Advcan QNX Driver User Manual

V1.02

ADVANTECH *Open eAutomation, Boundless Integration*

Contents

1. Introduction

Advcan QNX driver is design for ADVANTECH CAN device in QNX OS.
Now it supports the following devices.

PCM-3680	2 port Isolated ISA CAN bus Card.
PCL-841	2 port Isolated ISA CAN bus Card.
TPC-662G	1 port Isolated ISA CAN bus Device on TPC-662G.
PCI-1680	2 port Isolated PCI CAN bus Card.
UNO-2052(E)	2 port Isolated PCI CAN bus Device on UNO-2052(E).
EAMB-PH07	1 port Isolated PCI CAN bus Card.
ADVANTECH GENERAL CAN PORT (1 PORT)	1 port Isolated PCI CAN bus Card.
ADVANTECH GENERAL CAN PORT (2 PORT)	2 port Isolated PCI CAN bus Card.
ADVANTECH GENERAL CAN PORT (4 PORT)	4 port Isolated PCI CAN bus Card.
ADVANTECH GENERAL CAN PORT (1 PORT, support CANopen)	1 port Isolated PCI CAN bus Card and support CANopen.
ADVANTECH GENERAL CAN PORT (2 PORT, support CANopen)	2 port Isolated PCI CAN bus Card and support CANopen.
ADVANTECH GENERAL CAN PORT (4 PORT, support CANopen)	4 port Isolated PCI CAN bus Card and support CANopen.

This driver supports QNX 6.5.X Intel x86 hardware platform.

1.1. System Requirement

- Hardware platform: Intel x86
- QNX version: 6.5.X

1.2. Driver configuration

Advcan QNX driver can auto detect ADVANTECH PCI CAN device and init it’s CAN port

➢ For PCI CAN device, user can only run it in terminal as follows:

```
#. /advcan &
```

Then, driver will auto detect the can port and make device node in /dev/ folder, for example, if there is a PCI-1680 CAN device with 2 CAN port, after you run the driver, /dev/advcan0 and /dev/advcan1 will be established.

- If the CAN device is ISA devices, user should input the base address and irq the CAN port used to the driver.

Example:
Configure /dev/advcan0 device, the base address is 0xda000 and irq is 3.

```
#. /advcan -isa0 addr=0xda000 irq=3 &
```

Configure two ports: /dev/advcan0 and /dev/advcan1, the base address for advcan0 is 0xda000, the irq for advcan0 is 3, the base address for advcan1 is 0xda200, irq the advcan1 is 5.

```
#. /advcan -isa0 addr=0xda000 irq=3 -isa1 addr=0xda200 irq=5 &
```

2. AdvCan Data Structures

Here are the data structures with brief descriptions:

canmsg_t	The CAN message structure
CanStatusPar_t	IOCTL generic CAN controller status request parameter structure
Command_par_t	IOCTL Command request parameter structure
ConfigureRTR_par_t	IOCTL ConfigureRTR request parameter structure
Send_par_t	IOCTL Send request parameter structure

2.1. Canmsg_t Struct Reference

Detailed Description

The CAN message structure.
Used for all data transfers between the application and the driver using read() or write().

Data Fields

int **flags**

flags, indicating or controlling special message properties

```
#define MSG_RTR (1<<0)         /**< RTR Message */

#define MSG_OVR (1<<1)         /**< CAN controller Msg overflow error */

#define MSG_EXT (1<<2)         /**< extended message format */

#define MSG_SELF (1<<3)        /**< message received from own tx */

#define MSG_PASSIVE (1<<4)  /**< controller in error passive */

#define MSG_BUSOFF (1<<5)   /**< controller Bus Off */

#define MSG_BOVR (1<<7)       /**< receive/transmit buffer overflow */
```

int **cob**

CAN object number, used in Full CAN.

unsigned long **id**

CAN message ID, 4 bytes.

timeval **timestamp**

time stamp for received messages

short int **length**

number of bytes in the CAN message

unsigned char **data** [CAN_MSG_LENGTH]

message data,8 bytes.

```
#define CAN_MSG_LENGTH 8 /**< maximum length of a CAN frame */
```

2.2.CanStatusPar Struct Reference

Detailed Description

IOCTL generic CAN controller status request parameter structure.

Data Fields

unsigned int **baud**

actual bit rate

unsigned int **status**

CAN controller status register.

unsigned int **error_warning_limit**

the error warning limit

unsigned int **rx_errors**

content of RX error counter

unsigned int **tx_errors**

content of TX error counter

unsigned int **error_code**

content of error code register

unsigned int **rx_buffer_size**

size of rx buffer

unsigned int **rx_buffer_used**

number of messages

unsigned int **tx_buffer_size**

size of tx buffer

unsigned int **tx_buffer_used**

number of messages

unsigned long **retval**

return value

unsigned int **type**

CAN controller / driver type.

