

Linux 2.6 driver for HiCO.CAN boards

PCI, CompactPCI and PC104

emtrion

© Copyright 2006 **emtrion GmbH**

All rights reserved. This documentation may not be photocopied or recorded on any electronic media without written approval. The information contained in this documentation is subject to change without prior notice. We assume no liability for erroneous information or its consequences. Trademarks used from other companies refer exclusively to the products of those companies.

Date 22/04/2008 10:15

- 1 INTRODUCTION 4**
- 2 FIRST STEPS WITH THE DRIVER..... 5**
 - 2.1 BUILDING AND INSTALLING 5
 - 2.2 STARTING THE EXAMPLE APPLICATION 7
 - 2.3 GETTING DRIVER STATUS INFORMATION..... 8
- 3 DRIVER USAGE..... 9**
 - 3.1 DEVICE FILES 9
 - 3.2 CAN TELEGRAM STRUCTURE..... 10
 - 3.3 BLOCKING AND NON-BLOCKING OPERATIONS 11
 - 3.4 USING THE DRIVER WITH POLL() AND SELECT()..... 12
 - 3.5 IO SIGNALS (ASYNCHRONOUS NOTIFICATION)..... 13
 - 3.6 SETTING CAN PARAMETERS 13
 - 3.7 MULTIPROCESSING 14
- 4 DRIVER API..... 15**
 - 4.1 STANDARD SYSTEM CALLS..... 15
 - 4.2 IOCTL CALLS 17
 - 4.3 EXAMPLE APPLICATION 23
- 5 HARDWARE OVERVIEW..... 25**
 - 5.1 ERROR MESSAGES DURING OPERATION (LEDS)..... 25
 - 5.2 SETTING THE RESOURCES (PC/104 VARIANTS) 26
 - 5.3 BOARD NUMBER (X4)..... 26

1 Introduction

Due the differences between Windows and Unix OSs, neither the drivers API nor its function correspond for large parts to those of the windows drivers. The API for the driver is, like for most of the Linux drivers, the Linux file-system (POSIX) API. Thus, if you already have experience of Linux device drivers or inter-process communication with pipes or sockets you will probably feel yourself comfortable with the API.

The HiCOCAN-PCI/ISA – Linux 2.6 driver is implemented as a Loadable Kernel Module (LKM) which is accessed via device file nodes in the /dev directory (for example /dev/can1). CAN telegrams are written and read using a set of standard Linux system calls (open(), read(), ioctl() ...). The driver is multiprocessing capable (i.e. multiple processes can open it for use) and able to handle four HiCOCAN boards simultaneously.

2 First steps with the driver

2.1 Building and installing

1. Install Linux kernel sources

In order to build a kernel module you need the kernel source tree (kernel header will do as well) installed and configured for your running kernel. On a debian based system you would give command:

```
sudo apt-get install linux-headers-`uname -r`
```

On a SuSE System you would use the *YaST package manager*. With command 'uname -r' you can get the exact version of your running kernel. After installing - ensure that following command lists you the kernel source tree root:

```
ls /lib/modules/`uname -r`/build
```

2. Build the module

If you are not using the default location for the Linux kernel sources, you can specify it with KDIR variable (see Makefile). Change to the driver directory and build the module with make

```
cd driver
sudo make
```

3. Install the kernel Module

Install `hicocan_pci/isa.o` with the `insmod` command. Example for the PCI module

```
sudo insmod hicocan_pci.ko
```

example for the ISA/PC104 module

```
sudo insmod hicocan_isa.ko base_mem=0xDA000 irq_number=5 \  
board_number=0
```

example for ISA/PC104 module when multiple hicocan boards on the bus:

```
sudo insmod hicocan_isa.ko base_addresses=0xDA000,0xDF00,0,0 \  
interrupt_numbers=5,7,0,0
```

Linux 2.6 driver for HiCO.CAN-PCI/CPCI/PC104 boards

The sequence you write the base addresses and interrupt numbers correspond to the board numbers. Value 0 means that the board is no used.

