TIP816-SW-82
Linux Device Driver
TIP816 – Extended CAN

Version 1.0.x

Reference Manual
Issue 1.0

November 28, 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

TIP816-SW-82
Extended CAN
Linux Device Driver

Issue	Description	Date
1.0	First Issue	November 28, 2001

Table of Contents

1 Introduction

The TIP816-SW-82 Linux device driver allows the operation of a TIP816 Extended CAN IP on Linux operating systems with Intel and Intel-compatible x86 CPU.
The device driver was developed and tested on a Compact PCI system with Linux kernel Version 2.2.13 / 2.4.4 installed (SuSE Linux 6.3 / 7.2 Distribution).

Supported features:

- ✓ Transmission and receive of Standard and Extended Identifiers
- ✓ Up to 15 receive message queues with user defined size
- ✓ Variable allocation of receive message objects to receive queues
- ✓ Separate task queues for each receive queue and transmission buffer message object
- ✓ Standard bit rates from 5 kbit up to 1.6 Mbit and user defined bit rates
- ✓ Message acceptance filtering
- ✓ Definition of receive and remote buffer message objects
- ✓ Transmission and receive of Standard and Extended Identifiers
- ✓ Designed as Linux kernel module with dynamically loading (*insmod and rmmod*).
- ✓ Supports shared IRQ's.
- ✓ Creates devices with dynamically allocated or fixed major device numbers.

To understand all features of this device driver, it is very important to read the *Architectural Overview* of the Intel 82527 CAN controller, which is part of the engineering kit TIP816-EK.

2 Installation

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

Following files are located on the diskette:

tip816drv.c	Driver source code
tip816def.h	Driver include file
tip816.h	Driver include file for application program
I82527.h	Intel 82527 CAN controller definitions
load816	Script for driver loading
unload816	Script for driver unloading
makefile	Device driver make file
tip816-SW-82.pdf	This manual in PDF format
example/example.c	Example application
example/makefile	Example application makefile

In order to perform an installation, copy all files on the distribution diskette to the desired target directory.

2.1 Build and install the device driver

1. Login as *root*
2. Change to the target directory
3. To create and install the driver in the module directory */lib/modules/<version>* enter:

```
make install
```

Note

Some Linux distributions and kernel versions requires modification of the Makefile. If you got errors during compilation or installation of the driver please verify the macros *VER* and *INCLUDEDIR* in the Makefile.

2.2 Uninstall the device driver

1. Login as *root*
2. Change to the target directory
3. To remove the driver from the module directory */lib/modules/<version>* enter:

```
make uninstall
```

2.3 Install the device driver in the running kernel

To install the driver in the running kernel execute the shell script *load816*.

```
sh load816 mem=<ma1>,<ma2>,… id=<ia1>,<ia2>,… irq=<irq1>,<irq2>,…
```

This shell script removes a previously installed TIP816 driver, installs the new one into the kernel and creates nodes for up to eight TIP816 devices.

The arguments *mem, id* and *irq* defines the physical IP-MEM and IP-ID address respective IRQ of the TIP816 to initialize. The parameter *ma1, ia1, irq1* belongs to the first TIP816 the parameter *ma2, ia2, irq2* belongs to second device and so on. Up to 8 TIP816 devices can be initialized without modifying the driver source.

Example:

```
sh load816 mem=0xE9000000,0xE9800000 id=0xE8001100,0xE8002100 irq=9,9
```

The example above initializes the first TIP816 at physical address 0xE9000000 (IP-MEM space), 0xE8001100 (IP ID space) and IRQ 9 and a second TIP816 at physical address 0xE9800000, 0xE8002100 and IRQ 9.
Because both TIP816 plugged on the same IP carrier they share the same interrupt.

Created device nodes are:

```
/dev/tip816_0, /dev/tip816_1, ..., /dev/tip816_7
```

Note

The unmodified driver use dynamic allocation of major device numbers. To get the current used major number the script extracts the major number of the TIP816 driver from */proc/devices* to create the correct device nodes.

