The Embedded I/O Company

TEWS TECHNOLOGIES

TPMC816-SW-72

LynxOS Device Driver

Extended CAN

Version 1.0.x

User Manual

Issue 1.0

March 2003

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

TEWS TECHNOLOGIES LLC
1 E. Liberty Street, Sixth Floor
Reno, Nevada 89504 / USA
Phone: +1 (775) 686 6077
Fax: +1 (775) 686 6024
e-mail: usasales@tews.com
www.tews.com

TPMC816-SW-72

Extended CAN

LynxOS Device Driver

Issue	Description	Date
1.0	First Issue	March 25, 2003

Table of Contents

1 Introduction

The TPMC816-SW-72 LynxOS device driver allows the operation of a TPMC816 PMC module on LynxOS systems with DRM based PCI Interface.

The standard file (I/O) functions (open, close, read) provide the basic interface for opening and closing a file descriptor and for performing device input operations.

The TPMC816 device driver includes the following functions:

- 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
- Standard bit rates from 5 kbit up to 1.0 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

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 TPMC816-EK.

2 Installation

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

Following files are located on the diskette:

tpmc816.c	Driver source code
tpmc816.h	Definitions and data structures for driver and application
tpmc816def.h	Definitions and data structures for the driver
tpmc816_info.c	Device information definition
tpmc816_info.h	Device information definition header
tpmc816.cfg	Driver configuration file include
tpmc816.import	Linker import file
Makefile	Device driver make file
Makefile.ppc.dldd	Make file for dynamic driver installation (PowerPC)
Makefile.x86.dldd	Make file for dynamic driver installation (Intel x86)
example/example.c	Example application source
TPMC816-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 driver directory path */sys/drivers.xxx*, where *xxx* represents the BSP that supports the target hardware. For example: */sys/drivers.pp_drm/tpmc816*
2. Copy the following files to this directory: *tpmc816.c, tpmc816def.h, Makefile*
3. Copy *tpmc816.h* to */usr/include/*
4. Copy *tpmc816_info.c* to */sys/devices.xxx/* or */sys/devices* if */sys/devices.xxx* does not exist (xxx represents the BSP).
5. Copy *tpmc816_info.h* to */sys/dheaders/*
6. Copy *tpmc816.cfg* to */sys/cfg.ppc/*

2.1.1.1 Build the driver object

1. Change to the directory */sys/drivers.xxx/tpmc816,* 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 = ... tpmc816_info.x
3. And at the end of the Makefile

   tpmc816_info.o:$(DHEADERS)/tpmc816_info.h
4. 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:tpmc816.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
   ```
4. The **N** flag instructs *init* to run *mknod* and create all the nodes mentioned in the new *nodetab*.
5. After reboot you should find the following new devices (depends on the device configuration): */dev/tp816a1, /dev/tp816a2 …*

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 driver directory path */sys/drivers.xxx*, where *xxx* represents the BSP that supports the target hardware. For example: */sys/drivers.pp_drm/tpmc816*
2. Copy the following files to this directory:

   tpmc816.c
   tpmc816def.h
   tpmc816_info.c
   tpmc816_info.h
   tpmc816.import
   Makefile.ppc.dldd
   Makefile.x86.dldd

3. Copy *tpmc816.h* to */usr/include*
4. Change to the directory */sys/drivers.xxx/tpmc816*
5. To make the dynamic link-able driver enter:

   ```
   make –f Makefile.ppc.dldd  (Makefile.x86.dldd on x86 sytems!)
   ```

6. Create a device definition file for one major device:

   ```
   make –f Makefile.ppc.dldd tpmc816_info
   ./tpmc816_info > tp816a_info
   ```

7. To install the driver enter:

   ```
   drinstall –c tpmc816.obj
   ```

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

8. To install the major device enter:

   ```
   devinstall –c –d <driver-ID> tp816a_info
   ```

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

9. To create nodes for the devices enter:

   ```
   mknod /dev/tp816a c <major_no> 0...
   ```

If all steps are successful completed the TPMC816 is ready to use.

To uninstall the TPMC816 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 TPMC816 major device.

The implementation of the device information definition is done through a C structure which is defined in the header file tpmc816_*info.h*.

This structure contains following parameter:

PCIBusNumber	Contains the PCI bus number at which the TPMC816 is connected. Valid bus numbers are in range from 0 to 255.
PCIDeviceNumber	Contains the device number (slot) at which the TPMC816 is connected. Valid device numbers are in range from 0 to 31.

If both *PCIBusNumber* and *PCIDeviceNumber* are –1 then the driver will auto scan for the TPMC816 device. The first device found in the scan order will be allocated by the driver for this major device. Already allocated devices can't be allocated twice. This is important to know if you have more that one TMPC816 major device.

A device information definition is unique for every TPMC816 major device. The file *tpmc816_info.c* on the distribution disk contains two device information declarations, **tp816a_info** for the first major device and **tp816b_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 **tp816c_info**, **tp816d_info** and so on.

It is also necessary to modify the device and driver configuration files respectively the configuration include file *tpmc816.cfg*.

The following device declaration information uses the auto find method to detect the TPMC816 module on the PCI bus.