2.3.Command_par_t Struct Reference

Detailed Description

IOCTL Command request parameter structure.

Data Fields

int **cmd**

special driver command

will be one of them

# define CMD_START 1

# define CMD_STOP 2

# define CMD_RESET 3

# define CMD_CLEARBUFFERS 4

int **target**

special configuration target

unsigned long **val1**

parameter for the target

unsigned long **val2**

parameter for the target

int **error**

return value

unsigned long **retval**

return value

2.4. ConfigureRTR_par Struct Reference

Detailed Description

IOCTL ConfigureRTR request parameter structure.

Data Fields

unsigned **message**

CAN message ID.

canmsg_t * **Tx**

CAN message struct.

int **error**

return value for errno

unsigned long **retval**

return value

2.5. Send_par Struct Reference

Detailed Description

IOCTL Send request parameter structure.

Data Fields

canmsg_t *	**Tx** *CAN message struct.*
int	**error** *return value for error*
unsigned long	**retval** *return value*

3. AdvCan Functions

int open(const char *pathname, int flags)

int ioctl(int fd, int request, ...)

ssize_t read(int fd, void *buf, size_t nbyte)

size_t write(int fd, const char *buf, size_t nbyte)

int close(int fd)

int select(int nfds, fd_set * readfds, fd_set * writefds, fd_set * exceptfds, struct timeval *timeout)

3.1. open Function Reference

Functions

int **open**(const char *pathname, int flags);

opens the CAN device

Function Documentation

int open(const char *pathname, int flags);

opens the CAN device for following operations

Parameters:

pathname CAN device pathname, usual /dev/advcan?

flags is one of **O_RDONLY**, **O_WRONLY** or **O_RDWR** which request opening the file read-only, write-only or read/write, respectively.

The open call is used to "open" the device. Doing a first initialization. Additional an ISR function is assigned to the IRQ.

Returns:

Open return the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately).

ERRORS

the following errors can occur

- ENXIO the file is a device special file and no corresponding device exists.
- EBUSY I/O region for hardware is not available

3.2.ioctl Function Reference

Functions

int **ioctl**(int fd, int request, **...**)

the CAN controllers control interface

Function Documentation

int ioctl(int fd, int request, **...**);

the CAN controllers control interface

Parameters:

fd The descriptor to change properties

request special configuration request

... traditional a *char* *argp

The *ioctl* function manipulates the underlying device parameters of the CAN special device. In particular, many operating characteristics of character CAN driver may be controlled with *ioctl* requests. The argument *fd* must be an open file descriptor.

An ioctl request has encoded in it whether the argument is an **in** parameter or **out** parameter. Macros and defines used in specifying an *ioctl* request are located in the file advcan.h .

The following *requests* are defined:

- CAN_IOCTL_COMMAND some commands for start, stop and reset the CAN controller chip

- CAN_IOCTL_CONFIG configure some of the device properties like acceptance filtering, bit timings, mode of the output control register or the optional software message filter configuration.
- CAN_IOCTL_STATUS request the CAN controllers status
- CAN_IOCTL_SEND a single message over the *ioctl* interface
- CAN_IOCTL_RECEIVE poll a receive message

The third argument is a parameter structure depending on the request. These are

```
struct Command_par

struct Config_par

struct CanStatusPar

struct ConfigureRTR_par

struct Send_par
```

The structures above are described in **advcan.h**

The following items are some configuration targets:

Bit Timing

The bit timing can be set using the *ioctl*(CONFIG,..) and the targets are CONF_TIMING or CONF_BTR. CONF_TIMING should be used only for the predefined Bit Rates (given in kbit/s). With CONF_BTR it is possible to set the CAN controllers bit timing registers individually by providing the values in **val1** (BTR0) and **val2** (BTR1).

Setting the bit timing register

advcan provides direct access to the bit timing registers, besides an implicate setting using the *ioctl* CONF_TIMING and fixed values in Kbit/s. In this case ioctl(can_fd, CAN_IOCTL_CONFIG, &cfg); is used with configuration target CONF_BTR. The configuration structure contains two values, *val1* and *val2* . The following relation to the bit timing registers is used regarding the CAN controller:

```
                       val1            val2

SJA1000                    BTR0            BTR1
```

Acceptance Filtering

Basic CAN. In the case of using standard identifiers in Basic CAN mode for receiving CAN messages only the low bytes are used to set acceptance code and mask for bits ID.10 ... ID.3

PeliCAN. For acceptance filtering the entries AccCode and AccMask are used like specified in the controller's manual for **Single Filter Configuration**. Both are 4 byte entries. In the case of using standard identifiers for receiving CAN messages also all 4 bytes can be used. In this case two bytes are used for acceptance code and mask for all 11 identifier bits plus additional the first two data bytes. The SJA1000 is working in the **Single Filter** Mode.