For example the TIP816 IP-MEM space and IP-ID space addresses and associated interrupts can be extracted from the Linux */proc* file system

The following dump shows a SBS PCI40 IP carrier board

```
# cat /proc/pci
  . . .
  Bus  0, device  10, function  0:
    Class 0680: PCI device 124b:0040 (rev 11).
      IRQ 9.
      Master Capable.  Latency=64.
      Non-prefetchable 32 bit memory at 0xf4002000 [0xf40020ff].
      I/O at 0xec00 [0xecff].
      Non-prefetchable 32 bit memory at 0xe8000000 [0xebffffff].
```

The TIP816 I/O registers are located in the second memory space (BAR2) at PCI bus address 0xe8000000. The first IP-MEM space appears at offset 0x100000 the IP-ID space at offset 0x1100 from the beginning. The second IP spaces appears at offset 0x180000 / 0x2100. Because the PCI bus address is equal to the physical address on i386 systems we got the following addresses for installation. The associated IRQ is also extracted from the dump.

```
sh load816 mem=0xE9000000,0xE9800000 id=0xE8001100,0xE8002100 irq=9,9
```

Note

On PowerPC systems the PCI bus address are different to the physical addresses used for driver installation. In this case you have to convert the bus address to a appropriate physical address. Please refer to related documentation from the CPU board supplier and Linux distribution.

The example above is only suitable for SBS PCI 40 IP carriers on a specific CPU board and PCI slot. If you use other IP carrier boards or the same carrier on other CPU boards (probably) or PCI slots, you have to adapt the memory mapping. Please refer to related documentation.
Keep in mind to adapt the IP addresses if the carrier board is plugged to an other PCI slot or the platform has changed.

For some IP carrier boards it also necessary to initialize some registers before the TIP816 device driver can access the IP or can use interrupts. This initialization can be done within the TIP816 device driver (e.g. function init_PCI40() in tip816drv.c) or in special initialization modules which are started before the TIP816 driver. Usually this code initializes board interrupts and enable PCI spaces. Please refer to related documentation which initializations are necessary.

An other important point is alignment of the TIP816 controller registers in the IP-ID space of the carrier board. By default the driver assumes that the registers appears at even addresses with a gap of on byte. If you are not sure about the alignment of your carrier board enable the memory dump of controller registers by defining the symbol *TIP816_DEBUG_VIEW* in tip816drv.c. The memory dump will appear in the xconsole window.

With the defines *TIPxxx_ODD_MAP, OFF_STEP* and in tip816def.h its possible to adapt the driver to the hardware requirements of the based system (see also Chapter 4 Diagnostic).

Note

According to the mechanism your system uses to deliver output from kernel modules the output appear on the xconsole window (SUSE distribution) or it may go to one of the system log files, such as */var/log/messages*.

The Big/Little Endian configuration can also cause trouble for accesses to the IP-MEM space of the TIP816. The TIP816 has a 16-bit IP interface. If the endian mode doesn't match, low and high byte will be swapped and accesses will got wrong registers.
If the endian mode can't be changed by an appropriate configuration of the carrier board, the driver can "change" the endian mode by itself depending on the definition of the macro **SWAP_BYTE_LANE** in tip816def.h. By default this macro is defined and byte lanes will be swapped. This is a suitable configuration for SBS PCI40 and cPCI100/200 carrier boards.
Chapter 4 – Diagnostic will give you hints how you can determine the current endian mode of your carrier board.

2.4 Remove the device driver in the running kernel

To remove the driver from the running kernel login as root and execute the shell script *unload816*.

2.5 Change Major Device Number

The TPCM816 driver use dynamic allocation of major device numbers per default. If this isn't suitable for the application it's possible to define a major number for the driver.

To change the major number edit the file tip816drv.c, change the following symbol to appropriate value and enter `make install` to create a new driver.

TIP816_MAJOR Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

```
Example:

#define TIP816_MAJOR       122
```

Note

Be sure that the desired major number isn't used by an other driver. Please check */proc/devices* to see which numbers are free.

Keep in mind that's necessary to create new device nodes if the major number for the TIP816 driver has changed and the load816 script isn't used.

2.6 Receive Queue Configuration

Received CAN messages will be stored in receive queues. Each receive queue contains a FIFO and a separate task wait queue. The number of receive queues and the depth of the FIFO can be adapted by changing the following symbols in *tip816drv.c*.

NUM_RX_QUEUES Defines the number of receive queues for each device (default = 3). Valid numbers are in range between 1 and 15.

RX_FIFO_SIZE Defines the depth of the message FIFO inside each receive queue (default = 100). Valid numbers are in range between 1 and MAXINT.

Note

Enter ***make install*** to create a new driver

3 Device Input/Output functions

This chapter describes the interface to the device driver I/O system used for communication over the CAN Bus.

3.1 open()

NAME

open() - open a file descriptor

SYNOPSIS

#include <fcntl.h>

int open (*const char *filename, int flags*)

DESCRIPTION

The open function creates and returns a new file descriptor for the file named by *filename*.
The *flags* argument controls how the file is to be opened. This is a bit mask; you create the value by the bitwise OR of the appropriate parameters (using the | operator in C).
See also the GNU C Library documentation for more information about the open function and open flags.

