The Embedded I/O Company

TEWS TECHNOLOGIES

TDRV011-SW-42

VxWorks Device Driver

Extended CAN Bus

Version 1.0.x

User Manual

Issue 1.0.2

January 2007

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

TEWS TECHNOLOGIES LLC
9190 Double Diamond Parkway,
Suite 127, Reno, NV 89521, USA
www.tews.com
Phone: +1 (775) 850 5830
Fax: +1 (775) 201 0347
e-mail: usasales@tews.com

TDRV011-SW-42

VxWorks Device Driver

Extended CAN Bus

Supported Modules:
TPMC316
TPMC816
TPMC901

Issue	Description	Date
1.0.0	First Issue	December 21, 2006
1.0.1	Description: BSP dependent adjustment added	January 25, 2007
1.0.2	Description: BSP dependent adjustment changed	January 29, 2007

Table of Contents

1 Introduction

The TDRV011-SW-42 VxWorks device driver software allows the operation of the supported PMCs conforming to the VxWorks I/O system specification. This includes a device-independent basic I/O interface with *open*(), *close(),* and *ioctl()* functions.

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 from losing data, incoming messages will be stored in a message FIFO with a depth of 100 messages.

The TDRV011-SW-42 device driver supports the following features:

- sending and receiving CAN messages
- extended and standard message frames
- acceptance filtering
- message objects
- remote frame requests and so on

The TDRV011-SW-42 supports the modules listed below:

TPMC316-xx	2 Channel Extended CAN	(PMC, Conduction Cooled)
TPMC816-10	2 Channel Extended CAN	(PMC)
TPMC816-11	1 Channel Extended CAN	(PMC)
TPMC901-10	6 Channel Extended CAN	(PMC)
TPMC901-11	4 Channel Extended CAN	(PMC)
TPMC901-12	2 Channel Extended CAN	(PMC)

In this document all supported modules and devices will be called TDRV011. Specials for a certain devices will be advised.

To get more information about the features and use of supported devices it is recommended to read the manuals listed below.

User manual of the appropriate hardware

Engineering Manual of the appropriate hardware

Architectural Overview of the Intel 82527 CAN controller

2 Installation

Following files are located on the distribution media:

Directory path 'TDRV011-SW-42':

tdrv011drv.c	TDRV011 device driver source
tdrv011def.h	TDRV011 driver include file
tdrv011.h	TDRV011 include file for driver and application
tdrv011pci.c	TDRV011 PCI MMU mapping for Intel x86 based targets
i82527.h	Controller definitions
tdrv011exa.c	Example application
include/tdhal.h	Hardware dependent interface functions and definitions
TDRV011-SW-42-1.0.2.pdf	PDF copy of this manual
ChangeLog.txt	Release history
Release.txt	Release information

2.1 Include device driver in Tornado IDE project

For including the TDRV011-SW-42 device driver into a Tornado IDE project follow the steps below:

(1) Copy the files from the distribution media into a subdirectory in your project path.
(For example: ./TDRV011)

(2) Add the device drivers C-files to your project.
Make a right click to your project in the 'Workspace' window and use the 'Add Files ...' topic.
A file select box appears, and the driver files can be selected.

(3) Now the driver is included in the project and will be built with the project.

For a more detailed description of the project facility please refer to your Tornado User's Guide.

2.2 Special installation for Intel x86 based targets

The TDRV011 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_FAMILY**. If the content of this macro is equal to *I80X86* 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 device memory spaces can't be accessed.

To solve this problem a MMU mapping entry has to be added for the required TDRV011 PCI memory spaces prior the MMU initialization (*usrMmuInit()*) is done.

The C source file **tdrv011pci.c** contains the function *tdrv011PciInit().* This routine finds out all TDRV011 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()*).

The right place to call the function *tdrv011PciInit()* is at the end of the function *sysHwInit()* in **sysLib.c** (it can be opened from the project *Files* window).

Be sure that the function is called prior to MMU initialization otherwise the TDRV011 PCI spaces remains unmapped and an access fault occurs during driver initialization.

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

```
tdrv011PciInit();
```

Modifying the sysLib.c file will change the sysLib.c in the BSP path. Remember this for future projects and recompilations.