You should now be able to read the board status information from the drivers proc file entry `/proc/hicocan_[pci/isa]`. If the `insmod` failed, you can read from the log messages what could have been the probable cause with command `'dmesg | tail'`

Tip: you can use `'modinfo'` command to find out available parameters:
`sudo modinfo hicocan_[pci/isa].ko`

4. Create device nodes

Make device files for the CAN nodes on the board. Device nodes to be used are listed in the `/proc/hicocan` file. You can use following command to automatically create all required device nodes:

```
sed -ne 's/NODE:/sudo mknod/p' /proc/hicocan_pci > tmp.sh
sh tmp.sh
```

You also might want to change the file permissions for the device files so that user processes can access them:

```
sudo chown 666 /dev/hicocan*
```

The driver is now installed and ready to be used.

5. Test that the driver works

Connect the two CAN nodes with the delivered Y-Cable and start the test script in the driver directory:

```
$> ./test.sh
start with option '-v' if you want to see more output
mknod: `/dev/hicocan0': File exists
mknod: `/dev/hicocan1': File exists
testing with 10 kbit/s..ok
testing with 20 kbit/s..ok
testing with 50 kbit/s..ok
testing with 100 kbit/s..ok
testing with 120 kbit/s..ok
testing with 250 kbit/s..ok
testing with 500 kbit/s..ok
testing with 800 kbit/s..ok
testing with 1000 kbit/s..ok
```

2.2 Starting the example application

In example directory is a simple ‘ping-pong’ 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 ping pong application with the ‘`start.sh`’ script:

2.3 Getting driver status information

When the driver is loaded to the kernel, it creates a proc file `/proc/hicocan*`. From here you can read status information of the driver and the HiCOCAN board for troubleshooting. By reading the file you will get a "snapshot" status report on the screen:

```
major:254

Board number 0 running ok
DPM base (phys/mapped)    e4000000/cca10000
Cnfg base (phys/mapped)  e4008000/cc972000
Irq number                10, Board state regs    40 a5 00 02

CAN nod 0
Data overrun              ( )
Passive error             ( )
Buss off                  ( )
State register            0x34
Baud rate (kb/s)         100
Msgs in TraQ              0
Msgs in RecQ              0
Msgs in RecBuf            0

CAN nod 1
Data overrun              ( )
Passive error             ( )
Buss off                  ( )
State register            0x34
Baud rate (kb/s)         100
Msgs in TraQ              0
Msgs in RecQ              0
Msgs in RecBuf            0
```

3 Driver usage

When the driver is loaded to the kernel and the device files are created, a CAN node (i.e. the device files /dev/hicocan*) can be opened with the open() system call for use. 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 hicocan_fd;

//open CAN node 0 for reading and writing
hicocan_fd = open("/dev/hicocan0",O_RDWR);
if(hicocan_fd!=-1){
    //..deal with error..
}
.
.
// ** application code **
.
close(hicocan_fd)
```

3.1 Device files

The device files are named after the node numbers, which are further dependent of the jumpered board number (see [Hardware overview](#)). Following table shows the names and values:

BoardNo	NodeNo	Device file	Minor number
0	0	/dev/hicocan0	0
0	1	/dev/hicocan1	1
1	2	/dev/hicocan2	2
1	3	/dev/hicocan3	3
2	4	/dev/hicocan4	4
2	5	/dev/hicocan5	5
3	6	/dev/hicocan6	6
3	7	/dev/hicocan7	7

3.2 CAN telegram structure

The structure `sCanMsg` is type-defined in the header `hicocan.h`:

`uint8_t ff;`

Frame format. Normal frame with 11 bit identifier or extended frame with 29 bit identifier (`HiCOCAN_FORMAT_BASIC` or `HiCOCAN_FORMAT_EXTENDED`)

`uint8_t rtr;`

Remote transmission request or a normal frame (`HiCOCAN_REMOTE_FRAME` or `HiCOCAN_NORMAL_FRAME`)

`uint8_t dlc;`

Data length code 0-7 (i.e. how many of the eight data bytes contain data).

`uint32_t id;`

CAN id (the arbitration field). 11 bit in case of a normal frame and 29 bit in case of an extended frame.

`canTs_t ts;`

Time-stamp which indicates when the telegram was received (written by the HiCOCAN firmware). See `hicocan.h` for the definition. When you are sending telegrams you can ignore this.

3.3 *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) );  
.
```

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

The driver supports the poll() and select() system calls, so if you are using them, 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().