Example for extended message format

```
     Bits
mask  31 30 .....           4  3  2  1  0
code
-------------------------------------------
ID    28 27 .....           1  0  R  +--+-> unused
                               T
                               R

 acccode =  (id << 3) + (rtr << 2)
```

Example for base message format

```
     Bits
mask  31 30 .....           23 22 21 20 ... 0
code
-------------------------------------------
ID    10  9.....              1 0  R  +--+-> unused
                                T
                                R
```

You have to shift the CAN-ID by 5 bits and two bytes to shift them into ACR0 and ACR1 (acceptance code register)

```
 acccode =  (id << 21) + (rtr << 20)
```

In case of the base format match the content of bits 0...20 is of no interest, it can be 0x00000 or 0xFFFFF.

Returns:

On success, zero is returned. On error, -1 is returned, and error is set appropriately.

Example

```
    Config_par_t  cfg;
    volatile Command_par_t cmd;



    cmd.cmd = CMD_STOP;
    ioctl(can_fd, CAN_IOCTL_COMMAND, &cmd);


    cfg.target = CONF_ACCM;
    cfg.val1    = acc_mask;
    ioctl(can_fd, CAN_IOCTL_CONFIG, &cfg);
    cfg.target = CONF_ACCC;
    cfg.val2    = acc_code;
    ioctl(can_fd, CAN_IOCTL_CONFIG, &cfg);

    cmd.cmd = CMD_START;
    ioctl(can_fd, CAN_IOCTL_COMMAND, &cmd);
```

Other CAN_IOCTL_CONFIG configuration targets please refer to advcan.h in appendix and other source code of IOCTL; please refer to “examples” directory in advcan driver source code.

(see **advcan.h**)

```
CONF_LISTEN_ONLY_MODE   if set switch to listen only mode
                        (default false)
CONF_SELF_RECEPTION   if set place sent messages back in the rx queue
                        (default false)
CONF_BTR                   configure bit timing directly registers
CONF_TIMESTAMP           if set fill time stamp value in message structure
                        (default true)
CONF_WAKEUP               if set wake up waiting processes (default true)
```

3.3. read Function Reference

Functions

ssize_t **read**(int fd, void *buf, size_t nbyte)

the read system call

Function Documentation

ssize_t read(int fd, void *buf, size_t nbyte);

the read system call

Parameters:

fd	The descriptor to read from.
buf	The destination data buffer (array of CAN **canmsg_t**).
nbyte	The number of byte to read, nbyte=count* sizeof(canmsg_t),

Count is the number of CAN message to read.

read() attempts to read up to nbyte/sizeof(canmsg_t) CAN messages from file descriptor fd into the buffer starting at buf. The nbyte must an integral number of times to sizeof(canmsg_t). buf must be large enough to hold the *nbyte* data.

```
int got;
canmsg_t rx[80];         // receive buffer for read()

got = read(can_fd, rx , 80* sizeof( canmsg_t));
if( got > 0) {
  ...
} else {
    // read returned with error
    fprintf(stderr, "- Received got = %d\n", got);
    fflush(stderr);
}
```

Returns:

The number of bytes actually read, or -1 (errno is set).

On success, The number of bytes actually read is returned. It only support nonblock so it is not an error if this number is smaller than the requested; this may happen for example because fewer messages are actually available right now, or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately.

ERRORS

the following errors can occur

- EBADF **fd** isn't a valid open file descriptor

3.4. write Function Reference

Functions

size_t **write**(int fd, const char *buf, size_t nbyte)

write CAN messages to the network

Function Documentation

size_t write(int fd, const char *buf, size_t nbyte);

write CAN messages

Parameters:

fd The descriptor to write to.

buf The data buffer to write (array of CAN **canmsg_t**).

nbyte The number of byte to write, nbyte=count* sizeof(canmsg_t),
Count is the number of CAN message to write.

write writes up to nbyte/sizeof(canmsg_t) CAN messages to the CAN controller referenced by the file descriptor fd from the buffer starting at buf . It Only supports nobblock. If some data can be written without blocking the process, write() transfers what it can and returns the number of bytes written.

