CANusb Layer 2 API
User Manual

Version 4.0 April 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
www.softing.com

Printed in Germany 2002

HCN08E02 25.04.2002 RDF

Contents

List of figures

List of tables

Engineering notes:

Preface

About this manual

This user manual is written for users operating the CANusb interface in Windows 98 / ME and Windows 2000 / XP.

It includes the following topics:

- Chapter 1 gives a common introduction about the product 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 CANusb hardware. The main functionality is explained and the I/O connectors are defined.
- Chapter 4 includes the CAN Layer2 API function reference as well as a description of the main operational modes and their implementation sequences.
- Chapter 5 provides some helpful programming hints regarding API linking, interrupt programming and cyclic transmission.
- Useful information about the test and example program 'Can_test.exe' can be found in Chapter 6.
- Chapter 7 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 CANusb

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

Nowadays, **USB** (**U**niversal **S**erial **B**us) systems become more and more important due to their easy usage in mobile and desktop PC's.

The CANusb is an intelligent CAN interface board for IBM and compatible computers with a USB port. Together with the supplied driver libraries PC-based applications can easily be integrated into CAN networks.

The CANusb interface:

- Offers an application interface to a single CAN network.
- Provides a physical layer according to ISO 11898 CAN High Speed.
- Can be optionally equipped with another physical interface (CAN Low Speed).
- Relieves applications of real-time-sensitive tasks while receiving and transmitting CAN messages by means of buffering and filtering.
- Executes the CAN access procedures directly on its own processor. Thus, it provides good performance for time-critical tasks.
- Provides bit rates up to 1Mbit
- Supports event driven processing and cyclic transmission
- Provides onboard timer with 1μs resolution

- Is compatible to all CAN L2 APIs V4.x of CANcard2, CANcard (NEC), CANcard-SJA, CAN-ACx, CAN-ACx-PCI and CAN-ACx-104.
- Can be used with additional CAN standard software and operating system drivers.

1.2 Scope of Application

Each CANusb 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 network. The driver library supports:

- Initialization of the CAN chip
- 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)
- FIFO and object buffer operation

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

The physical interface consists of an electrically isolated CAN High Speed interfaces according to ISO 11898.

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

1.3 Supported Systems

The CANusb L2 API functions are integrated in a 32bit DLL according to standard calling 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)]
- Microsoft Visual Basic 5.0 upwards[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 56-326.

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

1.4 Scope of Delivery

Before you begin to install the CANusb you should make sure that all of the parts listed below are at hand.

The CANusb is delivered with the following components:

- CANusb interface supplied with Phillips SJA1000 (CAN 2.0B)
- USB cable (series A/B plugs)
- Compact disk with installation software and user manual (PDF)

2 How to Install CANusb

2.1 System Requirements

To run the CANusb interface your PC must meet the following requirements:

- 100% IBM-compatible
- At least one available USB port
- Windows 98 / ME or Windows 2000 / XP running
- At least 1.5 MByte free on hard disk

Windows 2000
To enhance the stability of the system USB stack it is recommended to install Service Pack 2 for Windows 2000.

2.2 Installation

1. Install the software by running ‘Setup.exe’ from installation disk and follow the instructions. The installation of the CANusb interface is processed within ‘Setup.exe’.
2. Start ‘\W32ApiDLL\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 CANusb works properly. Quit the test by pressing ‘q’ or proceed further tests (see Chapter 6).

If the test program returns any error code please refer to Chapter 7.

NOTE:
Due to the PnP mechanism of Windows the CANusb interface should not be plugged into the USB port before the software installation. If the CANusb interface is plugged in first the appearing installation assistant should be closed by clicking the 'Cancel' button.

2.3 How to Test the Installation

After installation of hardware and software the test program ‘Can_test.exe’ in ‘\W32ApiDLL’ 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 device. Errors while initializing the interface 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 Chapter 6).

2.4 Uninstall Support

2.4.1 Software

After successful installation the 'Add/Remove Software' entry in the 'Control Panel' includes the 'Softing CANusb L2 API V4.00' entry. Double clicking this entry the 'Unwise32.exe' in the '\Uninstall' directory of the CANusb software is processed and all steps of the preceding CAN L2 API installation are undone.

2.4.2 Hardware

2.4.2.1 Windows 98 / ME

1. Plug in the CANusb interface into a free USB port.
2. Open the 'System properties' in the 'Control Panel' and select the 'Device Manager' tab.
3. Select the CANusb interface within the 'SoftingFieldbusInterface' node with the right mouse button and select 'Remove' from the pop-up menue.
4. Unplug the CANusb interface from the USB port.

Repeat this procedure for each installed CANusb interface.

2.4.2.2 Windows 2000 / XP

1. From the 'Control Panel' choose 'Add/Remove Hardware'.
2. In the hardware assistant select 'Uninstall/Unplug a device' and then 'Uninstall a device'.
3. If the CANusb interface is not plugged in activate the 'Show hidden devices' checkbox below the device list.
4. Select the CANusb interface from the list and procceed with the assistant.
5. Unplug the CANusb interface from the USB port.

Repeat this procedure for each installed CANusb interface.

Engineering notes:

3 Hardware Notes

3.1 Environmental Conditions

For proper operation of the CANusb interface the following environmental conditions have to be observed:

- Operating temperature 0...+55°C
- Non-operating temperature -25...+85°C

  (transport and storage)
- Relative humidity (non condensing) 5... 95%
- Range of air pressure 860...1060hPa (mbar)
- Power supplied by the USB-Bus +5V (4,6..5,5V), max. 310mA

3.2 CAN Interface

3.2.1 Implementation

The physical CAN interface is realized by the transceiver PCA82C251 from Philips according to the CAN Highspeed Specification (ISO11898). Optocouplers are used to electrically isolate the signal lines Tx and Rx between the transceiver and the CAN controller. The supply voltage for the physical interface is provided by the PC via an isolated DC/DC converter or from an externally connected physical interface. Its ground potential is connected to PC ground via an RC network of 1MΩ resistance and 10nF capacity. The CANusb interface is connected to the CAN network via D-SUB 9 connectors.

3.2.2 Bus Termination

The CAN High Speed bus should be terminated with 124 Ω between CAN_H and CAN_L at each end of the network (see Figure 3-1). This termination resistance is usually realized in the network cable's plug.

NOTE:
Invalid bus termination may cause communication errors.

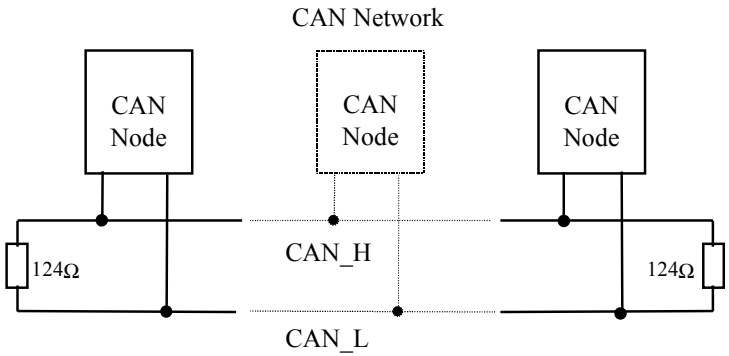

Figure 3-1: Bus termination of the CAN Highspeed connection

3.2.3 Connector Pinning

The D-Sub 9 connector's pinning is defined according to the CiA recommendation for the CAN High Speed (see Figure 3-2 and Table 3-1).