3.4.1 Implementing timeout with poll()

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 poll() system call which the driver supports (see man page of poll()). Following snippet gives an example how to implement this:

```
#define TIMEOUT 100 //100 milliseconds
struct pollfd pfd; //declare a pollfd structure
.
.
//initialise the pollfd structure and set it to wait
//data input (POLLIN event)
pfd.fd = hicocan_fd;
pfd.events = POLLIN;
pfd.revents = 0;
ret = poll( &pfd, 1, TIMEOUT );
if(ret==-1){
//..deal with the error..
}
else if(ret==0){
//.. the poll() call timed out..
}
else{
if(pfd.revents & POLLIN){
//..data available to be read..
}
.
.
.
```

3.5 IO Signals (Asynchronous Notification)

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:

```
.
fcntl( hicocan_fildes, F_SETOWN, getpid() );
//...and setting the FASYNC flag
fcntl( hicocan_fildes, F_SETFL,
      fcntl(fildes,F_GETFL) | FASYNC );
.
```

3.6 Setting CAN parameters

In the `hicocan.h` header file is type-defined structure `sCanParam`; which contains the available CAN parameters:

`uint8_t btr0, uint8_t btr1;`

bus timing registers of the SJA1000 CAN controller (Note: If you don't know how to set the timing registers, it's better to leave the alone and use only the baud parameter)

`uint8_t baud;`

CAN baudrate. Possible values are: `HiCOCAN_BAUD10K`, `HiCOCAN_BAUD20K`, `HiCOCAN_BAUD50K`, `HiCOCAN_BAUD100K`, `HiCOCAN_BAUD125K`, `HiCOCAN_BAUD250K`, `HiCOCAN_BAUD500K`, `HiCOCAN_BAUD800K`, `HiCOCAN_BAUD1M`
(See documentation for `IOC_SET_BAUDRATE` below)

`uint8_t accFm;`

Acceptance Filter Mode. Possible values:

`HiCOCAN_FILTERMODE_DUAL`,
`HiCOCAN_FILTERMODE_SINGLE`

(See documentation for `IOC_SET_ACCEPTANCE` below)

`uint8_t accCode;`

Acceptance Code

`Uitn8_t accMask;`

Acceptance Mask

The convention is that only the required parameters are written to the structure and then an corresponding ioctl() call is used to write them to the board (these calls take the parameter structure pointer as argument). Here's an example how the set the baud-rate:

```
.  
canparam.baud = HiCOCAN_BAUD1M;  
ret=ioctl(fildes, IOC_SET_BAUD, &canparam);  
if(ret== -1) perror("ioctl:IOC_SET_BAUD");  
.
```

3.7 Multiprocessing

The driver can be opened used by multiple processes 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_SET_TIMESTAMP, since these affect both CAN nodes on the board. It is recommended that one process is a "master", which, in case the board has to be reset or timestamp value altered, informs the other processes and then performs the actions required.

4 Driver API

4.1 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).

4.1.1 read()

reads one can message from the driver into the given buffer.
Example:

```
read(canFd, &canMsg , sizeof(canMsg))
```

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

4.1.2 write()

Writes one can message to CAN bus. Example:

```
write(canFd, &canMsg , sizeof(canMsg))
```

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

4.1.3 ioctl()

Give a specific command to the driver (ioctl calls are described in the hicocan.h API header file)

4.1.4 poll() and select()

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

4.1.5 open()

Opens a CAN node for use and returns a file descriptor which is the CAN nodes "handle". Example:

```
canFd=open("/dev/hicocan0",O_RDWR) .
```

see man page for more info.

4.1.6 close()

closes the can node.

