The Embedded I/O Company

TEWS TECHNOLOGIES

TPMC816-SW-42

VxWorks Device Driver

2 Channel CAN PMC

Version 1.4

User Manual

Issue 1.4

December 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-42

2 Channel CAN PMC

VxWorks Device Driver

This document contains information, which is proprietary to TEWS TECHNOLOGIES GmbH. Any reproduction without written permission is forbidden.

TEWS TECHNOLOGIES GmbH has made any effort to ensure that this manual is accurate and complete. However TEWS TECHNOLOGIES GmbH reserves the right to change the product described in this document at any time without notice.

TEWS TECHNOLOGIES GmbH is not liable for any damage arising out of the application or use of the device described herein.

©1998-2003 by TEWS TECHNOLOGIES GmbH

Issue	Description	Date
1.0	First Issue	September 1998
1.1	Changed file names	June 1999
1.2	Changed configuration	July 1999
1.3	Support for Intel x86 based targets	May 2000
1.4	General Revision	December 2003

Table of Content

1 Introduction

The TPMC816-SW-42 VxWorks device driver allows the operation of the TPMC816 PMC conforming to the VxWorks system specification. This includes a device-independent basic I/O interface with *open*(), *read(), write()* and *ioctl()* functions and a buffered I/O interface (*fopen(), printf(), scanf(),…).*

Special I/O operation that do not fit to the standard I/O calls will be performed by calling the *ioctl()* function with a specific function code and an optional function dependent argument.

This driver invokes a mutual exclusion and binary semaphore mechanism to prevent simultaneous requests by multiple tasks from interfering with each other.

To prevent the application program for losing data, incoming messages will be stored in a message FIFO with a depth of 100 messages.

This device driver supports all features of the Intel 82527 CAN controller:

- extended and standard message frames
- acceptance filtering
- message objects
- remote frame requests etc.

To understand and use 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 Documentation TPMC816-ED.

2 Installation

The software is delivered on a 3½" HD diskette.

Following files are located on the diskette:

tp816drv.c	TPMC816 Device Driver Source
tp816.h	TPMC816 Include File for driver and application
tp816def.h	TPMC816 Driver Include File
i82527.h	Extended CAN controller programming model
tp816exa.c	TPMC816 Example Application
Makefile	Makefile for Example Application (PowerPC targets)
tp816_pci.c	TPMC816 PCI MMU mapping for Intel x86 based targets
PCI_CONF*	Example for a PCI configuration routine (MVME3604)

For installation the files have to be copied to the desired target directory.

2.1 Install the driver to VxWorks system

To install the TPMC816 device driver to the VxWorks system following steps have to be done:

- Build the object code of the TPMC816 device driver
- Link or load the driver object file to the VxWorks system
- Call the *tp816Drv()* function to install the driver.

2.2 Including the driver in VxWorks

How to include the device drive in the VxWorks system is described in the VxWorks and Tornado manuals.

2.3 Example application

The example application uses the MVME2600/3600 BSP. If an older version of the BSP (1.1/0 up to 1.1/2) is used, the value *_OLD_BSP_* in *tp816exa.c* must be defined. If this value is undefined the newer BSP will be used.

Using a Motorola PMC-span, the PCI/PCI Bridge has to be initialized on the span.

Using other carriers the initialization matching has to be adapted to the BSP.

The example code holds two functions setting and reading the PCI configuration registers. The first function (*PCIsetupTPMC816()*) sets up the PCI configuration registers, the TPMC816 registers will appear at the specified address.

The second function (*searchPCI()*) will search for the TPMC816 and read and calculate the modules registers and correction data address, which must be used, when installing the driver.

2.4 Special installation for Intel x86 based targets

The TPMC816 device driver is fully adapted for Intel x86 based targets. This is done by conditional compilation directives inside the source code and controlled by the VxWorks global defined macro ***CPU***. If the contents of this macro are equal to *I80386*, *I80386* or *PENTIUM* special Intel x86 conforming code and function calls will be included.

The second problem for Intel x86 based platforms can't be solved by conditional compilation directives. Due to the fact that some Intel x86 BSP's doesn't map PCI memory spaces of devices which are not used by the BSP, the required CAN controller device registers can't be accessed.

