TIP810-SW-72
LynxOS Device Driver
TIP810 – PeliCAN Controller

Version 1.0

Reference Manual
Issue 1.0

October 22, 2001

TEWS TECHNOLOGIES GmbH
Am Bahnhof 7
D-25469 Halstenbek
Germany
Tel.: +49 (0)4101 4058-0
Fax.: +49 (0)4101 4058-19
http://www.tews.com
e-mail: info@tews.com

TIP810-SW-72
PeliCAN Controller
LynxOS Device Driver

Issue	Description	Date
1.0	First Issue	October 22, 2001

Table of Contents

1 Introduction

The TIP810-SW-72 LynxOS device driver allows the operation of a TIP810 PeliCAN Controller IP on PowerPC platforms.
The standard file (I/O) functions (open, close, read, write and ioctl) provide the basic interface for opening and closing a file descriptor and for performing device I/O and control operations.

The TIP810 device driver includes the following functions:

- ✓ Transmission and receive of Standard and Extended Identifiers
- ✓ Standard bit rates from 5 kbit up to 1 Mbit and user defined bit rates
- ✓ Message acceptance filtering
- ✓ Single-Shot transmission
- ✓ Listen only mode
- ✓ Message self reception
- ✓ Programmable error warning limit

To understand all features of this device driver, it is very important to read the functional description of the SJA1000 PeliCAN controller, which is part of the engineering kit TIP810-EK.

2 Installation

The software is delivered on a PC formatted 3½" HD diskette.

Following files are located on the diskette:

tip810.c	Driver source code
tip810.h	Definitions and data structures for driver and application
tip810def.h	Definitions and data structures for the driver
sja1000.h	SJA1000 CAN controller programming model
tip810_info.c	Device information definition
tip810_info.h	Device information definition header
tip810.cfg	Driver configuration file include
tip810.import	Linker import file
Makefile	Device driver make file
Makefile.dldd	Make file for dynamic driver installation
example/example.c	Example application source
tip810-sw-72.pdf	This Manual in PDF format

2.1 Device Driver Installation

The two methods of driver installation are as follows:

- Static Installation
- Dynamic Installation (only native LynxOS systems)

2.1.1 Static Installation

With this method, the driver object code is linked with the kernel routines and is installed during system start-up.

In order to perform a static installation, copy the following files to their target directories:

1. Create a new directory in the system drivers directory path */sys/drivers.xxx*, where *xxx* represents the BSP that supports the target hardware.
   For example: */sys/drivers.pp_drm/tip810*

2. Copy the following files to this directory:
   tip810.c, tip810def.h, sja1000.h, Makefile

3. Copy *tip810.h* to */usr/include/*

4. Copy *tip810_info.c* to */sys/devices.xxx/* or */sys/devices* if */sys/devices.xxx* does not exist (xxx represents the BSP).

5. Copy *tip810_info.h* to */sys/dheaders/*

6. Copy *tip810.cfg* to */sys/cfg.ppc/*

2.1.1.1 Build the driver object

1. Change to the directory */sys/drivers.xxx/tip810,* where *xxx* represents the BSP that supports the target hardware.

2. To update the library */sys/lib/libdrivers.a* enter:

```
make install
```

2.1.1.2 Create Device Information Declaration

1. Change to the directory */sys/devices.xxx* or */sys/devices* if */sys/devices.xxx* does not exist (xxx represents the BSP).

2. Add the following dependencies to the *Makefile*

```
DEVICE_FILES_all = ... tip810_info.x
```

And at the end of the Makefile

```
...
tip810_info.o:$(DHEADERS)/tip810_info.h
```

3. To update the library */sys/lib/libdevices.a* enter:

```
make install
```

2.1.1.3 Modify the Device and Driver Configuration File

In order to insert the driver object code into the kernel image, an appropriate entry in file CONFIG.TBL must be created.

1. Change to the directory */sys/lynx.os/* respective */sys/bsp.xxx,* where *xxx* represents the BSP that supports the target hardware.

2. Create an entry at the end of the file CONFIG.TBL

```
I:tip810.cfg
```

2.1.1.4 Rebuild the Kernel

1. Change to the directory */sys/lynx.os/* (*/sys/bsp.xxx*)

2. Enter the following command to rebuild the kernel:

```
make install
```

3. Reboot the newly created operating system by the following command (not necessary for KDIs):

```
reboot –aN
```

The **N** flag instructs *init* to run *mknod* and create all the nodes mentioned in the new *nodetab*.