EXAMPLE

```
{
 int fd;

 fd = open(“/dev/tip816_0, O_RDWR);
}
```

RETURNS

The normal return value from open is a non-negative integer file descriptor. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

E_NODEV The requested minor device does not exist.

This is the only error code returned by the driver, other codes may be returned by the I/O system during open. For more information about open error codes, see the *GNU C Library description – Low-Level Input/Output*.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.2 close()

NAME

close() – close a file descriptor

SYNOPSIS

#include <unistd.h>

int **close** *(int filedes)*

DESCRIPTION

The close function closes the file descriptor *filedes*.

EXAMPLE

```
{
 int fd;

 . . .

 if (close(fd) != 0) {
   /* handle close error conditions */
 }
}
```

RETURNS

The normal return value from close is 0. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

E_NODEV The requested minor device does not exist.

This is the only error code returned by the driver, other codes may be returned by the I/O system during close. For more information about close error codes, see the *GNU C Library description – Low-Level Input/Output*.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.3 read()

NAME

read() – read from a device

SYNOPSIS

#include <unistd.h>

ssize_t read(*int filedes, void *buffer, size_t size*)

DESCRIPTION

The read function reads a CAN message from the specified receive queue. A pointer to the callers message buffer (*T816_MSG_BUF*) and the size of this structure is passed by the parameters *buffer* and *size* to the device.

The *T816_MSG_BUF* structure has the following layout:

```
typedef struct {

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

} T816_MSG_BUF, *PT816_MSG_BUF;
```

Identifier
Receives the message identifier of the read CAN message .

Timeout
Specifies the amount of time (in ticks) the caller is willing to wait for execution of read. A value of 0 means wait indefinitely.

RxQueueNum
Specifies the receive queue number from which the data will be read. Valid receive queue numbers are in range between 1 and n. In which n depends on the definition of *NUM_RX_QUEUES* (see also 2.6).

Extended
Receives TRUE for extended CAN messages.

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

T816_SUCCESS	No messages lost
T816_FIFO_OVERRUN	One or more messages was overwritten in the receive queue FIFO. This problem occurs if the

	FIFO is too small for the application read interval.
T816_MSGOBJ_OVERRUN	One or more messages was overwritten in the CAN controller message object because the interrupt latency is too large. Keep in mind Linux isn't a real-time operating system. Use message object 15 (buffered) to receive this time critical CAN messages, reduce the CAN bit rate or upgrade the system speed.
T816_RAW_FIFO_OVERRUN	One or more messages was overwritten in the FIFO between the interrupt service routine and post-processing in the driver (bottom half).

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.

EXAMPLE

```
{
 int fd;
 ssize_t  NumBytes;
 T816_MSG_BUF  MsgBuf;

 . . .

 MsgBuf.RxQueueNum = 1;
 MsgBuf.Timeout = 200;      /* 2 sec*/

 NumBytes = read(fd, &MsgBuf, sizeof(MsgBuf));

 if (NumBytes > 0) {
   /* process received CAN message */
 }

 . . .

}
```

RETURNS

On success read returns the size of structure T816_MSG_BUF. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

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.
EFAULT	Invalid pointer to the message buffer.
ECONNREFUSED	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.
EAGAIN	Resource temporarily unavailable; the call might work if you try again later. This error occurs only if the device is opened with the flag *O_NONBLOCK* set.
ETIME	The allowed time to finish the read request is elapsed.
EINTR	Interrupted function call; an asynchronous signal occurred and prevented completion of the call. When this happens, you should try the call again.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.4 write()

NAME

write() – write to a device

SYNOPSIS

#include <unistd.h>

ssize_t write(*int filedes, void *buffer, size_t size*)

DESCRIPTION

The write function writes a CAN message to the device with descriptor *filedes*. A pointer to the callers message buffer (*T816_MSG_BUF*) and the size of this structure is passed by the parameters *buffer* and *size* to the device.

The write function dynamically allocates a free message object for this transmit operation. The search begins at message object 1 and ends at message object 14. The first found free message object is used. If currently no message object is available the write operation is blocked until any message object become free or a timeout occur.
If your application performs write operations you should left at least one message object free for transmit, preferably the first message object.

The *T816_MSG_BUF* structure has the following layout:

```
typedef struct {

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

} T816_MSG_BUF, *PT816_MSG_BUF;
```

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

Timeout
Specifies the amount of time (in ticks) the caller is willing to wait for execution of write. A value of 0 means wait indefinitely.

