CAN API for CAN2 / CAN4

VxWorks

Software Manual

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. 205
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
7667 W. Sample Road
Suite 127
Coral Springs, FL 33065
USA

Phone: +1-800-504-9856
Fax: +1-800-288-8235
E-mail: sales@esd-electronics.com

Manual file:	I:\texte\Doku\MANUALS\PROGRAM\CAN\Schicht2\ENGLISCH\VxWorks\CAN2VX11.en6
Date of print:	26.10.2000

Described software:	VME-CAN2/VME-CAN4 driver for VxWorks
Revision/date:	CAN-Interface V1.2x canTools (CAN2: V03x, CAN4: V01x) cmsTools (CAN2: V03x, CAN4: V01x)
Order number:	CAN2: P.1405.56, CAN4: P.1408.56

Changes in the software and/or documentation

Changes in this manual from previous version	Changes in the software	Changes in the documentation
First English version.	-	-

Contents

Page

1. Reference

/1/: electronic system design gmbh, CAL-Slave Manual, July 1998

/2/: electronic system design gmbh, CANopen-Slave Manual, March 1998

/3/: electronic system design gmbh, CANopen-Master Manual, August 2000

/4/: CiA DS-102, CAN Physical Layer for Industrial Applications, April 1994

/5/: CiA DS-201, CAN Reference Model, February 1996

/6/: CiA DS-202/1, CMS Service Specification, February 1996

/7/: CiA DS-202/2, CMS Protocol Specification, February 1996

/8/: CiA DS-203/3, CMS Data Types and Encoding Rules, February 1996

2. Introduction

In this document the C application programmers interface (API) for the driver of the CAN controller will be explained. The CAL-/CANopen-Masters /3/ respectively the CAL-/CANopen-Slaves /1/, /2/ from *esd gmbh* are based on this interface.

Concerning the OSI-layer model the driver contains functions from layer 2 which serve the blocking and non-blocking transmission and reception of data, and also functions from the layer-7-CMS-domain-transfer protocol of the *CAN-application layer* (CAL) /6/-/7/.

> The C-interface is implemented for several operating systems at several esd CAN boards. If there are limitations for the functions due to the operating system or the board type, this is marked in the text.

The prototypes of all functions and the *communication handles* are combined in the header file *can.h*.

3. Program Interface

This chapter describes the C-API for access to the CAN controller hardware. The meaning of returned values in case of error is described in chapter 4.

3.1 Initialization

The service described below serves as initialization of the CAN-controller hardware and has to be executed before every further function call.

---

canCalSetBaudrate()

Name: **canCalSetBaudrate()** - initializing CAN interface

Call:

```
int    canCalSetBaudrate
       (
       int          net,        /* number of CAN interface */
       int          baud        /* bit rate */
       )
```

Description: This function initializes the CAN interface and sets the bit rate for net *net* to the bit rate *baud* corresponding to following table. The number of the supported nets depends on the implementation and is described in the appendix.

baud [HEX]	Bit rate [kbit/s]
0	1000
1	6666
2	500
3	3333
4	250
5	166
6	125
7	100
8	666
9	50
A	333
B	20
C	125
D	10

Return: 0 or an error code described at page 15.

3.2 Reading and Writing of Data

The API calls described below serve the reading and writing of data on the CAN. Each function exists in a non-blocking and a blocking variant with timeout default. Data can be requested by giving out 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.

---

canCalOpen()

Name: **canCalOpen()** - generating a handle for read/write operations

Call:

```
HCAN    canCalOpen
        (
        int             net,    /* number of CAN interface */
        int             txid,   /* Tx identifier of handle */
        int             rxid,   /* Rx identifier of handle */
        unsigned short  typ,    /* properties */
        int             txtout, /* timeout of Tx identifier */
        int             rxtout  /* timeout of Rx identifier */
        )
```

Description: This function returns a handle for following I/O requests.

Net transmits the net number if several nets are available.

txid and *rxid* are the CAN identifiers in the range of 0-2047 assigned to this handle. For writing API calls the Tx identifier is used and for reading API calls the RX identifier of the handle is used. If only one Tx identifier or one Rx identifier is to be assigned to the handle, the unused identifier has to be set to a negative value.