Return Values

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

4.2 *ioctl* calls

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

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_STOP	
Description	Stop listening CAN traffic and disable transmit
Parameters	none

IOC_RESET BOARD	
Description	Reset the board and the firmware and reset the transmit and receive buffers. Note that this command affects both of the CAN nodes on the board
Parameters	none

IOC_GET_CAN_PARAM	
Description	Reads the current CAN parameters from the board.
Parameters	struct canParam *params pointer to a sCanParameter structure where the parameters are to be written.
Notes	The canParams is used by many of the ioctl calls. Here is the definition: <pre>struct canParam { uint8_t btr0; /* bus timing register 0 */ uint8_t btr1; /* bus timing register 1 */ uint8_t baud; /* CAN baudrate / 10 */ uint8_t accFm; /* acceptance filter mode */ uint32_t accCode; /* acceptance code */ uint32_t accMask; /* acceptance mask */ };</pre>

IOC_GET_CAN_STATE	
Description	Reads the current status information of the CAN node.
Parameters	struct canState *state pointer to a canState structure where the status information is to be written
Notes	<pre> struct canState { uint8_t state; /* CAN node state */ uint8_t rxErr; /* receive error counter */ uint8_t txErr; /* transmit error counter */ uint16_t recBuf; /* messages in driver buffer */ uint16_t traQ; /* messages in boards Tx buf */ uint16_t recQ; /* messages in boards Rx buf */ }; </pre>
Notes	<p>Here are the flag definitions for the state variable:</p> <p>C_ERR_OVERR Overrun has occurred</p> <p>C_ERR_PSV Tx or Rx error counter has reached its limit and CAN node is in "Error Passive" state. No messages are transmitted.</p> <p>C_BUSOFF Bord is in "Bus Off" state due to Rx/Tx errors. No Messages are transmitted ot received.</p> <p>The other flags in the state variable do not have any meaning for applications.</p>

IOC_SET_BAUD	
Description	Set the CAN nodes baud rate.
Parameters	<pre>struct canParam *param</pre> Pointer to the CAN parameter structure which contains the new baudrate. Valid values are: HiCOCAN_BAUD10K HiCOCAN_BAUD20K HiCOCAN_BAUD50K HiCOCAN_BAUD100K HiCOCAN_BAUD125K HiCOCAN_BAUD250K HiCOCAN_BAUD500K HiCOCAN_BAUD800K HiCOCAN_BAUD1M
Example	<pre>struct canParam param; param.baud= HiCOCAN_BAUD500K ioctl(can_fd,IOC_SET_BAUD,&param);</pre>

IOC_SET_ACCEPTANCE	
Description	Set the acceptance mask and code. By default all messages are let through.
Parameters	struct canParam *param pointer to the canParama structure which contains the new acceptance mask and code.
Example	<pre>wanted_id=0xaaaa; mask=0x0; // All bits are of importance // (only one telegram will pass) canparam.accMask = mask<<3&(1<<2); //rtr ignored canparam.accCode = wanted_id<<3; canparam.accFm = HiCOCAN_FILTERMODE_SINGLE; ret=ioctl(fildes, IOC_SET_ACCEPTANCE, &canparam);</pre>
Notes	FILTERMODE_SINGLE is a 'normal' filtering mode. In FILTERMODE_DUAL - the two first databytes of a CAN telegram are also used in filtering (for standard frame messages). See datasheet for SJA1000 for more information on the dual mode. We recommend that you only use just the single mode

IOC_PASSIVE	
Description	Set the CAN node in passive mode. CAN traffic is listened and received it is however not affected in any way.
Parameters	none

IOC_SET_TIMING_REGS	
Description	Sets the timing registers btr0 and btr1 on the SJA1000 can controller. Normally you don't need to use this ioctl call. In some cases it might be necessary to tweak the timing parameters, if there are lots of errors on the CAN bus due to timing incompatibilities between different CAN nodes.
Parameters	struct canParam *param Pointer to the canParam structure where which contains the btr0 and btr1 values.