```
TP816_INFO tp816a_info = {
     -1,       /*  auto find the TPMC816 on any PCI bus    */
     -1,
};
```

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 the file and produces a new set of driver and device configuration tables and a corresponding *nodetab*.

To install the TPMC816 driver and devices into the LynxOS system, the configuration include file *tpmc816.cfg* must be included in the *CONFIG.TBL* (see also 2.1.1.3).

The file *tpmc816.cfg* on the distribution disk contains the driver entry (C:tpmc816:\....) and one enabled major device entry (D:TPMC816 1:tp816a_info::) with two minor device entries (N:tp816a1:0 and N:tp816a2:1).

If the driver should support more than one major device (TPMC816) 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 TPMC816 devices.

The name of the device information declaration (info-block-name) must match to an existing C structure in the file *TPMC816_info.c*.

This example shows a driver entry with one major device and 2 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:tpmc816:\
     :tp816open:tp816close:tp816read:tp816write:\
     ::tp816ioctl:tp816install:tp816uninstall
D:TPMC816 1:tp816a_info::
N:tp816a1:0
N:tp816a2:1
```

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

```
/dev/tp816a1
/dev/tp816a2
```

2.2 Receive Queue Configuration

Received CAN messages will be stored in receive queues. Each receive queue contains a FIFO. The number of receive queues and the depth of the FIFO can be adapted by changing the following symbols in *tpmc816def.h*.

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.

3 TPMC816 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 (TPMC816 device) named in ***path*** for reading and writing. The value of ***oflags*** indicates the intended use of the file. In case of a TPMC816 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 TPMC816 device already exists this argument is ignored.

EXAMPLE

```
int  fd

fd = open ("/dev/tp816a1", 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;

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 <tpmc816.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 (*TP816_MSG_BUF*) and the size of this structure is passed by the parameters ***buff*** and ***count*** to the device.

The *TP816_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];
} TP816_MSG_BUF, *PTP816_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.

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 **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Extended

Receives TRUE for extended CAN messages.

Status

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

TP816_SUCCESS	No messages lost
TP816_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.
TP816_MSGOBJ_OVERRUN	One or more messages were overwritten in the CAN controller message object because the interrupt latency is too large. Use message object 15 (buffered) to receive this time critical CAN messages, reduce the CAN bit rate or upgrade the system speed.

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;
int  result;
TP816_MSG_BUF  MsgBuf;


MsgBuf.RxQueueNum  = 1;
MsgBuf.Timeout     = 200;

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

if(result == sizeof(TP816_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* contains a standard read error code (see also LynxOS System Call – read) or one of the following TPMC816 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()

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 (*TP816_MSG_BUF*) and the size of this structure is passed by the parameters ***buff*** and ***count*** to the device.

The write function dynamically allocates a free message object for the transmit operation. The search begins at message object 1 and ends at message object 14. The first found free message object is used. If no message object is available the write operation returns with error.

If your application performs write operations you should left at least one message object free for transmit, preferably the first message object.

The *TP816_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];
} TP816_MSG_BUF, *PTP816_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.

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;
int  result;
TP816_MSG_BUF MsgBuf;

/*
**   Write two data bytes with identifier 1234 to the
**   CANbus and wait max. 200 ticks on execution
*/
MsgBuf.Identifier  = 1234;
MsgBuf.Timeout     = 200;
MsgBuf.Extended    = TRUE;
MsgBuf.MsgLen      = 2;
MsgBuf.Data[0]     = 0xaa;
MsgBuf.Data[1]     = 0x55;

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

if( result != sizeof(TP816_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 TPMC816 specific error code:

ENXIO	Illegal device
EINVAL	Invalid argument. This error code is returned if the size of the message buffer is too small.
ENOSPC	No free message object available for transmit.
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 occurs 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()

3.5 ioctl()

NAME

ioctl() - I/O device control

SYNOPSIS

#include <ioctl.h>

#include <tpmc816.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 *tpmc816.h* :

Value	Meaning
TP816_BITTIMING	Setup new bit timing
TP816_SETFILTER	Setup acceptance filter masks
TP816_GETFILTER	Get the current acceptance filter masks
TP816_BUSON	Enter the bus on state
TP816_BUSOFF	Enter the bus off state
TP816_FLUSH	Flush one or all receive queues
TP816_CANSTATUS	Returns the contents of the CAN controller status register
TP816_DEFRXBUF	Define a receive buffer message object
TP816_DEFRMTBUF	Define a remote transmit buffer message object
TP816_UPDATEBUF	Update a remote or receive buffer message object
TP816_RELEASEBUF	Release an allocated message buffer object

See behind for more detailed information on each control code.

RETURNS

ioctl returns 0 if successful, or –1 on error.

The TPMC816 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 TP816_BITTIMING

NAME

TP816_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 callers parameter buffer (*TP816_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 *TP816_TIMING* structure has the following layout:

```
typedef struct {
    unsigned short    TimingValue;
    unsigned short    TreeSamples;
} TP816_TIMING, *PTP816_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 1.6 MBit per second. The include file 'tpmc816.h' contains predefined transfer rate symbols (TP816_5KBIT .. TP816_1_6MBIT).

For other transfer rates please follow the instructions of the Intel 82527 Architectural Overview, which is also part of the engineering kit TPMC816-EK.

ThreeSamples

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

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;
TP816_TIMING  BitTimingParam;


BitTimingParam.TimingValue   = TP816_100KBIT;
BitTimingParam.ThreeSamples  = FALSE;

result = ioctl(fd, TP816_TIMING, (char*)&BitTimingParam);

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

SEE ALSO

tpmc816.h for predefined bus timing constants

Intel 82527 Architectural Overview - *4.13 Bit Timing Overview*

3.5.2 TP816_SETFILTER

NAME

TP816_SETFILTER - 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 (*TP816_ACCEPT_MASKS*) is passed by the parameter pointer ***arg*** to the driver.

The *TP816_ACCEPT_MASKS* structure has the following layout:

```
typedef struct {
    unsigned long     Message15Mask;
    unsigned long     GlobalMaskExtended;
    unsigned short    GlobalMaskStandard;
} TP816_ACCEPT_MASKS, *PTP816_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.

The TPMC816 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;
TP816_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, TP816_SETFILTER, (char*)&AcceptMasksParam);

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

SEE ALSO

Intel 82527 Architectural Overview - *4.9 Acceptance Filtering*

3.5.3 TP816_GETFILTER

NAME

TP816_GETFILTER - 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 (*TP816_ACCEPT_MASKS*) is passed by the parameter pointer ***arg*** to the driver.

The *TP816_ACCEPT_MASKS* structure has the following layout:

```
typedef struct {
    unsigned long     Message15Mask;
    unsigned long     GlobalMaskExtended;
    unsigned short    GlobalMaskStandard;
} TP816_ACCEPT_MASKS, *PTP816_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.

The TPMC816 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;
TP816_ACCEPT_MASKS AcceptMasksParam;


result = ioctl(fd, TP816_GETFILTER, (char*)&AcceptMasksParam);

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

SEE ALSO

Intel 82527 Architectural Overview - *4.9 Acceptance Filtering*

3.5.4 TP816_BUSON

NAME

TP816_BUSON - Enter the bus on state

DESCRIPTION

This ioctl function sets the specified CAN controller into the BusON 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 pointer can be NULL.

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, TP816_BUSON, NULL);

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

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

Intel 82527 Architectural Overview - *3.2 Software Initialization*

3.5.5 TP816_BUSOFF

NAME

TP816_BUSOFF - Enter the bus off state

DESCRIPTION

This ioctl function sets the specified CAN controller into the BusOff state.