The table below shows additional Tx- and/or Rx-identifier properties which are determined by means of parameter *typ*. Moreover, some drivers support additional options which are described in the appendix.

Parameter	Description
TXRX_DOMAIN	This identifier couple is to be used for the CMS-domain transfer described in section 3.4.
TX_RECV_RTR	An RTR frame can be received on the Tx identifier.
TX_AUTOANSWER	After receiving an RTR frame on this Tx identifier, the actual data is automatically transmitted.
RX_SEND_RTR	An RTR frame can be transmitted on the Rx identifier.

txtout and *rxtout* determine the timeout interval in ms in blocking read/write operations.

Return: In case of error NULLHANDLE, otherwise a valid handle.

canCalClose()

Name: **canCalClose()** - closing a handle for read/write operations

call:

```
int     canCalClose
        (
        HCAN            hcan        /* CAN handle */
        )
```

Description: This function frees a CAN handle and all assigned resources.

Return: 0 or an error code described at page 15.

canCalUpdate()

Name: **canCalUpdate()** - updating the Tx data

call:

```
int     canCalUpdate
        (
        HCAN            hcan,       /* CAN handle */
        const void *    buf,        /* pointer to current data */
        int             len         /* number of bytes (1..8) */
        )
```

Description: This function updates the internal Tx-data structure of the driver of the Tx identifier designated by *hcan*. For this matter *len* bytes are copied from memory address *buf*.

Return: 0 or an error code described at page 15.

canCalSend()

Name: **canCalSend()** - non-blocking transmission of a message

Call:

```
int     canCalSend
        (
        HCAN            hcan,       /* CAN handle */
        const void *    buf,        /* pointer to current data */
        int             len         /* number of bytes (1..8) */
        )
```

Description: This function transmits *len* bytes from memory address *buf* on the Tx identifier of *hcan*. At the same time the internal Tx-data structure of the driver is updated. The successful termination of the transmission command is not being waited for.

Return: 0 or an error code described at page 15.

canCalWrite()

Name: **canCalWrite()** - blocking transmission of a message

call:

```
int     canCalWrite
        (
        HCAN            hcan,       /* CAN handle */
        const void *    buf,        /* pointer to transmission data */
        int             len         /* number of bytes (1..8) */
        )
```

Description: This function transmits *len* bytes from the memory address to the *buf* on the Tx identifier of *hcan*. At the same time the internal Tx data structure of the driver is updated. The function only returns after the transmission request has been successfully terminated or the Tx-timeout interval which had been specified when opening the handle has been exceeded.

Return: 0 or an error code described at page 15.

canCalWaitRTR()

Name: **canCalWaitRTR()** - waiting for the reception of an RTR frame

Call:

```
int     canCalWaitRTR
        (
        HCAN        hcan            /* CAN handle */
        )
```

Description: This function waits for the reception of an RTR frame on the Tx identifier of *hcan*. Flag `TX_RECV_RTR` has to be set when opening the handle. The function only returns after an RTR frame has been received or the Tx-timeout interval specified when opening the handle has been exceeded.

Return: 0 or an error code described at page 15.

---

canCalTake()

Name: **canCalTake()** - non-blocking reading of updated Rx data

Call:

```
int     canCalTake
        (
        HCAN        hcan,       /* CAN handle */
        void *      buf,        /* pointer to free memory area */
        int *       len         /* number of copied bytes (1..8) */
        )
```

Description: This function copies the last-valid Rx data of the Rx identifier from *hcan* to the memory area specified by *buf*. This memory area must have a size of at least 8 bytes. The number of copied bytes is stored in *len*.

Return: 0 or an error code described at page 15.

canCalRead()

Name: **canCalRead()** - blocking reading of updated Rx data

Call:

```
int     canCalRead
        (
        HCAN        hcan,       /* CAN handle */
        void *      buf,        /* pointer to free memory area */
        int *       len         /* number of copied bytes (1..8) */
        )
```

Description: This function copies the updated data of the Rx identifier from *hcan* to the memory area specified by *buf*. This memory area must have a size of at least 8 bytes. The number of copied bytes is stored in *len*. If the data in the Rx buffer of the driver are valid when the function is requested, the function returns immediately. Otherwise the function only returns, if the Rx timeout interval which has been specified when opening the handle has been exceeded or, if new data has been received within this interval.