To solve this problem a MMU mapping entry has to be added for the required TPMC816 PCI memory spaces prior the MMU initialization (*usrMmuInit()*) is done.

The C source file **tp816pci.c** contains the function *tp816PciInit().* This routine finds out all TPMC816 devices and adds MMU mapping entries for all used PCI memory spaces. Please insert a call to this function after the PCI initialization is done and prior to MMU initialization (*usrMmuInit()*).

If the Tornado 2.0 project facility is used, the right place to call the function *tp816PciInit()* is at the end of the function *sysHwInit()* in **sysLib.c** (can be opened from the project *Files* window).

If Tornado 1.0.1 compatibility tools are used insert the call to *tp816PciInit()* at the beginning of the root task (*usrRoot()*) in **usrConfig.c**.

Be sure that the function is called prior to MMU initialization otherwise the TPCM816 PCI spaces remains unmapped and an access fault occurs during driver initialization.

Please insert the following call at a suitable place in either **sysLib.c** or **usrConfig.c**:

```
tp816PciInit();
```

Don’t use the Makefile on the distribution disk for Intel x86 based targets.

To link the driver object modules to VxWorks, simply add all necessary driver files to the project. If Tornado 1.0.1 *Standard BSP Builds...* is used add the object modules to the macro *MACH_EXTRA* inside the BSP Makefile (MACH_EXTRA = tp816drv.o tp816pci.o ...).

3 I/O system functions

This chapter describes the driver-level interface to the I/O system. The purpose of these functions is to install the driver in the I/O system, add and initialize devices.

3.1 tp816Drv()

NAME

tp816Drv() - installs the TPMC816 driver in the I/O system.

SYNOPSIS

void tp816Drv(void)

DESCRIPTION

This function installs the TPMC816 driver in the I/O system.

The call of this function is the first thing the user has to do before adding any device to the system or performing any I/O request.

RETURNS

OK or ERROR (if the driver cannot be installed)

INCLUDE FILES

tp816.h

3.2 tp816DevCreate()

NAME

tp816DevCreate() - adds a TPMC816 device to the system and initializes device hardware.

SYNOPSIS

```
STATUS tp816DevCreate
    (
    char            *name,      /* name of the device to create                    */
    unsigned int    memAddr,    /* physical device memory address                  */
    unsigned int    bitTiming,  /* transfer rate (contents of Bus Timing Registers)*/
    unsigned int    channel,    /*TPMC816 channel number                           */
    unsigned char   level,      /* interrupt level (hardware dependent)            */
    unsigned char   vector      /* interrupt vector (hardware dependent)           */
    )
```

DESCRIPTION

This routine is called to add a device to the system that will be serviced by the TPMC816 driver. This function must be called before performing any I/O request to this driver.

There are several device dependent arguments required for the device initialization and allocation of the system resources.

PARAMETER

The argument **name** specifies the name, which will select the device in future calls.

The argument **memAddr** specifies the address where the TPMC816 registers can be located in memory (see chapter “Installation and Example Application”).

The argument **bitTiming** selects the transfer rate of the CAN bus. Its value is loaded into the Intel 82527 CAN controllers Bus Timing Register 0 and Bus Timing Register 1. Possible transfer rates are between 5k bit per second and 1.6M bit per second. The include file *tp816.h* contains predefined transfer rates. For other transfer rates please follow the instructions of the Intel 82527 product specification, which is also part of the Engineering Documentation TPMC816-ED.

The argument **channel** specifies the TPMC816 channel of the device to be created.

The argument **level** specifies the interrupt level where the TPMC816 interrupts will be generated to. Please refer to the CPU and BSP description to get the right value.

The argument **vector** specifies the interrupt vector which will be used when an interrupt is generated. Please refer to the CPU and BSP description to get the right value.

EXAMPLE

```
#include "tp816.h"

...

int status;

...

/*-------------------------------------------------------
  Create a device "/tp816/1" for TPMC816 first channel
  - transfer rate = 100 KBit/s
  - TPMC816 registers mapped to 0xC1000000
  -------------------------------------------------------*/
status = tp816DevCreate("/tp816/1",
                        0xC1000000,
                        TP816_100KBIT,
                        0,
                        0xF,
                        0xF);

...
```

RETURNS

OK or ERROR

INCLUDE FILES

tp816.h

4 I/O interface functions

This chapter describes the interface to the basic I/O system used for communication over the CAN bus.