RxQueueNum
Unused for this control function. Can be 0.

Extended
Contains TRUE (1) for extended CAN messages..

Status
Unused for this control function. Can be 0.

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.

EXAMPLE

```
{
 int fd;
 ssize_t  NumBytes;
 T816_MSG_BUF  MsgBuf;

 . . .

 MsgBuf.Identifier = 1234;
 MsgBuf.Timeout = 200;      /* 2 sec*/
 MsgBuf.Extended = TRUE;
 MsgBuf.MsgLen = 2;
 MsgBuf.Data[0] = 0xaa;
 MsgBuf.Data[1] = 0x55;

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

 if (NumBytes > 0) {
   /* CAN message successful transmitted */
 }
 . . .
}
```

RETURNS

On success write returns the size of structure T816_MSG_BUF. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL	Invalid argument. This error code is returned if the size of the message buffer is too small.
EFAULT	Invalid pointer to the message buffer.
ECONNREFUSED	The controller is in bus off state and unable to transmit messages.
EAGAIN	Resource temporarily unavailable; the call might work if you try again later. This error occurs only if the device is opened with the flag *O_NONBLOCK* set.
ETIME	The allowed time to finish the write request is elapsed. This occur if currently no message object is available or if the CAN bus is overloaded and the priority of the message identifier is too low.
EINTR	Interrupted function call; an asynchronous signal occurred and prevented completion of the call. When this happens, you should try the call again.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.5 ioctl()

NAME

ioctl() – device control functions

SYNOPSIS

#include <sys/ioctl.h>

int ioctl(int filedes, int request [, void *argp])

DESCRIPTION

The **ioctl** function sends a control code directly to a device, specified by *filedes*, causing the corresponding device to perform the requested operation.
The argument *request* specifies the control code for the operation. The optional argument *argp* depends on the selected request and is described for each request in detail later in this chapter.

The following ioctl codes are defined in *TIP816.h* :

Value	**Meaning**
T816_IOCSBITTIMING	Setup a new bit timing
T816_IOCSSETFILTER	Setup acceptance filter masks
T816_IOCGGETFILTER	Get the current acceptance filter masks
T816_IOCBUSON	Enter the bus on state
T816_IOCBUSOFF	Enter the bus off state
T816_IOCFLUSH	Flush one or all receive queues
T816_IOCGCANSTATUS	Returns the contents of the CAN controller status register
T816_IOCSDEFRXBUF	Define a receive buffer message object
T816_IOCSDEFRMTBUF	Define a remote transmit buffer message object
T816_IOCSUPDATEBUF	Update a remote or receive buffer message object
T816_IOCTRELEASEBUF	Release an allocated message buffer object

See behind for more detailed information on each control code.

Note

To use these TIP816 specific control codes the header file TIP816.h must be included in the application

RETURNS

On success, zero is returned. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL	Invalid argument. This error code is returned if the requested ioctl function is unknown. Please check the argument *request*.

Other function dependant error codes will be described for each ioctl code separately. Note, the TIP816 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.5.1 T816_IOCSBITTIMING

NAME

T816_IOCSBITTIMING - Setup a 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 callers parameter buffer (*T816_BITTIMING*) is passed by the argument *argp* to the driver.

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

The *T816_BITTIMING* structure has the following layout:

```
typedef struct {

    unsigned short    TimingValue;
    unsigned short    TreeSamples;

} T816_BITTIMING, *PT816_BITTIMING;
```

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 1.6 MBit per second. The include file 'TIP816.H' contains predefined transfer rate symbols (T816_5KBIT .. T816_1_6MBIT).
For other transfer rates please follow the instructions of the Intel 82527 Architectural Overview, which is also part of the engineering kit TIP816-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;
 T816_BITTIMING  BitTimingParam;

 . . .

 BitTimingParam.TimingValue = T816_100KBIT;
 BitTimingParam.ThreeSamples = FALSE;
```

```
 result = ioctl(fd, T816_IOCSBITTIMING, &BitTimingParam);

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

 . . .

}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.

SEE ALSO

ioctl man pages
tip816.h for predefined bus timing constants
Intel 82527 Architectural Overview - *4.13 Bit Timing Overview*

3.5.2 T816_IOCSSETFILTER

NAME

T816_IOCSSETFILTER - Setup acceptance filter masks

DESCRIPTION

This ioctl function modifies the acceptance filter masks of the specified CAN controller device.
The acceptance masks allow message objects to receive messages with a larger range of message identifiers instead of just a single message identifier. A "0" value means "don't care" or accept a "0" or "1" for that bit position. A "1" value means that the incoming bit value "must-match" identically to the corresponding bit in the message identifier.