Return: 0 or an error code described at page 15.

---

canCalSendRTR()

Name: **canCalSendRTR()** - non-blocking request of data

Call:

```
int     canCalSendRTR
        (
        HCAN        hcan        /* CAN handle */
        )
```

Description: This function requests data by transmitting an RTR frame on the Rx identifier of handle *hcan*. This is only possible, if the flag `RX_SEND_RTR` is set when opening the handle.

Return: 0 or an error code described at page 15.

canCalReadRem()

Name: **canCalReadRem()** - blocking request of data

Call:

```
int     canCalRead
        (
        HCAN            hcan,       /* CAN handle */
        void *          buf,        /* pointer to a free memory area */
        int *           len         /* number of copied bytes (1..8) */
        )
```

Description: This function requests data by transmitting an RTR frame on the Rx identifier of handle *hcan*. This is only possible, if flag `RX_SEND_RTR` is set when opening the handle. If valid data are available at the time of the request, these data are marked as invalid first. The function only returns after new data has been received or the timeout interval specified when opening the handle has been exceeded. Valid data are copied to the memory area specified by *buf*. This memory area must have a size of at least 8 bytes. The number of copied bytes is stored in *len*.

Return: 0 or an error code described at page 15.

canCalWriteRead()

Name: **canCalWriteRead()** - succeeding write/read operations

Call:

```
int     canCalWrite
        (
        HCAN            hcan,       /* CAN handle */
        const void *    tbuf,       /* pointer to transmission data */
        int             tlen,       /* number of transmission bytes (1..8) */
        void *          rbuf,       /* pointer to memory for reception data */
        int *           len         /* number of received bytes (1..8) */
        )
```

Description: This function transmits *tlen* bytes of the transmission data to which *tbuf* points at the Tx identifier of handle *hcan*. At the same time the internal Tx-data structure of the driver is updated. Afterwards a reading request on the Rx identifier of *hcan* is executed. If valid Rx data are available at the time of the request, these are aborted first. The function only returns after new data has been received or the Rx-timeout interval specified when opening the handle has been exceeded. Valid data is copied to the memory area specified by means of *rbuf*. This memory area must have a size of at least 8 bytes. The number of copied bytes is stored in *rlen*.

Return: 0 or an error code described at page 15.

3.3 Simultaneous Reading of Several Rx Identifiers (*Multi read)*

By means of the services described below it is possible to wait for the reception of several Rx identifiers by linking all identifiers into a waiting queue. An identifier used this way cannot be used for an individual reading request anymore. The received data of all Rx identifiers to be waited for are stored into a message queue.

---

canCalMultiOpen()

Name: **canCalMultiOpen()** - generating a multi read handle

Call:

```
SCAN     canCalMultiOpen
         (
         int                net,          /* number of CAN interface */
         unsigned short     typ,          /* reserved */
         int                queusize,     /* size of waiting queue */
         int                timeout       /* timeout of reading order */
         )
```

Description: This function returns a handle for following multi read requests for net *net*.

Parameter *typ* is reserved at the moment and has to be set to zero.

queuesize determines the size of the waiting queue in buffered CAN telegrams. The greater this parameter is selected the more improbable is a data loss and the greater is the resource need of the driver.

timeout determines the timeout interval in ms in reading order.

Return: In case of error `NULLHANDLE`, otherwise a valid handle.

---

canCalMultiClose()

Name: **canCalMultiClose()** - closing a multi read handle

Call:

```
int    canCalMultiClose
       (
       SCAN               scan          /* multi read handle */
       )
```

Description: This function frees a multi read handle and all assigned resources. All Rx identifiers have to be removed from the waiting queue before.

