C Interface Library for DOS and Win 3.11

Software Manual

Manual Rev. 1.2 - esd gmbh

Manual File:	I:\texte\Doku\MANUALS\PROGRAM\CAN\Schicht2\ENGLISCH\dos\UNIDOS12.en9
Date of Print:	08.03.99

Described Software:	C interface library for DOS and Windows 3.11
Revision/Date:	V1.2

Implementation at the Following Boards	Order no.
CAN-ISA/200	C.2011.xx
CAN-ISA/331	C.2010.xx
CAN-PC104/200	C.2013.xx
CAN-PC104/331	C.2012.xx
DN-PC104/331	C.2014.xx
CAN-PCI/331	C.2020.xx
CAN-PCI/360	C.2022.xx
PMC-CAN/331	C.2025.xx
CAN-CPCI/331	C.2027.xx
-	-

Changes in the chapters

The changes in the user's manual listed below affect changes in the software, as well as changes in the description of the facts only.

Alternations in the appendix versus previous revisions	Alternations in software	Alternations in documentation
Correction of canRead description.	-	x

Technical details are subject to change without notice.

N O T E

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. **esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of **esd** gmbh.

esd does not convey to the purchaser of the product described herein any license under the patent rights of **esd** gmbh nor the rights of others.

esd electronic system design gmbh
Vahrenwalder Str. 207
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd-electronics.com
Internet: www.esd-electronics.com

USA / Canada:
esd electronics Inc.
12 Elm Street
Hatfield, MA 01038-0048
USA

Phone: +1-800-732-8006
Fax: +1-800-732-8093
E-mail: us-sales@esd-electronics.com
Internet: www.esd-electronics.us

Content

Page

1. Introduction 3

2. Functions of the Programming Interface 4
2.1 Initialization 4
canHwInit() 4
canEnableIrq() 4
canDisableIrq() 5
canInit() 6
canInitEx() 7
2.2 Reading and Writing Data 8
canEnableId() 8
canDisableId() 8
canSetAcceptanceEx() 9
canRead() 10
canReadEx() 11
canWrite() 12
canWriteEx() 13

3. AC2-Compatible Function Calls of the Programming Interface 15
3.1 Initialization 15
CAN_start_chip() 15
CAN_reset_board() 15
3.2 Reading and Writing Data 16
CAN_set_acceptance(), CAN_set_acceptance2() 16
CAN_read() 17
CAN_send_data(), CAN_send_data2() 20
CAN_send_remote(), CAN_send_remote2() 21

4. Returned Values 22

Appendix - Installation and Implementation A-1

This page is intentionally left blank.

1. Introduction

This manual describes the C-interface to the driver of the CAN controller.

The C-interface is implemented at several esd CAN boards. If there are any restrictions depending on the used board type, this is marked in the text.

The appendix describes the installation of the driver as well as properties which are specific for implementation.

The archetypes of all functions as well as the *Communication Handle* are combined in the header file *can.h*.

2. Functions of the Programming Interface

The following chapter describes the C-programming interface to the CAN controller. The meaning of the returned values in case of an error will be described in chapter 4.

2.1 Initialization

---

canHwInit()

Name: **canHwInit()** - Hardware initialization of the CAN interface

Synopsis:

```
int    canHwInit
       (
       unsigned int  base_addr        /* module address */
       )
```

Description: This function has to be called at the beginning as the first library function.

Return: 0 or an error code as described at page 22.

---

canEnableIrq()

Name: **canEnableIrq()** - Setting the interrupt level

Synopsis:

```
int    canEnableIrq
       (
       int    level     /* IRQ level */
       )
```

Description: By means of this function the interrupt level of the CAN module is set.

Return: 0 or an error code as described at page 22.

canDisableIrq()

Name: **canDisableIrq()** - Disable interrupt level

Synospis:

```
int    canDisableIrq
       (
       )
```

Description: By means of this function the previously programmed interrupt level is disabled again. This function should only be called, if *canEnableIrq* had successfully been called before.