The shield is connected to earth via the PC housing. To prevent high compensation currents due to earth loops, the cable shield can be connected to pin 5 instead of the D-Sub 9 shield. This potential is connected to PC ground via an RC network of 1MΩ resistance and 10nF capacity.

Table 3-1: 9-pin D-Sub connector acc. to CIA recommendation

Pin	Signal
1	N.C.
2	CAN_L
3	Isolated GND (DCDC)
4	N.C.
5	Drain connected to connector shield (1M/10n to isolated GND)
6	Isolated GND (DCDC)
7	CAN_H
8	N.C.
9	N.C.

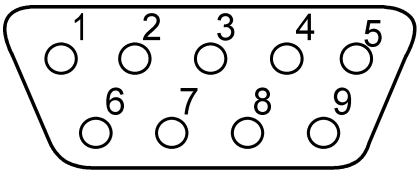

Figure 3-2: Pinning of the 9-pin D-sub connector

Engineering notes:

4 Software Description

4.1 About the CANusb API

The CANusb **API** (**A**pplication **P**rogramming **I**nterface) is supplied as a C function library for Windows 98 / ME and Windows 2000 / XP.

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 API is designed to be conform for all of Softing's CAN interfaces (PCMCIA, ISA, PCI, PC/104 etc.) and provides the following functionality:

- Initialization of CAN parameters, e.g. bit rate, output control and acceptance filter
- 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. Practical hints for linking and embedding are provided in Chapter 5.

4.2 API Driver Concept

The API functions are supplied in the 32bit Windows DLL *CANusb.dll*. This API library accesses the CANusb interface via the hardware driver *canusbw.sys* (see Figure 4-1).

The hardware driver is placed in the Windows system directory. We recommend to copy the API DLL *CANusb.dll* always to the local directory of the application to prevent access errors due to existence of API DLLs of different versions.

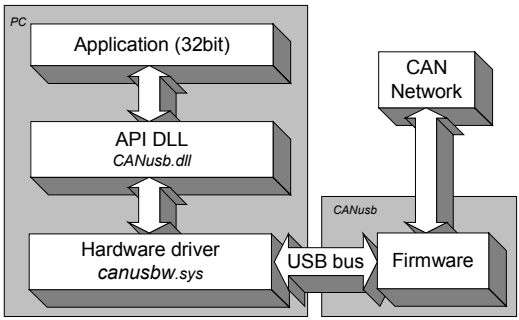

Figure 4-1: Access structure of the API software

4.3 Operational Modes of the Interface

The CANusb interface 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. For initialization of the operational modes see Chapter 4.4.3.

4.3.1 FIFO Mode

The communication between the CAN bus and the PC application is processed via FIFO buffer (Figure 4-2). The message that is entered first, is the next to be processed further. The *Transmit-FIFO* bears a maximum of 1022 entries. The *Receive-FIFO* is configurable in size from 255 entries up to 65535.

NOTE:
The variable size of the *Receive-FIFO* is only supported by the Layer 2 API for CANusb and not for the other CAN interface types. (see Chapter 4.5.10 and 5.4)

4.3.1.1 Transmission Request

The *Transmit-FIFO* handles all transmit requests of the application entered by *CANPC_send_data*.

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

4.3.1.2 Transmit Acknowledge

Acknowledges on successful transmissions are transferred to the application through the *Receive-FIFO*. They can be read out of the FIFO using *CANPC_read_ac*.

The transmit acknowledges must be switched on by *CANPC_enable_fifo_transmit_ack* during initialization (see Figure 4-6). Otherwise, a successful transmission is not confirmed to the application.

4.3.1.3 Received Messages

Received messages and bus events are transferred to the application through the *Receive-FIFO*. They can be read out of the FIFO using *CANPC_read_ac*.

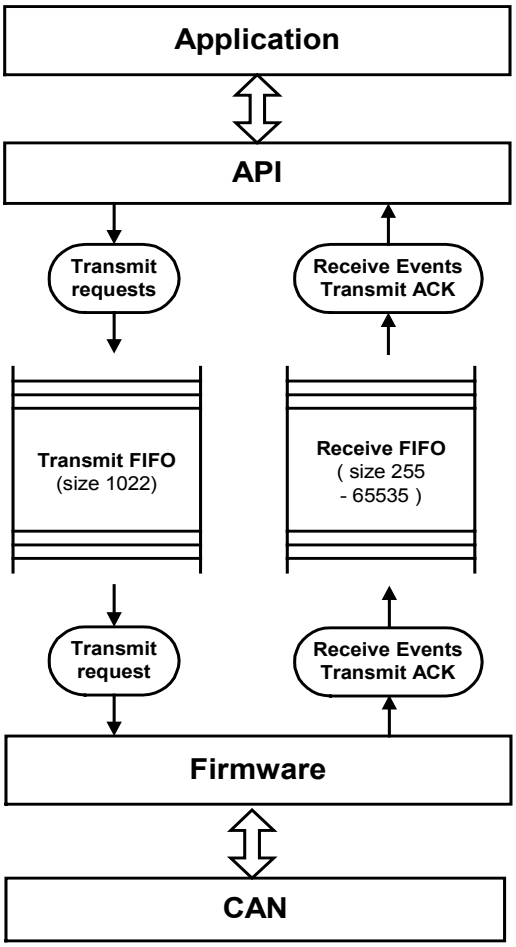

Figure 4-2: FIFO mode

4.3.2 Dynamic Object Buffer Mode

In dynamic object buffer mode the CAN messages and their data are stored in two object lists, i.e. transmission and reception list (see Figure 4-3). 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 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 referring their object number which is returned by *CANPC_define_object* to the application program. A maximum of 200 transmit and receive objects may be defined.

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. The applied mechanism are chosen and configured by *CANPC_initialize_interface*.

4.3.2.1 Transmission Request

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

If *TransmitReqFifoEnable* is set in *CANPC_initialize_interface*, the transmit request for an object is transferred to the CAN controller through a FIFO. The FIFO has a maximum of 255 entries. An overrun of the FIFO is recognized and reported to the application.

Otherwise, the transmit object list is polled for objects to be sent. Polling is processed from low to high object numbers.

4.3.2.2 Transmit Acknowledge

On a successful transmission of an object a corresponding acknowledge can be posted to the application. It is read by *CANPC_read_ac*.

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

If the transmit acknowledge FIFO is configured (*TransmitAckFifoEnable*), the transmit acknowledges are transferred through a FIFO to the application. 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.

Otherwise, the transmit object list is polled for acknowledged objects. Polling is processed from low to high object numbers.

4.3.2.3 Received Messages

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

The report of a received object and triggering 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. 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.

Otherwise, the receive object list is polled for received objects. Polling is processed from low to high object numbers.

4.3.2.4 Remote Frames

If automatic transmission on received remote frames is configured by definition for an object (*AutoRemoteEnable* in *CANPC_define_object*), 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 itself.

If FIFO for auto remote transmission is configured (*TransmitRmtFifoEnable* in *CANPC_initialize_interface*), the incoming remote frames are passed on for auto transmission through a FIFO. 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.

Otherwise, the remote request is stored in the transmit object list, which is polled for transmission of data frames. Polling is processed from low to high object numbers.

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.