2.3 BSP dependent adjustments

The driver includes a file called *include/tdhal.h* which contains functions and definitions for BSP adaptation. It may be necessary to modify them for BSP specific settings. Most settings can be made automatically by conditional compilation set by the BSP header files, but some settings must be configured manually. There are two way of modification, first you can change the *include/tdhal.h* and define the corresponding definition and its value, or you can do it, using the command line option *–D*.

There are 3 offset definitions (*USERDEFINED_MEM_OFFSET*, *USERDEFINED_IO_OFFSET*, and *USERDEFINED_LEV2VEC*) that must be configured if a corresponding warning message appears during compilation. These definitions always need values. Definition values can be assigned by command line option *-D<definition>=<value>*.

definition	**description**
USERDEFINED_MEM_OFFSET	The value of this definition must be set to the offset between CPU-Bus and PCI-Bus Address for PCI memory space access
USERDEFINED_IO_OFFSET	The value of this definition must be set to the offset between CPU-Bus and PCI-Bus Address for PCI I/O space access
USERDEFINED_LEV2VEC	The value of this definition must be set to the difference of the interrupt vector (used to connect the ISR) and the interrupt level (stored to the PCI header)

Another definition allows a simple adaptation for BSPs that utilize a *pciIntConnect()* function to connect shared (PCI) interrupts. If this function is defined in the used BSP, the definition of *USERDEFINED_SEL_PCIINTCONNECT* should be enabled. The definition by command line option is made by *-D<definition>*.

Please refer to the BSP documentation and header files to get information about the interrupt connection function and the required offset values.

2.4 System resource requirement

The table gives an overview over the system resources that will be needed by the driver.

Resource	Driver requirement	Devices requirement
Memory	< 1 KB	*[1])
Stack	< 1 KB	---
Semaphores	*[2])	4

*[1]) The memory requirement mainly depends on the defined FIFO size.

*< ((20Byte * MAX_FIFO_MSG) + 100)*

*[2]) The number of semaphores depends on the maximum number of read and write jobs active at the same time. Each job needs one extra semaphore.

Memory and Stack usage may differ from system to system, depending on the used compiler and its setup.

The following formula shows the way to calculate the common requirements of the driver and devices.

*<total requirement> = <driver requirement> + (<number of devices> * <device requirement>)*

The maximum usage of some resources is limited by adjustable parameters. If the application and driver exceed these limits, increase the according values in your project.

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 tdrv011Drv()

NAME

tdrv011Drv() - installs the TDRV011 driver in the I/O system

SYNOPSIS

#include tdrv011.h

void tdrv011Drv(void)

DESCRIPTION

This function installs the TDRV011 driver in the I/O system. It is the first thing we have to do before adding any device to the system or performing any I/O request.

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

EXAMPLE

```
#include "tdrv011.h"

STATUS            result;

…

/*-------------------
  Initialize Driver
  -------------------*/
result = tdrv011Drv();
if (result == ERROR)
{
     /* Error handling */
}

…
```

RETURNS

OK or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error codes are stored in *errno* and can be read with the function *errnoGet()*.

Error code	Description
S_tdrv011Drv_NOMEM	Can't allocate memory for devices
S_tdrv011Drv_NXIO	No supported module found

SEE ALSO

VxWorks Programmer's Guide: I/O System

3.2 tdrv011DevCreate()

NAME

tdrv011DevCreate() – Add a TDRV011 device to the VxWorks system

SYNOPSIS

```
#include “tdrv011.h”

STATUS tdrv011DevCreate
(
        char            *name,
        unsigned int    devIdx,
        unsigned int    bitTiming
)
```

DESCRIPTION

This routine adds the selected device to the VxWorks system. The device hardware will be setup and prepared for use.

This function must be called before performing any I/O request to this device.

PARAMETER

name

This string specifies the name of the device that will be used to identify the device, for example for *open()* calls.

devIdx

This index number specifies the device to add to the system.
The index number depends on the search priority of the modules. The modules will be searched in the following order:

- TPMC316-xx
- TPMC816-xx
- TPMC901-xx

If modules of the same type are installed the channel numbers will be assigned in the order the VxWorks *pciFindDevice()* function will find the devices.

Example: (A system with 1 TPMC901-10, 1 TPMC816-11, and 2 TPMC316-10) will assign the following device indexes:

Module	Device Index
TPMC316-10 (1st)	0, 1
TPMC316-10 (2nd)	2, 3
TPMC816-11	4
TPMC901-10	5, 6, …, 10

bitTiming

This parameter specifies the initial bit timing of the device. Standard values are defined in tdrv011.h.

