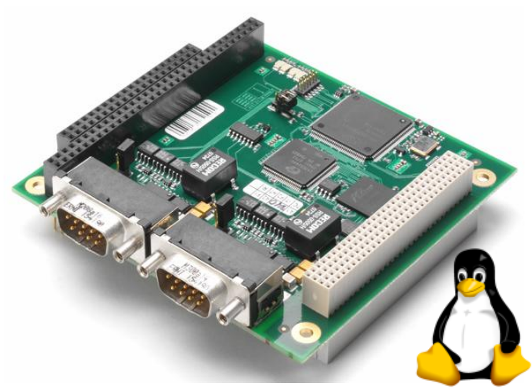
Linux CAN driver manual

HiCO.CAN MiniPCI, PCI-104 and PC/104+ boards



Date 23/09/2010 11:59

10/09/2010/Ny	Added pinout for HiCO-CAN-MiniPCI-4C

1 Introduction

The driver is implemented as a Loadable Kernel Module (LKM), which is able to handle multiple boards and multiple clients (applications) simultaneously on the system. Interface to the driver is stream interface (or character device interface) that has already been established as the ‘standard’ interface for CAN bus on Linux. The function of the Driver API follows the POSIX standard as far as possible – CAN traffic is read and written with the `read()` and `write()` system calls and Hardware specific commands (like setting the bitrate) are given with diverse `ioctl()` calls.

HiCO.CAN-MiniPCI

2 First steps with the driver

2.1 Building and installing the kernel module

In order to build and install the module, give following commands:

```
$> cd driver

# Build the module
$> make

# load the module in kernel and create the device nodes
$> sudo make install
```

You should now be able to read the status of the can nodes in /proc/hcan/* files and you should also find some /dev/can* device nodes.

Note: Before you can build a kernel module, *you need to have the kernel sources installed* for the kernel you are running. You can install the sources normally with the package manager of your Linux distribution (Yast on SuSE, apt-get on Debian based systems and so on.

Note: You might need to set the KERNELDIR variable in the Makefile to point to your kernel sources

2.2 Starting the example application

In example directory is a simple application which you can use for testing and starting point for your own applications.

1. Compile the example application by running make in the directory where the source files are (i.e. cd example; make)
2. Connect the CAN nodes with the delivered Y-cable
3. Start the application:

```
#> example /dev/can0 /dev/can1
received message ID=000000ab rtr=0 ts=5597 data 8 bytes: 00 01
02 03 04 05 06 07
```

2.3 Getting driver status information

You can read some status information from the /dev/hcanpci/* device files. It contains some useful debug information in case something doesn't work.

```
# Board specific data in a file named after the PCI name of
# the device (same as with command 'lspci')
$> cat /proc/hcanpci/board_02_0c
board 0000:02:0c.0
nodes: can0, can1

# Firmware versions. fw1 is the boards bootloader (permanent).
# fw2 is the actual firmware, which can be updated
fw1 version: 743
fw1 date: 20 09.07.07
fw2 version: 743
fw2 date: 20 09.07.07
pci rev: 0xc
fw state: f2f2 - firmware (fw2)
error code: 0000 - no error
interrupt: 9

# The can* files provide useful information on the
# CAN node status
$> cat /proc/hcanpci/can0
can0: node 0 on board 0000:02:0c.0

# Current operation mode (reset, active, passive or baudscan)
# and bitrate (aka baudrate)
mode: reset
bitrate: 20kbps

# Controller status - ok/errPassive/busOff
status: 3c - ok
errCnt tx/rx: 0/0

# status for boards receive and transmit buffers
dpm Tx buf: 0/22
dpm Rx buf: 0/200
sram Rx buf: 0/428

# Counter for debugging purposes (received/sent/filtered).
# NOTE: These are 16 bit counters!
rec/snt/flt: 0/0/0
```

3 Programming practice

3.1 Opening and closing a CAN node

The CAN device nodes (/dev/can*) are opened with the open() system call. The open() returns a file descriptor which is a handle of the opened CAN node. The CAN node is closed with the close() system call. Example:

```
int canFd;

//open CAN node 0 for reading and writing
canFd = open("/dev/can0",O_RDWR);
assert(canFd>0);

/* Configure CAN node */
.
.
.
.// ** application code **
.
close(canFd);
```

Note
Once configured, a CAN node can be opened and closed multiple times, without affecting the CAN bus or the configuration in any way.