4. After reboot you should find the following new devices (depends on the device configuration): */dev/t810a1, [/dev/t810b1, …]*

2.1.2 Dynamic Installation

This method allows you to install the driver after the operating system is booted. The driver object code is attached to the end of the kernel image and the operating system dynamically adds this driver to its internal structures. The driver can also be removed dynamically.

The following steps describe how to do a dynamic installation:

1. Create a new directory in the system drivers directory path */sys/drivers.xxx*, where *xxx* represents the BSP that supports the target hardware.
   For example: */sys/drivers.pp_drm/tip810*

2. Copy the following files to this directory:
   - tip810.c
   - tip810def.h
   - sja1000.h
   - tip810_info.c
   - tip810_info.h
   - tip810.import
   - Makefile.dldd

3. Copy *tip810.h* to */usr/include*

4. Change to the directory */sys/drivers.xxx/tip810*

5. To make the dynamic link-able driver enter :

   ```
   make –f Makefile.dldd
   ```

6. Create a device definition file for one major device

   ```
   gcc –DDLDD –o tip810_info tip810_info.c
   ./tip810_info > t810a_info
   ```

7. To install the driver enter:

   ```
   drinstall –c tip810.obj
   ```

   If successful *drinstall* returns a unique <*driver-ID*>

8. To install the major device enter:

   ```
   devinstall –c –d <driver-ID> t810a_info
   ```

   The <*driver-ID*> is returned by the *drinstall* command

9. To create nodes for the devices enter:

   ```
   mknod /dev/t810a1 c <major_no> 0
   ...
   ```

If all steps are successful completed the TIP810 is ready to use.

To uninstall the TIP810 device enter the following commands:

```
devinstall –u –c <device-ID>
drinstall –u <driver-ID>
```

2.1.3 Device Information Definition File

The device information definition contains information necessary to install the TIP810 major device.
The implementation of the device information definition is done through a C structure, which is defined in the header file *tip810_info.h*.

This structure contains the following parameter:

IpIoVirtualAddress Contains the kernel virtual address of the IP I/O space (CAN controller registers). This address depends on the configuration of the IP carrier board. In case of a VMEbus carrier this space usually appears in the VMEbus short I/O space A16/D16.

IpIdVirtualAddress Contains the kernel virtual address of the IP ID space (ID-PROM). This address depends on the configuration of the IP carrier board. In case of a VMEbus carrier this space usually appears in the VMEbus short I/O space A16/D16.

IpInterruptVector Contains the vector at which the TIP810 generate interrupts. If the TIP810 is plugged on a VMEbus carrier any free vector from 64 to 255 can be used. The drivers setup the vector register in the TIP810 with this vector during driver initialization.

Note

If the TIP810 is plugged on a VMEbus carrier be sure that the appropriate VMEbus driver *uvme* or *vme* is started (CONFIG.TBL). See also *Chapter 5 – Accessing Hardware* in the "Writing Device Drivers for LynxOS" manual and the man pages *uvmedrvr* and *vmedrvr* for information about the VMEbus configuration and mapping.

A device information definition is unique for every TIP810 major device. The file *tip810_info.c* on the distribution disk contains two device information declarations, **t810a_info** for the first major device and **t810b_info** for the second major device.

If the driver should support more than two major devices it is necessary to copy and paste an existing declaration and rename it with unique name for example **t810c_info**, **t810d_info** and so on.

NOTE. It is also necessary to modify the device and driver configuration file respectively the configuration include file *tip810.cfg*.

The following device declaration information expected that the IP spaces appear at virtual address 0xCFFF8000 for IP I/O and at virtual address 0xCFFF8080 for IP ID space. The interrupt vector 64 is the first usable vector on the VMEbus.

```
T810_INFO t810a_info = {

    0xCfff8000,                 /*  IP I/O Space     */
    0xCfff8080,                 /*  IP ID Space      */
    64                          /*  Interrupt Vector */
};
```

2.1.4 Configuration File: CONFIG.TBL

The device and driver configuration file *CONFIG.TBL* contains entries for device drivers and its major and minor device declarations. Each time the system is rebuild, the *config* utility reads this file and produces a new set of driver and device configuration tables and a corresponding *nodetab*.