EXAMPLE

```
#include "tdrv011.h"

STATUS             result;

…

/*-------------------------------------------------------
  Create the device "/tdrv011/0" for the first CAN device
..* Initial Bittiming: 250 KBps
  -------------------------------------------------------*/
result = tdrv011DevCreate(   "/tdrv011/0",
                             0,
                             TDRV011_250KBIT);
if (result == OK)
{
     /* Device successfully created */
}
else
{
     /* Error occurred when creating the device */
}

…
```

RETURNS

OK or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error codes are stored in *errno* and can be read with the function *errnoGet()*.

Error code	Description
S_tdrv011Drv_NODRV	The driver has not been started
S_tdrv011Drv_NXIO	The specified device is not available

SEE ALSO

VxWorks Programmer's Guide: I/O System

3.3 tdrv011PciInit()

NAME

tdrv011PciInit() – Generic PCI device initialization

SYNOPSIS

void tdrv011PciInit()

DESCRIPTION

This function is required only for Intel x86 VxWorks platforms. The purpose is to setup the MMU mapping for all required TDRV011 PCI spaces (base address register) and to enable the TDRV011 device for access.

The global variable *tdrv011Status* obtains the result of the device initialization and can be polled later by the application before the driver will be installed.

Value	Meaning
> 0	Initialization successful completed. The value of tdrv011Status is equal to the number of mapped PCI spaces
0	No TDRV011 device found
< 0	Initialization failed. The value of (tdrv011Status & 0xFF) is equal to the number of mapped spaces until the error occurs. Possible cause: Too few entries for dynamic mappings in sysPhysMemDesc[]. Remedy: Add dummy entries as necessary (syslib.c).

EXAMPLE

```
extern void tdrv011PciInit();

…

tdrv011PciInit();

…
```

4 I/O Functions

4.1 open()

NAME

open() - open a device or file.

SYNOPSIS

```
int open
(
        const char   *name,
        int          flags,
        int          mode
)
```

DESCRIPTION

Before I/O can be performed to the TDRV011 device, a file descriptor must be opened by invoking the basic I/O function *open()*.

PARAMETER

name

Specifies the device which shall be opened, the name specified in tdrv011DevCreate() must be used

flags

Not used

mode

Not used

EXAMPLE

```
int      fd;

…

/*-----------------------------------------
  Open the device named "/tdrv011/0" for I/O
  -----------------------------------------*/
fd = open("/tdrv011/0", 0, 0);
if (fd == ERROR)
{
     /* Handle error */
}

…
```

RETURNS

A device descriptor number or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error code can be read with the function *errnoGet()*.

The error code is a standard error code set by the I/O system (see VxWorks Reference Manual.

SEE ALSO

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

4.2 close()

NAME

close() – close a device or file

SYNOPSIS

```
STATUS close
(
     int          fd
)
```

DESCRIPTION

This function closes opened devices.

PARAMETER

fd

This file descriptor specifies the device to be closed. The file descriptor has been returned by the *open()* function.

EXAMPLE

```
int       fd;
STATUS    retval;

…

/*----------------
  close the device
  ----------------*/
retval = close(fd);
if (retval == ERROR)
{
     /* Handle error */
}

…
```

RETURNS

OK or ERROR. If the function fails, an error code will be stored in *errno*.

ERROR CODES

The error code can be read with the function *errnoGet()*.

The error code is a standard error code set by the I/O system (see VxWorks Reference Manual).

SEE ALSO

ioLib, basic I/O routine - close()

4.3 ioctl()

