

AdvCan

User Manual



Introduction

Advcan driver is a migration from can4linux 3.4 and compatible with can4linux 3.4. The application for can4linux driver can run on advcan driver without any modification. The current stable version of Advcan is 2.6.

Now it supports following boards.

- PCI-1680 2 port Isolated PCI CAN-bus Card
- MIC-3680 2 port Isolated PCI CAN-bus Card.
- UNO-2052(E) 2 port Isolated UNO-2052(E) onboard PCI
 CAN device.
- PCM-3680 2 port Isolated ISA CAN-bus Card.

This driver supports Linux Kernel 2.4.xx, 2.6.xx, Intel x86 hardware platform. Any problem occurs, please contact ADVANTECH at:
support@ADVANTECH.com.tw.

In ADVANTECH's ftp sites, you may always find latest driver at:
<ftp://ftp.ADVANTECH.com> or <ftp://ftp.ADVANTECH.com.tw>.

System Reaurement

- Hardware platform: Intel x86
- Kernel version: 2.4.xx, 2.6.xx

This project was done to control devices via CAN.

The driver itself is highly configurable using the /proc interface of the LINUX kernel.

The following listing shows a typical configuration with four boards:

\$ grep . /proc/sys/Can/*								
/proc/sys/Can/AccCode:	-1	-1	-1	-1	-1	-1	-1	-1
/proc/sys/Can/AccMask:	-1	-1	-1	-1	-1	-1	-1	-1
/proc/sys/Can/Base:	800	672	832	896	684	696	856	872
/proc/sys/Can/Baud:	125	125	125	250	125	125	125	250
/proc/sys/Can/Chipset:	SJA1000							
/proc/sys/Can/IOModel:	pppppppp							
/proc/sys/Can/IRQ:	5	7	3	5	5	7	3	5
/proc/sys/Can/Outc:	250	250	250	0	250	250	250	0
/proc/sys/Can/Overrun:	0	0	0	0	0	0	0	0
/proc/sys/Can/RxErr:	0	0	0	0	0	0	0	0
/proc/sys/Can/Timeout:	100	100	100	100	100	100	100	100
/proc/sys/Can/TxErr:	0	0	0	0	0	0	0	0
/proc/sys/Can/dbgMask:	0							
/proc/sys/Can/version:	advcan-2.6							

The following sections are describing the sysctl entries.

AccCode/AccMask

contents of the message acceptance mask and acceptance code registers of 82x200/SJA1000 compatible CAN controllers (see can_ioctl()).

Base

CAN controllers base address for each board. Depending of the IOModel entry that can be a memory or I/O address.

(don't modify it if device is PCI boards)

Baud

used bit rate for this board in Kbit/s

Chipset

name of the supported CAN chip used with this boards Read only for this version.

IOModel

one letter for each port. Read-only. Read the CAN register access model. The following models are currently supported:

- m - memory access, the registers are directly mapped into memory(PCM-3680)
- p - port I/O, 80x86 specific I/O address range (PCI-1680,MIC-3680,UNO-2052(E))

IRQ

used IRQ numbers, one value for each board. (don't modify it if device is PCI boards)

Outc

value of the output control register of the CAN controller.

Overrun

counter for overrun conditions in the CAN controller

RxErr

counter for CAN controller rx error conditions

Timeout

time out value for waiting for a successful transmission

TxErr

counter for CAN controller tx error conditions

version

read only entry containing the drivers version number

Please see also at `can_iocctl()` for some additional descriptions.

For initially writing these sysctl entries after loading the driver (or at any time) a shell script utility does exist. It uses a board configuration file that is written *over* `/proc/sys/Can`.

```
utils/cansetup advpci.conf
```

or, like used in the Makefile:

```
CONFIG := advpci.conf
```

```
# load host specific CAN configuration
```

```
load:
```

```
    @echo "Loading etc/${CONFIG}.conf CAN configuration"
    utils/cansetup etc/${CONFIG}.conf
    echo 0 >/proc/sys/Can/dbgMask
```

Example *.conf files are located in the etc/ directory.