After execution of this control function the CAN controller is completely removed from the CAN bus and cannot communicate until the control function *TP816_BUSON* is executed.

The optional argument pointer can be NULL.

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

EXAMPLE

```
int fd;
int result;


result = ioctl(fd, TP816_BUSOFF, NULL);

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

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

Intel 82527 Architectural Overview - *3.2 Software Initialization*

3.5.6 TP816_FLUSH

NAME

TP816_FLUSH - Flush one or all receive queues

DESCRIPTION

This ioctl function flushes the message FIFO of the specified receive queue.

The optional argument pointer ***arg*** passes the receive queue number to the device driver on which the FIFO's to be flushed.

EXAMPLE

```
int fd;
int result;
char  RxQueueNum;


/* flush receive queues 1 */
RxQueueNum = 1;

result = ioctl(fd, TP816_IOCFLUSH, &RxQueueNum);

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

ERRORS

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

3.5.7 TP816_DEFRXBUF

NAME

TP816_DEFRXBUF - 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 double-buffered.

A pointer to the caller's message description (*TP816_BUF_DESC*) is passed by the argument pointer ***arg*** to the driver.

The *TP816_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];
} TP816_BUF_DESC, *PTP816_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 **Fehler! Verweisquelle konnte nicht gefunden werden.**).

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;
TP816_BUF_DESC  BufDesc;


/*
**   Define message object 15 to receive the extended message identifier
**   1234 and store received messages in receive queue 1
*/
BufDesc.MsgObjNum  = 15;
BufDesc.RxQueueNum = 1;
BufDesc.Identifier = 1234;
BufDesc.Extended   = TRUE;

result = ioctl(fd, TP816_DEFRXBUF, (char*)&BufDesc);

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

ERRORS

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

Intel 82527 Architectural Overview - *4.18 82527 Message Objects*

3.5.8 TP816_DEFRMTBUF

NAME

TP816_DEFRMTBUF - 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 *TP816_UPDATEBUF* or cancelled with *TP816_RELEASEBUF*.

A pointer to the caller's message description (*TP816_BUF_DESC*) is passed by the argument pointer ***arg*** to the driver.

The *TP816_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];
} TP816_BUF_DESC, *PTP816_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.

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;
TP816_BUF_DESC  BufDesc;


/*
**   Define message object 10 to transmit the extended message identifier
**   777 after receiving of a remote frame with der same identifier
*/
BufDesc.MsgObjNum  = 10;
BufDesc.Identifier = 777;
BufDesc.Extended   = TRUE;
BufDesc.MsgLen     = 1;
BufDesc.Data[0]    = 123;

result = ioctl(fd, TP816_DEFRMTBUF, (char*)&BufDesc);

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

ERRORS

EINVAL	Invalid argument. This error code is returned if the message object number or the message length (*MsgLen*) is out of range.
EADDRINUSE	The requested message object is already occupied.

SEE ALSO

Intel 82527 Architectural Overview - *4.18 82527 Message Objects*

3.5.9 TP816_UPDATEBUF

NAME

TP816_UPDATEBUF - 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 caller's message description (*TP816_BUF_DESC*) is passed by the argument pointer ***arg*** to the driver.

The *TP816_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];
} TP816_BUF_DESC, *PTP816_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 15.

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 byte 0, Data[1] contains message data byte 1 and so on.

This parameter is used only for remote transmission object updates.

EXAMPLE

```
int fd;
int result;
TP816_BUF_DESC  BufDesc;


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

result = ioctl(fd, TP816_UPDATEBUF, (char*)&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, TP816_UPDATEBUF, (char*)&BufDesc);

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

ERRORS

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

Intel 82527 Architectural Overview - *4.18 82527 Message Objects*

3.5.10 TP816_RELEASEBUF

NAME

TP816_RELEASEBUF - Release an allocated message buffer object

DESCRIPTION

This TPMC816 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 *TP816_DEFRXBUF* and *TP816_DEFRMTBUF* control functions.

A pointer to the caller's message description (*TP816_BUF_DESC*) is passed by the argument pointer ***arg*** to the driver.

The *TP816_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];
} TP816_BUF_DESC, *PTP816_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 parameters are not used and could be left blank.

EXAMPLE

```
int fd;
int result;
TP816_BUF_DESC  BufDesc;


BufDesc.MsgObjNum  = 14;

result = ioctl(fd, TP816_RELEASEBUF, (char*)&BufDesc);

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

ERRORS

EINVAL	Invalid argument. This error code is returned if the message object number is out of range or the requested message object is not defined.
EBUSY	The message object is currently busy transmitting data or another task is waiting for a received message.