NAME

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

SYNOPSIS

```
#include “tdrv011.h”

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

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

fd

This file descriptor specifies the device to be used. The file descriptor has been returned by the *open()* function.

request

This argument specifies the function that shall be executed. Following functions are defined:

Function	Description
TDRV011_READ	Read received CAN message
TDRV011_WRITE	Write CAN message
TDRV011_FLUSH	Flush receive FIFO
TDRV011_SETFILTER	Set acceptance filter
TDRV011_GETFILTER	Get current acceptance filter setting
TDRV011_BITTIMING	Set new bit timing
TDRV011_DEFINE_MSG	Define a CAN message object
TDRV011_UPDATE_MSG	Update a CAN message object
TDRV011_CANCEL_MSG	Cancel a CAN message object
TDRV011_BUSON	Set device BUD ON
TDRV011_STATUS	Get state of a CAN message object
TDRV011_CAN_STATUS	Get state of the CAN controller

arg

This parameter depends on the selected function (request). How to use this parameter is described below with the function.

RETURNS

OK or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error code can be read with the function *errnoGet()*.

The error code is a standard error code set by the I/O system (see VxWorks Reference Manual). Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_ICMD	Unknown ioctl() command specified

SEE ALSO

ioLib, basic I/O routine - ioctl()

4.3.1 TDRV011_READ

This I/O control function reads a CAN message from the specified device. The function specific control parameter **arg** is a pointer on a *TDRV011_IO_BUFFER* structure for this function.

If the device is blocked by another read request or no message is available in the read buffer, the requesting task will be blocked until a message is received or the request times out. If the flag *TDRV011_F_NOWAIT* is set, read returns immediately.

```
typedef struct
{
        unsigned long       flags;
        unsigned long       timeout;
        unsigned long       identifier;
        unsigned char       extended;
        unsigned char       length;
        unsigned char       data[8];
} TDRV011_IO_BUFFER;
```

flags

The parameter flags define a flag field used to control the read operation.
If the flag *'TDRV011_F_FLUSH*' is set, a flush of the device message FIFO will be performed before initiating the read request.
If the flag '*TDRV011_F_NOWAIT*' is set, read returns immediately, if the device is blocked by an other read request or no message is available.

timeout

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

identifier

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

extended

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

length

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

data

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

EXAMPLE

```
#include “tdrv011.h”

int                fd;
int                retval;
int                i;
TDRV011_IO_BUFFER  rw;  /* I/O parameter block for read */


/*------------------------------------------------------------
  Read a message from a TDRV011 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      = TDRV011_F_FLUSH;
rw.timeout    = 500; /* ticks */

retval = ioctl(fd, TDRV011_READ, (int)&rw);
if (retval != ERROR)
{    /* process received message */
     printf("Message received:\n",
     printf("  Identifier: %d ", rw.identifier);
     printf("  Message length: %d byte\n", rw.length);
     printf("  Message data: ");
     for (i = 0; i << rw.length; i++)
     {
          printf("%02Xh", rw.data[i]);
     }
     printf("\n");
}
else
{
     /* handle error */
}
```

ERROR CODES

Error code	Description
S_tdrv011Drv_BUSOFF	The device is in BUS OFF state
S_tdrv011Drv_NODATA	No data is available
S_tdrv011Drv_TIMEOUT	The read request timed out

4.3.2 TDRV011_WRITE

This I/O control function writes a CAN message to specified device. The function specific control parameter **arg** is a pointer on a *TDRV011_IO_BUFFER* structure for this function.

If the device is blocked by another write request the requesting task will be blocked until the request times out. If the flag *TDRV011_F_NOWAIT* is set, write returns immediately.

```
typedef struct
{
        unsigned long     flags;
        unsigned long     timeout;
        unsigned long     identifier;
        unsigned char     extended;
        unsigned char     length;
        unsigned char     data[8];
} TDRV011_IO_BUFFER;
```

flags

The parameter flags define a flag field used to control the write operation.
If the flag '*TDRV011_F_NOWAIT*' is set, write returns immediately, if the device is blocked by other write request.

timeout

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

identifier

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

extended

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

length

The parameter length specifies the size of message data in bytes.

data

The data bytes that shall be send with the message.

EXAMPLE

```
#include “tdrv011.h”

#define            HELLO     "HELLOOOO"

int                fd;
int                retval;
int                i;
TDRV011_IO_BUFFER  rw;  /* I/O parameter block for write */

…

/*-------------------------------------------------------------
  Write a message to a TDRV011 device.
  - if there is a problem with the transmitter the write
    request times out
-------------------------------------------------------------*/
rw.flags      = 0;
rw.timeout    = 100;
rw.identifier = 1234;       /* extended identifier */
rw.extended   = TRUE;
rw.length     = 8;
memcpy(rw.data, HELLO, 8);

retval = ioctl(fd, TDRV011_WRITE, (int)&rw);
if (retval == ERROR)
{
     /* handle error */
}