Contents

Chapter	1	AdvCan Data Structures	2
	1.1	Canmsg_t Struct Reference	3
	1.1.1	Detailed Description	3
	1.2	Command_par_t Struct Reference	4
	1.2.1	Detailed Description	4
	1.3	ConfigureRTR_par Struct Reference	5
	1.3.1	Detailed Description	5
	1.4	Send_par Struct Reference	6
	1.4.1	Detailed Description	6
Chapter	2	AdvCan Functions.....	8
	2.1	open Function Reference.....	9
	2.1.1	Function Documentation.....	9
	2.2	ioctl Function Reference	10
	2.2.1	Function Documentation.....	10
	2.3	read Function Reference.....	13
	2.3.1	Function Documentation.....	14
	2.4	write Function Reference	15
	2.4.1	Function Documentation.....	15
	2.5	close File Reference	16
	2.5.1	Function Documentation.....	16
	2.6	select Function Reference	17
	2.6.1	Function Documentation.....	17
Chapter	3	Examples Reference	20
Appendix	1	can4linux.h	24

CHAPTER 1

AdvCan Data Structures

Chapter 1 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

1.1 Canmsg_t Struct Reference

1.1.1 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 <i>flags, indicating or controlling special message properties</i> <i>#define MSG_RTR (1<<0) /**< RTR Message */</i> <i>#define MSG_OVR (1<<1) /**< CAN controller Msg</i> <i>overflow error */</i> <i>#define MSG_EXT (1<<2) /**< extended message format */</i> <i>#define MSG_SELF (1<<3) /**< message received from own</i> <i>tx */</i> <i>#define MSG_PASSIVE (1<<4) /**< controller in error passive */</i> <i>#define MSG_BUSOFF (1<<5) /**< controller Bus Off */</i> <i>#define MSG_BOVR (1<<7) /**< receive/transmit buffer</i> <i>overflow */</i>
int	cob <i>CAN object number, used in Full CAN.</i>
unsigned long	id <i>CAN message ID, 4 bytes.</i>
timeval	timestamp <i>time stamp for received messages</i>
short int	length <i>number of bytes in the CAN message</i>
unsigned char	data [CAN_MSG_LENGTH] <i>message data, 8 bytes.</i> <i>#define CAN_MSG_LENGTH 8 /**< maximum length of a CAN</i> <i>frame */</i>

1.2 Command_par_t Struct Reference

1.2.1 Detailed Description

IOCTL Command request parameter structure.

Data Fields

int	cmd <i>special driver command will be one of them</i> # define CMD_START 1 # define CMD_STOP 2 # define CMD_RESET 3 # define CMD_CLEARBUFFERS 4
int	target <i>special configuration target</i>
unsigned long	val1 <i>parameter for the target</i>
unsigned long	val2 <i>parameter for the target</i>
int	error <i>return value</i>
unsigned long	retval <i>return value</i>

1.3 ConfigureRTR_par Struct Reference

1.3.1 Detailed Description

IOCTL ConfigureRTR request parameter structure.

Data Fields

unsigned	message <i>CAN message ID.</i>
canmsg_t *	Tx <i>CAN message struct.</i>
int	error <i>return value for errno</i>
unsigned long	retval <i>return value</i>

1.4 Send_par Struct Reference

1.4.1 Detailed Description

IOCTL Send request parameter structure.

Data Fields

unsigned	message <i>CAN message ID.</i>
canmsg_t *	Tx <i>CAN message struct.</i>
int	error <i>return value for errno</i>
unsigned long	retval <i>return value</i>

CHAPTER 2

AdvCan Functions

Chapter 2 AdvCan Functions

int **open**(const char *pathname, int flags)
int **ioctl**(int fd, int request, ...)
ssize_t **read**(int fd, void *buf, size_t count)
size_t **write**(int fd, const char *buf, size_t count)
int **close**(int fd)
int **select**(int nfds, fd_set * readfds, fd_set * writefds, fd_set *
exceptfds, struct timeval *timeout)

2.1 open Function Reference

Functions

unsigned	open (const char *pathname, int flags); <i>opens the CAN device</i>
----------	---

2.1.1 Function Documentation

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

opens the CAN device for following operations

Parameters:

pathname	device pathname, usual /dev/can
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 according the to values in the /proc/sys/Can file system. 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.
- **EINVAL** illegal minor device number
- **EINVAL** wrong IO-model format in /proc/sys/Can/IOmodel
- **EBUSY** IRQ for hardware is not available
- **EBUSY** I/O region for hardware is not available

2.2 ioctl Function Reference

Functions

int	open (const char *pathname, int flags); <i>opens the CAN device</i>
-----	---

2.2.1 Function Documentation

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

the CAN controllers control interface

Parameters:

<i>fd</i>	The descriptor to change properties
<i>request</i>	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 can4linux.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 **can4linux.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 imprecise 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:

	<code>val1</code>	<code>val2</code>
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 controllers 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 can4linux.h in appendix and other source code of IOCTL , please refer to °examples°± directory in advcan driver source code.

