STEINHOFF
AUTOMATION- & FIELDBUS-SYSTEMS

Installation / User's Guide

DACHS® CAN

C-API for CAN Layer 2

Version 1.0.1

for **QNX® Neutrino RTOS v.6.4**

for the dual CAN Janus-MM PC/104 Board
from Diamond Systems

Order No.:
CAN–SJA1000-104-API-JAN-NTO-[1.0.2-NTO6.4]

DACHS®
...the better Idea!

QNX® Build a more reliable world™

Author: Dipl.-Inform. Armin Steinhoff

© STEINHOFF AUTOMATION & FIELDBUS-SYSTEMS • P.O.Box 228 • D-65601 Elz/Germany
+49-6431-529366 • Fax +49-6431-57454 • support@steinhoff.de • **www.DACHS.info** • www.steinhoff-automation.com

Disclaimer

STEINHOFF (STEINHOFF Automation & Fieldbus-Systems) makes no representations or warranties with respect to the contents or use of this manual or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result.

This manual contains information which is subject to change without notice. Although every precaution has been taken in the preparation of this manual, STEINHOFF assume no responsibility for any omissions or errors, nor do STEINHOFF, its employees, its contractors or the authors of this documentation assume liability for damages resulting from the use of the information contained herein - special, direct, indirect or consequential damages, losses, cost, charges, claims, demands, or claim for lost profits, fees or expenses of any nature or kind.

Notice of Copyright

Trademarks

QNX and Neutrino are registered trademarks of QNX Software Systems Ltd.
DACHS is a registered trademark of Steinhoff A.
All other trademarks belong to their respected owners.

CONTENTS

1 Preface

The CAN development kit contains the

- CAN call library : CAN.a

and several sources of programming examples and related make files.

It must be installed e.g. with the 'pax' utility after unzipping the archive by gunzip.

The CAN library provides you with serveral calls for

- the handling and configuring of the CAN controllers
- retrieving of status informations from the controllers
- reading and writing of CAN frames at the data link layer and
- defining of CAN code and mask filters

Supported are CAN frames compliant to the CAN specification 2.0 B .

The supported CAN controller board is based on two Philips CAN controller chips SJA1000.

2 Service-Calls

The CAN resource manager provides you with the access to a FIFO for reading and writing of standard CAN frames from and to the CAN bus. Each FIFO can hold up to 128 CAN frames. The capturing of frames can be defined by an acceptance code and mask register.

Attention: after registering a pulse at the driver side, there must be always an active MsgReceivePulse() call. The driver sends a pulse event for every received frame, that means the message queue at client side will grow infinitely if there is no active MsgReceivePulse() call. This can lead to a system lockup. Please deregister the pulse event if there is no active MsgReceivePulse() call.

2.1 ConnectDriver()

Function: ConnectDriver opens the access to the CAN driver specified by 'driver_name' and to a physical channel specified by 'ch' (1 ..4). It must be called at first before submitting other calls. The returned connection handle 'hdl' is specific to the target controller and must be used with all other subsequent calls of the API. The channel number ch must only be set if the CAN board has more than one channel.

Prototype:

```
int ConnectDriver(short ch, char * driver_name, canhdl_t * hdl);
```

Return value:

- ERR_NO_DRV:	CAN driver not started or not reachable
- ERR_CHANNEL	invalid channel number (1.. 4)
- ERR_OK:	driver connected
- if ERR_OK	hdl contains a valid connection handle

2.2 DisConnectDriver()

Function: DisConnectDriver close the access to the CAN driver.

Prototype: short DisConnectDriver(canhdl_t * hdl);

Return value:

- ERR_NOT_CON	driver not opened
- ERR_OK:	driver disconnected

2.3 CanWrite()

Function: write a CAN frame to to the bus.

The CAN frames are supported in the standard and the extended frame format. There is a common structure for the definition of CAN frames:

```
struct can_object {
      struct  frinf  inf;  // frame info: frame format, RTR bit, data length code
      unsigned int  id; // frame id:  integer of max 11 or 29 bits
      BYTE data[24]; // data of the frame,
                     // (8 bytes data + 16 bytes reserved)
};

struct  frinf {
      BYTE  DLC :4;  // data length code
      BYTE  res:2;   // reserved
      BYTE  RTR:1;   // RTR bit;  1 == RTR frame
      BYTE  FF:1;    // frame format: StdFF == 0,  ExtFF == 1
};
```

Prototype: short CanWrite(canhdl_t hdl, struct can_object *frm);

Return value:

- ERR_NOT_CON	driver not opened
- ERR_NOTIFY	pulse registration failed
- ERR_DEVCTL	connection broken
- ERR_FULL	CAN FIFO full, frame not accepted
- ERR_OK	call successful

2.4 CanRead()

Function: read a CAN frame from the receive FIFO.

The frame is read into the frame buffer with the structure can_object. This call is a non-blocking call. With the pulse parameter you can arm once the driver with a pulse for notification. The application must wait for that pulse in a MsgReceivePulse call. Only one user task can do this at the same time and it must be also the only task which submits read calls. The pulse will be triggered if at least one frame is received by the CAN controller.

Frame structure: see CanWrite.

```
Prototype:  short CanRead( canhdl_t hdl,              // connection id
                           struct can_object *frm,    // CAN frame buffer
                           struct sigevent  * pulse); // user pulse for
                                                      // receive events
```

Return value:

- ERR_NOT_CON	driver not opened
- ERR_DEVCTL	connection broken
- ERR_EMPTY	driver FIFO empty, frame not available
- ERR_OK	call successful
- ERR_OVERRUN	overrun of the receive FIFO

2.5 CanGetConfig(), CanSetConfig()

Function: reading and writing of configuration data of the driver.