3.2 Blocking and non-blocking operations

With the fcntl() system call (see man page for fcntl) you can set the hicocan device file-descriptor in non-blocking mode. In blocking mode (default), the read() and write() system calls will block (i.e. the system call will not return) until there's a CAN telegram available for reading or there is space left for writing. In non-blocking mode the calls will return immediately with error EAGAIN if there's nothing to read or there's no space left for writing. Following snippet shows how to change between the blocking and non-blocking modes:

```
.
//set the file descriptor in non-blocking mode
flags = fcntl(hicocan_fd, F_GETFL);
fcntl(hicocan_fd, F_SETFL, (flags | O_NONBLOCK) );
.
.
//and back to blocking mode
flags = fcntl(hicocan_fd, F_GETFL);
fcntl(hicocan_fd, F_SETFL, (flags & ~O_NONBLOCK) );
.
```

3.3 Using the driver with poll() and select()

The driver supports the poll() and select() system calls and you can add a hicocan file descriptor to the list of file descriptors you are monitoring. For more information see man pages of poll() and select(). From the two functions, select is more portable.

3.3.1 Implementing timeout with select()

In some cases it is desired that the read or write operation blocks only for a given time and if the expected event (i.e. data ready for reading or space left for writing) doesn't occur within this time the call returns. The read() and write() system calls do not directly support this, but you can achieve the same behaviour with the select() system call which the driver supports (see man page of select()). Following snippet gives an example how to implement this:

```
/* Read one CAN message. If the receive buffer is empty, wait
for data for time pointed by the timeout argument in
milliseconds */
int read_timeout(int fd, struct can_msg *buf, unsigned int
timeout)
{
    fd_set fds;
    struct timeval tv;
    int sec,ret;
    FD_ZERO(&fds);
    /* Convert from milliseconds */
    sec=timeout/1000;
    tv.tv_sec=sec;
    tv.tv_usec=(timeout-(sec*1000))*1000;

    FD_SET(fd,&fds);
    ret=select(fd+1,&fds,0,0,&tv);
    if(ret==0){
       return 0; /* timed out! */
    } else if (ret<0) {
       return errno;
    } else {
       /* Data available - read the CAN message */
       ret=read(fd,buf,sizeof(struct can_msg));
       return ret;
    }
}
```

3.4 Asynchronous Notification (IO Signals)

If enabled, the kernel module sends a signal SIGIO when a CAN telegram is received. The signal are enabled with the fcntl() system call. Following snippet shows how to enable the IO signals:

```
.
/* Set this process as the owner.. */
fcntl( hicocan_fildes, F_SETOWN, getpid() );
/*..and setting the FASYNC flag */
fcntl( hicocan_fildes, F_SETFL,
       fcntl(fildes,F_GETFL) | FASYNC );
.
```

Generally, IO signal handling can get messy and hard to implement. It is normally better to use the standard blocking operations, non-blocking operations with select() or poll(), or put a thread sleeping for input.

3.5 Multiprocessing

The driver can be opened used by multiple processes/threads simultaneously. When opening CAN nodes for use with multiple processes, extra care has to be taken with ioctl calls such as IOC_RESET_BOARD, IOC_RESET_TIMESTAMP, since these affect both CAN nodes on the board.

Although the driver is multithread and multiprocessing capable, it is recommended to use only one reader/write per CAN node.

4 Driver API

4.1 CAN telegram structure

CAN Messages are sent and received by using a C-structure **struct can_msg** which is defined in the API header file **hico_api.h**. Following list explains the fields:

msg.**id**	CAN Id or arbitration field of the message
msg.**data[0:7]**	Message databytes
msg.**dlc**	Data length code. Indicates how many bytes of the msg.data field are valid data.
msg.**ff**	Frame Format. 0 is a standard CAN frame with 11 bit identifier and 1 is an extended CAN frame with 29 bit identifier.
msg.**rtr**	Remote Transmission Request flag
msg.**dos**	Data Overrun Status. Indicates that data overrund has occurred and one or more CAN messages were lost. This normally happens if the application doesn't empty the Rx buffers (i.e. read messages) fast enough.
msg.**ts**	Time stamp of the received message. Ignored in Tx messages.
msg.**node**	Number of the node which received the message.
msg.**iopin**	Status of the nodes IO pin. On *Fault tolerant CAN* this is the line error status (1->ok, 0->line error detected).

4.2 *Standard system calls*

Like earlier explained, the API is a set of Linux file-system API calls. The following list of functions shows which calls can be used with a hicocan file descriptor (See also man pages for these system calls).

