The Embedded I/O Company

TEWS TECHNOLOGIES

TDRV011-SW-42

VxWorks Device Driver

Extended CAN Bus

Version 2.0.x

User Manual

Issue 2.0.0

June 2010

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

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
1.1.0	BUSOFF function added	November 21, 2008
1.2.0	Transmit Message Object configuration added	April 23, 2009
2.0.0	VxBus, SMP Support and API description added	June 7, 2010

Table of Contents

1 Introduction

1.1 Device Driver

The TDRV011-SW-42 VxWorks device driver software allows the operation of the supported PMC conforming to the VxWorks I/O system specification.

The TDRV011-SW-42 release contains independent driver sources for the old legacy (pre-VxBus) and the new VxBus-enabled driver model. The VxBus-enabled driver is recommended for new developments with later VxWorks 6.x release and mandatory for VxWorks SMP systems.

Both drivers, legacy and VxBus, share the same application programming interface (API) and device-independent basic I/O interface with open(), close() and ioctl() functions. The basic I/O interface is only for backward compatibility with existing applications and should not be used for new developments.

Both drivers invoke 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

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 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':

TDRV011-SW-42-2.0.0.pdf	PDF copy of this manual
TDRV011-SW-42-VXBUS.zip	Zip compressed archive with VxBus driver sources
TDRV011-SW-42-LEGACY.zip	Zip compressed archive with legacy driver sources
ChangeLog.txt	Release history
Release.txt	Release information

The archive TDRV011-SW-42-VXBUS.zip contains the following files and directories:

Directory path './tews/tdrv011':

tdrv011drv.c	TDRV011 device driver source
tdrv011def.h	TDRV011 driver include file
tdrv011.h	TDRV011 include file for driver and application
tdrv011api.c	TDRV011 API file
i82527.h	Controller definitions
Makefile	Driver Makefile
40tdrv011.cdf	Component descriptions file for VxWorks development tools
tdrv011.dc	Configuration stub file for direct BSP builds
tdrv011.dr	Configuration stub file for direct BSP builds
include/tvxbHal.h	Hardware dependent interface functions and definitions
apps/tdrv011exa.c	Example application

The archive TDRV011-SW-42-LEGACY.zip contains the following files and directories:

Directory path './tdrv011':

tdrv011drv.c	TDRV011 device driver source
tdrv011def.h	TDRV011 driver include file
i82527.h	Controller definitions
tdrv011.h	TDRV011 include file for driver and application
tdrv011pci.c	TDRV011 device driver source for x86 based systems
tdrv011api.c	TDRV011 API file
tdrv011exa.c	Example application
include/tdhal.h	Hardware dependent interface functions and definitions

2.1 Legacy vs. VxBus Driver

In later VxWorks 6.x releases, the old VxWorks 5.x legacy device driver model was replaced by VxBus-enabled device drivers. Legacy device drivers are tightly coupled with the BSP and the board hardware. The VxBus infrastructure hides all BSP and hardware differences under a well defined interface, which improves the portability and reduces the configuration effort. A further advantage is the improved performance of API calls by using the method interface and bypassing the VxWorks basic I/O interface.

VxBus-enabled device drivers are the preferred driver interface for new developments.

The checklist below will help you to make a decision which driver model is suitable and possible for your application:

Legacy Driver	VxBus Driver
▪ VxWorks 5.x releases ▪ VxWorks 6.5 and earlier releases ▪ VxWorks 6.x releases without VxBus PCI bus support	▪ VxWorks 6.6 and later releases with VxBus PCI bus ▪ SMP systems (only the VxBus driver is SMP safe!)

TEWS TECHNOLOGIES recommends not using the VxBus Driver before VxWorks release 6.6. In previous releases required header files are missing and the support for 3rd-party drivers may not be available.

2.2 VxBus Driver Installation

Because Wind River doesn't provide a standard installation method for 3rd party VxBus device drivers the installation procedure needs to be done manually.

In order to perform a manual installation extract all files from the archive TDRV011-SW-42-VXBUS.zip to the typical 3rd party directory *installDir/vxworks-6.x/target/3rdparty* (whereas *installDir* must be substituted by the VxWorks installation directory).

After successful installation the TDRV011 device driver is located in the vendor and driver-specific directory *installDir/vxworks-6.x/target/3rdparty/tews/tdrv011.*

At this point the TDRV011 driver is not configurable and cannot be included with the kernel configuration tool in a Wind River Workbench project. To make the driver configurable the driver library for the desired processer (CPU) and build tool (TOOL) must be built in the following way:

(1) Open a VxWorks development shell (e.g. C:\WindRiver\wrenv.exe -p vxworks-6.7)

(2) Change into the driver installation directory
installDir/vxworks-6.x/target/3rdparty/tews/tdrv011

(3) Invoke the build command for the required processor and build tool
make CPU=cpuName TOOL=tool

For Windows hosts this may look like this:

```
C:> cd \WindRiver\vxworks-6.7\target\3rdparty\tews\tdrv011
```

```
C:> make CPU=PENTIUM4 TOOL=diab
```