Return: 0 or an error code as described at page 22.

canInit()

Name: **canInit()** - Initialization of the CAN-interface baudrate

Synopsis:

```
int    canInit
       (
       int    net,          /* number of CAN interface */
       int    baudrate      /* baudrate index */
       )
```

Description: This function initializes the CAN interface and sets the bitrate for the net *net* to the bitrate *baudrate* according to the following table. The performance during transmission of other values as well as the number and assignment of the supported nets depends on the implementation.

baud [HEX]	bitrate [kbit/s]
0	1000
1	666.6
2	500
3	333.3
4	250
5	166
6	125
7	100
8	66.6
9	50
A	33.3
B	20
C	12.5
D	10

Return: 0 or an error code as described at page 22.

canInitEx()

Name: **canInitEx()** - Initialization of the baudrate via BT0/BT1 register

Synopsis:

```
int    canInitEx
       (
       int    net,          /* number of CAN interface */
       int    bt0_bt1       /* baudrate via controller register */
       )
```

Description: By means of this function the CAN interface is initialized and the bitrate for net *net* is set to the value that is defined via *bt0_bt1*.
In *bt0_bt1* the content of the registers of BTR0 and BTR1 of the CAN controller are coded:

bt0_bt1 = (BTR0 x 256) + BTR1

The description of the register values of BTR0 and BTR1 can be taken from the manual of the CAN controller.

Return: 0 or an error code as described at page 22.

2.2 Reading and Writing Data

The calls described below are used to read and write data on the CAN. By transmitting an RTR frame on an RX identifier data can be requested and it is possible to wait for the reception of an RTR frame by means of a Tx identifier.

---

canEnableId()

Name: **canEnableId()** - Enabling an identifier for Rx and RTR transfers

Synopsis:

```
int   canEnableId
      (
      int   net,     /* number of CAN interface */
      int   id       /* Rx identifier */
      )
```

Description: By means of this function identifiers for Rx and RTR transfers are enabled.

Via *net* the net number is transmitted if more than one net is available.
id is the desired CAN identifier in the range of 0-2047.

Return: 0 or an error code as described at page 22.

---

canDisableId()

Name: **canDisableId()** - Disabling an identifier for Rx or RTR transfers

Synopsis:

```
int   canDisableId
      (
      int   net,     /* number of CAN interface */
      int   id       /* Rx identifier */
      )
```

Description: This function disables identifiers for Rx and RTR transfers.

Via *net* the netnumber is transmitted if more than one net is available.
id is the desired CAN identifier in the range of 0-2047.

Return: 0 or an error code as described at page 22.

canSetAcceptanceEx()

Name: **canSetAcceptanceEx()** - Setting the acceptance filter for 29-bits identifiers

Synopsis:

```
int    canSetAcceptanceEx
       (
       int     net,          /* number of CAN interface */
       int     handle,       /* must always be set to '0' */
       long    mask,         /* selection of the Id bits to be compared */
       long    id            /* values which are to be used for comparison */
       )
```

Description: By this function the acceptance filter for 29-bits identifiers is set.
This function has not been designed for 11-bits identifiers.

Via *net* the netnumber is transmitted if more than one net is available.

handle is not yet supported and must always be set to '0'.

In *mask* the identifier bits which are to be compared with a default value are set to '1'. Identifier bits whose value is not to be set, have to be specified by '0'.

In *id* a desired value can be determined for the bits in *mask* marked by a '1'. For these bits the value '0' or '1' can be specified in *id*. For all other bits the value '0' has to be specified in *id*.
Only the identifiers which have the required values are taken into the receive buffer.

Return: 0 or an error code as described at page 22.

canRead()

Name: **canRead()** - Reading the current Rx frame

Synopsis:

```
int     canRead
        (
        int             net,        /* number of CAN interface */
        int             *id,        /* Rx identifier */
        int             *length,    /* number of data bytes (1…8) */
        unsigned char   *data       /* received data */
        )
```

Description: The function reads the last Rx data of the Rx identifier. The function does not wait.