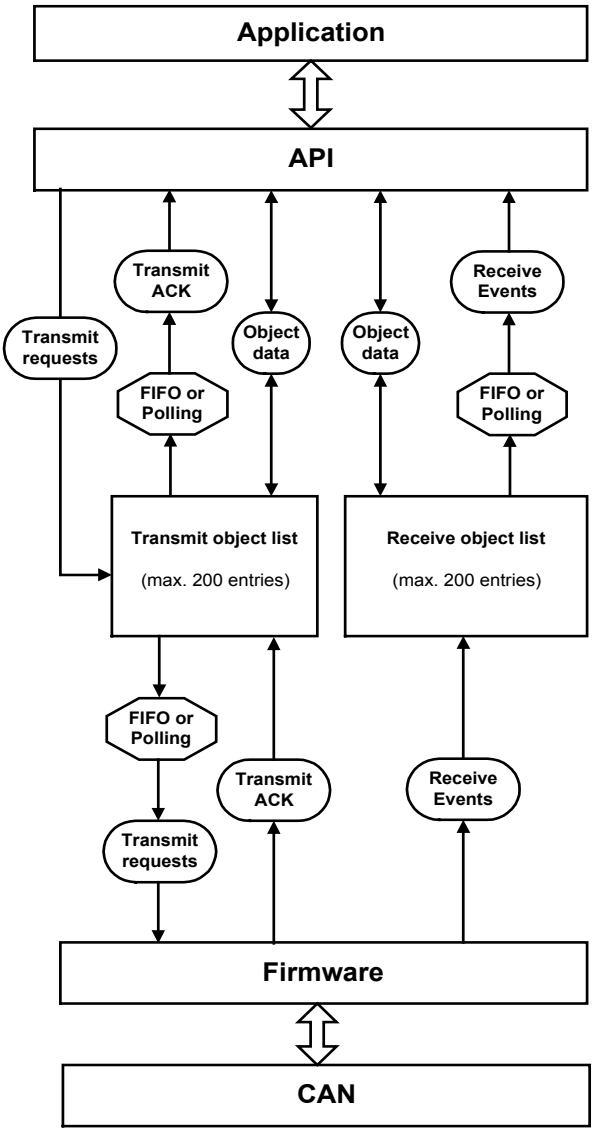

Figure 4-3: Dynamic object buffer mode

4.3.3 Static Object Buffer Mode (only for 11-bit identifiers)

In static object buffer mode the CAN messages and their data are stored in 2 object lists, one for transmission and one for reception (see Figure 4-4).

In opposition to the dynamic object buffer, the object lists hold 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 can be obtained.

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

The handling of transmission requests, received messages, transmit acknowledges and remote frames 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

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 is transferred to the CAN controller through a FIFO. 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.

4.3.3.2 Transmit Acknowledge

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.

4.3.3.3 Received Messages

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*).

4.3.3.4 Remote frames

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 (*TransmitRmtFifoEnable*), 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.

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.

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.

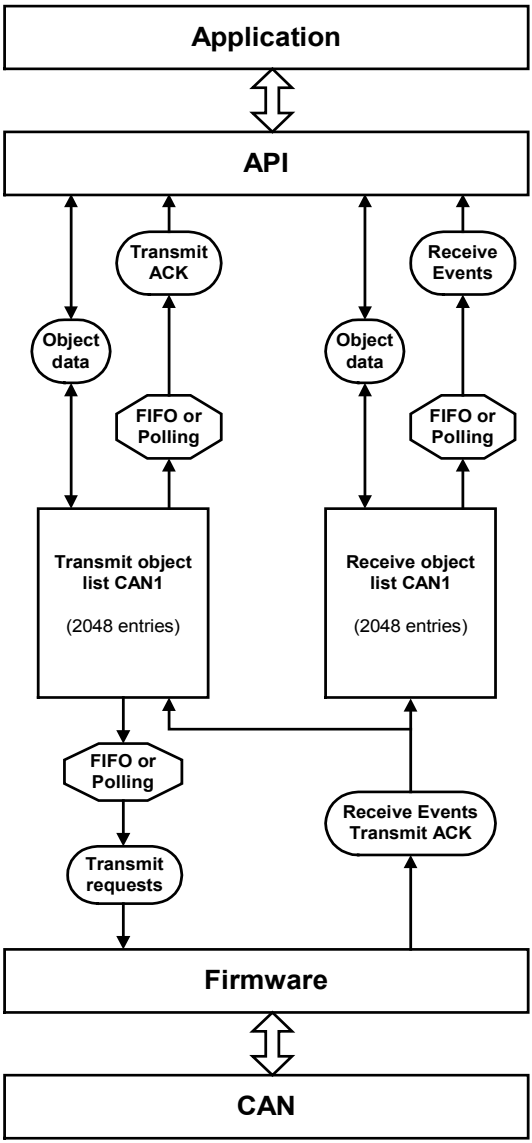

Figure 4-4: Static object buffer mode

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, even for extended identifier. No relation between identifier and defined object number has to be processed by the application.

4.4 Implementation

The CANusb has to be programmed in a specific sequence of instructions for proper operation. Figure 4-5 shows the main flow chart for CAN access with the API.

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 download the firmware.

4.4.2 CAN Initialization

The CAN chip is placed into reset status using *CANPC_reset_chip*. Subsequently, CAN specific parameters are initialized using *CANPC_initialize_chip* for bit timing, *CANPC_set_acceptance* for filtering CAN messages and *CANPC_set_output_control* for the physical signal specification. Optionally, some SW and HW information can be accessed by *CANPC_get_serial_number* and *CANPC_get_version*.

All functions (see Table 4-1) can be programmed in any succession.

Table 4-1: Functions of CAN initialization sequence

Function	Application
CANPC_initialize_chip	necessary
CANPC_set_acceptance	necessary
CANPC_set_output_control	necessary
CANPC_get_serial_number	optional
CANPC_get_version	optional