4.2.1 return values

All of the system calls to the driver may return with error EIO which means that the boad is in error state. In this case you need to reset the board. Other than that the return values follow the POSIX convention.

When a system call returns a EIO, the application can use ioctl calls IOC_GET_CAN_STATUS or IOC_GET_BOARD_STATUS to check what might be the reason.

4.2.2 read()

read() system call reads one can message from the driver into the given buffer. If the node is in blocking mode, the function will block until there is data available. Example:

```
struct can_msg canMsg;

ret=read(canFd, &canMsg , sizeof(canMsg));
assert(ret==sizeof(struct can_message);
```

Return value is number of read bytes (i.e. sizeof(canMsg)) or <-1 on error (errno is set).

4.2.3 write()

Writes one can message to CAN bus. If the transmit buffer is full and the node is in blocking mode, the function will block until there is space available. Example:

```
struct can_msg canMsg;

ret=write(canFd, &canMsg , sizeof(canMsg));
assert(ret==sizeof(struct can_message);
```

Return value is number of written bytes (i.e. sizeof(canMsg)) or <-1 on error (errno is set)

4.2.4 ioctl()

Give a HW specific command to the driver. See chapter “**Ioctl calls**” for the different calls.

4.2.5 poll() and select()

See man pages for the functions. You can use a hicocan* file descriptor with poll() or select() just like any other file descriptor which support these calls.

If there is space in the transmit buffer, these calls will return the “output-event” (POLLOUT for poll) set for the given file descriptor. If there are messages to be read from the boards Rx buffers these calls will return the “input-event” (POLLIN for poll)set for the given file descriptor.

4.2.6 open()

Opens a CAN node for use and returns a file descriptor which is the CAN nodes "handle". open() doesn’t perform any CAN specific actions. Example:

```
canFd=open("/dev/can0",O_RDWR) );
assert(canFd>0);
```

4.2.7 close()

Closes the can node. This function doesn’t perform any CAN specific actions.

4.3 Ioctl calls

Ioctl calls are also described in the hicocan.h API header file.

IOC_RESET_BOARD	
Description	Reset the board and the firmware. *Note that this command affects both of the CAN nodes on the board.*
Parameters	none

IOC_START	
Description	Start listening CAN traffic. Messages that pass the acceptance masking are saved into the drivers receive buffer. Transmit enabled.
Parameters	none

IOC_START_PASSIVE	
Description	Start the CAN node in passive mode. Listen to the CAN bus traffic but don't affect it in any way. CAN messages are received normally but they are not acknowledged (ACK bit in the CAN telegram stream is not set). Transmission is disabled.
Parameters	none

IOC_START_BAUDSCAN

Description	This command will put the CAN node in baudscan mode. The node will probe for the correct baudrate on the CAN bus, but doesn't affect the traffic in any way – it just listens. Of course, in order to the scan to succeed there has to be traffic on the bus. The first succesfully received message is put in the boards receive queue after which the node is in active state (like after givining IOC_START command)
Parameters	None

IOC_STOP

Description	Stop listening CAN traffic and disable transmit
Parameters	none

IOC_GET_MODE

Description	Get current operation mode of the CAN node (defined by the IOC_START_* and IOC_STOP commands). Possible states are CM_BAUDSCAN, CM_PASSIVE, CM_ACTIVE and CM_RESET.
Parameters	int *mode Variable where the mode is to be saved.

Example

```
int mode;

ret=ioctl(RxNode,IOC_GET_BITRATE, &mode);
assert(ret==EOK);
```

IOC_SET_BITRATE

Description	Set the bitrate (aka. baudrate) for the given CAN node.
Parameters	`Int *bitrate` Pointer to an integer with the desired bitrate. The bitrate can be one of following: BITRATE_10k BITRATE_20k BITRATE_50k BITRATE_100k BITRATE_125k BITRATE_250k BITRATE_500k BITRATE_800k BITRATE_1000k
Notes	Set the bitrate *before* starting the CAN node

IOC_GET_BITRATE

Description	Get current bitrate.
Parameters	`int *bitrate` Variable where the bitrate is to be saved. See IOC_SET_BITRATE for the values.
Example	

```
int bitrate;

ret=ioctl(canFd,IOC_GET_BITRATE, &bitrate);
assert(ret==EOK);
```

IOC_GET_CAN_STATUS

Description Get current state of the CAN node. The returned value includes flags for error passive mode, bus off mode and the TxRx Error counters. The flags are defined in the api header file.

Parameters uint32_t *status

variable where the CAN state is to be saved.