**net* is a parameter that returnes the number of the CAN net ('0' or '1') on which messages have been received.

**id* is a parameter that returnes the Rx identifier of the received message in the range of 0-2047.

The number of copied bytes is stored in **length*. Values for **length* which are larger than $0x10 show RTR frames.
E.g. $0x12 = > RTR frame with the data length '2'

The received data is stored in **data*.

Status =	-1	no data
	0	Rx data has been read
	5	acknowledge for last Tx telegram on this identifier

Return: 0 or an error code as described at page 22.

canReadEx()

Name: **canReadEx()** - Reading the current Rx frames with 29-bit identifiers

Synopsis:

```
int   canReadEx
      (
      int            *net,      /* number of CAN interface */
      long           *id,       /* 29-bit Rx identifiers */
      int            *length,   /* number of data bytes (1…8) */
      unsigned char  *data      /* received data */
      )
```

Description: This function reads the last Rx data of the 29-bit Rx identifiers. This function does not wait.

**net* is a parameter that returnes the number of the CAN net ('0' or '1') on which messages have been received.

**id* is a parameter that returnes the Rx identifier of the received message in the range of 0…(2^{29}-1).

The number of copied bytes is stored in **length*. Values for **length* which are larger than $0x10 show RTR frames.
E.g. $0x12 = > RTR frame with the data length '2'

The received data are stored in **data*.

Status =	-1	no data
	0	Rx data has been read
	5	acknowledge for last Tx telegram on this identifier
	15	extended identifier: data has been received
	16	extended identifier: Tx frame has been transmitted

Return: 0 or an error code as described at page 22.

canWrite()

Name: **canWrite()** - Transmitting Tx frames

Synopsis:

```
int   canWrite
      (
      int                net,        /* number of CAN interface */
      int                id,         /* Tx identifiers */
      int                length,     /* number of data bytes (1…8) */
      unsigned const char *data      /* data to be transmitted */
      )
```

Description: By means of this function Tx frames are transmitted.

net contains the number of the CAN net ('0' or '1'). If only one net is available, '0' has to be entered always.

In *id* the desired Tx identifier is specified in the range of 0-2047.

In *length* the desired number of data bytes to be transmitted is specified. Values for *length* which are larger than $0x10show RTR frames.
E.g. $0x12 = > transmit RTR frame with the data length '2'.

The data is stored in **data*.

Return: 0 or an error code as described at page 22.

canWriteEx()

Name: **canWriteEx()** - Transmitting Tx frames on 29-bit Tx identifiers

Synopsis:

```
int   canWriteEx
      (
      int                net,        /* number of CAN interface */
      long               id,         /* 20-bit Tx identifiers */
      int                length,     /* number of data bytes (1…8) */
      unsigned const char *data      /* data to be transmitted */
      )
```

Description: By means of this function Tx frames are transmitted.

net contains the number of the CAN net ('0' or '1'). If only one net is available, '0' has to be entered always.

In *id* the desired Tx identifier is specified in the range of 0…(2^{29}-1).

In *length* the desired number of data bytes to be transmitted is specified. Values for *length* which are larger than $0x10 show RTR frames.
E.g. $0x12 = > transmit RTR frame with the data length '2'.

The data is stored in **data*.

Return: 0 or an error code as described at page 22.

This page is intentionally left blank.

3. AC2-Compatible Function Calls of the Programming Interface

The software is offering the function calls described below to users who have been working with the AC2 software previously. The functionality of these routines is compatible to AC1/2 routines of similar names (e.g. previously *AC2_send_data*, now *CAN_send_data*).

You should use these routines only in order to put up *existing* I/O functions and *not for new developments*!

If these routines are used, the routines *canEnableId*, *canDisableId*, *canRead* and *canWrite* must not be used!

3.1 Initialization

CAN_start_chip()

Name: **CAN_start_chip()** - Starting CAN operations

Synopsis:
```
int   CAN_start_chip
      (     void
      )
```

Description: By means of this function CAN actions are activated. The CAN controller leaves the RESET status. From now on transmissions can be started and received messages are monitored.

Return:
0: start successful
-3: timeout error during access to the PCB
-4: timeout error during access to the CAN controller

CAN_reset_board()

Name: **CAN_reset_board()** - Resetting the boad into the initialization status

Synopsis:
```
int   CAN_reset_board
      (     void
      )
```

Description: By means of this function the board is reset into initialization status. The firmware is loaded into the board, therefore this function might last a little longer.