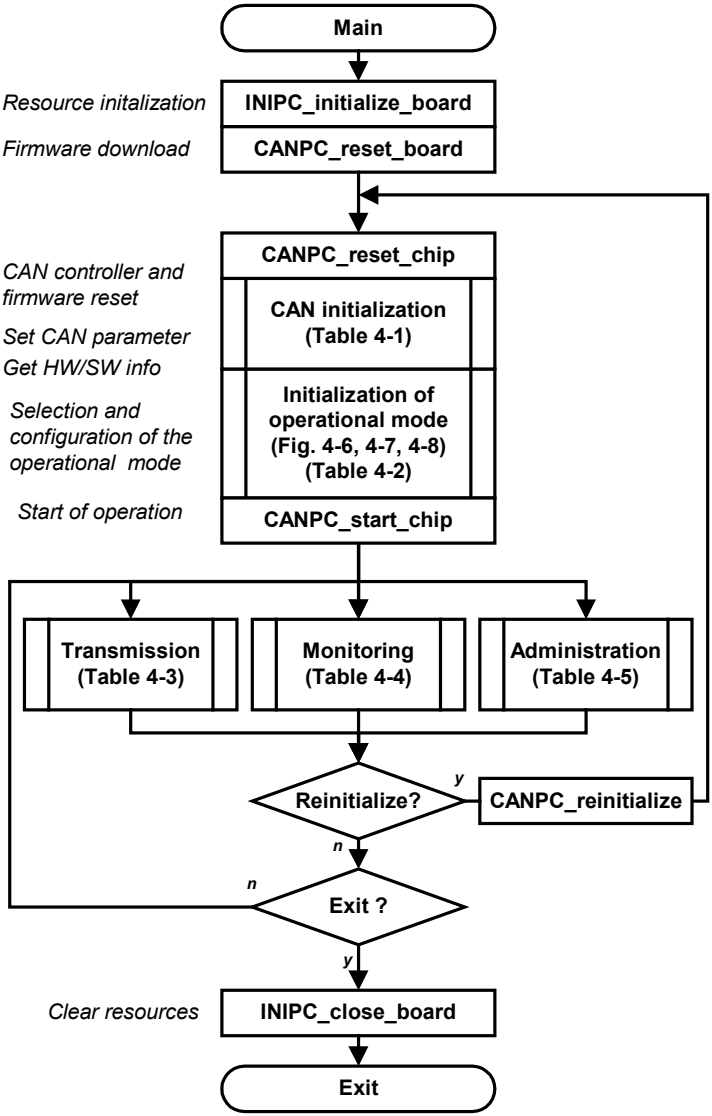

Figure 4-5: Main CAN programming sequence

4.4.3 Initialization of Operational Mode

After the CAN specific initialization the operational mode is to be configured as FIFO, dynamic (DOB) or static object buffer (SOB). The function *CANPC_enable_fifo* must be called to enable the FIFO mode. Otherwise, the object buffer is chosen which can be defined to be dynamically by *CANPC_enable_dyn_obj_buf* (see Figure 4-6 to Figure 4-8).

The chosen operational mode is configured by necessary and optional functions listed in Table 4-2. Usage and parameter sets are explained in Chapter 4.3 and 4.5.

If an WIN32 interrupt is used, it is configured by *CANPC_set_interrupt_event*. This function may be called at any time of the program but is preferably placed within the initialization routine.

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.

Table 4-2: Functions of operational mode initialization

Function	Application		
	FIFO	DOB	SOB
CANPC_enable_fifo	x	-	-
CANPC_enable_dyn_obj_buf	-	x	-
CANPC_set_rcv_fifo_size	o	-	-
CANPC_enable_fifo_transmit_ack	o	-	-
CANPC_enable_error_frame_detection	o	-	-
CANPC_initialize_interface	-	x	x
CANPC_define_object	-	x	o
CANPC_set_interrupt_event	o	o	o

x: necessary o: optional -: not possible

4.4.3.1 FIFO Mode Specific Configuration Functions

As an option, the stack can be initialized to confirm successful transmissions to the application (*CANPC_enable_fifo_transmit_ack*).

To overcome larger delays in processing the received messages the *Receive-FIFO* can be expanded optionally by *CANPC_set_rcv_fifo_size.*

Furthermore the error frame detection can be enabled by *CANPC_enable_error_frame_detection.*

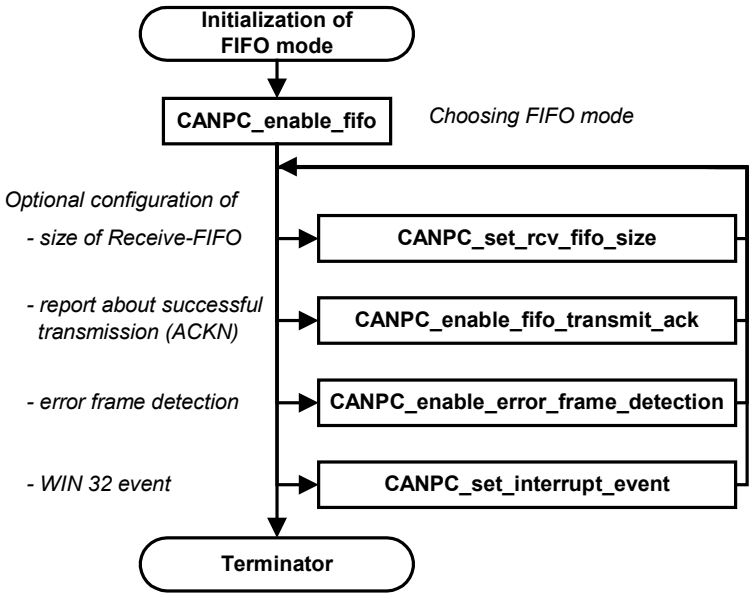

Figure 4-6: Initialization sequence for FIFO mode

4.4.3.2 Object Buffer Mode Specific Configuration Functions

The operating modes of object buffer are enabled using *CANPC_initialize_interface*. Beforehand the object buffer can be switched to dynamic object buffer (see Figure 4-7) by calling *CANPC_enable_dyn_obj_buf*. Otherwise the static object buffer is chosen by default (see Figure 4-8).

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

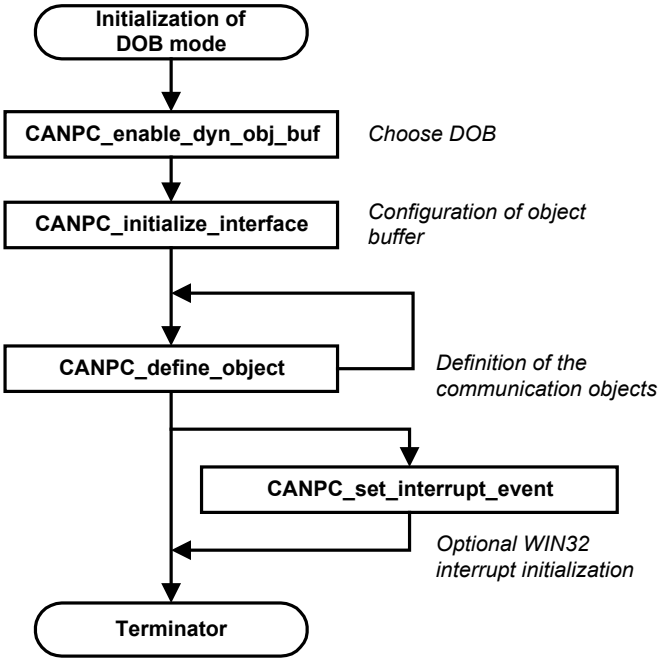

Figure 4-7: Initialization sequence for DOB mode

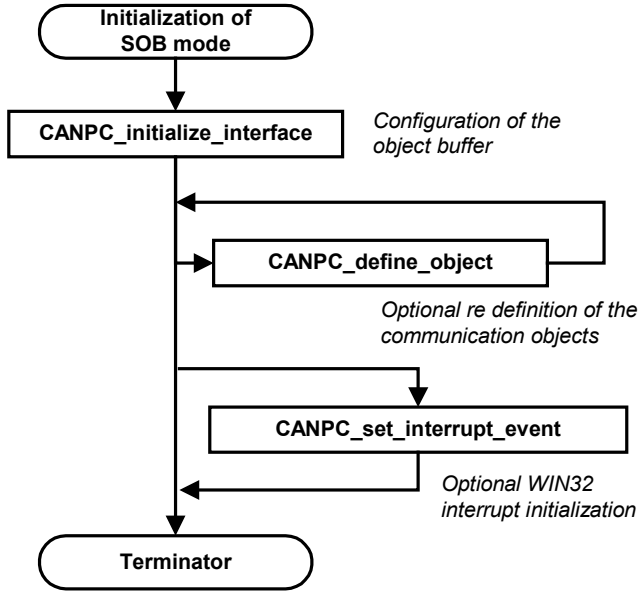

Figure 4-8: Initialization sequence for SOB mode

4.4.4 Transmission

The API enables the application to transmit data and remote frames with standard or extended identifier. Identifier length and value, data length and data contents are selectable at customers choice.