4.1 open()

NAME

open() - opens a device or file.

SYNOPSIS

```
int open
    (
    const char  *name,      /* name of the device to open                  */
    int         flags,      /* not used for TPMC816 driver, must be 0      */
    int         mode        /* not used for TPMC816 driver, must be 0      */
    )
```

DESCRIPTION

Before I/O can be performed to the TPMC816 device, a file descriptor must be opened by invoking the basic I/O function *open().*

PARAMETER

The parameter **name** selects the device which shall be opened.

The parameters **flags** and **mode** are not used and must be 0.

EXAMPLE

```
...

/*------------------------------------------------------------
  Open the device named "/tpmc816/1" for I/O
  ------------------------------------------------------------*/
fd = open("/tpmc816/1", 0, 0);

...
```

RETURNS

A device descriptor number or ERROR (if the device does not exist or no device descriptors are available)

INCLUDE FILES

vxworks.h

tp816.h

SEE ALSO

ioLib, basic I/O routine - *open()*

4.2 read()

NAME

read() – reads a message from the specified TPMC816 device.

SYNOPSIS

```
int read
    (
    int         fd,         /* device descriptor from opened TPMC816 device    */
    char        *buffer,    /* pointer to the message buffer                   */
    size_t      maxbytes    /* not used                                        */
    )
```

DESCRIPTION

Once a TPMC816 device has been initialized and a message object with receive direction is defined (*ioctl()* function *CMD_DEFINE_MSG*). The tasks can read received messages from the device.

If the device is blocked by another read request or no message is available in the read buffer, the requesting task will be blocked by a semaphore until a message was read or the request times out. If the flag *TP816_NOWAIT* is set, read returns immediately.

PARAMETER

The parameter **fd** is a file descriptor specifying the device which shall be used.

The argument **buffer** points to a driver-specific I/O parameter block. This data structure is named *tp816_IO_BUFFER* (see below).

The parameter **maxbytes** is not used by the TPMC816 Device Driver.

data structure *tp816_IO_BUFFER*

```
typedef struct
{
unsigned long    flags;          /* I/O flags */
unsigned long    timeout;        /* timeout in ticks */
unsigned long    identifier;     /* standard or extended identifier    */
unsigned char    extended;       /* TRUE, if extended 19 bit identifier        */
unsigned char    length;         /* 0..8 byte length of valid data      */
unsigned char    data[8];        /* data buffer        */
} tp816_IO_BUFFER;
```

The parameter **flags** defines a flag field used to control the read operation. If the flag *TP816_FLUSH* is set, a flush of the device message FIFO will be performed before initiating the read request. If the flag *TP816_NOWAIT* is set, read returns immediately, if the device is blocked by another read request or no message is available.

The parameter **timeout** specifies the timeout interval, in units of clock ticks. A timeout value of 0 means wait indefinitely.

The parameter **identifier** returns the message identifier (standard or extended) of the message received.

The parameter **extended** is *TRUE* (1) for extended identifier and *FALSE* (0) for standard identifier.

The parameter **length** returns the size of message data received in bytes.

The data bytes of the message received will be returned in **data**.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

unsigned long      result, retval, i;
tp816_IO_BUFFER    rw;           /* I/O parameter block for read */
int                fd;

...

/*-------------------------------------------------------------
  Read a message from a TPMC816 device.
  - flush the input ring buffer before reading
  - if there is no message in the read buffer, the read
    request times out after 500 ticks
-------------------------------------------------------------*/
rw.flags      = TP816_FLUSH;
rw.timeout    = 500; /* ticks */

result = read(fd, &rw, 0);

if (result == OK)
{
/* process received message */
  printf(" Message with identifier %d received \n", rw.identifier);
  printf(" Message length = %d byte\n", rw.length);
  printf(" Message data   = ");
  for (i = 0; i << rw.length; i++)
  {
    printf("%c", rw.data[i]);
  }
  printf("\n\n");
}
else
{
  /* handle the read error */
}