Return:
0: initialization successful
-1: error during loading the firmware (PCB or firmware have not been found)

3.2 Reading and Writing Data

The calls described below are used in order to read and write data onto the CAN. Data can be requested by transmitting an RTR frame on an Rx identifier and it is possible to wait for the reception of an RTR frame by means of a Tx identifier.

CAN_set_acceptance(), CAN_set_acceptance2()

Name: **CAN_set_acceptance()** - Setting the acceptance filter for net 0
CAN_set_acceptance2() - Setting the acceptance filter for net 1

Synopsis:

```
int   CAN_set_acceptance
      (
      int   AccCode,      /* acceptance-code register */
      int   AccMask       /* acceptance-mask register  */
      )

int   CAN_set_acceptance2
      (
      int   AccCode,      /* acceptance-code register */
      int   AccMask       /* acceptance-mask register  */
      )
```

Description: By means of these functions the acceptance registers of the CAN controller are set.
CAN_set_acceptance sets the first CAN channel and *CAN_set_acceptance2* sets the second CAN channel.
By means of these registers a receive filter for data and remote frames can be generated. Only data of identifiers whose 8 MSB are corresponding to the filter are accepted.
All bits which are marked by a '0' in the acceptance-mask register have to correspond in value to the according bits in the acceptance-code register in order to be able to pass through the filter. A '1' in the acceptance-mask register means that the according bit will not be checked.

The exact register description can be taken from the data sheets of the CAN controller (82C200, 8xC592 or SJA100 by Philips).

AccCode: acceptance-code register [0 to FF_{Hex}]
AccMask: acceptance-mask register [0 to FF_{Hex}]

Return:
0: successful call
-3: timeout error during access to the PCB
-4: timeout error during access to the CAN controller

CAN_read()

Name: **CAN_read()** - Receive messages and evaluate transfers

Synopsis:

```
int    CAN_read
       (
       param_struct        *write_ac_param        /* transfer structure */
       )
```

Description: By means of this function the application is informed about the transmission and reception of messages and various error conditions.

Optionally this function can be linked into an interrupt-service routine on the PC and the the CAN module can be informed about the events by means of an interrupt. Alternatively the function can be requested via polling.

The return codes 1…12 (RC1 to RC12) define the relevant elements of the returned structure.

Elements of structure *param_struct*:

- int *Ident*: Identifier of a received or transmitted frame (RC1, RC2, RC3, RC8).

- int *DataLength*: Number of data bytes received (RC1) or requested (RC2).

- int *RecOverrun_flag*: The data received last has not been read by the PC and has been overwritten by new data (RC1, RC2).

- int *RCV_fifo_lost_msg*: Number of lost messages of the Rx FIFO (RC1, RC2, RC3, RC8).

- byte *RCV_data[8]*: Received data bytes (RC1).

- int *AckOverrunFlag*: The acknowledgement of the frame transmitted last has not yet been read by the application (RC3).

- int *XMT_ack_fifo_lost_acks*: Number of lost messages in the transmit-acknowledge FIFO (RC3).

- int *XMT_rmt_fifo_lost_remotes*: Number of lost transmission orders in the remote-transmit FIFO.

- int *Bus_state*: CAN status (RC5):
  0: error active
  1: error passive
  2: bus off

- int *Error_state*: Further error causes (RC7):
  0: no error
  -3: overrun of BASIC CAN 82C200

- int *Can*: CAN channel 1 or 2 (RC1, RC2, RC3, RC4, RC5, RC7, RC8). If *CAN_read* is called, the status described below are handled in the described order.

- *Changing the bus status (RC5):*
  If the firmware discovers a change of the bus status, it passes the information to the structure variable *Bus_state*.

- *Reception of a data or error frame (RC1 or RC2):*
  If a message has been received in the receive FIFO, or if the reception of a frame has been discoverd via polling, the following parameters are taken over into the structure: identifier, overrun flag, the number of lost messages of the receive FIFO and the received data.