To install the TIP810 driver and devices into the LynxOS system, the configuration include file *tip810.cfg* must be included in the *CONFIG.TBL* (see also 2.1.1.3).
The file *tip810.cfg* on the distribution disk contains the driver entry (C:tip810:\....) and one enabled major device entry (D:TIP810 1:t810a_info::) with one minor device entry (N: t810a1:0).

If the driver should support more than one major device (TIP810) the following entries for major and minor devices must be enabled by removing the comment character (#). By copy and paste an existing major and minor entry and renaming the new entries, it is possible to add any number of additional TIP810 devices.

NOTE. The name of the device information declaration (info-block-name) must match to an existing C structure in the file *tip810_info.c*.

This example shows a driver entry with one major device and 6 minor devices:

```
#Format:
#C:driver-name:open:close:read:write:select:control:install:uninstall
#D:device-name:info-block-name:raw-partner-name
#N:node-name:minor-dev

C:tip810:\
  :t810open:t810close:t810read:t810write:\
  ::t810ioctl:t810install:t810uninstall
D:TIP810 1:t810a_info::
N:t810a1:0
D:TIP810 2:t810b_info::
N:t810b1:0
```

The configuration above creates the following node in the */dev* directory.

```
/dev/t810a1
/dev/t810b1
```

2.2 Receive Queue Configuration

Received CAN messages will be stored in a FIFO buffer. The depth of the FIFO can be adapted by changing the following symbols in *tip810def.h*.

T810_RX_FIFO_SIZE Defines the depth of the message FIFO buffer (default = 100). Valid numbers are in range between 1 and MAXINT.

3 TIP810 Device Driver Programming

LynxOS system calls are all available directly to any C program. They are implemented as ordinary function calls to "glue" routines in the system library, which trap to the OS code.
Note that many system calls use data structures, which should be obtained in a program from appropriate header files. Necessary header files are listed with the system call synopsis.

3.1 open()

NAME

open() - open a file

SYNOPSIS

```
#include <sys/file.h>
#include <sys/types.h>
#include <fcntl.h>

int open ( char *path, int oflags[, mode_t mode] )
```

DESCRIPTION

Opens a file (TIP810 device) named in ***path*** for reading and writing. The value of ***oflags*** indicates the intended use of the file. In case of a TIP810 devices ***oflags*** must be set to **O_RDWR** to open the file for both reading and writing.
The ***mode*** argument is required only when a file is created. Because a TIP810 device already exists this argument is ignored.

EXAMPLE

```
int  fd

/*
**  open the device named "/dev/t810a1" for I/O
*/

fd = open ("/dev/t810a1", O_RDWR);
```

RETURNS

open returns a file descriptor number if successful, or –1 on error.

SEE ALSO

LynxOS System Call - open()

3.2 close()

NAME

close() – close a file

SYNOPSIS

int close(int fd)

DESCRIPTION

This function closes an opened device.

EXAMPLE

```
int               result;


/*
**     close the device
*/

result = close(fd);
```

RETURNS

close returns 0 (OK) if successful, or –1 on error

SEE ALSO

LynxOS System Call - close()

3.3 read()

NAME

read() - read from a file

SYNOPSIS

#include <tip810.h>

int read (int fd, char *buff, int count)

DESCRIPTION

The read function reads a CAN message from the specified receive queue. A pointer to the callers message buffer (*T810_MSG_BUF*) and the size of this structure is passed by the parameters ***buff*** and ***count*** to the device.

The *T810_MSG_BUF* structure has the following layout:

```
typedef struct {

    unsigned long    Identifier;
    unsigned char    IOFlags;
    unsigned char    MsgLen;
    unsigned char    Data[8];
    long             Timeout;
    unsigned char    Satus;

} T810_MSG_BUF, *PT810_MSG_BUF;
```

Identifier
Receives the message identifier of the read CAN message.

IOFlags
Receives CAN message attributes as set of bit flags. The following attribute flags are possible.

T810_EXTENDED	Set if the received message is a extended message frame. Reset for standard message frames.
T810_REMOTE_FRAME	Set if the received message is a remote transmission request (RTR) frame.