A pointer to the callers parameter buffer (*T816_ACCEPT_MASKS*) is passed by the parameter *argp* to the driver.

The *T816_ACCEPT_MASKS* structure has the following layout:

```
typedef struct {

    unsigned long     Message15Mask;
    unsigned long     GlobalMaskExtended;
    unsigned short    GlobalMaskStandard;

} T816_ACCEPT_MASKS, *PT816_ACCEPT_MASKS;
```

Message15Mask
This parameter specifies the value for the Message 15 Mask Register. The Message 15 Mask Register is a local mask for message object 15. This 29 bit identifier mask appears in bit 3..31 of this parameter.
The Message 15 Mask is "ANDed" with the Global Mask. This means that any bit defined as "don't care" in the Global Mask will automatically be a "don't care" bit for message 15.(See also Intel 82527 Architectural Overview).

GlobalMaskExtended
This parameter specifies the value for the Global Mask-Extended Register. The Global Mask-Extended Register applies only to messages using the extended CAN identifier. This 29 bit identifier mask appears in bit 3..31 of this parameter.

GlobalMaskStandard
This parameter specifies the value for the Global Mask-Standard Register. The Global Mask-Standard Register applies only to messages using the standard CAN identifier. The 11 bit identifier mask appears in bit 5..15 of this parameter.

Note

The TIP816 device driver copies the parameter directly into the corresponding registers of the CAN controller, without shifting any bit positions. For more information see the Intel 82527 Architectural Overview - *4.7…4.10*

EXAMPLE

```
{

 int fd;
 int result;
 T816_ACCEPT_MASKS AcceptMasksParam;

 . . .

 /* Standard identifier bits 0..3 don't care */
 AcceptMasksParam.GlobalMaskStandard = 0xfe00;

 /* Extended identifier bits 0..3 don't care */
 AcceptMasksParam.GlobalMaskExtended = 0xffffff80;

 /* Message object 15 identifier bits 0..7 don't care */
 AcceptMasksParam.Message15Mask = 0xfffff800;


 result = ioctl(fd, T816_IOCSSETFILTER, &AcceptMasksParam);

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

 . . .

}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.

SEE ALSO

ioctl man pages
Intel 82527 Architectural Overview - *4.9 Acceptance Filtering*

3.5.3 T816_IOCGGETFILTER

NAME

T816_IOCGGETFILTER - Get the current acceptance filter masks

DESCRIPTION

This ioctl function returns the current acceptance filter masks of the specified CAN Controller.

A pointer to the callers parameter buffer (*T816_ACCEPT_MASKS*) is passed by the parameter *argp* to the driver.

The *T816_ACCEPT_MASKS* structure has the following layout:

```
typedef struct {

    unsigned long     Message15Mask;
    unsigned long     GlobalMaskExtended;
    unsigned short    GlobalMaskStandard;

} T816_ACCEPT_MASKS, *PT816_ACCEPT_MASKS;
```

Message15Mask
This parameter receives the value for the Message 15 Mask Register. The Message 15 Mask Register is a local mask for message object 15. This 29 bit identifier mask appears in bit 3..31 of this parameter.

GlobalMaskExtended
This parameter receives the value for the Global Mask-Extended Register. The Global Mask-Extended Register applies only to messages using the extended CAN identifier. This 29 bit identifier mask appears in bit 3..31 of this parameter.

GlobalMaskStandard
This parameter receives the value for the Global Mask-Standard Register. The Global Mask-Standard Register applies only to messages using the standard CAN identifier. The 11 bit identifier mask appears in bit 5..15 of this parameter.

Note

The TIP816 device driver copies the masks directly from the corresponding registers of the CAN controller into the parameter buffer, without shifting any bit positions. For more information see the Intel 82527 Architectural Overview - *4.7…4.10*

EXAMPLE

```
{

 int fd;
 int result;
 T816_ACCEPT_MASKS AcceptMasksParam;

 . . .

 result = ioctl(fd, T816_IOCGGETFILTER, &AcceptMasksParam);

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

 . . .

}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.

SEE ALSO

ioctl man pages
Intel 82527 Architectural Overview - *4.9 Acceptance Filtering*

3.5.4 T816_IOCBUSON

NAME

T816_IOCBUSON - 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 init bit in the control 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 can be omitted for this ioctl function.

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, T816_IOCBUSON);

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

 . . .