- *Acknowledging the transmission of a Tx data frame* (RC3):
  If a transmission acknowledgement of a frame has been discovered in the transmit-acknowledge FIFO or via polling all objects, the identifier and the number of lost messages, if available, is written into the transmit-acknowledge FIFO or into *XMT_ack_fifo_lost_acks*.

- *Overrun of the remote-transmit-FIFO (RC4):*
  (Transmission order on the basis of a received RTR frame)
  If the remote-transmit FIFO overruns, the number of lost messages is stored into *XMT_rmt_fifo_ lost_remotes*.

- *Further error causes (RC7):*
  The error causes stated above are recognized via their according codes in the structure variable *Error_state*.

- *Messages on CAN net 1 and CAN net 2, if the second operates in FIFO mode (RC1, RC2, RC3, RC5, RC7, RC8).*
  If a message has been recognized, the following messages are only evaluated after *CAN_read* had been called again.

Return:
(Return codes - RCs of the command)

0: No new event on the CAN coupler module.
1: Standard-data frame has been received.
2: Standard-remote frame has been received.
3: Transmission of a standard-data frame has been acknowledged.
4: Overrun of transmission FIFO.
5: Change in bus status.
7: Another error status.
8: Transmission of a standard-remote frame has been acknowledged.

CAN_send_data(), CAN_send_data2()

Name: **CAN_send_data()** - Transmitting data on net 0
CAN_send_data2() - Transmitting data on net 1

Synopsis:

```
int   CAN_send_data
      (
      int       Ident,        /* Tx identifier */
      int       DataLength,   /* number of data bytes to be transmitted  */
      byte      *pData        /* pointer to data structure  */
      )

int   CAN_send_data2
      (
      int       Ident,        /* Tx identifier */
      int       DataLength,   /* number of data bytes to be transmitted  */
      byte      *pData        /* pointer to data strucure  */
      )
```

Description: By means of this command a data frame transmits on the CAN with the specified parameters.

Ident transmits the Tx identifier.
In *DataLength* the number of bytes to be transmitted is specified.
pData is the pointer to the address field of the data.

Return: 0: function successful
-1: transmit FIFO is full

CAN_send_remote(), CAN_send_remote2()

Name: **CAN_send_remote()** - Transmission of RTR on net 0
CAN_send_remote2() - Transmission of RTR on net 1

Synopsis:

```
int   CAN_send_remote
      (
      int         Ident,          /* Tx identifier */
      int         DataLength      /* number of data bytes to be transmitted  */
      )

int   CAN_send_remote2
      (
      int         Ident,          /* Tx identifier */
      int         DataLength      /* number of data bytes to be transmitted  */
      )
```

Description: By means of this command a remote frame is transmitted to the CAN with the specified parameters. The transmitted remote frame has always the length 0. Nevertheless the number of transmitted data bytes (here 0) is also transmitted in parameter *DataLength*.

Ident transmits the Tx Identifier.
In *DataLength* the number of bytes to be transmitted is specified (here always '0').

Return: 0: function successful
-1: transmit FIFO is full

4. Returned Values

A function with a return value of type *int* will return an error code according to the following table. Not every code may be suggestive for every function.

Error	Description
CAN_OK	No error
CAN_NO_DEVICE	CAN controller is not available or not initialized
CAN_PARA_ERROR	Invalid parameter during function call
CAN_SYS_ERROR	Internal error

Appendix - Installation and Implementation

Content Page

A 1 Installation A-2
A 1.1 Installation of the Hardware A-2
A 1.2 Installation of the Software A-2

A 2 Implementation A-2
A 2.1 Number and Assignments of Nets A-2

A 1 Installation

A 1.1 Installation of the Hardware

The hardware installation is described in the documentation 'Hardware Installation and Technical Data'.

A 1.2 Installation of the Software

At the moment the software has two files which only have to be copied into the same directory whose name can be freely chosen.

A 2 Implementation

A 2.1 Number and Assignments of Nets

The driver supports up to two nets. The first net has the logical net number '0' and the second net has the logical number '1'.