MsgLen
Receives the number of message data bytes (0..8).

Data[8]
This buffer receives up to 8 data bytes. Data[0] receives message Data 0, Data[1] receives message Data 1 and so on.

Timeout
Specifies the amount of time (in ticks) the caller is willing to wait for execution of read.

Status

Receives status information about overrun conditions either in the CAN controller or intermediate software FIFO's.

T810_SUCCESS	No messages lost
T810_FIFO_OVERRUN	One or more messages was overwritten in the receive FIFO. This problem occurs if the FIFO is too small for the application read interval.
T810_MSGOBJ_OVERRUN	One or more messages were overwritten in the CAN controller message object because the interrupt latency is too large. Reduce the CAN bit rate or upgrade the system speed.

EXAMPLE

```
int              fd;
int              result;
T810_MSG_BUF     MsgBuf;


...


MsgBuf.Timeout = 200;

result = read(fd, (char*)&MsgBuf, sizeof(MsgBuf));


/*
**  Check the result of the last device I/O operation
*/

if( result == sizeof(T810_MSG_BUF)) {

  /* process received CAN message */

}
else {
  printf( "\nRead failed --> Error = %d.\n", errno );
}

...
```

RETURNS

When *read* succeeds, the size of the read buffer is returned. If read fails, -1 (SYSERR) is returned.

On error, *errno* will contain a standard read error code (see also LynxOS System Call – read) or one of the following TIP810 specific error codes:

ENXIO	Illegal device

EINVAL	Invalid argument. This error code is returned if either the size of the message buffer is too small, or the specified receive queue is out of range.
ETIMEDOUT	The maximum allowed time to finish the read request is exhausted.
ENETDOWN	The controller is in bus off state and no message is available in the specified receive queue. Note, as long as CAN messages are available in the receive queue FIFO, bus off conditions were not reported by a read function. This means you can read all CAN messages out of the receive queue FIFO during bus off state without an error result.

SEE ALSO

LynxOS System Call - read()
TIP810 example application

3.4 write()

NAME

write() – write to a file

SYNOPSIS

int write (int fd, char *buff, int count)

DESCRIPTION

The write function writes a CAN message to the device with descriptor ***fd***. A pointer to the callers message buffer (*T810_MSG_BUF*) and the size of this structure is passed by the parameters ***buff*** and ***count*** to the device.

The *T810_MSG_BUF* structure has the following layout:

```
typedef struct {

    unsigned long    Identifier;
    unsigned char    IOFlags;
    unsigned char    MsgLen;
    unsigned char    Data[8];
    long             Timeout;
    unsigned char    Satus;

} T810_MSG_BUF, *PT810_MSG_BUF;
```

Identifier
Contains the message identifier of the CAN message to write.

IOFlags
Contains a set of bit flags, which defines message attributes and controls the write operation. To set more that one bit flag the predefined macros must be binary OR'ed.

T810_EXTENDED	Transmit an extended message frame. If this macro isn't set or the "dummy" macro T810_STANDARD is set a standard frame will be transmitted.
T810_REMOTE_FRAME	A remote transmission request (RTR bit is set) will be transmitted.
T810_SINGLE_SHOT	No re-transmission will be performed if an error occurred or the arbitration will be lost during transmission (single-shot transmission).
T810_SELF_RECEPTION	The message will be transmitted and simultaneously received if the acceptance filter is set to the corresponding identifier.

MsgLen
Contains the number of message data bytes (0..8).

Data[8]
This buffer contains up to 8 data bytes. Data[0] contains message Data 0, Data[1] contains message Data 1 and so on.

Timeout
Specifies the amount of time (in ticks) the caller is willing to wait for execution of write.

Status
Unused for this control function. Can be 0.

EXAMPLE

```
int              fd;
int              result;
T810_MSG_BUF     MsgBuf;

...

/*
**     Write two data bytes with extended identifier 1234 to
**     the CANbus and wait max. 200 ticks on execution.
**     The transmitted frame will be received simultaneously.
*/
MsgBuf.Identifier = 1234;
MsgBuf.Timeout = 200;
MsgBuf.IOFlags = T810_EXTENDED | T810_SELF_RECEPTION;
MsgBuf.MsgLen = 2;
MsgBuf.Data[0] = 0xaa;
MsgBuf.Data[1] = 0x55;

result = write(fd, &MsgBuf, sizeof(MsgBuf));

if( result != sizeof(T810_MSG_BUF)) {
  printf( "\nWrite failed --> Error = %d.\n", errno );
}
```

RETURNS

When *write* succeeds, the size of the write buffer is returned. If write fails, -1 (SYSERR) is returned.

