C**ontroller

API

Application **P**rogramming **I**nterface

CAN

Controller **A**rea **N**etwork

CAN-AC

CAN Application **C**ontroller

CAN-AC-PCI

CAN Application **C**ontroller **P**eripheral **C**omponent **I**nterface

CiA

CAN **i**n **A**utomation

DIP

Dual-**I**nline **P**ackage

DPRAM

Dual-**P**ort **R**andom **A**ccess **M**emory

ISA

Industry **S**tandard **A**rchitecture

ISO

International **S**tandards **O**rganization

PB

Piggy**B**ack

PC

Personal **C**omputer

PCB

Printed **C**ircuit **B**oard

PCI

Peripheral **C**omponent **I**nterconnection

RAM

Random **A**ccess **M**emory

SAB

Siemens **A**dvanced **B**oard

SRAM

Static **R**andom **A**ccess **M**emory

Index