...
```

RETURNS

ERROR or number of data_bytes read [0..8]

INCLUDE FILES

vxworks.h

tp816.h

SEE ALSO

ioLib, basic I/O routine - *read()*

4.3 write()

NAME

write() – writes a message to the specified TPMC816 device.

SYNOPSIS

```
int write
        (
        int         fd,         /* device descriptor from opened TPMC816 device    */
        char        *buffer,    /* pointer to the message buffer                   */
        size_t      bytes       /* not used                                        */
        )
```

DESCRIPTION

This routine writes a message to the specified TPMC816 device.

PARAMETER

The parameter **fd** is a file descriptor specifying the device which shall be used.

The argument **buffer** points to a driver-specific I/O parameter block. This data structure is named *tp816_IO_BUFFER* (see below).

The parameter **bytes** is not used by the TPMC816 Device Driver.

data structure *tp816_IO_BUFFER*

```
typedef struct
{
unsigned long     flags;          /* I/O flags */
unsigned long     timeout;        /* timeout in ticks */
unsigned long     identifier;     /* standard or extended identifier    */
unsigned char     extended;       /* TRUE, if extended 19 bit identifier        */
unsigned char     length;         /* 0..8 byte length of valid data      */
unsigned char     data[8];        /* data buffer        */
} tp816_IO_BUFFER;
```

The parameter **flags** defines a flag field used to control the write operation. If the flag *TP816_NOWAIT* is set, write returns immediately, if the device is blocked by another write request or no message is available.

The parameter **timeout** specifies the timeout interval, in units of clock ticks. A timeout value of 0 means wait indefinitely.

The parameter **identifier** returns the message identifier (standard or extended) of the transmit message.

The parameter **extended** is *TRUE* (1) for extended identifier and *FALSE* (0) for standard identifier.

The parameter **length** defines the length valid data in the message buffer.

The data bytes of the message received are delivered in **data**.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

unsigned long      result, retval, i;
tp816_IO_BUFFER    rw;              /* I/O parameter block for write */
int                fd;

...

#define       HELLO         "HELLOOOO"

...

/*-------------------------------------------------------------
  Write a message to a TPMC816 device.
  - if there is a problem with the transmitter the write
    request times out
-------------------------------------------------------------*/
rw.flags      = 0;
rw.timeout    = 100;        /* ticks */
rw.identifier = 1234;       /* extended identifier */
rw.extended   = TRUE;
rw.length     = 8;

memcpy(rw.data, HELLO, 8);

result = write(fd, &rw, 0);

if (result == ERROR)
{
/* handle device error */
}

...
```

RETURNS

ERROR or number of data_bytes written [0..8]

INCLUDE FILES

vxworks.h

tp816.h

SEE ALSO

ioLib, basic I/O routine - *write()*

4.4 ioctl()

NAME

ioctl() - performs an I/O control function.

SYNOPSIS

```
int ioctl
(
  int       fd,         /* device descriptor from opened TPMC816 device    */
  int       request,    /* select of control function    */
  int       arg         /* parameter buffer              */
)
```

DESCRIPTION

Special I/O operation that do not fit to the standard basic I/O calls (read, write) will be performed by calling the *ioctl()* function.

PARAMETER

The parameter **fd** specifies the device descriptor of the opened TPMC816 device.

The parameter **request** specifies the function which shall be executed.

The structure **arg** depends on the selected request (see description below).

RETURNS

OK or ERROR (if an error occurred)

INCLUDE FILES

vxworks.h

tp816.h

SEE ALSO

ioLib, basic I/O routine - *ioctl()*

4.4.1 FIO_BITTIMING

This I/O control function modifies the Bit Timing Register of the CAN controller. The function specific control parameter **arg** is a pointer on a *DC_ARGS* structure.

data structure *DC_ARGS*

```
typedef struct
{
unsigned long    cmd;          /* unused  */
unsigned long    flags;        /* I/O flags */
unsigned long    arg;          /* BITTIMING value        */
} DC_ARGS;
```

The **cmd** argument is not used.