Return: 0 or an error code described at page 15.

canCalMultiAdd()

Name: **canCalMultiAdd()** - adding an Rx identifier to a multi read queue

Call:

```
int     canCalMultiAdd
        (
        SCAN            scan,       /* multi read handle */
        int             idf         /* CAN identifier */
        )
```

Description: This function adds the Rx identifier *idf* to the waiting queue of the multi read handle *scan*.

Return: 0 or an error code described at page 15.

canCalMultiDelete()

Name: **canCalMultiDelete()** - deleting an Rx identifier from a multi read queue

Call:

```
int     canCalMultiDelete
        (
        SCAN            scan,       /* multi read handle */
        int             idf         /* CAN identifier */
        )
```

Description: This function deletes the Rx identifier *idf* from the waiting queue of the multi read handle *scan*.

Return: 0 or an error code described at page 15.

canCalMultiRead()

Name: **canCalMultiRead()** - waiting for the reception of Rx data

Call:

```
int    canCalMultiRead
       (
       SCAN        scan,      /* multi read handle */
       int *       idf,       /* received CAN identifier */
       int *       len,       /* number of received bytes */
       char *      data       /* pointer to the receive buffer  */
       )
```

Description: This function copies *len* Bytes received data from the message queue of the multi read handle *scan* to the memory area given by data. The according identifier is returned in *idf*. If no valid data are available, the function only returns after the reception of valid data or after exceeding the timeout interval specified when opening the multi read handle.

Return: 0 or an error code described at page 15.

3.4 Auxiliary Functions

The API requests described below are auxiliary functions which cannot be allocated to any of the previous sections.

---

canCalCopy()

Name: **canCalCopy()** - copying with possibly swapping the byte order

Call:

```
int    canCalCopy
       (
       const char *      src,      /* pointer to memory with source data */
       char *            dest,     /* pointer to memory for destination data */
       const char *      datatype  /* data type designator */
       )
```

Description: This function copies data from the memory area pointed to by *src* into the memory area pointed to by *dest*. At the same time the required byte- swapping is initiated in processors which manage data internally different as the byte order on the CAN required in /5/. For this the datatype designator is transmitted to the routine in *datatype*. This datatype designator consists of a zero-terminated string in which every component of the source data is represented by a single `CHAR` which is the length of the component in bytes. The total length of source data accordingly results from *datatype* and the memory area for source and destination data has to be accordingly sized.

Example:

Following structure is to be transmitted on the CAN:

```
struct myStruct
   {
   long      a;
   char      b;
   short     c;
   };
```

A datatype designation for this example would have to be:

```
const char myDatatype = {4, 1, 2, 0};
```

The number of copied data bytes in this example is 7 and the memory area for source and destination data has to be selected accordingly large.

Return: Always 0.

canCalGetVersion()

Name: **canCalGetVersion()** - determining driver version

Call:

```
int     canCalGetVersion
        (
        unsigned short *    version,    /* memory cell for version number */
        char *              name        /* memory cell for driver name */
        )
```

Description: By means of this function it is possible to determine the version number and driver name for the respective net. In *version* the net number has to be given.

The version number is returned in *version* as 16-bit value in which the individual bits have following meaning:

Bit 15...12	Bit 11...8	Bit 7...0
level	revision	modification

A textual description of the driver is copied to *name*. The buffer size made available for this by the application has to have at least 32 bytes. If `NULL` is given as argument, no copy operation is executed.

Return: 0 or an error code described at page 15.

4. Returned Values

All functions which are called for the request of a handle return NULLHANDLE in case of error.

A function with a returned value of type *int* returns an error code corresponding to following table in which not all codes are useful for all API functions.