…
```

ERROR CODES

Error code	Description
S_tdrv011Drv_BUSOFF	The device is in BUS OFF state
S_tdrv011Drv_IPARAM	Illegal parameter value specified
S_tdrv011Drv_TIMEOUT	The write request timed out

4.3.3 TDRV011_FLUSH

This I/O control function flushes the receive FIFO. All messages in the FIFO will be discarded. The function specific control parameter **arg** is not used for this function.

EXAMPLE

```
#include “tdrv011.h”

int                fd;
int                retval;

…

retval = ioctl(fd, TDRV011_FLUSH, 0);
if (retval == ERROR)
{
     /* handle error */
}

…
```

4.3.4 TDRV011_SETFILTER

This I/O control function modifies the acceptance filter masks of the CAN Controller. The function specific control parameter **arg** is a pointer on a *TDRV011*_ACCEPT_MASKS structure for this function.

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 value of ‘1’ means that the incoming bit value "must-match" identically to the corresponding bit in the message identifier.

```
typedef struct
{
        unsigned short          GlobalMaskStandard;
        unsigned long           GlobalMaskExtended;
        unsigned long           Message15Mask;
} TDRV011_ACCEPT_MASKS;
```

GlobalMaskStandard

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.

GlobalMaskExtended

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.

Message15Mask

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 “tdrv011.h”

int                     fd;
int                     retval;
TDRV011_ACCEPT_MASKS    acceptMasks;

…

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 */

retval = ioctl(fd, TDRV011_SETFILTER, (int)&acceptMasks);
if (retval == ERROR)
{
     /* handle error */
}

…
```

4.3.5 TDRV011_GETFILTER

This I/O control function reads the acceptance filter masks from the CAN Controller. The function specific control parameter **arg** is a pointer on a *TDRV011*_ACCEPT_MASKS structure for this function.

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 value of ‘1’ means that the incoming bit value "must-match" identically to the corresponding bit in the message identifier.

```
typedef struct
{
        unsigned short          GlobalMaskStandard;
        unsigned long           GlobalMaskExtended;
        unsigned long           Message15Mask;
} TDRV011_ACCEPT_MASKS;
```

GlobalMaskStandard

The parameter GlobalMaskStandard returns 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.

GlobalMaskExtended

The parameter GlobalMaskExtended returns 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.

Message15Mask

The parameter Message15Mask returns 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 “tdrv011.h”

int                     fd;
int                     retval;
TDRV011_ACCEPT_MASKS    acceptMasks;

…

retval = ioctl(fd, TDRV011_GETFILTER, (int)&acceptMasks);
if (retval == ERROR)
{
     /* handle error */
}
```

4.3.6 TDRV011_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 *TDRV011_ARGS* structure for this function.

```
typedef struct
{
        unsigned long       cmd;
        unsigned long       flags;
        unsigned long       arg;
} TDRV011_ARGS;
```

cmd

The *cmd* argument is not used.

flags

If the I/O flag *TDRV011_F_THREE_SAMPLES* is set in the flags argument the CAN bus is sampled three times per bit time instead of one time.

arg

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 20 KBit per second and 1.0 MBit per second. The include file tdrv011.h contains predefined transfer rates. (For other transfer rates please follow the instructions of the Intel 82527 Architectural Overview)

EXAMPLE

```
#include “tdrv011.h”

int                 fd;
int                 retval;
TDRV011_ARGS        argBuf;

…

/*--------------------------------------
  Setup the transfer rate to 500 KBit/s
  --------------------------------------*/
argBuf.arg    = TDRV011_500KBIT;
argBuf.flags  = 0;

retval = ioctl(fd, TDRV011_BITTIMING, (int)&argBuf);
if (retval == ERROR)
{
     /* handle error */
}
…
```

4.3.7 TDRV011_DEFINE_MSG

This I/O control function allocates and defines parameters of a CAN message object. User-definable message objects are message object 1...13 and 15. Message object 14 is reserved for internal use of the device driver (write). The function specific control parameter **arg** is a pointer on a *TDRV011_ARGS* structure for this function.

A defined message object will be active until the I/O control function *TDRV011_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).