Table 4-3 lists all functions for transmission purposes and their valid operational mode. Usage and parameter sets of the functions are explained in Chapter 4.3 and 4.5.

Table 4-3: List of transmit functions

Function	Application		
	FIFO	DOB	SOB
CANPC_send_data	o	-	-
CANPC_send_remote	o	-	-
CANPC_supply_object_data	-	o	o
CANPC_send_object	-	o	o
CANPC_write_object	-	o	o
CANPC_send_remote_object	-	o	o
CANPC_define_cyclic	-	o	-

x: necessary o: optional -: not possible

4.4.5 Receiving

The API enables the application to receive data and remote frames with standard or extended identifier. Also acknowledges of successful transmissions are reported if configured during initialization (see Chapter 4.3). Identifier length and value, data length and data contents of the related frames as well as the event time can be monitored.

Table 4-4 lists all functions for frame receiving and their valid operational modes. Usage and parameter sets of the functions are explained in Chapters 4.3 and 4.5.

To monitor the bus events *CANPC_read_ac* or *CANPC_read_rcv_data* should be polled in a loop or have to be implemented in a event controlled thread/service routine.

NOTE:
In object buffer mode the event service routine should not interrupt any API function since this may cause false function return codes .

Table 4-4: List of receive functions

Function	Application		
	FIFO	DOB	SOB
CANPC_read_ac	o	o	o
CANPC_read_rcv_data	-	o	o

x: necessary o: optional -: not possible

4.4.6 Administration

The API provides numerous functions to support the customer in analyzing and administrating the operational state of the bus and/or the interface. Beside the status information of *CANPC_read_ac* following services can be accessed:

- Evaluation of the onboard time and bus states
- Evaluation and reset of the FIFO levels and the lost message counter in FIFO mode
- Evaluation of current transmit object data and presetting of receive object data

Table 4-5 lists all functions for these tasks and their valid operational mode. Usage and parameter sets of the functions are explained in Chapters 4.3 and 4.5.

Table 4-5: List of administration functions

Function	Application		
	FIFO	DOB	SOB
CANPC_get_time	o	o	o
CANPC_get_bus_state	o	o	o
CANPC_read_rcv_fifo_level	o	-	-
CANPC_reset_rcv_fifo	o	-	-
CANPC_reset_lost_msg_counter	o	-	-
CANPC_read_xmt_fifo_level	o	-	-
CANPC_reset_xmt_fifo	o	-	-
CANPC_supply_rcv_object_data	-	o	o
CANPC_read_xmt_data	-	o	o

x: necessary o: optional -: not possible

4.4.7 Reinitialization

Often it is necessary to change object settings or CAN parameters or simply to reset the CAN controller from Bus-Off state. In this case the flow chart can be reentered at *CANPC_reset_chip* by calling *CANPC_reinitialize* (Figure 4-5). The operational mode (firmware) as well as the CAN parameters are reset by *CANPC_reset_chip*. Subsequently, calling this function requires again calling of the initialization functions and *CANPC_start_chip*.

4.4.8 Exit board

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

4.5 API functions Reference

4.5.1 INIPC_initialize_board

int INIPC_initialize_board(

CANPC_RESSOURCES cp_resources)

The resources required by the CANusb are enabled using *INIPC_initialize_board*. Thus, it must be called before any other API function.

If the function fails it returns an error code which corresponds to the error cause. The error codes of *INIPC_initialize_board* are documented in the header file *Canlay2.H*.

The distinction is done by *uSocket* and *ulDPRAMMemBase* of *CANPC_RESSOURCES* (see Table 4.6).

Function Parameters:

Table 4-6: Elements of structure CANPC_RESSOURCES

Type/Name	Description
unsigned short uSocket	Select CANusb interface. 0xFFFF: Take first CANusb interface found by the driver. 0x0: Device selected by its serial number in *ulDPRAMMemBase*.
unsigned short uInterrupt	Not used.
unsigned long ulDPRAMemBase	Serial number of the CANusb interface. (only used with uSocket=0x0)
unsigned long ulDPRMemSize	Not used.

ChipType uChip	Not used.
unsigned short uIOAdress	Not used.
unsigned short uRegisterBase	Not used.

Function Return Codes:

0:	Initialization successful
others:	See Canlay2.h and Chapter 7

4.5.2 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 API 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 7.

Function Return Codes:

0:	Loading and reset successful
Others:	see Chapter 7

4.5.3 CANPC_reset_chip

int CANPC_reset_chip(void)

This function terminates a possible bus operation and places the CAN chip into reset status. It also performs a firmware reset.

After the reset the bit timing, acceptance register, output control register and operational mode have to be configured before the CAN controller is started by *CANPC_start_chip*.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.4 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 information about the version numbers of hard-, soft- and firmware, license and CAN chip type of the CANusb interface.

It can be called optionally after the firmware and the CAN controller is reset by *CANPC_reset_chip* but before starting the operation by *CANPC_start_chip*.

Function Parameters:

- **sw_version:**

Pointer to the entry of the version number of the API 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 CANusb interface

01H:	Licensed for operation with interface software

- **can_chip_type:**

Pointer to entry containing the identifying digits of the CAN chip type.

can_chip_type[0]:	CAN 1
can_chip_type[1]:	CAN 2 (not valid for CANusb)

0x1000:	SJA1000
0x0:	No chip (e.g. CAN2)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-602:	Unable to open USB pipe. Restart or re-plug the CANusb.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.5 CANPC_get_serial_number

int CANPC_get_serial_number(

unsigned long *SerialNumber)

This function returns the serial number of the CANusb interface in **SerialNumber*. It can be called after firmware and CAN controller reset by *CANPC_reset_chip*.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.6 CANPC_initialize_chip

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

Function Parameters:

Table 4-7: 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]

These functions define the bit timing (baud rate) of the CAN chip. 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 shortened during resynchronization.

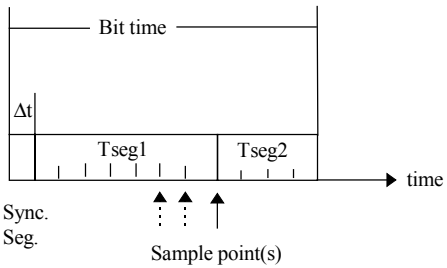

Figure 4-9: Bit period

Table 4-8: 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:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.7 CANPC_set_output_control

int CANPC_set_output_control (int OutputControl)

Function Parameters:

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

  Default: 0xFB or -1
  (CAN high-speed)

  No ACKN: 0x03
  (only listening)

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.

If the CANusb 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 FA_{Hex} or FB_{Hex}. If you like to adapt the interface to a different bus physic consult the SJA data sheet for the required OCR setting. The OCR specification of the Philips SJA1000 is described in Table 4-9 to 4-12.

Setting the OCR=03H switches off the transmission lines Tx0 and Tx1 of the CAN controller. Hence, the CANusb 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. Choosing the default values for CAN High automatically by passing the default parameter –1 assures compatibility with Softing's other CAN interfaces using CAN High Speed Standard.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.

-603: Communication via USB pipe broken.
Restart or re-plug the CANusb.

-605: Internal driver error

Output control specification of Philips SJA1000:

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 (see Table 4-11).