Error	Description
CAN_OK	no error
CAN_NO_MEMORY	not enough memory for internal resource need
CAN_NO_DEVICE	CAN controller not available or not initialized
CAN_PARA_ERROR	invalid parameter in function request
CAN_SYS_ERROR	internal error
CAN_RX_TIMEOUT	timeout in reception
CAN_TX_TIMEOUT	timeout in transmission
CAN_TX_ERROR	error during transmission
CAN_CONTR_OFF_BUS	error on CAN. Controller not at bus anymore
CAN_CONTR_BUSY	access to CAN controller not possible
CAN_CONTR_WARN	error on CAN
CAN_OLD_DATA	the read data have already been read
CAN_NO_ID_ENABLED	multi read handle without CAN identifier
CAN_INVALID_HANDLE	the transmitted handle is invalid

5. Programming Interface for the CMS Domain Protocol

By means of the following calls the protocol described in /6/ - /8/ is realised. This can also be achieved by means of the I/O-operations described in 3.1, however, task changes will be reduced on driver level and performance will increase. The protocol distinguishes between a client, which initiates the upload and download of data, and a server, which waits for upload and download requests.

canCalClientDownload()

Name: **canCalClientDownload()** - initiating a downloading transfer

Call:

```
int   canCalClientDownload
      (
      HCAN          hcan,     /* CAN-handle */
      long          mux,      /* CMS multiplexor */
      const void *  buf,      /* pointer to transmit data */
      long          len       /* number of bytes to be transmitted */
      )
```

Description: This function initiates the transmission of *len* bytes of the data pointed at by *buf*, according to the CMS domain protocol with the multiplexor *mux* for the handle *hcan*. When opening the handle, `RXTX_DOMAIN` has to be set.

Return: 0 or an error code described in the appendix.

canCalClientUpload()

Name: **canCalClientUpload()** - initiating an uploading transfer

Call:

```
int   canCalClientUpload
      (
      HCAN          hcan,     /* CAN-handle */
      long          mux,      /* CMS-multiplexor */
      void *        buf,      /* pointer to buffer for receive data */
      long *        len       /* length of receive buffer */
      )
```

Description: This function requests the transmission of data downloaded by *mux* from a server in accordance with the domain protocol on handle *hcan*. When opening the handle, `RXTX_DOMAIN` has to be set. The received data is copied into the memory *buf*, whose length is determined in *len* when called. The number of received bytes is stored in *len*.

Return: 0 or an error code described in the appendix.

canCalServerWait()

Name: **canCalServerWait()** - waiting for a transfer operation

Call:

```
int   canCalServerWait
      (
      HCAN          hcan,     /* CAN-handle */
      long *        mux,      /* CMS-multiplexor */
      int *         up        /* transmission direction */
      )
```

Description: This function waits for the handle *hcan* as a server for a transfer operation to be initiated by a client in accordance with the CMS domain protocol. When opening the handle, the flag `RXTX_DOMAIN` has to be set. The function only returns after a client has initiated a domain transfer or when the value for Rx-timeout, as specified when the handle was opened, has been exceeded.
In *mux* the multiplexor of the data is stored which the client wants to upload or download.

The desired direction of transmission is returned in *up* (0 = download, 1 = upload). In order to continue the domain transfer depending on *up* the application has to call the routines **canCalServerUpload()** or **canCalServerDownload()**.

Return: 0 or an error code described in the appendix.

---

canCalServerUpload()

Name: **canCalServerUpload()** - continuing a receive transfer

Call:

```
int   canCalServerUpload
      (
      HCAN          hcan,     /* CAN-handle */
      void *        buf,      /* pointer to memory for receive data */
      long *        len       /* number of transmitted bytes */
      )
```

Description: This function has to be called after **canCalServerWait()** has returned successfully, if the client initiated a download. When opening the handle *hcan*, the flag `RXTX_DOMAIN` has to be set. The received data is copied into the memory determined by *buf*, whose length is specified in *len* when called. After the transmission has been finished, the number of actually received bytes is stored in *len*.

Return: 0 or an error code described in the appendix.

canCalServerDownload()

Name: **canCalServerDownload()** - continuing a transmission transfer

Call:

```
int     canCalServerDownload
        (
        HCAN            hcan,       /* CAN-handle */
        const void *    buf,        /* pointer to transmission data */
        long            len         /* number of bytes to be transmitted */
        )
```

Description: This function has to be called after **canCalServerWait()** has successfully returned, if the client initiated an upload. When opening the handle *hcan* , the flag `RXTX_DOMAIN` has to be set.
The transmission data is downloaded from the buffer *buf*, and the number of the bytes to be transmitted is determined in *len*.

Return: 0 or an error code described in the appendix.