```
typedef struct
{
        unsigned long       cmd;
        unsigned long       flags;
        unsigned long       arg;
} TDRV011_ARGS;
```

cmd

The *cmd* argument is not used.

flags

By combination (binary OR) of the I/O flags *TDRV011_F_RECEIVE*, *TDRV011_F_TRANSMIT* and *TDRV011_F_REMOTE* the application can select one of four possible types of message objects.

TDRV011_F_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.
TDRV011_F_RECEIVE TDRV011_F_REMOTE	This is also a receive object, but a remote frame is sent to request a remote node to send the corresponding data.
TDRV011_F_TRANSMIT	This is a transmit object. The transmission of the message data starts immediately after definition of the message object.
TDRV011_F_TRANSMIT TDRV011_F_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.

arg

A pointer to a message object description data structure *TDRV011_MSGDEF* is passed to the driver by the function-dependent parameter *arg*.

```
typedef struct
{
        unsigned char     MsgNum;
        unsigned long     identifier;
        unsigned char     extended;
        unsigned char     length;
        unsigned char     data[8];
} TDRV011_MSGDEF;
```

MsgNum

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

identifier

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

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.

length

The parameter length is only necessary for transmit message objects, if *TDRV011_F_TRANSMIT* is set. The parameter length specifies the number of bytes in data.

data

The parameter data is only used for transmit objects, it holds the message data.

EXAMPLE

```
#include "tdrv011.h"

int                fd;
int                retval;
TDRV011_MSGDEF     MsgDef;
TDRV011_ARGS       argBuf;

…
```

```
…

/*-------------------------------------------------------------
  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.
  -------------------------------------------------------------*/
argBuf.arg         = (unsigned long)&MsgDef;
argBuf.flags       = TDRV011_F_RECEIVE;

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

retval = ioctl(fd, TDRV011_DEFINE_MSG, (int)&argBuf);
if (retval == ERROR)
{
     /* handle error */
}

…

argBuf.arg         = (unsigned long)&MsgDef;
argBuf.flags  = TDRV011_F_RECEIVE | TDRV011_F_REMOTE;

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

retval = ioctl(fd, TDRV011_DEFINE_MSG, (int)&argBuf);
if (retval == ERROR)
{
     /* handle error */
}

…
```

```
…

argBuf.arg         = (unsigned long)&MsgDef;
argBuf.flags   = TDRV011_F_TRANSMIT;

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

retval = ioctl(fd, TDRV011_DEFINE_MSG, (int)&argBuf);
if (retval == ERROR)
{
     /* handle error */
}

…

argBuf.arg         = (unsigned long)&MsgDef;
argBuf.flags   = TDRV011_F_TRANSMIT | TDRV011_F_REMOTE;

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

retval = ioctl(fd, TDRV011_DEFINE_MSG, (int)&argBuf);
if (retval == ERROR)
{
     /* handle error */
}

…
```

ERROR CODES

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified
S_tdrv011Drv_MSGBUSY	The specified message is already in use

4.3.8 TDRV011_UPDATE_MSG

This I/O control function updates the message data of a previously defined transmission object or starts transmission of a remote frame for receive message objects. The function specific control parameter **arg** is a pointer on a *TDRV011_ARGS* structure for this function.

The status of message object (for example transmission completed) can be determined by use of the I/O control function '*TDRV011_STATUS*'

```
typedef struct
{
        unsigned long       cmd;
        unsigned long       flags;
        unsigned long       arg;
} TDRV011_ARGS;
```

cmd

The *cmd* argument is not used.

flags

If the I/O flags *TDRV011_REMOTE* is set, transmission will be started by a request of a remote node, otherwise transmission of the message data starts immediately.

arg

A pointer to a message object description data structure *TDRV011_MSGDEF* is passed to the driver by the function-dependent parameter *arg*.

```
typedef struct
{
        unsigned char       MsgNum;
        unsigned long       identifier;
        unsigned char       extended;
        unsigned char       length;
        unsigned char       data[8];
} TDRV011_MSGDEF;
```

MsgNum

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

identifier

The parameter is unused.

extended

The parameter is unused.

length

This parameter specifies the number of bytes the new data contains.

data

The buffer data contains the new message data.

EXAMPLE

```
#include “tdrv011.h”

int                fd;
int                retval;
TDRV011_MSGDEF     MsgDef;
TDRV011_ARGS       argBuf;

…

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

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

retval = ioctl(fd, TDRV011_UPDATE_MSG, (int)&argBuf);
if (retval == ERROR)
{
     /* handle error */
}