Table 4-9: Output control Philips SJA1000

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

Table 4-10: 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	x	not implemented

Table 4-11: 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

4.5.8 CANPC_set_acceptance

```
int          CANPC_set_acceptance(
                unsigned int     AccCodeStd,
                unsigned int     AccMaskStd,
                unsigned long    AccCodeXtd,
                unsigned long    AccMaskXtd)
```

The function *CANPC_set_acceptance* initializes the acceptance filter of the CAN controller. 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.

Table 4-12: 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 $1FFFFFFF_{Hex}$]

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.9 CANPC_enable_fifo

int CANPC_enable_fifo(void)

FIFO operation of the interface is activated calling this function If this function is not used, then the CANusb operates with an object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.10 CANPC_set_rcv_fifo_size

int CANPC_set_rcv_fifo_size(int FifoSize)

To accommodate to larger delays in processing the received messages the *Receive-FIFO* in FIFO mode is configurable in size. *CANPC_set_rcv_fifo_size* must be called if other sizes than the default size of 255 entries is to be used.

NOTE:
This function is only applicable in FIFO mode. It is only supported by the Layer 2 API for CANusb.

Function Parameters:

- FifoSize:

The value defines the size of the *Receive-FIFO* as defined in the following table:

Value	No of entries
0	255 (default)
1	511
2	1023
3	2047
4	4095
5	8191
6	16383
7	32767
8	65535

Function Return Codes:

0:	Function successful
-1:	Parameter error
-2:	FIFO mode not enabled
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.

4.5.11 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

4.5.12 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.

Function Return Codes:

0:	Function successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.13 CANPC_enable_fifo_transmit_ack

int CANPC_enable_fifo_transmit_ack(void)

CANPC_enable_fifo_transmit_ack enables the reporting of successful transmit jobs (acknowledge) 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.

NOTE:
This function is only applicable in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.14 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 Chapter 4.3.2). If this function is not used, then the CANusb is operates with the static object buffer or in the FIFO mode (if *CANPC_enable_fifo* has been called).

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.15 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 of the object buffer (see Chapters 4.3.2 and 4.3.3). 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 Chapters 4.3.2 and 4.3.3).

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*. Under certain conditions this can cause a longer running time of *CANPC_read_ac*, and can therefore result in lower throughput rates (see Chapters 4.3.2 and 4.3.3).

- **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 Chapter 4.3.3)

0: Polling of only those receive objects which have been defined using *CANPC_define_object* (see Chapter 4.3.3)

- **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 application (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 4.3.3).

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 only transmitted after the first call of *CANPC_supply_object_data* or *CANPC_write_object* initialized the object data.

- **TransmitReqFifoEnable:**

  1: Transmit jobs for data frames or remote frames are transferred to the CAN bus through the transmit job FIFO (see Chapters 4.3.2 and 4.3.3)

  0: Transmit jobs for data frames or remote frames are recognized by the firmware via polling of the objects in the transmit object lists (see Chapters 4.3.2 and 4.3.3).

- **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 Chapter 4.3.3)

  0: Polling of only those transmit objects that have been defined using *CANPC_define_object* (see Chapters 4.3.2 and 4.3.3)

- **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 by *CANPC_read_ac* or *CANPC_read_xmt_data* (see Chapters 4.3.2 and 4.3.3).

  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 Chapters 4.3.2 and 4.3.3).

  0: Acknowledgements of transmitted data frames or remote frames are recognized by polling of the objects (see Chapters 4.3.2 and 4.3.3). 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.

- **TransmitRmtFifoEnable:**

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 Chapters 4.3.2 and 4.3.3).

0: Incoming remote frames are stored in object lists, which are polled for transmission of data frames (see Chapters 4.3.2 and 4.3.3).

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-6	Parameter conflict
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.16 CANPC_define_object

```
int     CANPC_define_object(
                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.

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 equal 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 CANusb transmits a data frame with the same identifier independently from the PC (see Chapters 4.3.2 and 4.3.3).

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. 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 4.3.2, 4.3.3).

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
-4:	Communication error between host and interface
-6.	Parameter conflict
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.17 CANPC_set_interrupt_event

int CANPC_set_interrupt_event(HANDLE InterruptEvent)

This function gives a HANDLE (pointer) of a WIN32 event to the driver which is set if new CAN events were posted to the PC by the CANusb.

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.

NOTE
This function is only useful in WIN32 applications.

For more detailed information about the interrupt handling refer to Chapter 5.2.

0: Function successful

-1: Function not successful

4.5.18 CANPC_start_chip

int CANPC_start_chip(void)

The function *CANPC_start_chip* puts the CAN controller into operational mode. From now on transmit jobs can be issued and received messages are monitored.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error
-612:	Not enough memory. Check available memory resources.
-613:	Failed to start communication between host and interface.

4.5.19 CANPC_send_data

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

NOTE:
This function is only applicable in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.20 CANPC_send_remote

```
int     CANPC_send_remote(
                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*. 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:
This function is only applicable in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.21 CANPC_supply_object_data

```
int     CANPC_supply_object_data(
                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 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 Chapter 4.3).

NOTE:
This function is only applicable in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Request overrun
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.22 CANPC_send_object

```
int     CANPC_send_object(
                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 data length of *DataLength* bytes. The data transmitted are the last entered into the 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 Chapter 4.3).

NOTE:
This function is only applicable in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Request overrun
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.23 CANPC_write_object

```
int     CANPC_write_object(
                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 Chapter 4.3).

NOTE:
This function is only applicable in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Request overrun
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.24 CANPC_send_remote_object

```
int     CANPC_send_remote_object(
                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 Chapter 4.3).

NOTE:
This function is only applicable in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Last request still pending
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.25 CANPC_define_cyclic

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

The function *CANPC_define_cyclic* defines cyclic transmission of a communication object previously defined by *CANPC_define_object*.

The cyclic transmission is started and stopped by the value of *Rate.* The settings of the newly defined cyclic transmission 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 is only applicable 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:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.26 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 4-13). 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, RC3, RC8, RC9, RC10, 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 acknowledge messages in transmit-acknowledge-FIFO in object buffer mode due to FIFO overrun(RC3, RC10).

Only valid in mode object buffer configured with *TransmitAck-FifoEnable*=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. With CANusb this parameter is always 1 (only single channel).

(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. This is the actual onboard time when the event occurred. The timer is reset in *CANPC_start_chip*. (RC1, RC2, RC9, RC12, RC3, RC5, RC8, RC10, RC11, RC15)

Table 4-13: 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 used
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
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.27 CANPC_read_rcv_data

```
int     CANPC_read_rcv_data(
                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 dynamic object buffer mode it depends on the succession of definition (see Chapter 4.3).

NOTE:
This function is only applicable 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 not active
-7:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.28 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 CANusb in the parameter *time*. The unit of *time* is µs. The timer is reset by *CANPC_reset_chip*.

Please note that the evaluation of the actual time stamp can be delayed by the USB communication (max. 20 ms).

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.29 CANPC_get_bus_state

int CANPC_get_bus_state(int Can);

Function Parameters:

- CAN: CAN bus number; set always to 1 with CANusb.

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:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.30 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* (host) waiting to be read by *CANPC_read_ac*.

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

NOTE:
This function is only applicable in FIFO mode.

Function Return Codes:

0 .. *max*[1]:	Messages in receive FIFO
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

1: value of max depends on Receive-FIFO size set with CANPC_set_rcv_fifo_size (see Chapter 4.5.10). Default value is 255.

4.5.31 CANPC_reset_rcv_fifo

int CANPC_reset_rcv_fifo(void);

CANPC_reset_rcv_fifo resets the receive FIFO (host side) in FIFO mode.

NOTE:
This function is only applicable in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.32 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 only applicable in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.33 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.

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

The FIFO level can be reset to 0 by *CANPC_reset_xmt_fifo* which clears the FIFO (host side).

NOTE:
This function is only applicable in FIFO mode.

Function Return Codes:

0 ... 1022:	Jobs in transmit FIFO
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.34 CANPC_reset_xmt_fifo

int CANPC_reset_xmt_fifo(void);

CANPC_reset_xmt_fifo resets the transmit FIFO (host and firmware) in FIFO mode.

NOTE:
This function is only applicable in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.35 CANPC_supply_rcv_object_data

```
int     CANPC_supply_rcv_object_data(
                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 Chapter 4.3).