(see can4linux.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)

2.3 read Function Reference

Functions

size_t	read (int fd, void *buf, size_t count) <i>the read system call</i>
--------	--

2.3.1 Function Documentation

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

the read system call

Parameters:

<i>fd</i>	The descriptor to read from.
<i>buf</i>	The destination data buffer (array of CAN canmsg_t).
<i>count</i>	The number of CAN message to read.

read() attempts to read up to count CAN messages(**not bytes!**) from file descriptor fd into the buffer starting at buf. buf must be large enough to hold the "count" times the size of one CAN messages structure canmsg_t.

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

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

ERRORS

the following errors can occur

- EINVAL buf points not to an large enough area

Returns:

On success, the number of CAN messages read is returned (zero indicates end of file). It is not an error if this number is smaller than the messages 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.

2.4 write Function Reference

Functions

size_t	write (int fd, const char *buf, size_t count) <i>write CAN messages to the network</i>
--------	--

2.4.1 Function Documentation

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

write CAN messages to the network

Parameters:

<i>fd</i>	The descriptor to write to.
<i>buf</i>	The data buffer to write (array of CAN canmsg_t).
<i>count</i>	The number of CAN message to write.

write writes up to count CAN messages(**not bytes!**) to the CAN controller referenced by the file descriptor fd from the buffer starting at buf.

Errors

the following errors can occur

- EBADF fd is not a valid file descriptor or is not open for writing.
- EINVAL fd is attached to an object which is unsuitable for writing.
- EFAULT buf is outside your accessible address space.
- EINTR The call was interrupted by a signal before any data was written.

Returns:

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

2.5 close File Reference

Functions

int	close(int fd) <i>close a file descriptor</i>
-----	---

2.5.1 Function Documentation

int close(int fd);

close a file descriptor

Parameters:

<i>fd</i>	The descriptor to close.
-----------	--------------------------

close closes a file descriptor, so that it no longer refers to any device and may be reused.

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

2.6 select Function Reference

Functions

int	<code>select(int nfds, fd_set * readfds, fd_set * writefds, fd_set * exceptfds, struct timeval *timeout)</code> <i>the select system call</i>
-----	--

2.6.1 Function Documentation

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

W

Parameters:

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

Errors

- **ERRORS EBADF** An invalid file descriptor was given in one of the sets.
- **EINTR** A non blocked signal was caught.
- **EINVAL** n is negative or the value contained within timeout is invalid.
- **ENOMEM** select was unable to allocate memory for internal tables.

CHAPTER 3

Examples Reference

Chapter 3 Examples Reference

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

- **receive**

polling or nonblocking mode to receive message

- **transmit**

polling mode to transmit message

- **receive-select**

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

- **transmit-select**

simple transmit using the select() call

- **send_ioctl**

simple transmit using the SEND ioctl command

- **baud**

simple driver test: change the bit rate registers with ioctl()

the change itself stays after program is finished.

you can check it by read

cat /proc/sys/Can/Baud

before and after using this command

- **acceptance**

simple driver test: change the message filter with ioctl()

the change itself stays after program is finished.

you can check it by read

cat /proc/sys/Can/AccCode /proc/sys/Can/AccMask

before and after using this command

- **ctest**

very simple and basic driver test: read a message every sleep ms

- **singlefilter**

In this example, when set accept code to only accept the message which id = 10 and rtr = 0

using /dev/can0 device to accept message

using /dev/can1 to transmit message

- **selfreception**

self reception example

when you run this example, you should run another receive example

to receive the message transmit by selfreception and at the same time, selfreception will

also receive the message transmit by itself.