If the I/O flag *TP816_THREE_SAMPLES* is set in the **flags** argument the CAN bus is sampled three times per bit time instead of one time.

The parameter **arg** holds the new values for the Bit Timing Register 0 (bit 8..15) and for the Bit Timing Register 1 (bit 0..7). Possible transfer rates are between 5 KBit per second and 1.6 MBit per second. The include file *tp816.h* contains predefined transfer rates. For other transfer rates please follow the instructions of the Intel 82527 Architectural Overview, which is also part of the Engineering Documentation TPMC816-ED.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

int       result;
int       fd;
DC_ARGS   dc;

...

/*-------------------------------------
  Setup the transfer rate to 500 KBit/s
  -------------------------------------*/
dc.arg    = TP816_500KBIT;
dc.flags  = 0;

result = ioctl(fd, FIO_BITTIMING, &dc);

...
```

4.4.2 FIO_SETFILTER

This I/O control function modifies the acceptance filter masks of the CAN Controller.

The acceptance masks allow message objects to receive messages with a 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 data structure *ACCEPT_MASKS* is passed to the driver by the function-dependent control parameter **arg**.

data structure *ACCEPT_MASK*

```
typedef struct
{
unsigned short     GlobalMaskStandard;
unsigned long      GlobalMaskExtended;
unsigned long      Message15Mask;
} ACCEPT_MASKS;
```

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

The parameter **GlobalMaskExtended** 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 appears in bit 3..31 of this parameter.

The parameter **Message15Mask** 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 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.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

int           result;
int           fd;
ACCEPT_MASKS  acceptMasks;

...

/*-----------------------------
  Setup acceptance filter masks
  -----------------------------*/
AcceptMasks.GlobalMaskStandard   = 0xfe00;      /* bit 0..3 don't care */

AcceptMasks.GlobalMaskExtended   = 0xffffff80;  /* bit 0..3 don't care */

AcceptMasks.Message15Mask        = 0xfffff800;  /* bit 0..7 don't care */

result = ioctl(    fd,
                   FIO_SETFILTER,
                   (unsigned long)&AcceptMasks);

...
```

4.4.3 FIO_GETFILTER

This I/O control function returns the contents of the acceptance filter masks of the CAN Controller in the data structure AcceptMasks.

A pointer to the data structure *ACCEPT_MASKS* is passed to the driver by the function-dependent control parameter **arg.**

data structure *ACCEPT_MASK*
typedef struct
{
unsigned short GlobalMaskStandard;
unsigned long GlobalMaskExtended;
unsigned long Message15Mask;
} ACCEPT_MASKS;

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

The parameter **GlobalMaskExtended** 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 appears in bit 3..31 of this parameter.

The parameter **Message15Mask** 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 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)

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

int           result;
int           fd;
ACCEPT_MASKS  acceptMasks;

...

/*--------------------------
  Get acceptance filter masks
  --------------------------*/
result = ioctl(    fd,
                   FIO_GETFILTER,
                   (unsigned long)&AcceptMasks);

...
```

4.4.4 FIO_BUSON

After an abnormal rate of occurrences of errors on the CAN bus, the CAN controller enters the *busoff* state. This I/O control function resets the init bit in the Control Register. The CAN controller begins the *busoff* recovery sequence. The bus recovery sequence resets the transmit and receive error counters. If the CAN controller counts 128 packets of 11 consecutive recessive bits on the CAN bus, the *busoff* state is exited.

No special argument is required in the control parameter **arg**.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

int  result;
int  fd;

...

/*---------------------
  Enter the buson state
  ---------------------*/
result = ioctl(fd, FIO_BUSON, 0);

...
```

4.4.5 FIO_FLUSH

This I/O control function flushes the device message FIFO.

No special argument is required for this call in the control parameter **arg**.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"
...
int  result
int  fd;
...