SEE ALSO

ioctl man pages

3.5.11 TP816_CANSTATUS

NAME

TP816_CANSTATUS - 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 content of the controller status register is received in an unsigned char variable. A pointer to this variable is passed by the argument pointer ***arg*** to the driver.

EXAMPLE

```
int fd;
int result;
unsigned char CanStatus;


result = ioctl(fd, TP816_CANSTATUS, (char*)&CanStatus);

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

SEE ALSO

Intel 82527 Architectural Overview - *4.3 Status Register*

4 Debugging and Diagnostic

This driver was successfully tested on Motorola PowerPC and Intel x86 compactPCI systems with LynxOS V4.0.0, V3.1.0 or V3.0.1 installed.

If the driver will not work properly, usually a PCI bus or interrupt problem, you can enable debug outputs by removing the comments around the symbols *DEBUG*, *DEBUG_PCI* and *DEBUG_TPMC*.

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 PCI Header, the address of each base address register and a memory dump of all mapped memory and I/O spaces of the TPMC816 like this (see also *TPMC816 User Manual – PCI Configuration*).

```
TPMC816 Device Driver Install
Bus = 0  Dev = 16  Func = 0
[00] = 905010B5
[04] = 02800000
[08] = 02800001
[0C] = 00000008
[10] = 02042000
[14] = 0000C001
[18] = 02043000
[1C] = 00000000
[20] = 00000000
[24] = 00000000
[28] = 00000000
[2C] = 03301498
[30] = 00000000
[34] = 00000000
[38] = 00000000
[3C] = 00000109
PCI Base Address 0 (PCI_RESID_BAR0)

B8142000 : 00 FE FF 0F 00 00 00 00 00 00 00 00 00 00 00 00
B8142010 : 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
B8142020 : 00 00 00 00 00 00 00 00 02 78 30 80 00 00 00 00
B8142030 : 00 00 00 00 00 00 00 00 00 00 00 00 81 00 00 00
B8142040 : 81 01 00 00 00 00 00 00 00 00 00 00 49 00 00 00
B8142050 : 00 00 78 18 00 00 00 00 00 00 00 00 00 00 00 00
B8142060 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B8142070 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCI Base Address 1 (PCI_RESID_BAR1)

B0108000 : 00 FE FF 0F 00 00 00 00 00 00 00 00 00 00 00 00
B0108010 : 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
B0108020 : 00 00 00 00 00 00 00 00 02 78 30 80 00 00 00 00
B0108030 : 00 00 00 00 00 00 00 00 00 00 00 00 81 00 00 00
B0108040 : 81 01 00 00 00 00 00 00 00 00 00 00 49 00 00 00
B0108050 : 00 00 78 18 00 00 00 00 00 00 00 00 00 00 00 00
```

```
B0108060 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B0108070 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCI Base Address 2 (PCI_RESID_BAR2)

B8143000 : 01 00 61 61 00 00 FF FF FF FF F8 00 00 00 00
B8143010 : 55 55 00 E0 00 08 00 10 00 00 00 00 02 00 00 01
B8143020 : 55 55 00 08 98 08 00 00 00 20 00 00 20 00 00 00
B8143030 : 55 55 0B 00 02 00 00 00 00 34 02 00 00 01 00 00
B8143040 : 55 59 00 80 00 00 00 00 00 00 00 00 01 10 20 00
B8143050 : 55 55 00 00 08 00 00 00 00 80 00 00 00 00 00 00
B8143060 : 55 55 23 81 00 F0 00 00 00 00 01 00 00 00 02 FF
B8143070 : 95 59 00 02 80 00 00 40 00 00 00 00 00 00 00 FF

Moduletype TPMC816-10 with 2 CAN Channel
LynxOS POWERPC Version 4.0.0
```

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