can4linux.h

Appendix 1 can4linux.h

```
00001 /*
00002  * can4linux.h - can4linux CAN driver module
00003  *
00004  * This file is subject to the terms and condi-
00005  * tions of the GNU General Public
00006  * License. See the file "COPYING" in the main
00007  * directory of this archive
00008  * for more details.
00009  *
00010  *
00011  *
00012  *-----
00013  *
00014  *
00015  *
00016  *
00017  *-----
00018  */
00019
00020
00021
00022
00023
00024
00025
00026
00027
00028
00029
00030
00031
00032
00033
00034 # ifndef __CAN_H
00035 #  define __CAN_H
00036
00037
00038 #  define CAN4LINUXVERSION 0x0304 /*(Version
00039 3.3)*/
00040 #  ifndef __KERNEL__
00041 #  include <sys/time.h>
00042 #  endif
00043  /*----- the can message structure */
00044
00045 #  define CAN_MSG_LENGTH 8
00046 #  define MSG_RTR (1<<0)
00047 #  define MSG_OVR (1<<1)
00048 #  define MSG_EXT (1<<2)
```



```

00051 # define MSG_SELF (1<<3)
00052 # define MSG_PASSIVE (1<<4)
00053 # define MSG_BUSOFF (1<<5)
00054 # define MSG_ (1<<6)
00055 # define MSG_BOVR (1<<7)
00059 # define MSG_ERR_MASK (MSG_OVR +
                           MSG_PASSIVE +
                           MSG_BUSOFF +
                           MSG_BOVR)

00060
00066 typedef struct {
00068     int    flags;
00069     int    cob;
00070     unsigned long id;
00071     struct timeval timestamp;
00072     short int length;
00073     unsigned char data[CAN_MSG_LENGTH];
00074 } canmsg_t;
00075
00076
00077
00080 /* Use 'c' as magic number, follow chapter 6 of
      LDD3 */
00081 # define CAN4L_IOC_MAGIC 'c'
00082
00083 # define CAN_IOCTL_COMMAND 0
00084 # define CAN_IOCTL_CONFIG 1
00085 # define CAN_IOCTL_SEND 2
00086 # define CAN_IOCTL_RECEIVE 3
00087 # define CAN_IOCTL_CONFIGURERTR 4
00088 # define CAN_IOCTL_STATUS 5
00090 /*----- CAN ioctl parameter types */
00091
00093 struct Command_par {
00094     int    cmd;
00095     int    target;
00096     unsigned long val1;
00097     unsigned long val2;
00098     int    error;
00099     unsigned long retval;
00100 };
00101
00102
00105 typedef struct Command_par Command_par_t ;
00108 typedef struct Command_par Config_par_t ;

```

```

00113     typedef struct CanStatusPar {
00114         unsigned int baud;
00115         unsigned int status;
00116         unsigned int error_warning_limit;
00117         unsigned int rx_errors;
00118         unsigned int tx_errors;
00119         unsigned int error_code;
00120         unsigned int rx_buffer_size;
00121         unsigned int rx_buffer_used;
00122         unsigned int tx_buffer_size;
00123         unsigned int tx_buffer_used;
00124         unsigned long retval;
00125         unsigned int type;
00126     } CanStatusPar_t;
00127
00130 # define    CAN_TYPE_UNSPEC            0
00131 # define    CAN_TYPE_SJA1000          1
00132 # define    CAN_TYPE_FlexCAN          2
00133 # define    CAN_TYPE_TouCAN           3
00134 # define    CAN_TYPE_82527            4
00135 # define    CAN_TYPE_TwinCAN          5
00136 # define    CAN_TYPE_BlackFinCAN      6
00137
00138
00141     typedef struct Send_par {
00142         canmsg_t *Tx;
00143         int      error;
00144         unsigned long retval;
00145     } Send_par_t ;
00146
00149     typedef struct Receive_par {
00150         canmsg_t *Rx;
00151         int      error;
00152         unsigned long retval;
00153     } Receive_par_t ;
00154
00157     typedef struct ConfigureRTR_par {
00158         unsigned message;
00159         canmsg_t *Tx;
00160         int      error;
00161         unsigned long retval;
00162     } ConfigureRTR_par_t ;
00163

```

```

00167 # define    CMD_START        1
00168 # define    CMD_STOP          2
00169 # define    CMD_RESET          3
00170 # define    CMD_CLEARBUFFERS  4
00171
00172
00173
00174
00178 # define    CONF_ACC           0 /* mask and code */
00179 # define    CONF_ACCM           1 /* mask only */
00180 # define    CONF_ACCC           2 /* code only */
00181 # define    CONF_TIMING          3 /* bit timing */
00182 # define    CONF_OMODE           4 /* output control
                                register */
00183 # define    CONF_FILTER          5
00184 # define    CONF_FENABLE         6
00185 # define    CONF_FDISABLE        7
00186 # define    CONF_LISTEN_ONLY_MODE 8 /* for SJA1000
                                Pelican
                                */
00187 # define    CONF_SELF_RECEPTION 9 /* */
00188 # define    CONF_BTR             10 /* set direct
                                bit timing
                                registers
                                (SJA1000) */
00189
00190 # define    CONF_TIMESTAMP        11 /* use TS in
                                received messages
                                */
00191 # define    CONF_WAKEUP           12 /* wake up
                                processes */
00192
00193 # endif      /* __CAN_H */

```