NOTE:
This function is only applicable in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.36 CANPC_read_xmt_data

```
int     CANPC_read_xmt_data(
                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 (see Chapters 4.3.2 and 4.3.3) 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 dynamic object buffer mode it depends on the succession of definition (see Chapter 4.3).

NOTE:
This function is only applicable 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:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.37 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.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-4:	Communication error between host and interface
-99:	Board not initialized: INIPC_initialize_board() was not yet called or a INIPC_close_board() was done.
-603:	Communication via USB pipe broken. Restart or re-plug the CANusb.
-605:	Internal driver error

4.5.38 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 hardware resources a second time without system exit (e.g. applications with LABView a.o.).

Function Return Codes:

0:	Function successful

5 Programming Notes

5.1 API Linking

5.1.1 General

The API DLL exports the C API functions compliant to the **'stdcall' calling convention**. This standard is supported by nearly all compiler types and visualization tools. Basically the parameter succession on the stack, stack cleaning and naming of the functions in the export table are defined by the calling convention.

The API functions appear in the **export table** of the DLL with their function name as defined in the API function prototype declaration (undecorated function names), e.g.

INIPC_initialize_board

CANPC_reset_board

CANPC_initialize_chip

For backward compatibility and compatibility to the CAN L2 APIs V4.xx of the other CAN interfaces the DLL exports the C API functions also with their 'decorated stdcall' names as:

'_FunctionName@ParameterBytes'

Example (32bit): *_INIPC_initialize_board@4*

_CANPC_reset_board@0

_CANPC_initialize_chip@20

The supplied definition file CAN*usb.def* contains both the decorated and undecorated API function names.

An application can be linked to the CANusb Layer 2 API DLL in two ways:

- **Implicit Linking**

The DLL is loaded and mapped into the address space of the application process at startup. Implicit Linking requires a .lib Library File (*CANusb.lib*) to be provided when building the application executable.

- **Explicit Linking**

The DLL and API function entry points are loaded at run-time within the application.

5.1.2 MS Visual C/C++

The API DLL is supplied together with a Visual C/C++ import library (*CANusb.lib*) which may be linked to the application. Otherwise explicit linking via the Win32 functions *LoadLibrary* and *GetProcAddress* may be used.

5.1.3 Borland C/C++/Builder

Borland C/C++ is supported from version 4.5 upwards.

The supplied import library of the API DLL cannot be linked in Borland projects since they are of Microsoft standard. You can try to produce a Borland compatible import library of the DLL using Borland tools *IMPDEF* and *IMPLIB* in the *BIN* directory of the Borland installation:

1. IMPDEF tempimp.def canusb.dll
2. IMPLIB -f canusb.lib tempimp.def

As an alternative, the supplied *canusb.def* may be linked instead of the import library to the Borland project.

Otherwise explicit linking via the Win32 functions *LoadLibrary* and *GetProcAddress* may be used.

5.1.4 MS Visual Basic

In Visual Basic the API functions are linked by declaring them in a basic module.

Example:

```
Declare Function INIPC_initialize_board Lib "canusb.dll"
(resources As CANPC_RESSOURCES) As Long

Declare Function CANPC_reset_board Lib "canusb.dll" ( ) As Long
```

For backward compatibility and compatibility to the CAN L2 APIs V4.xx of the other CAN interfaces the API functions may be declared by their decorated names (see Chapter 5.1.1).

Example:

```
Declare Function INIPC_initialize_board Lib "canusb.dll" Alias
"_INIPC_initialize_board@4"
(resources As CANPC_RESSOURCES) As Long

Declare Function CANPC_reset_board Lib "canusb.dll" Alias
"_CANPC_reset_board@0" () As Long
```

The parameters can be defined as:

```
Type data_struct
Data(7) As Byte
End Type

Type CANPC_RESSOURCES
   SocketNumber        As Integer
   InterruptLine       As Integer
   DpramBase           As Long
   DpramSize           As Long
   ChipType            As Integer
   IOAddress           As Integer
   RegisterBase        As Integer
End Type
```

```
Type param_struct
   Ident                As Long
   DataLength           As Long
   RecOverrun_flag      As Long
   RCV_fifo_lost_msg    As Long
   Data                 As data_struct
   AckOverrun_flag      As Long
   XMT_ack_fifo_lost_acks  As Long
   XMT_rmt_fifo_lost_remotes As Long
   Bus_state            As Long
   Error_state          As Long
   Can                  As Long
   Time                 As Long
End Type
```

The declarations are supplied with the L2 API DLL as *canusb.bas*. A complete application example for Visual Basic 6.0 is also supplied. For compatibility reasons the declarations with the decorated names of the API functions are used in *canusb.bas* (see Chapter 5.1.1).

5.1.5 Delphi

In Delphi the API functions are linked by declaring them in a Delphi Pascal-Unit.

Example:

```
function INIPC_initialize_board(ressources:
pCANPC_RESSOURCES):Integer;...
                                    ... stdcall;external'canusb.dll' name
'_INIPC_initialize_board@4';

function CANPC_reset_board:Integer; stdcall;external'canusb.dll' name
'_CANPC_reset_board@0';

function CANPC_reset_chip:Integer; stdcall;external'canusb.dll' name
'_CANPC_reset_chip@0';

function CANPC_initialize_chip(presc:Integer;
sjw:Integer;tseg1:Integer;tseg2:Integer;sam:Integer):Integer;...
                                    ...stdcall;external'canusb.dll' name
'_CANPC_initialize_chip@20';
```

The naming convention of the functions in the DLL's export table must be observed (see section 5.1.1). Furthermore, the parameter structures in INIPC_initialize_board and CANPC_reset_board can be defined as:

```
DataType = array [0..7] of Byte;

CANPC_RESSOURCES = record       //Parameter structure for
                                  INIPC_initialize_interface
  uSocket: SmallInt;
  uInterrupt: SmallInt;
  ulDPRAMBaseAdr: LongInt;
  ulDPRAMSize: LongInt;
  uChip: SmallInt;
  IOAddress: SmallInt;
  uRegisterBase: SmallInt;
 end;
```