}
```

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

ioctl man pages
Intel 82527 Architectural Overview - *3.2 Software Initialization*

3.5.5 T816_IOCBUSOFF

NAME

T816_IOCBUSOFF - 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 T816_IOCBUSON is executed.
The optional argument can be omitted for this ioctl function.

Note

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

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 result = ioctl(fd, T816_IOCBUSOFF);

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

 . . .

}
```

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

ioctl man pages
Intel 82527 Architectural Overview - *3.2 Software Initialization*

3.5.6 T816_IOCFLUSH

NAME

T816_IOCFLUSH - Flush one or all receive queues

DESCRIPTION

This ioctl function flushes the message FIFO of the specified receive queue(s).
The optional argument *argp* passes the receive queue number to the device driver on which the FIFO's to be flushed. If this parameter is 0 the FIFO's of all receive queues of the device will be flushed, otherwise only the FIFO of the specified receive queue will be flushed.

EXAMPLE

```
{

 int fd;
 int result;

 . . .

 /* flush all receive queues */

 result = ioctl(fd, T816_IOCFLUSH, (int)0);

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

 . . .

}
```

ERRORS

EINVAL	Invalid argument. This error code is returned if the specified receive queue is out of range.

SEE ALSO

ioctl man pages

3.5.7 T816_IOCGCANSTATUS

NAME

T816_IOCGCANSTATUS - Returns the contents of the CAN status register

DESCRIPTION

This ioctl function returns the actual contents of the CAN controller status register for diagnostic purposes.
The contents of the controller status register is received in a unsigned char variable. A pointer to this variable is passed by the argument *argp* to the driver.

EXAMPLE

```
{

 int fd;
 int result;
 unsigned char CanStatus;

 . . .

 result = ioctl(fd, T816_IOCGCANSTATUS, &CanStatus);

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

 . . .

}
```

ERRORS

EFAULT	Invalid pointer to the unsigned char variable which receives the contents of the CAN status register. Please check the argument *argp*.

SEE ALSO

ioctl man pages
Intel 82527 Architectural Overview - *4.3 Status Register (01H)*

3.5.8 T816_IOCSDEFRXBUF

NAME

T816_IOCSDEFRXBUF - Define a receive buffer message object

DESCRIPTION

This ioctl function defines a CAN message object to receive a single message identifier or a range of message identifiers (see also Acceptance Mask). All CAN messages received by this message object are directed to the associated receive queue and can be read with the standard read function (see also 3.3).

Before the driver can receive CAN messages it's necessary to define at least one receive message object. If only one receive message object is defined at all preferably message object 15 should be used because this message object is buffered.

A pointer to the callers message description (*T816_BUF_DESC*) is passed by the argument *argp* to the driver.

The *T816_BUF_DESC* structure has the following layout:

```
typedef struct {

    unsigned long   Identifier;
    unsigned char   MsgObjNum;
    unsigned char   RxQueueNum;
    unsigned char   Extended;
    unsigned char   MsgLen;
    unsigned char   Data[8];

} T816_BUF_DESC, *PT816_BUF_DESC;
```

Identifier
Specifies the message identifier for the message object to be defined.

MsgObjNum
Specifies the number of the message object to be defined. Valid object numbers are in range between 1 and 15.

RxQueueNum
Specifies the associated receive queue for this message object. All CAN messages received by this object are directed to this receive queue. The receive queue number is one based, valid numbers are in range between 1 and n. In which n depends on the definition of *NUM_RX_QUEUES* (see also 2.6).

Note

It's possible to assign more than one receive message object to one receive queue.

Extended
Set to TRUE for extended CAN messages.

MsgLen
Unused for this control function. Set to 0.

Data[8]
Unused for this control function.

EXAMPLE

```
{

 int fd;
 int result;
 T816_BUF_DESC  BufDesc;

 . . .

 BufDesc.MsgObjNum    = 15;
 BufDesc.RxQueueNum   = 1;
 BufDesc.Identifier   = 1234;
 BufDesc.Extended     = TRUE;

 /* Define message object 15 to receive the extended        */
 /* message identifier 1234 and store received messages     */
 /* in receive queue 1                                      */

 result = ioctl(fd, T816_IOCSDEFRXBUF, &BufDesc);

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

 . . .

}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.
EINVAL	Invalid argument. This error code is returned if either the message object number, or the specified receive queue is out of range.
EADDRINUSE	The requested message object is already occupied.

SEE ALSO

ioctl man pages
Intel 82527 Architectural Overview - *4.18 82527 Message Objects*

3.5.9 T816_IOCSDEFRMTBUF

NAME

T816_IOCSDEFRMTBUF - Define a remote transmit buffer message object

DESCRIPTION

This ioctl function defines a remote transmission CAN message buffer object. A remote transmission object is similar to normal transmission object with exception that the CAN message is transmitted only after receiving of a remote frame with the same identifier.
This type of message object can be used to make process data available for other nodes which can be polled around the CAN bus without any action of the provider node.
The message data remain available for other CAN nodes until this message object is updated with the control function *T816_IOCSUPDATEBUF* or cancelled with *T816_IOCTRELEASEBUF*.

A pointer to the callers message description (*T816_BUF_DESC*) is passed by the argument *argp* to the driver.

The *T816_BUF_DESC* structure has the following layout:

```
typedef struct {

    unsigned long   Identifier;
    unsigned char   MsgObjNum;
    unsigned char   RxQueueNum;
    unsigned char   Extended;
    unsigned char   MsgLen;
    unsigned char   Data[8];

} T816_BUF_DESC, *PT816_BUF_DESC;
```

Identifier
Specifies the message identifier for the message object to be defined.

MsgObjNum
Specifies the number of the message object to be defined. Valid object numbers are in range between 1 and 14.
Keep in mind that message object 15 is only available for receive message objects.

RxQueueNum
Unused for remote transmission message objects. Set to 0.

Extended
Set to TRUE for extended CAN messages.

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.

EXAMPLE

```
{

 int fd;
 int result;
 T816_BUF_DESC  BufDesc;

 . . .

 BufDesc.MsgObjNum    = 10;
 BufDesc.Identifier   = 777;
 BufDesc.Extended     = TRUE;
 BufDesc.MsgLen       = 1;
 BufDesc.Data[0]      = 123;

 /* Define message object 10 to transmit the extended       */
 /* message identifier 777 after receiving of a remote      */
 /* frame with der same identifier                          */

 result = ioctl(fd, T816_IOCSDEFRMTBUF, &BufDesc);

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

 . . .

}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.
EINVAL	Invalid argument. This error code is returned if the message object number is out of range.
EADDRINUSE	The requested message object is already occupied.
EMSGSIZE	Invalid message size. MsgLen must be in range between 0 and 8.

SEE ALSO

ioctl man pages
Intel 82527 Architectural Overview - *4.18 82527 Message Objects*

3.5.10 T816_IOCSUPDATEBUF

NAME

T816_IOCSUPDATEBUF - Update a remote or receive buffer message object

DESCRIPTION

This ioctl function updates a previous defined receive or remote transmission message buffer object.
To update a receive message object a remote frame is transmitted over the CAN bus to request new data from a corresponding remote transmission message object on other nodes.
To update a remote transmission object only the message data and message length of the specified message object is changed. No transmission is initiated by this control function.

A pointer to the callers message description (*T816_BUF_DESC*) is passed by the argument *argp* to the driver.

The *T816_BUF_DESC* structure has the following layout:

```
typedef struct {

    unsigned long   Identifier;
    unsigned char   MsgObjNum;
    unsigned char   RxQueueNum;
    unsigned char   Extended;
    unsigned char   MsgLen;
    unsigned char   Data[8];

} T816_BUF_DESC, *PT816_BUF_DESC;
```

Identifier
Unused for this control function. Set to 0.

MsgObjNum
Specifies the number of the message object to be updated. Valid object numbers are in range between 1 and 14.
Keep in mind that message object 15 is available only for receive message objects.

RxQueueNum
Unused for this control function. Set to 0.

Extended
Set to TRUE for extended CAN messages.

MsgLen
Contains the number of message data bytes (0..8). This parameter is used only for remote transmission object updates.

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.
This parameter is used only for remote transmission object updates.

EXAMPLE

```
{

 int fd;
 int result;
 T816_BUF_DESC  BufDesc;

 . . .

 /* Update a receive message object */
 BufDesc.MsgObjNum    = 14;

 result = ioctl(fd, T816_IOCSUPDATEBUF, &BufDesc);

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


 /* Update a remote message object */
 BufDesc.MsgObjNum    = 10;
 BufDesc.MsgLen       = 1;
 BufDesc.Data[0]      = 124;

 result = ioctl(fd, T816_IOCSUPDATEBUF, &BufDesc);

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


 . . .

}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.
EINVAL	Invalid argument. This error code is returned if either the message object number is out of range or the requested message object is not defined.
EMSGSIZE	Invalid message size. *MsgLen* must be in range between 0 and 8.

SEE ALSO

ioctl man pages
Intel 82527 Architectural Overview - *4.18 82527 Message Objects*

3.5.11 T816_IOCTRELEASEBUF

NAME

T816_IOCTRELEASEBUF - Release an allocated message buffer object

DESCRIPTION

This TIP816 control function releases a previous defined CAN message object. Any CAN bus transactions of the specified message object become disabled. After releasing the message object can be defined again with *T816_IOCSDEFRXBUF* and *T816_IOCSDEFRMTBUF* control functions.

A pointer to the callers message description (*T816_BUF_DESC*) is passed by the argument *argp* to the driver.

The *T816_BUF_DESC* structure has the following layout:

typedef struct {