Prototype: short CanGetConfig(canhdl_t hdl, struct config *conf);
short CanSetConfig(canhdl_t hdl, struct config *conf);

The struct config is defined in the include file canstr.h;
it contains the following components:

controller mode bits

```
  -  BYTE LOM        // listen only mode if bit set to 1
  -  BYTE STM        // self test mode if bit set to 1
  -  BYTE AFM        // acceptance filter mode,
                     // single filter mode if bit set to 1, dual else

 -  BYTE BTR0;       // SJW bit and baud rate prescale value
 -  BYTE BTR1;       // SAM bit and TSEG1/TSEG2 values
                     // applicable values should be taken from
                     // precalculated timing tables.

 -  BYTE EWL;        // error warning limit; default 96
 -  BYTE RXERR;      // receive error counter
 -  BYTE TXERR;      // transmit error counter
 -  BYTE ACR[4];     // current acceptance code
 -  BYTE AMR[4];     // current mask value
```

Return value:

- ERR_NOT_CON	driver not opened
- ERR_DEVCTL	connection broken
- ERR_OK	call successful

2.6 CanRestart()

Function: initialisation and restart of the CAN driver.

The initialisation is done with the current configuration parameters.
The status of the transmit and receive error counters are returned.

Prototype: short CanRestart(canhdl_t hdl);

Return value:

- ERR_NOT_CON	driver not opened
- ERR_DEVCTL	connection broken
- ERR_OK	call successful

2.7 RegRdPulse()

Function: register a pulse for delayed reads by CanRead().

Prototype: short RegRdPulse(canhdl_t hdl, struct sigevent * pulse);

Return value:

- ERR_NOTIFY:	not possible to register the pulse
- ERR_OK:	call successful

2.8 DeRegRdPulse()

Function: de-register a pulse for delayed reads by CanRd.

Prototype: short DeRegRdPulse(canhdl_t hdl);

Return value:

- ERR_NOT_CON	driver not opened
- ERR_DEVCTL	connection broken
- ERR_OK:	call successful

2.9 ResetAccPattern()

Function: resets the acceptance code and mask registers.
The controller will then accept all kind of message
(Basic Mode).

Prototype: short CanGetStatus(canhdl_t hdl, struct * status);

Return value:

- ERR_NOT_CON	driver not opened
- ERR_DEVCTL	connection broken
- ERR_OK:	call successful

2.10 CanGetStatus()

Function: read the status register of the CAN controller

Prototype: short CanGetStatus(canhdl_t hdl, struct * status);

```
struct status
{
   BYTE  STR;                 // status register
   BYTE  ErrState;            // 1 if there was a error state interrupt
   BYTE  OverRunState;        // 1 if there was a overrun state interrupt
   BYTE  WakeUpState;         // 1 if there was a wakeup state interrupt
   BYTE  ErrPassiveState;     // 1 if there was a passive error interrupt
   BYTE  ArbitLostState;      // 1 if there was a arbitration lost  interrupt
   BYTE  BusErrState;         // 1 if there was a bus error interrupt
   int   LostFrames;          // number of lost frames (ERR_OVERRUN)
};
```

Return value:

- ERR_NOT_CON	driver not opened
- ERR_DEVCTL	connection broken
- else	status byte

The status byte register STR must be interpreted by the following structure:

```
   BYTE  RBS     : 1;   receive buffer status / 0 = empty
   BYTE  DOS     : 1;   data overun status / 0 = absent
   BYTE  TBS     : 1;   transmit buffer status / 1 = buffer released
   BYTE  TCS     : 1;   transmission complete status / 1 = completed
   BYTE  RS      : 1;   receive status / 1 = receiving
   BYTE  TS      : 1;   transmit status / 1 = transmitting
   BYTE  ES      : 1;   error status / 1 = at least one error counter
                                       exeeds its limit
   BYTE  BS      : 1;   bus status / 1 = bus off
```

(minor bits first ...)

2.11 Programming example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/neutrino.h>

#include "candef.h"
#include "canstr.h"
#include "canglob.h"

char vers[]="Copyright A. Steinhoff";

struct  can_object msg;
unsigned char OutByte;
struct status  st;

//    Task          : CAN2-Driver example task
//    Function      : this task sends frames to partner task
//     Author       : Armin Steinhoff
//     Copyright    : Armin Steinhoff
//     Date         : 08-08 / 2004
//     Changes      :

void main(int argc, char **argv)
{
  short resp;
  canhdl_t hdl;

   if(ConnectDriver(1, "CANDRV", &hdl)) < 0)  // physical channel 1
      exit(-1)

   printf("CAN status: %04x ", CanGetStatus(hdl, &st) );

   // Output Frame
  msg.frame_inf.inf.DLC = 1;
  msg.frame_inf.inf.FF  = StdFF;  // standard frame
  msg.frame_inf.inf.RTR = 0;
  msg.id = 500;   // CAN ID
  msg.data[0] = OutByte++;
   for(;;)
   {

         resp = CanWrite(hdl,  &msg );

         printf("CAN Status: %04x resp: %04x\n", CanGetStatus(hdl, &st), resp);
         delay(2);
   }
}
```

®

DACHS

...the better Idea!

QNX
PARTNER
NETWORK

QNX

QNX SOFTWARE SYSTEMS