Example

```
uint32_t status;

ret=ioctl(canFd,IOC_GET_CAN_STATUS,&status);
assert(ret==0);
if(status&CS_ERROR_PASSIVE ||
  status&CS_ERROR_BUS_OFF){
    errx(1,"can node in error state – check the cable!");
}
```

IOC_GET_BOARD_STATUS

Description Get the current state of the board firmware. For an application, the only usable value that this ioctl call returns is BS_RUNNING_OK. If the value is something else the board is in exception/error state and must be reset to get functional again

Parameters uint32_t *state

Variable where the CAN state is to be saved.

IOC_SET_FILTER

Description	Add an acceptance filter into the list of active acceptance filters. By default, the acceptance masking is not active and all the messages are let through. There are two different types of filters: FTYPE_AMASK – works like the traditional acceptance mask and code. filter.mask defines which bits are of importance and filter.code defines their value (like the sja1000 acceptance mask) FTYPE_RANGE – CAN telegrams between filter.lower and filter.upper are let through. If the board internal filter table for the node (4 filters per CAN node) is full, this function will return with error EBUSY.
Parameters	struct can_filter *filter pointer to a filter structure
Example	

```
/* Set a acceptance filter of type ‘range’. Messages
with id’s between 0xa and 0xf are let through*/
filter.type=FTYPE_RANGE;
filter.lower=0xa;
filter.upper=0xf;
ret=ioctl(RxNode,IOC_SET_FILTER,&filter);
assert(ret==0);
```

IOC_CLEAR_FILTERS

Description	Clear all acceptance filters for the node and set the acceptance in default state (all messages let through)
Parameters	none

IOC_MSGS_IN_RXBUF

Description	Return the number of messages that can be read from the boards receive buffer.
Parameters	`int *nMsgs` variable where to save the value.

IOC_RESET_TIMESTAMP

Description	Reset the CAN message timestamp timer. *Note that this command affects both of the nodes.*
Parameters	none

4.4 Example Application

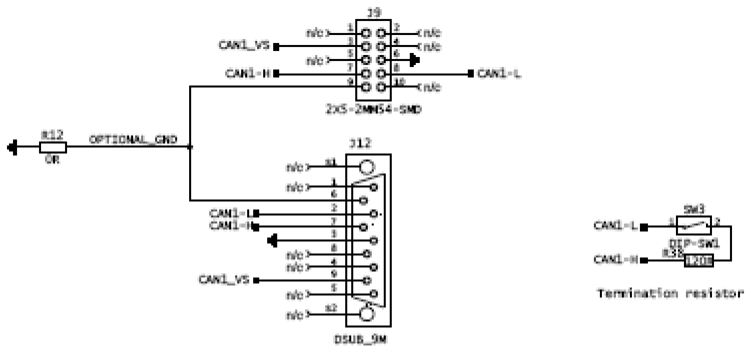
Following C-Program shows a trivial example how to open and configure a CAN node and send one message to the CAN bus.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <err.h>
#include "hico_api.h"

int main(int argc, char *argv[])
{
    int ret,i,val;
    int canFd;
    struct can_msg msg;


    /* Open the CAN nodes for reading and writing */
    canFd = open("/dev/can0", O_RDWR);
    if(canFd<0){
       err(1, "could not open can node");
    }

    /* Set bitrate */
    val=BITRATE_500k;
    ret=ioctl(canFd,IOC_SET_BITRATE,&val);
    if(ret!=0){
       err(1, "could not set bitrate");
    }

    /* Start the node */
    ret=ioctl(canFd,IOC_START);
    if(ret!=0){
       err(1, "IOC_START");
    }

    /* Compose a CAN message with some dummy data */
    memset(&msg,0,sizeof(struct can_msg));
    msg.ff = FF_NORMAL;
    msg.id = 0xab;
    msg.dlc = 8;
    for(i=0;i<msg.dlc;i++){
       msg.data[i]=i;
    }
    /* Write the message to the CAN bus */
    ret=write(canFd,&msg,sizeof(struct can_msg));
    if(ret!=sizeof(msg)){
      err(1, "Failed to send message");
    }