On error, *errno* will contain a standard write error code (see also LynxOS System Call – write) or the following TIP810 specific error code:

ENXIO	Illegal device
EINVAL	Invalid argument. This error code is returned if the size of the message buffer is too small or the message length is greater than 8 bytes.
ENETDOWN	The controller is in bus off state and unable to transmit messages.
ETIMEDOUT	The allowed time to finish the write request is elapsed. This occur if the CAN bus is overloaded and the priority of the message identifier is too low, no other node is online or the controller enters the BusOff state.

SEE ALSO

LynxOS System Call - write()
TIP810 example application

3.5 ioctl()

NAME

ioctl() - I/O device control

SYNOPSIS

```
#include <ioctl.h>
#include <tip810.h>

int ioctl ( int fd, int request, char *arg )
```

DESCRIPTION

ioctl provides a way of sending special commands to a device driver. The call sends the value of ***request*** and the pointer ***arg*** to the device associated with the descriptor ***fd***.

The following ioctl codes are defined in *TIP810.h* :

Value	**Meaning**
T810_BITTIMING	Setup a new bit timing
T810_SETFILTER	Setup acceptance filter
T810_BUSON	Enter the bus on state
T810_BUSOFF	Enter the bus off state
T810_FLUSH	Flush one or all receive queues
T810_CANSTATUS	Returns CAN controller status information
T810_ENABLE_SELFTEST	Enable self test mode
T810_DISABLE_SELFTEST	Disable self test mode
T810_ENABLE_LISTENONLY	Enable listen only mode
T810_DISABLE_LISTENONLY	Disable listen only mode
T810_SETLIMIT	Set new error warning limit

See behind for more detailed information on each control code.

RETURNS

ioctl returns 0 if successful, or –1 on error.
The TIP810 ioctl function returns always standard error codes. See LynxOS system call ioctl of a detailed description of possible error codes.

SEE ALSO

LynxOS System Call - ioctl().

3.5.1 T810_BITTIMING

NAME

T810_BITTIMING - Setup new bit timing

DESCRIPTION

This ioctl function modifies the bit timing register of the CAN controller to setup a new CAN bus transfer speed. A pointer to the caller's parameter buffer (*T810_TIMING*) is passed by the argument pointer ***arg*** to the driver.

Keep in mind to setup a valid bit timing value before changing into the Bus On state.

The *T810_TIMING* structure has the following layout:

```
typedef struct {

    unsigned short    TimingValue;
    unsigned short    TreeSamples;

} T810_TIMING, *PT810_TIMING;
```

Timing Value
This parameter holds the new values for the bit timing register 0 (bit 0..7) and for the bit timing register 1 (bit 8..15). Possible transfer rates are between 5 Kbit per second and 1 Mbit per second. The include file 'tip810.h' contains predefined transfer rate symbols (T810_5KBIT .. T810_1MBIT).
For other transfer rates please follow the instructions of the *SJA1000 Product Specification*, which is also part of the engineering kit TIP810-EK.

ThreeSamples
If this parameter is TRUE (1) the CAN bus is sampled three times per bit time instead of one.

Note

Use one sample point for faster bit rates and three sample points for slower bit rate to make the CAN bus more immune against noise spikes.

EXAMPLE

```
{

 int fd;
 int result;
 T810_TIMING  BitTimingParam;

 . . .

 BitTimingParam.TimingValue = T810_100KBIT;
 BitTimingParam.ThreeSamples = FALSE;

 result = ioctl(fd, T810_TIMING, (char*)&BitTimingParam);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

SEE ALSO

tip810.h for predefined bus timing constants
SJA1000 Product Specification Manual – *6.5.1/2 BUS TIMING REGISTER*

3.5.2 T810_SETFILTER

NAME

T810_SETFILTER - Setup acceptance filter

DESCRIPTION

This ioctl function modifies the acceptance filter of the specified CAN controller device.

The acceptance filter compares the received identifier with the acceptance filter and decides whether a message should be accepted or not. If a message passes the acceptance filter it is stored in the RXFIFO.
The acceptance filter is defined by the acceptance code registers and the acceptance mask registers. The bit pattern of messages to be received are defined in the acceptance code register.
The corresponding acceptance mask registers allow to define certain bit positions to be "don't care" (a 1 at a bit position means "don't care").

A pointer to the caller's parameter buffer (*T810_FILTER*) is passed by the parameter pointer ***arg*** to the driver.

The *T810_FILTER* structure has the following layout:

```
typedef struct {

   int              SingleFilter;
   unsigned long    AcceptanceCode;
   unsigned long    AcceptanceMask;

 } T810_FILTER, *PT810_FILTER;
```

SingleFilter
Set TRUE (1) for single filter mode.
Set FALSE (0) for dual filter mode.

AcceptanceCode
The contents of this parameter will be written to acceptance code register of the controller.

AcceptanceMask
The contents of this parameter will be written to the acceptance mask register of the controller.

Note

A detailed description of the acceptance filter and possible filter modes can be found in the SJA1000 Product Specification Manual.

EXAMPLE

```
{

 int fd;
 int result;
 T810_FILTER AcceptFilter;

 . . .

 /* Not relevant because all bits are "don't care" */
 AcceptFilter.AcceptanceCode = 0x0;

 /* Mark all bit position don't care */
 AcceptFilter.AcceptanceMask = 0xffffffff;

 /* Single Filter Mode */
 AcceptFilter.SingleFilter = 1;  // TRUE


 result = ioctl(fd, T810_SETFILTER, (char*)&AcceptFilter);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

SEE ALSO

SJA1000 Product Specification Manual – *6.4.15 ACCEPTANCE FILTER*

T810_BUSON

NAME

T810_BUSON - Enter the bus on state

DESCRIPTION

This ioctl function sets the specified CAN controller into the Bus On state.

After an abnormal rate of occurrences of errors on the CAN bus or after driver startup, the CAN controller enters the Bus Off state. This control function resets the "reset mode" bit in the mode register. The CAN controller begins the busoff recovery sequence and resets the transmit and receive error counters. If the CAN controller counts 128 packets of 11 consecutive recessive bits on the CAN bus, the Bus Off state is exited.
The optional argument pointer can be NULL.

Note

Before the driver is able to communicate over the CAN bus
after driver startup, this control function must be executed.

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 result = ioctl(fd, T810_BUSON, NULL);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

ERRORS

ENETDOWN	Unable to enter the BUSON mode.

SEE ALSO

SJA1000 Product Specification Manual – *6.4.3 MODE REGISTER (MOD)*

3.5.3 T810_BUSOFF

NAME

T810_BUSOFF - Enter the bus off state

DESCRIPTION

This ioctl function sets the specified CAN controller into the Bus Off state.

After execution of this control function the CAN controller is completely removed from the CAN bus and cannot communicate until the control function T810_BUSON is executed.
The optional argument pointer can be NULL.

Note

Execute this control function before the last close to the CAN controller channel.

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 result = ioctl(fd, T810_BUSOFF, NULL);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

ERRORS

EIO	Unable to enter the BUSOFF mode.

SEE ALSO

SJA1000 Product Specification Manual – *6.4.3 MODE REGISTER (MOD)*

3.5.4 T810_FLUSH

NAME

T810_FLUSH - Flush the received message FIFO

DESCRIPTION

This ioctl function flushes the FIFO buffer of received messages.

The optional argument pointer can be NULL.

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 result = ioctl(fd, T810_FLUSH, NULL);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

ERRORS

This ioctl function returns no function specific error codes.

T810_CANSTATUS

NAME

T810_CANSTATUS - Returns CAN controller status information

DESCRIPTION

This ioctl function returns the actual contents of several CAN controller registers for diagnostic purposes.
A pointer to the callers status buffer (*T810_STATUS*) and the size of this structure is passed by the parameters ***buff*** and ***count*** to the device.