/*-----------------------------
  Flush the receive message FIFO
  -----------------------------*/
result = ioctl(fd, FIO_FLUSH, 0);
...
```

4.4.6 FIO_DEFINE_MSG

With this I/O control function it is possible to allocate and set up user-definable message objects. User-definable message objects are message object 1..13 and 15. Message object 14 is reserved for internal use of the device driver (write). This function requires a pointer on a *DC_ARGS* structure in the function dependent control parameter **arg**.

data structure *DC_ARGS*

```
typedef struct
{
unsigned long    cmd;          /* unused   */
unsigned long    flags;        /* I/O flags */
unsigned long    arg;          /* pointer on MSGDEF structure     */
} DC_ARGS;
```

The **cmd** argument is not used.

By combination of the I/O **flags** *TP816_RECEIVE*, *TP816_TRANSMIT,* and *TP816_REMOTE* the application program can select one of four possible types of message objects.

TP816_RECEIVE	This is a receive message object, that will receive just a single message identifier or a range of message identifiers (see also Acceptance Mask). Receive message data can be read by the standard read function.
TP816_RECEIVE **and** TP816_REMOTE	This is also a receive object, but a remote frame is sent to request a remote node to send the corresponding data.
TP816_TRANSMIT	This is a transmit object. The transmission of the message data starts immediately after definition of the message object.
TP816_TRANSMIT **and** TP816_REMOTE	This is also a transmit object, but the transmission of the message data will be started if the corresponding remote frame (same identifier) was received.

A pointer to a message object description data structure *MSGDEF* is passed to the driver by the function-dependent parameter **arg**.

data structure *MSGDEF*

```
typedef struct
{
unsigned char    MsgNum;
unsigned long    identifier;
unsigned char    extended;
unsigned char    length;
unsigned char    data[8];
} MSGDEF;
```

The parameter **MsgNum** specifies the number of the message object to define (1..13 and 15).

The parameter **identifier** specifies the message identifier (standard or extended).

If the parameter **extended** is *TRUE* (1) an extended frame message identifier will be used. If extended is *FALSE* (0) a standard frame message identifier will be used.

The parameter **length** is only necessary for transmit message objects, if *TP816_TRANSMIT* is set. The parameter **length** specifies the number of bytes data contains.

The parameter **data** is only used if a transmit object is used. It holds the message data.

A defined message object will be active until the I/O control function *FIO_CANCEL_MSG* marks it as invalid to stop communication transactions.

One of the first things the application has to do after device initialization is to define one or more receive message objects with specific identifiers that should be received by this device (see also Intel 82527 Architectural Overview - Message Objects).

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

int      result;
int      fd;
MSGDEF   MsgDef;
DC_ARGS  dc;

...

/*-------------------------------------------------------
  Define message object 1..4
  1 - Receive message object, use extended message
      identifier, wait for receiving of a message with
      specified identifier.
  2 - Receive message object, use standard message
      identifier, send a remote frame to request a
      remote node to send the corresponding data.
  3 - Transmit message object, use standard message
      identifier, start transmission.
  4 - Transmit message object, use extended message
      identifier, start transmission if the corresponding
      remote frame (same identifier) was received.
  -------------------------------------------------------*/
dc.arg        = (unsigned long)&MsgDef;
dc.flags  = TP816_RECEIVE;

MsgDef.MsgNum = 1;
MsgDef.identifier  = 10;
MsgDef.extended    = TRUE;

result = ioctl(fd, FIO_DEFINE_MSG, &dc);

...

dc.flags  = TP816_RECEIVE | TP816_REMOTE;

MsgDef.MsgNum      = 2;
MsgDef.identifier  = 100;
MsgDef.extended    = FALSE;

result = ioctl(fd, FIO_DEFINE_MSG, &dc);

...
```

```
dc.flags  = TP816_TRANSMIT;

MsgDef.MsgNum      = 3;
MsgDef.identifier  = 50;
MsgDef.extended    = FALSE;
MsgDef.length      = 1;
MsgDef.data[0]     = 'x';

result = ioctl(fd, FIO_DEFINE_MSG, &dc);

...

dc.flags  = TP816_TRANSMIT | TP816_REMOTE;

MsgDef.MsgNum      = 4;
MsgDef.identifier  = 500;
MsgDef.extended    = TRUE;
MsgDef.length      = 1;
MsgDef.data[0]     = 0;

result = ioctl(fd, FIO_DEFINE_MSG, &dc);

...
```

4.4.7 FIO_UPDATE_MSG

This I/O control function updates only the message data of a previously defined transmission object or starts transmission of a remote frame for receive message objects. The function dependent control parameter **arg** requires a pointer on a *DC_ARGS* structure.

data structure *DC_ARGS*

```
typedef struct
{
unsigned long    cmd;          /* unused                    */
unsigned long    flags;        /* I/O flags                 */
unsigned long    arg;          /* pointer on MSGDEF structure  */
} DC_ARGS;
```

The **cmd** argument is not used.