…
```

ERROR CODES

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified
S_tdrv011Drv_MSGBUSY	The specified message is in use
S_tdrv011Drv_MSGNOTDEF	The message object has not been defined

4.3.9 TDRV011_CANCEL_MSG

This I/O control function marks the specified message object as invalid and stops transaction regarding the object. The function specific control parameter **arg** specifies the new message object number. Allowed message numbers are 1…13 and 15.

EXAMPLE

```
#include “tdrv011.h”

int                fd;
int                retval;

…

/*------------------------
  Cancel message object 3
  ------------------------*/
retval = ioctl(fd, TDRV011_CANCEL_MSG, 3);
if (retval == ERROR)
{
     /* handle error */
}

…
```

ERROR CODES

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified

4.3.10 TDRV011_BUSON

This function set the specified device to buson state. The function specific control parameter **arg** is not used for this function.

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 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

EXAMPLE

```
#include “tdrv011.h”

int                fd;
int                retval;

…

retval = ioctl(fd, TDRV011_BUSON, 0);
if (retval == ERROR)
{
     /* handle error */
}

…
```

ERROR CODES

Error code	Description
S_tdrv011Drv_BUSOFF	The device can not be set buson

4.3.11 TDRV011_STATUS

This I/O control function returns the actual state of the specified transmit message object. The function specific control parameter **arg** is a pointer on a *TDRV011_STAT* structure for this function.

The status of message object (for example transmission completed) can be determined by use of the I/O control function '*TDRV011_STATUS*'

```
typedef struct
{
        unsigned long       message_sel;
        unsigned long       status;
} TDRV011_STAT;
```

message_sel

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 *TDRV011_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 *TDRV011_F_NOWAIT* set.

status

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

EXAMPLE

```
#include "tdrv011.h"

int                 fd;
int                 retval;
TDRV011_STAT        msgStat;

…

/*-----------------------------
  Get state of message object 3
  -----------------------------*/
msgStat.message_sel = 3;

retval = ioctl(fd, TDRV011_STATUS, (int)&msgStat);
if (retval == ERROR)
{
     /* handle error */
}

…
```

ERROR CODES

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified
S_tdrv011Drv_MSGNOTDEF	The message object has not been defined
S_tdrv011Drv_IDLE	Status of the message object is idle
S_tdrv011Drv_BUSY	The device or message object is busy

4.3.12 TDRV011_CAN_STATUS

This I/O control function returns the contents of the CAN controller status register. The function specific control parameter **arg** is a pointer to an unsigned long value which will be set to the registers content.

EXAMPLE

```
#include “tdrv011.h”

int                fd;
int                retval;
unsigned long      canStat;

…

retval = ioctl(fd, TDRV011_CAN_STATUS, (int)&canStat);
if (retval == ERROR)
{
     /* handle error */
     printf("CAN-state: %Xh", canStat);
}

…
```

5 Appendix

5.1 Additional Error Codes

Error code	Error value	Description
S_tdrv011Drv_IDLE	0x00000000	Status of the message object is idle
S_tdrv011Drv_NXIO	0x00110001	No TDRV011 device found at the specified base address
S_tdrv011Drv_IDEVICE	0x00110002	Invalid or duplicate minor device number
S_tdrv011Drv_ICMD	0x00110003	Unknown ioctl() function code
S_tdrv011Drv_NOTINIT	0x00110004	Device was not initialized
S_tdrv011Drv_NOMEM	0x00110005	Unable to allocate memory
S_tdrv011Drv_TIMEOUT	0x00110006	I/O request times out
S_tdrv011Drv_BUSY	0x00110009	The device or message object is busy
S_tdrv011Drv_NOSEM	0x0011000A	Unable to create a semaphore
S_tdrv011Drv_NODATA	0x00110010	No data available, only possible if '*TDRV011_F_NOWAIT*' option selected
S_tdrv011Drv_IPARAM	0x00110013	Invalid device initialization data
S_tdrv011Drv_PARAM_MISMATCH	0x00110014	Mismatch of common initialization parameter
S_tdrv011Drv_OVERRUN	0x00110015	Data overrun
S_tdrv011Drv_BUSOFF	0x00110016	Controller is in busoff state
S_tdrv011Drv_IMSGNUM	0x00110017	Invalid message object number
S_tdrv011Drv_MSGBUSY	0x00110018	Message object already defined
S_tdrv011Drv_MSGNOTDEF	0x00110019	Message object not defined
S_tdrv011Drv_NODRV	0x0011001A	Driver has not been installed