    /* Close the nodes after usage */
    close(canFd);

    return 0;
}
```

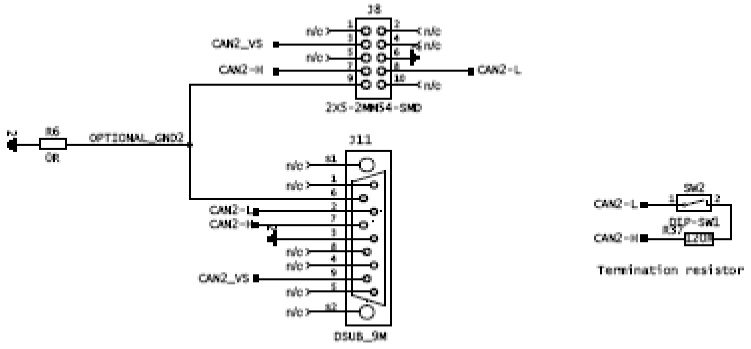
5 PCI104 and PC/104+ hardware.

5.1 CAN connector pinout

Connectors for the first CAN channel



Connectors for the second CAN channel



5.2 Fault Tolerant CAN

HiCO.CAN-PCI and PC/104+ boards are also available as Fault tolerant versions. Please refer to the hardware datasheet for more information

6 HiCO.CAN-MiniPCI Hardware

Two independent CAN channels compliant with the CAN 2.0B specification on a high quality miniPCI (type IIIA) hardware in industrial temperature range. Integrated CAN Tranceivers with optional TTL-signals for galvanic isolation.

6.1 CAN connector pinout

The onboard female connector for the CAN signals connector is **Molex 53780-1470** and the male connector is **Molex 51146-1400** (for more information visit www.molex.com).

Pin	2 Channel	4 Channel
1	+3,3V	+3,3V
2	CAN1 + / CAN1 TD	CAN1 + / CAN1 TD
3	CAN1 - / CAN1 RD	CAN1 - / CAN1 RD
4	CAN1 IO or CAN1 ERR	CAN1 IO or CAN1 ERR
5	Ground	Ground
6	CAN2 + / CAN2 TD	CAN2 + / CAN2 TD
7	CAN2 - / CAN2 RD	CAN2 - / CAN2 RD
8	CAN2 IO or CAN2 ERR	CAN2 IO or CAN2 ERR
9	Ground	Ground
10	reserved, do not use	CAN3 + / CAN3 TD
11	reserved, do not use	CAN3 - / CAN3 RD
12	reserved, do not use	CAN4 + / CAN4 TD
13	reserved, do not use	CAN4 - / CAN4 RD
14	reserved, do not use	Ground

6.2 Fault Tolerant CAN (optional)

The Board is provided by default with a CAN-transceiver on each canal and the pins 2/3 and 6/7 provide the signals CANx+/CANx-. The board can be optionally delivered without the CAN transceivers and the TTL level CAN TD and CAN RD signals lead to the pins 2/3 and 6/7 of the onboard connector thus enabling the implementation of external galvanic isolation.

There are two IO-pins (one for each CAN node), which can be used as error lines for Fault-Tolerant CAN or for other customer specific purposes.

7 Firmware

7.1 overview

Firmware consists of two parts; the bootloader (FW1) and the actual firmware (FW2). FW1 is permanent and can be updated only with special hardware. FW2, however, can be updated without any risk of damaging the boot loader. The Processor on the board communicates with the host PC via 8Kbytes of Dual Ported Memory (DPM).

7.2 Onboard leds

7.2.1 Normal use

When the board firmware is running the application firmware (FW2), the **green leds** indicate traffic on the CAN bus. When a telegram is sent or received the led is turned on for a short time (~20ms).

The **red led** is turned on for a short time when a buffer overflow occurred. The red led is also turned on if the node is in error mode (error passive or Bus-Off).

With firmware version 1162 and above the **red led** also indicates a line error (*Fault tolerant CAN only*). If line error is detected the led blinks shortly two with ~1s interval.

7.2.2 Firmware exception

In case the firmware run into an exception both of the red leds are turned on and off at constant interval (~200ms). In case of an exception, the state of the green leds is undefined. However, if both of the green leds are on – it is likely that the failure happened in the bootloader (FW1) during firmware update or after unsuccessful fw update (i.e. no valid FW2 found).

7.2.3 Updating firmware

You can update the boards firmware with the fwupdate script in the driver sources. Here is an example:

```
./fwupdate /dev/can0 hcanpci-fw2-v1135en-edi1823.bin
```

In some occasions, if the firmware is corrupted or it doesn’t exist, you need to install the kernel module in fw update mode first:

```
cd linux/driver
make install FW_UPDATE=1
```

8 Support

At www.support.emtrion.de you can download the latest version of the driver, firmware and this manual. After purchasing a HiCO.CAN product from emtrion you can register at the site and we will give you the needed access rights.