If the I/O **flags** *TP816_REMOTE* is set, transmission will be started by a request of a remote node, otherwise transmission of the message data starts immediately.

A pointer to a message object description data structure *MSGDEF* is passed to the driver by the function dependent parameter **arg**.

data structure *MSGDEF*

```
typedef struct
{
unsigned char    MsgNum;
unsigned long    identifier;   /* not used */
unsigned char    extended;     /* not used */
unsigned char    length;
unsigned char    data[8];
} MSGDEF;
```

For the function *FIO_UPDATE_MSG* only the parameters **MsgNum**, **length** and **data** are used.

The parameter **MsgNum** specifies the number of the message object to update (1..13 and 15).

The parameter **length** specifies the size of the message data.

The buffer **data** contains the new message data.

The status of message object (for example transmission completed) can be determined by using of the I/O control function *FIO_STATUS*.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

int      result;
int      fd;
MSGDEF   MsgDef;
DC_ARGS  dc;

...

/*-----------------------------------------------
  Update message object 3 and start transmission
  -----------------------------------------------*/
dc.arg        = (unsigned long)&MsgDef;
dc.flags      = 0;

MsgDef.MsgNum = 3;
MsgDef.data[0]= 'y';
MsgDef.length = 1;

result = ioctl(fd, FIO_UPDATE_MSG, &dc);

...

/*-----------------------------------------------
  Update message object 4
  Data will be sent by a request of a remote node
  -----------------------------------------------*/
dc.arg        = (unsigned long)&MsgDef;
dc.flags      = TP816_REMOTE;

MsgDef.MsgNum = 4;
MsgDef.data[0]= 1;
MsgDef.length = 1;

result = ioctl(fd, FIO_UPDATE_MSG, &dc);

...
```

4.4.8 FIO_CANCEL_MSG

This I/O control function marks a specified message object as invalid and stops communication transactions.

The number of the message object must be set in the function dependent control parameter **arg**.

Additional to the user-definable message objects 1..13 and 15 the control function *FIO_CANCEL_MSG* cancel the internal used message object 14 by selecting either message object number 14 or 0. This facility is important for write requests with the I/O flag *TP816_NOWAIT* set, to cancel pending and not longer used transmission requests (for example in *BUSOFF* case).

The message object number 0 always selects message object 14!

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

int      result;
int      fd;

...

/*---------------------------------------
  Cancel message object 2 (not longer used)
  ---------------------------------------*/
result = ioctl(fd, FIO_CANCEL_MSG, 2);

...
```

4.4.9 FIO_STATUS

This function returns the actual status of the specified transmission message object.

This function requires a pointer to a *TP816_STATUS* structure as function dependent control parameter **arg**.

Data structure *TP816_STATUS*
typedef struct
{
unsigned long message_sel;
unsigned long status;
} TP816_STATUS;

The number of the message object must be set in **message_sel**. Additional to the user-definable message objects 1..13 and 15 the function *FIO_STATUS* returns the status of the internal used message object 14 by selecting either message object number 14 or 0. This facility is important for write request with the I/O flag *TP816_NOWAIT* set.

The message state is returned in **status**. The returned values are the same as used for error and status codes.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

int           result;
int           fd;
TP816_STATUS  msg_stat;

...

/*--------------------------------------
  Get status of  message object 3
  --------------------------------------*/
msg_stat.message_sel = 3;

result = ioctl(fd, FIO_STATUS, &msg_stat);

...
```

4.4.10 FIO_CAN_STATUS

This function returns the actual contents of the CAN Controller Status Register.

A pointer to an unsigned long variable, that receives the Status Register contents, is passed to the driver by the function-dependent control parameter **arg**.

EXAMPLE

```
#include <vxworks.h>
#include "tp816.h"

...

int               result;
int               fd;
unsigned long     can_state

...