The *T810_STATUS* structure has the following layout:

```
typedef struct {

    unsigned char   ArbitrationLostCapture;
    unsigned char   ErrorCodeCapture;
    unsigned char   TxErrorCounter;
    unsigned char   RxErrorCounter;
    unsigned char   ErrorWarningLimit;
    unsigned char   StatusRegister;
    unsigned char   ModeRegister;
    unsigned char   RxMessageCounterMax;

} T810_STATUS, *PT810_STATUS;
```

ArbitrationLostCapture
Contents of the arbitration lost capture register. This register contains information about the bit position of losing arbitration.

ErrorCodeCapture
Contents of the error code capture register. This register contains information about the type and location of errors on the bus.

TxErrorCounter
Contents of the TX error counter register. This register contains the current value of the transmit error counter.

RxErrorCounter
Contents of the TX error counter register. This register contains the current value of the receive error counter.

ErrorWarningLimit
Contents of the error warning limit register.

StatusRegister
Contents of the status register.

ModeRegister
Contents of the mode register.

RxMessageCounterMax
Contains the peak value of messages in the RXFIFO. This internal counter value will be reset to 0 after reading.

EXAMPLE

```
{

 int fd;
 int result;
 T810_STATUS CanStatus;

 . . .

 result = ioctl(fd, T810_CANSTATUS, (char*)&CanStatus);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

SEE ALSO

SJA1000 Product Specification Manual

3.5.5 T810_ENABLE_SELFTEST

NAME

T810_ENABLE_SELFTEST - Enable self test mode

DESCRIPTION

This ioctl function enables the self test facility of the SJA1000 CAN controller.

In this mode a full node test is possible without any other active node on the bus using the self reception facility (see also 3.4). The CAN controller will perform a successful transmission even if there is no acknowledge received.
Also in self test mode the normal functionality is given, that means the CAN controller is able to receive messages from other nodes and can transmit message to other nodes if any connected.

The optional argument pointer can be NULL.

Note

This ioctl command will be accepted only in reset mode (BUSOFF). Enter T810_BUSOFF first otherwise you will get an error (EACCES).

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 result = ioctl(fd, T810_ENABLE_SELFTEST, NULL);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

ERRORS

EACCES	The CAN controller is in operating mode. This mode can be changed only in reset mode.

SEE ALSO

SJA1000 Product Specification Manual – *6.4.3 MODE REGISTER (MOD)*

3.5.6 T810_DISABLE_SELFTEST

NAME

T810_DISABLE_SELFTEST - Disable self test mode

DESCRIPTION

This ioctl function disables the self test facility of the SJA1000 CAN controller, which was before enabled with the ioctl command T810_ENABLE_SELFTEST.

The optional argument pointer can be NULL.

Note

This ioctl command will be accepted only in reset mode (BUSOFF). Enter T810_BUSOFF first otherwise you will get an error (EACCES).

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 result = ioctl(fd, T810_DISABLE_SELFTEST, NULL);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

ERRORS

EACCES	The CAN controller is in operating mode. This mode can be changed only in reset mode.

SEE ALSO

SJA1000 Product Specification Manual – *6.4.3 MODE REGISTER (MOD)*

3.5.7 T810_ENABLE_LISTENONLY

NAME

T810_ENABLE_LISTENONLY - Enable listen only mode

DESCRIPTION

This ioctl function enables the listen only facility of the SJA1000 CAN controller.

In this mode the CAN controller would give no acknowledge to the CAN-bus, even if a message is received successfully. Message transmission is not possible. All other functions can be used like in normal mode.
This mode can be used for software driver bit rate detection and 'hot-plugging'.

The optional argument pointer can be NULL.

Note

This ioctl command will be accepted only in reset mode (BUSOFF). Enter T810_BUSOFF first otherwise you will get an error (EACCES).

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 result = ioctl(fd, T810_ENABLE_LISTENONLY, NULL);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

ERRORS

EACCES	The CAN controller is in operating mode. This mode can be changed only in reset mode.

SEE ALSO

SJA1000 Product Specification Manual – *6.4.3 MODE REGISTER (MOD)*

3.5.8 T810_DISABLE_ LISTENONLY

NAME

T810_DISABLE_ LISTENONLY - Disable listen only mode

DESCRIPTION

This ioctl function disables the listen only facility of the SJA1000 CAN controller, which was before enabled with the ioctl command T810_ENABLE_ LISTENONLY.

The optional argument pointer can be NULL.

Note

This ioctl command will be accepted only in reset mode (BUSOFF). Enter T810_BUSOFF first otherwise you will get an error (EACCES).

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 result = ioctl(fd, T810_DISABLE_LISTENONLY, NULL);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

ERRORS

EACCES	The CAN controller is in operating mode. This mode can be changed only in reset mode.

SEE ALSO

SJA1000 Product Specification Manual – *6.4.3 MODE REGISTER (MOD)*

3.5.9 T810_SETLIMIT

NAME

T810_SETLIMIT - Set new error warning limit

DESCRIPTION

This ioctl function sets a new error warning limit in the corresponding CAN controller register. The default value (after hardware reset) is 96.
The new error warning limit will be set in an unsigned char variable. A pointer to this variable is passed by the argument pointer ***arg*** to the driver.

Note

This ioctl command will be accepted only in reset mode (BUSOFF). Enter T810_BUSOFF first otherwise you will get an error (EACCES).

EXAMPLE