```
PARAM_STRUCT = record        //Parameter structure for CANPC_read_ac
  Ident:                                                    LongInt;
  DataLength:                                               Integer;
  RecOverrun:                                               Integer;
  LostMessage:                                              Integer;
  RcvData:                                                  DataType;
  AckOverrun:                                               Integer;
  LostAck:                                                  Integer;
  LostRemotes:                                              Integer;
  BusState:                                                 Integer;
  Error_state:                                              Integer;
  Can:                                                      Integer;
  TimeStamp:                                                LongInt;
end;
```

5.1.6 LABView

The API functions can be accessed via the *Call Library Function* node. Configuring the node requires the following definitions:

- DLL name and path as *canusb.dll*
- Function name according to *stdcall*
- Calling convention as *stdcall*.
- Parameters and return value as defined in software description in the user manual (Chap.4).

The parameter structures of INIPC_initialize_board and CANPC_read_ac can be defined as arrays and need to be evaluated byte wise. The array's size must fit or exceed the parameter structure's size. Otherwise, access violations during operation may occur.

An application example for LABView 5.0 can be downloaded from www.softing.com.

5.1.7 Others

Other application and compiler environments such as LABWindowsCVI, HPVEE, WATCOM, Testpoint a.o. are supported as well. Application hints and examples may be requested from Softing's technical CAN support.

5.2 Interrupt Processing

5.2.1 WIN32 Events

For many applications it is useful to be informed by a WIN32 event about occurrence of CAN events. Otherwise, the CANusb must be polled for new events which requires more PC processor time.

The driver triggers a WIN32 event to the application on the following CAN events:

- Reception of data, remote and error frames
- Acknowledge on successful transmissions if enabled
- Change of bus state

NOTE:
In object buffer mode the event service routine should not interrupt any API function since this may cause false function return codes .

5.2.2 WIN32 Event Programming

If the CANusb driver detects an new received CAN event 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 new CAN events.

As a prerequisite the WIN32 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 WIN32 event is triggered by the driver. Then, the necessary interrupt activities can be processed and the thread gets back into WAIT status.

NOTE:
In object buffer mode the service routine should not interrupt any API function since this may cause false function return codes . Hence, parallel process control is required which can be realized by critical sections (see C++ manual).

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

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

5.3 Debugging Hint

CANusb uses internal threads for communication. I.e. while debugging, if the application halts at a user defined breakpoint or is operated by single stepping, CAN messages and bus events may not immediately be transfered via the API.

5.4 Cyclic Transmission

Cyclic transmission tasks of data and remote frames can be executed onboard by the firmware in dynamic object buffer mode. The number of cycles can be set as well as the cycle rate.

The cyclic transmission is programmed by following steps:

1. Definition of the transmit object by *CANPC_define_object* during the initialization of the dynamic object buffer
2. Definition of cyclic transmission (number of cycles and cycle time) of the previously defined transmit object by *CANPC_define_cyclic* (after *CANPC_start_chip*).
3. Start of the cyclyc transmission by *CANPC_send_object* or *CANPC_write_object*.
4. Stop unlimited cyclic transmission by *CANPC_define_cyclic* with a cycle time of 0 ms.

The transmitted data contents are defined and modified during cyclic transmission by *CANPC_supply_object* or *CANPC_write_object*. The cyclic transmission is stopped automatically if the defined number of 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.

NOTE:
Cyclic transmission is only provided in dynamic object buffer mode (DOB).

5.5 Compatibility Note

There is full compatibility between all CAN APIs V4.x of CANcard, CAN-AC2, CAN-AC2-PCI and CAN-AC2-104 and V4.00 of CANusb if the following is observed.

- It is possible to operate an application with different hardware platforms just by renaming the API DLL accordingly.
- The application must use initialization parameters in INIPC_initialize_board set to values which are valid for all platforms to be supported. Especially, *cp_ressorces.uSocket* should be set to -1 (AUTO_SOCKET).
- The default physical layer must be used by setting output control to -1 in CANPC_set_output_control.
- If time stamps are to be used with CAN-AC2 (ISA) they must be enabled explicitly by CANPC_enable_timestamps.
- Error frame detection is not possible with CAN-AC2 (ISA).
- Large *Receive_FIFO* support for Fifo Mode as descriped in Chapters 4.3.1 and 4.5.10 is only available with the Layer 2 API DLL for the CANusb interface. Don' t use the function *CANPC_set_rcv_fifo_size* in applications with have to operate with different CAN interfaces.
- The CAN API V4.x is also desigend for CAN interfaces equipped with two CAN channels. For this see the CAN API manuals of CAN-AC2, CAN-AC2-PCI, etc. Therefore calling API functions directed to CAN channel CAN2 (like CANPC_send_data2()) has no effects but causes no harm also on CANusb. All functions to CAN2 return 0. CANPC_get_bus_state() will return 2 (=Bus off) if called with parameter Can = 2 (CAN2).

6 Test program Can_test.exe

6.1 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 (WIN32) it realizes basically the flow charts of the operational modes described in Chapter 4.

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. Its functionality is not documented except in the source code (see Chapter 6.3).

NOTE:
The example program Can_test is dedicated to start up and test the proper operation of the CAN interface. 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.

6.2 Testing Installation and Communication

It is recommended to use the program also as an installation and operation test running the interface with another CAN node. It should be assured that a valid bus termination is applied.

First the program displays some useful information about hardware, software and firmware version as well as about serial number and license of the CAN interface. Otherwise, the error number and cause are 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.

The **baudrate** can be set to certain values. Further, the **operational mode** has to be chosen:

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

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

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

If the selection is omitted (just pressing ENTER) the default settings are used (1 Mbit/s, FIFO mode, interrupt usage).

NOTE:
The display of 'Chip is running' states that any initialization routines have been executed successfully. The installation works fine.

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.

6.3 Sample Code

The C source code of the program is sampled in the '\Sample\C' directory of the installed software. It shows exemplary the programming of the operational modes as well as it provides basics of the WIN32 interrupt programming. The code is conform for all Softing CAN interfaces. Making an executable only requires compiling and linking of the source files with the related library. As the WIN32 interrupt programming in the sample involves a separate thread a multithreaded run-time library of the development environment has to be used.

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 used for 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 an additional CAN node. 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 56-326.

Engineering notes:

7 Error Return Codes

This chapter defines the detailed error return codes of INIPC_initialize_board (Table 7-1) and CANPC_reset_board (Table 7-2) due to the variety of possible error causes while initializing the CANusb 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 CANusb at all.

7.1 INIPC_initialize_board

Table 7-1: Error codes of INIPC_initialize_board

Function return code	Error cause
FE00	No CAN device or driver found
FE08	Wrong driver DLL version
FE09	Wrong driver version
FE0A	Driver not found
FE0B	Not enough memory
FE0C	Too many devices
FE0E	Device already exists
FE0F	Device already open
FE11	Resource conflict
FE12	Resource access error

7.2 CANPC_reset_board

Table 7-2: Error codes of CANPC_reset_board

Error return code	Error cause
-4	Communication error between host and interface.
-6	Firmware data format error. Reinstall API software.
-7	Firmware checksum error. Reinstall API software.
-20	Bad response from card. Re-plug CANusb
-21	SRAM seems to be damaged. Try to reinstall the CANusb. If error is remaining contact Softing.
-22	Invalid firmware start address. Reinstall hardware.
-23	Invalid record type. Reinstall hardware and API software.
-24	No response after firmware start. Reinstall hardware and API software.
-25	Bad response after firmware start. Reinstall hardware and API software.
-99	Board not initialized.
-603	Communication via USB pipe broken.
-614	Wrong version of boot firmware. Upgrade the CANusb with CANusbUpg102.exe supplied on the installation disk.

Glossary

AC

Application **C**ontroller

API

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

CAN

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

CAN-AC

CAN Application **C**ontroller

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

OS

Operating System

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

USB

Universal **S**erial **B**us

Index

D

E

F

H