Returns:

On success, the bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and error is set appropriately.

Errors

the following errors can occur

- EBADF fd is not a valid file descriptor or is not open for writing.

3.5.close Function Reference

Functions

int close(int fd)

close a file descriptor

Function Documentation

int close(int fd);

close a file descriptor

Parameters:

fd The descriptor to close.

close closes the file specified by the given file descriptor..

Returns:

close returns zero on success, or -1 if an error occurred.

ERRORS

the following errors can occur

- BADF fd isn't a valid open file descriptor

3.6.select Function Reference

Functions

int select(int nfds, fd_set * readfds, fd_set * writefds, fd_set * exceptfds, struct timeval *timeout)

the select system call

Function Documentation

int select(int nfds, fd_set * readfds, fd_set * writefds, fd_set * exceptfds, struct timeval *timeout)

the select system call

Parameters:

ndfs	The number of file handle to be watched
readfds	be watched to see if characters become available for reading
writefds	be watched to see if a write will not block
exceptfds	be watched for exceptions
timeout	an upper bound on the amount of time elapsed before select returns

Returns:

The number of ready descriptors in the descriptor sets, 0 if the timeout expired, or -1 if an error occurs (errno is set).

Errors:

EBADF

One of the descriptor sets specified an invalid descriptor.

EFAULT

One of the pointers given in the call referred to a nonexistent portion of the address space for the process.

EINTR

A signal was delivered before any of the selected events occurred, or before the time limit expired.

EINVAL

A component of the pointed-to time limit is outside the acceptable range: *t_sec* must be between 0 and 10^8, inclusive; *t_usec* must be greater than or equal to 0, and less than 10^6.

4. Examples Reference

These are some simple examples to test the communication between two CAN channels.

- **receive**

Non block mode to receive message

- **transmit**

Non block mode to transmit message

- **receive-select**

Simple receiving using the select() call to wait for CAN messages

- **transmit-select**

Simple transmit using the ioctl() call.

- **baud**

Change CAN port’s baud rate.

- **getstate**

Show CAN port configuration and state.

- **filter**

Set the acceptance code and mask with ioctl().