/*-----------------------------------------------
  Get contents of CAN Controller Status Register
  -----------------------------------------------*/
result = ioctl(fd, FIO_CAN_STATUS, &can_state);

...
```

5 Appendix

This chapter describes the symbols which are defined in the file *tpmc816.h*.

5.1 Predefined Symbols

Bit Timing symbol definitions

TP816_1_6MBIT	0x0011	1.6 MBit/s	max. cable length: 20 m
TP816_1MBIT	0x0014	1 MBit/s	max. cable length: 36 m
TP816_500KBIT	0x001c	500 KBit/s	max. cable length: 130 m
TP816_250KBIT	0x011c	250 KBit/s	max. cable length: 270 m
TP816_125KBIT	0x031c	125 KBit/s	max. cable length: 530 m
TP816_100KBIT	0x432f	100 KBit/s	max. cable length: 620 m
TP816_50KBIT	0x472f	50 KBit/s	max. cable length: 1.3 km
TP816_20KBIT	0x532f	20 KBit/s	max. cable length: 3.3 km
TP816_10KBIT	0x672f	10 KBit/s	max. cable length: 6.7 km
TP816_5KBIT	0x7f7f	5 KBit/s	max. cable length: 10 km

I/O control function codes

FIO_FLUSH	9	Flush the read message FIFO
FIO_SETFILTER	11	Setup acceptance filter masks
FIO_BITTIMING	12	Setup CAN Bit Timing parameter
FIO_DEFINE_MSG	13	Define a message object
FIO_UPDATE_MSG	14	Update a message object
FIO_CANCEL_MSG	15	Cancel a message object
FIO_BUSON	16	Enter *Bus On* state
FIO_STATUS	17	Get status information of a message object
FIO_CAN_STATUS	18	Get contents of CAN Controller Status Register
FIO_GETFILTER	19	Get contents of acceptance filter masks

I/O flags

TP816_RECEIVE	(1 << 0)	Message object direction is receive
TP816_TRANSMIT	(1 << 1)	Message object direction is transmit
TP816_REMOTE	(1 << 2)	If direction is receive a remote frame will be transmitted. If direction is transmit the message data will be send after the receive of a remote frame which matches with the identifier.
TP816_FLUSH	(1 << 3)	Flush the read message FIFO
TP816_NOWAIT	(1 << 4)	Do not wait, if the device is blocked or no data are available for a read requests. Return immediately after starting of transmission for write requests.
TP816_THREE_SAMPLES	(1 << 5)	Three samples are used for determining the valid bit value.

5.2 Status and Error Codes

If the device driver creates an error the error codes are stored in the *errno*. They can be read with the VxWorks function *errnoGet()* or *printErrno().*

S_tp816Drv_IDLE	0x00000000	Status of the message object is idle
S_tp816Drv_NXIO	0x08160001	No TPMC816 IP was found at the specified base address
S_tp816Drv_IDEVICE	0x08160002	Invalid or duplicate minor device number
S_tp816Drv_ICMD	0x08160003	Unknown *cntrl* function code
S_tp816Drv_NOTINIT	0x08160004	Device was not initialized
S_tp816Drv_NOMEM	0x08160005	Unable to allocate memory
S_tp816Drv_TIMEOUT	0x08160006	I/O request times out
S_tp816Drv_BUSY	0x08160009	The device or message object is busy
S_tp816Drv_NOSEM	0x0816000A	Unable to create a semaphore
S_tp816Drv_NODATA	0x08160010	No data available, only possible if *TP816_NOWAIT* option selected
S_tp816Drv_IPARAM	0x08160013	Invalid device initialization data
S_tp816Drv_PARAM_MISMATCH	0x08160014	Mismatch of common initialization parameter
S_tp816Drv_OVERRUN	0x08160015	Data overrun
S_tp816Drv_BUSOFF	0x08160016	Controller is in *busoff* state
S_tp816Drv_IMSGNUM	0x08160017	Invalid message object number
S_tp816Drv_MSGBUSY	0x08160018	Message object already defined
S_tp816Drv_MSGNOTDEF	0x08160019	Message object not defined