```
    unsigned long    Identifier;
    unsigned char    MsgObjNum;
    unsigned char    RxQueueNum;
    unsigned char    Extended;
    unsigned char    MsgLen;
    unsigned char    Data[8];
```

} T816_BUF_DESC, *PT816_BUF_DESC;

MsgObjNum
Specifies the number of the message object to be released. Valid object numbers are in range between 1 and 15.

All other parameter are not used and could be left blank.

EXAMPLE

```
{
 int fd;
 int result;
 T816_BUF_DESC  BufDesc;

 . . .

 BufDesc.MsgObjNum    = 14;

 result = ioctl(fd, T816_IOCTRELEASEBUF, &BufDesc);

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

 . . .
}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.
EINVAL	Invalid argument. This error code is returned if the message object number is out of range.
EBADMSG	The requested message object is not defined.
EBUSY	The message object is currently busy transmitting data.

SEE ALSO

ioctl man pages

4 Diagnostic

If the TIP816 does not work properly it is helpful to get some status information from the driver.
You will got the debug output below by defining the macro *TIP816_DEBUG_VIEW* in tip816.c

Note

According to the mechanism your system uses to deliver output from kernel modules the output appear on the xconsole window (SUSE distribution) or it may go to one of the system log files, such as */var/log/messages*.

```
###   Initialize TIP816(0) at physical address =
0xE9000000/0xE8001100,  irq=11   ###

IP MEM Memory Space
C905B000 : 00 01 61 61 00 01 FF FF FF FF F8 FF 00 00 00 00
C905B010 : 55 55 20 00 00 00 00 00 00 00 00 00 00 00 00 00
C905B020 : 55 55 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C905B030 : 55 55 02 00 00 40 00 00 00 00 00 00 00 00 00 40
C905B040 : 55 55 00 00 00 00 00 00 00 00 08 00 00 00 00 00
C905B050 : 55 95 04 00 40 00 00 00 00 00 00 00 00 00 00 00
C905B060 : 55 55 00 00 20 80 00 00 00 00 00 00 00 00 FF 20
C905B070 : 55 55 08 10 40 01 00 00 00 00 00 00 00 00 FF 00
C905B080 : 99 55 00 08 00 00 00 00 00 00 00 00 00 00 FF 20
C905B090 : 55 55 08 01 10 00 00 04 00 00 00 00 00 00 00 00
C905B0A0 : 55 95 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C905B0B0 : 55 55 00 08 48 01 00 00 00 00 00 00 00 00 00 00
C905B0C0 : 55 55 00 04 00 00 00 00 00 00 00 00 00 00 FF 00
C905B0D0 : 55 65 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C905B0E0 : 55 56 00 20 00 10 00 00 00 00 00 00 00 00 00 00
C905B0F0 : 55 95 A9 42 00 B4 00 00 00 00 00 00 00 00 FF 00

IP ID Memory Space
C9059100 : 49 00 50 00 41 00 43 00 B3 00 1B 00 10 00 00 00
C9059110 : 00 00 00 00 0D 00 CC 00 0A 00 00 00 00 00 00 00
C9059120 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C9059130 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The endian mode of the carrier board can be find out by the bordered bytes in the IP-MEM output above. If the order of these bytes is F8 FF the endian mode doesn' t match and the macro *SWAP_BYTE_LANE* must be defined to compansate this.
If the order of these bytes is FF F8 byte swapping isn't necessary. The definition of the macro *SWAP_BYTE_LANE* must be removed (comment it out).

The alignment of the IP-ID space can be find out by the bordered byte sequence in the IP-ID space output above. If the order of these bytes is 49 00 50 00 41 00 43 00 the IP-ID space is aligned to even addresses and the macro *TIPxxx_ODD_MAP* must be removed. If the order of these bytes is 00 49 00 50 00 41 00 43 the IP-ID space is aligned to odd address. In this case the macro *TIPxxx_ODD_MAP* must be defined in tip816def.h.

Note

The TIP816 register dump can be used only after the first start of the driver after a power-up of your system!