```
{

 int fd;
 int result;
 unsigned char limit;

 . . .

 limit = 200;

 result = ioctl(fd, T810_SETLIMIT, (char*)&limit);

 if (result < 0) {
   /* handle ioctl error */
 }

 . . .

}
```

ERRORS

EACCES	The CAN controller is in operating mode. This mode can be changed only in reset mode.

SEE ALSO

SJA1000 Product Specification Manual – *6.4.3 MODE REGISTER (MOD)*

3.6 Step by Step Driver Initialization

The following code example illustrates all necessary steps to initialize a CAN device for communication.

```
/*
**  ( 1.) Setup CAN bus bit timing
*/
BitTimingParam.TimingValue = T810_100KBIT;
BitTimingParam.ThreeSamples = FALSE;

result = ioctl(fd, T810_TIMING, (char*)&BitTimingParam);



/*
**  ( 2.) Setup acceptance filter masks
*/
AcceptFilter.AcceptanceCode = 0x0;
AcceptFilter.AcceptanceMask = 0xFFFFFFFF;
AcceptFilter.SingleFilter = 1;

result = ioctl(fd, T810_SETFILTER, (char*)&AcceptFilter);



/*
**  ( 3.) Enter Bus On State
*/

result = ioctl(fd, T810_BUSON, NULL);
```

Now you should be able to send and receive CAN messages with appropriate calls to write() and read() functions.

4 Debugging and Diagnostic

This driver was successful tested on a Motorola MVME3600-1 (PMCSPAN) and MVME2305-900 board in a native LynxOS environment and a Windows Cross development with LynxOS V3.0.1 and V3.1.0a installed.

If the driver will not work properly please enable debug outputs by removing the comments around the symbol *DEBUG.*
The debug output should appear on the console. If not please check the symbol *KKPF_PORT* in *uparam.h*. This symbol should be configured to a valid COM port (e.g. *SKDB_COM1*).

The debug output displays the device information data for the current major device, a memory dump of the IP ID space (contents of the ID-PROM) and a memory dump of the IP I/O space (CAN controller registers) like this.

```
TIP810 Device Driver Install
IP IO Virtual Address    = CFFF8100
IP ID  Virtual Address   = CFFF8180
Interrupt Vector         = 64

IP ID space (ID-PROM)...
CFFF8180 : FF 49 FF 50 FF 41 FF 43 FF B3 FF 01 FF 20 FF 00
CFFF8190 : FF 00 FF 00 FF 0C FF 9D FF 0A FF 00 FF 00 FF 00

IP IO space (CAN controller registers)...
CFFF8100 : FF 21 FF FF FF 0C FF E0 FF 00 FF 00 FF 00 FF 00
CFFF8110 : FF 00 FF 00 FF FF FF FF FF FF FF FF FF FF FF FF
CFFF8120 : FF FF FF FF FF FF FF FF FF FF FF FF FF FE FF FF
CFFF8130 : FF 00 FF 00 FF 00 FF 04 FF FF FF FF FF FF FF 00
```

Note

The debug output above is only an example. Debug output on other systems may be different for addresses and data in some locations.

If driver installation was successful but you can't send or receive messages you should check the wiring and termination of the CANbus lines.
If all seems to be correct but you can't communicate over the CANbus please call the ioctl function *T810_CANSTATUS* direct after the write function returned to get extended error information.