Appendix

advcan.h

```
// #############################################################################
// *****************************************************************************
//                  Copyright ( c ) 2008, Advantech Automation Corp.
//      THIS IS AN UNPUBLISHED WORK CONTAINING CONFIDENTIAL AND PROPRIETARY
//               INFORMATION WHICH IS THE PROPERTY OF ADVANTECH AUTOMATION CORP.
//
//    ANY DISCLOSURE, USE, OR REPRODUCTION, WITHOUT WRITTEN AUTHORIZATION FROM
//               ADVANTECH AUTOMATION CORP., IS STRICTLY PROHIBITED.
// *****************************************************************************
// #############################################################################
/***
 * File:              advcan.h
 * Author:            jianfeng.dai
 * Created:           2008-10-20
 * Revision:          1.00
 *
 * Description:       Driver header file for development
 */
#include <sys/time.h>
#ifndef __CAN_H
#define __CAN_H

#define CAN_MSG_LENGTH 8     /**< maximum length of a CAN frame */
```

```
#define MSG_RTR      (1<<0)   /**< RTR Message */
#define MSG_OVR      (1<<1)   /**< CAN controller Msg overflow error */
#define MSG_EXT      (1<<2)   /**< extended message format */
#define MSG_SELF     (1<<3)   /**< message received from own tx */
#define MSG_PASSIVE  (1<<4)   /**< controller in error passive */
#define MSG_BUSOFF   (1<<5)   /**< controller Bus Off  */
#define MSG_         (1<<6)   /**<  */
#define MSG_BOVR     (1<<7)   /**< receive/transmit buffer overflow */

//mask used for detecting CAN errors in the canmsg_t flags field
#define MSG_ERR_MASK (MSG_OVR + MSG_PASSIVE + MSG_BUSOFF + MSG_BOVR)

/**
* The CAN message structure.
* Used for all data transfers between the application and the driver
* using read() or write().
*/
typedef struct
{
    int             flags;                 /** flags, indicating or controlling special
message properties */
    int             cob;                   /**< CAN object number, used in Full CAN  */
    unsigned   long id;                    /**< CAN message ID, 4 bytes  */
    struct timeval  timestamp;             /**< time stamp for received messages */
    short      int  length;                /**< number of bytes in the CAN message */
    unsigned   char data[CAN_MSG_LENGTH];  /**< data, 0...8 bytes */
} canmsg_t;


#define ADVCAN_IOC_MAGIC 'c'
#define CAN_IOCTL_COMMAND       __DIOT(_DCMD_CHR,ADVCAN_IOC_MAGIC + 1,Command_par_t)    /**<
IOCTL command request */
#define CAN_IOCTL_CONFIG        __DIOT(_DCMD_CHR,ADVCAN_IOC_MAGIC + 2,Config_par_t)     /**<
IOCTL configuration request */
#define CAN_IOCTL_SEND          __DIOT(_DCMD_CHR,ADVCAN_IOC_MAGIC + 3,int)              /**<
IOCTL request */
#define CAN_IOCTL_RECEIVE       __DIOT(_DCMD_CHR,ADVCAN_IOC_MAGIC + 4,int)              /**<
IOCTL request */
#define CAN_IOCTL_CONFIGURERTR  __DIOT(_DCMD_CHR,ADVCAN_IOC_MAGIC + 5,int)              /**<
IOCTL request */
#define CAN_IOCTL_STATUS        __DIOTF(_DCMD_CHR,ADVCAN_IOC_MAGIC + 6,CanStatusPar_t)
/**< IOCTL status request */
#define ADVCAN_IOC_MAXNR        6
```

```
/*CAN ioctl parameter types */
/*IOCTL Command request parameter structure */
struct Command_par
{
    int cmd;                /**< special driver command */
    int target;             /**< special configuration target */
    unsigned long val1;     /**< 1. parameter for the target */
    unsigned long val2;     /**< 2. parameter for the target */
    int error;              /**< return value */
    unsigned long retval;   /**< return value */
};

/* IOCTL Command request parameter structure */
typedef struct Command_par Command_par_t ; /**< Command parameter struct */

/* IOCTL CConfiguration request parameter structure */
typedef struct Command_par  Config_par_t ; /**< Configuration parameter struct */

/*IOCTL generic CAN controller status request parameter structure */
typedef struct CanStatusPar
{
   unsigned int   baud;                  /**< actual bit rate */
   unsigned int   status;                /**< CAN controller status register */
   unsigned int   error_warning_limit;   /**< the error warning limit */
   unsigned int   rx_errors;             /**< content of RX error counter */
   unsigned int   tx_errors;             /**< content of TX error counter */
   unsigned int   error_code;            /**< content of error code register */
   unsigned int   rx_buffer_size;        /**< size of rx buffer  */
   unsigned int   rx_buffer_used;        /**< number of messages */
   unsigned int   tx_buffer_size;        /**< size of tx buffer  */
   unsigned int   tx_buffer_used;        /**< number of messages */
   unsigned long  retval;                /**< return value */
   unsigned int   type;                  /**< CAN controller / driver type */
} CanStatusPar_t;


/**
 IOCTL Send request parameter structure */
typedef struct Send_par
{
   canmsg_t *Tx;            /**< CAN message struct  */
   int error;               /**< return value for errno */
   unsigned long retval;    /**< return value */
} Send_par_t ;
```

```
/**
 IOCTL Receive request parameter structure */
typedef struct Receive_par
{
   canmsg_t *Rx;          /**< CAN message struct  */
   int error;             /**< return value for errno */
   unsigned long retval;  /**< return value */
} Receive_par_t ;

/**
IOCTL ConfigureRTR request parameter structure */
typedef struct ConfigureRTR_par
{
   unsigned message;      /**< CAN message ID */
   canmsg_t *Tx;          /**< CAN message struct  */
   int error;             /**< return value for errno */
   unsigned long retval;  /**< return value */
} ConfigureRTR_par_t ;

/**
---------- IOCTL Command subcommands and there targets */

# define CMD_START         1
# define CMD_STOP          2
# define CMD_RESET         3
# define CMD_CLEARBUFFERS  4

/**
---------- IOCTL Configure targets */
# define CONF_ACC                 0  /* mask and code */
# define CONF_ACCM                1  /* mask only */
# define CONF_ACCC                2  /* code only */
# define CONF_TIMING              3  /* bit timing */
# define CONF_OMODE               4  /* output control register */
# define CONF_FILTER              5
# define CONF_FENABLE             6
# define CONF_FDISABLE            7
# define CONF_LISTEN_ONLY_MODE    8  /* for SJA1000 PeliCAN */
# define CONF_SELF_RECEPTION      9
# define CONF_BTR                 10 /* set direct bit timing registers(SJA1000) */
# define CONF_TIMESTAMP           11 /* use TS in received messages */
# define CONF_WAKEUP              12 /* wake up processes */
# define CONF_ACC_FILTER          20 /* Acceptance filter mode: 1-single, 0-dual*/
```

```
#endif      /* __CAN_H */
```