---

canCalServerWaitAndUpload()

Name: **canCalServerWaitAndUpload()** - waiting and executing an upload

Call:

```
int     canCalServerWaitAndUpload
        (
        HCAN            hcan,       /* CAN-handle */
        long *          mux,        /* CMS-multiplexor */
        void *          buf,        /* pointer to buffer for receive data */
        long *          len         /* number of transmitted bytes */
        )
```

Description: This function waits for the handle *hcan* as server for a download to be initiated by a client in accordance with the CMS-domain transfer protocol and copies the received data into the buffer determined in *buf*, whose length is determined in *len* when called. After the transmission has been finished the number of actually received bytes is returned in *len* and the multiplexor is returned in *mux*. When opening the handle, the flag `RXTX_DOMAIN` has to be set. The function only returns, when a client initiated a domain upload or when the value for the Rx-timeout, specified when the handle was opened, has been exceed.

The advantage of this call compared to a combination of **canCalServerWait()** and **canCalServerUpload()** is in avoiding a task change, because the initiation is acknowledged on interrupt level and the calling task only has to be considered again after the complete data transfer has been finished.

Return: 0 or an error code described in the appendix.

Appendix Implementation

Contents

Page

A 1.1 Number and Assignment of Networks

The driver for the VME-CAN2 supports up to sixteen networks. The network number for both channels is set by means of the coding switches in the front panel (see hardware manual). The network number cannot be configured by means of software.

The driver for the VME-CAN4 supports up to 32 networks. The network number for the four channels ca be configured by means of software (see canTools manual).

A 1.2 Interaction and Multi-Processing Ability

Interaction means the ability of a driver to pass on transmission commands on a certain CAN-identifier in a certain network via the CAN-bus as well as internally to other processes which have acknowledged reception for this CAN-identifier. A useful consequence of this property is that a CAL-master and a CAL-slave can be initiated in the same network, for example.

Multi-processing ability means the property of the driver that transmission and receive commands can be given in the same CAN-identifiers. A useful consequence of this property is that two CAL-slaves can be initiated in the same network, for example.

The CAN2/CAN4-driver supports interaction as well as multi-processing ability.

If the driver has not been adapted to using interaction, the functionality to wait with simultaneous upload without additional task change (canCalServerWaitAndUpload) will not be supported.

A 1.3 Configuring the Bit Rate

After a RESET the VME-CAN2 has got the bit rate which has been set by means of coding switches SW2 and SW4 (see hardware manual) and can be overwritten by **canCalSetBaudrate()**. If these coding switches are set to 0xF, the CAN2 performs passively on the bus until the bit rate is set accordingly by means of software.

The VME-CAN4 is always passive on board after a RESET so that the bit rate must always be set by **canCalSetBaudrate()**.

If a value between 0x0000 and 0x000E is specified for **canCalSetBaudrate()** for the parameter *baud*, the bit rate will be set according to the following table. Each value between 0x0011 and 0x7F7F will be directly entered into the *Bit Timing Register* of the CAN-controller.

Index [HEX]	Bit rate [kbit/s]	82C200- or SJA1000-register		typical values of the attainable line length $_{lax}$ [m]	minimum attainable line length $_{Lein}$ [m]
		BTR0 [HEX]	BTR1 [HEX]		
0	1000	00	14	37	20
1	800	00	18	59	42
2	666.6	00	1C	80	65
3	500	01	18	130	110
4	333.3	01	1C	180	160
5	250	02	1C	270	250
6	166	03	1C	420	400
7	125	04	1C	570	550
8	100	45	2F	710	700
9	66.6	09	1C	1000	980
A	50	4B	2F	1400	1400
B	33.3	18	1C	2000	2000
C	20	5F	2F	3600	3600
D	12.5	31	1C	5400	5400
E	10	00	16	7300	7300

The specifications in the table are based on limit values of the bit timing of the CAN-protocol, the delays of the local CAN-interface and the delays of the cable. The delay of the cable has been assumed with about 5.5 ns/m. Further influences, such as the terminal resistor, the resistivity, the cable geometry or external disturbances during transmission have not been considered in these specifications!