IOC_GET_LINE_ERR	
Description	Get the CAN line error status (Fault Tolerant CAN only)
Parameters	uint8_t *status Pointer to a variable where the status is to be saved. 1=Line error detected, 0=ok

IOC_CLEAR_OVERRUN	
Description	Clear the nodes overrun flag (in canState->state)
Parameters	none

4.3 Example Application

The example/pingong.c will give you an full example how to use the driver API. Here is an example how to send one message:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <err.h>
#include "hicocan.h"

int main(int argc, char *argv[])
{
    //Structure for the CAN parameters (see hicocan.h).
    struct canParam canparam;

    //structure for the CAN message (see hicocan.h).
    struct canMsg canMsg;

    // baudrate from hicocan.h
    int baud=HiCOCAN_BAUD20K;

    //file descriptors
    int canfd;

    // Open a can node for reading and writing
    canfd = open("/dev/hicocan0", O_RDWR);
    if (canfd < 0) err(1, "open");

    // Set baudrate
    canparam.baud = (uint8_t)baud;
    ret = ioctl(canfd, IOC_SET_BAUD, &canparam);
    if (ret == -1)
        err(1, "IOC_SET_BAUD");

    //start the CAN node
    if (ioctl(canfd, IOC_START))
        err(1, "IOC_START");

    // fill a CAN telegram
    canMsg.ff = HiCOCAN_FORMAT_BASIC;
    canMsg.rtr = 0;
    canMsg.id = 0;
    canMsg.dlc = 1;
    canMsg.data[0] = 1;
```

```
// and write it to CAN bus
ret = write(canfd, &canMsg, sizeof(struct canMsg));
if (ret == -1)
    err(1, "write");

//stop and close the CAN node when leaving
ioctl(canfd, IOC_STOP);
close(canfd);
}
```

5 Hardware overview

NOTE

For a more detailed hardware description please see the HiCO.CAN Manual for Windows. Here are only the most important features.

5.1 Error Messages During Operation (leds)

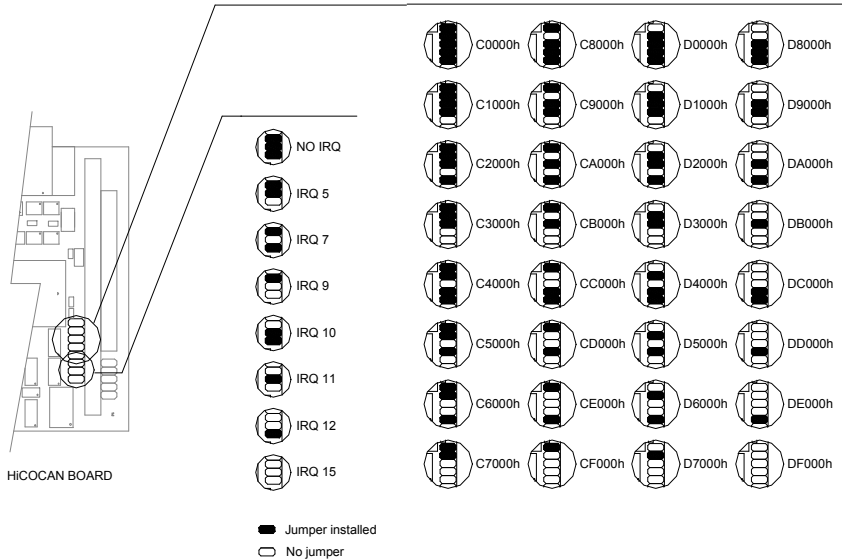
The bootstrap loader or the firmware writes the information on the errors occurred into the status cells in the board's communication area. This information can then be requested via specific function calls or is yielded in the form of return values of a function.

In addition, LEDs will go on or off, depending on the type of error. Table below lists the corresponding error states.

Red LED	Yellow LED	Significance
On	Off	Hardware error or bootstrap loader faulty
Is flashing	Off	Checksum error in bootstrap loader area
Is flashing	On	Checksum error in firmware area
Off	Is flashing	Missing or faulty configuration data
Off	On	No error has occurred
On	On	Firmware has detected an overload

5.2 Setting the Resources (PC/104 variants)

The following figure shows the settings of the resources (addresses, interrupts) on the board. This can only be done for HiCOCAN-104-2H, HiCOCAN-104-2L of the HiCOCAN boards.



5.3 Board Number (X4)

Jumper	open	closed	Jumper	open	closed	Board No.
1-3	x		2-4	x		0
1-3		x	2-4	x		1
1-3	X		2-4		x	2
1-3		x	2-4		x	3