To compile SMP-enabled libraries, the argument VXBUILD=SMP must be added to the command line

```
C:> make CPU=PENTIUM4 TOOL=diab VXBUILD=SMP
```

To integrate the TDRV011 driver with the VxWorks development tools (Workbench), the component configuration file *40tdrv011.cdf* must be copied to the directory *installDir/vxworks-6.x/target/config/comps/VxWorks*.

```
C:> cd \WindRiver\vxworks-6.7\target\3rdparty\tews\tdrv011
C:> copy 40tdrv011.cdf \Windriver\vxworks-6.7\target\config\comps\vxWorks
```

In VxWorks 6.7 and newer releases the kernel configuration tool scans the CDF file automatically and updates the *CxrCat.txt* cache file to provide component parameter information for the kernel configuration tool as long as the timestamp of the copied CDF file is newer than the one of the *CxrCat.txt*. If your copy command preserves the timestamp, force to update the timestamp by a utility, such as *touch*.

In earlier VxWorks releases the CxrCat.txt file may not be updated automatically. In this case, remove or rename the original *CxrCat.txt* file and invoke the make command to force recreation of this file.

```
C:> cd \Windriver\vxworks-6.7\target\config\comps\vxWorks
C:> del CxrCat.txt
C:> make
```

After successful completion of all steps above and restart of the Wind River Workbench, the TDRV011 driver and API can be included in VxWorks projects by selecting the *"TEWS TDRV011 Driver" and "TEWS TDRV011 API"* components in the *"hardware (default) - Device Drivers"* folder with the kernel configuration tool.

2.2.1 Direct BSP Builds

In development scenarios with the direct BSP build method without using the Workbench or the vxprj command-line utility, the TDRV011 configuration stub files must be copied to the directory *installDir/vxworks-6.x/target/config/comps/src/hwif.* Afterwards the *vxbUsrCmdLine.c* file must be updated by invoking the appropriate make command.

```
C:> cd \WindRiver\vxworks-6.7\target\3rdparty\tews\tdrv011
C:> copy tdrv011.dc \Windriver\vxworks-6.7\target\config\comps\src\hwif
C:> copy tdrv011.dr \Windriver\vxworks-6.7\target\config\comps\src\hwif

C:> cd \Windriver\vxworks-6.7\target\config\comps\src\hwif
C:> make vxbUsrCmdLine.c
```

2.3 Legacy Driver Installation

2.3.1 Include device driver in VxWorks projects

For including the TDRV011-SW-42 device driver into a VxWorks project (e.g. Tornado IDE or Workbench) follow the steps below:

(1) Extract all files from the archive TDRV011-SW-42-LEGACY.zip to your project directory.

(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 in the tdrv011 directory 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 VxWorks User's Guide (e.g. Tornado, Workbench, etc.)

2.3.2 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.

2.3.3 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.

2.3.4 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 KB
Stack	< 1 KB	---
Semaphores	2	1

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 API Documentation

3.1 General Functions

3.1.1 tdrv011Open()

Name

tdrv011Open() – opens a device.

Synopsis

```
TDRV011_DEV tdrv011Open
(
        char            *DeviceName
)
```

Description

Before I/O can be performed to a device, a file descriptor must be opened by a call to this function.

Parameters

DeviceName

This parameter points to a null-terminated string that specifies the name of the device. The naming convention for Legacy and VxBus devices is slightly differently. Basically the device name consists of the three parts, a prefix and a 0-based module and channel number separated by slash characters (e.g. “/tdrv011/0/0”, “tdrv011/0/1” etc).
For the VxBus driver the first probed TDRV011 module gets the module number 0 the second probed TDRV011 module gets module number 1 and so forth. The CAN channels on those TDRV011 modules are numbered from 0 for the first channel to <n> for the last channel, where <n> depends on the number CAN channels provided by the TDRV011 module.

For a configuration with a 2-channel TPMC816 (first module) and a 6-channel TPMC901 the following devices will be created:
TPMC816: /tdrv011/0/0, /tdrv011/0/1
TPMC901: /tdrv011/1/0, /tdrv011/1/1, /tdrv011/1/2, /tdrv011/1/3, /tdrv011/1/4, /tdrv011/1/5

For the Legacy driver the device name is specified by the application program during device creation with the function tdrv011DevCreate(). For devices created by the TDRV011 example application the module number is always 0 and the channels are numbered from 0 for the first channel on the first TDRV011 module to <n> for the last channel on the last TDRV011 module, where <n> depends on the overall number of CAN channels provided by all TDRV011 modules.

For the example configuration above the following devices will be created:
TPMC816: /tdrv011/0/0, /tdrv011/0/1
TPMC901: /tdrv011/0/2, /tdrv011/0/3, /tdrv011/0/4, /tdrv011/0/5, /tdrv011/0/6, /tdrv011/0/7

To see which TDRV011 devices are available in the system call the “devs” command from the VxWorks shell. To get detailed information of associated TDRV011 devices the tdrv011Show function (only VxBus driver) can be called form the VxWorks shell.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;

/*
** open file descriptor to a device
*/
pDev = tdrv011Open(“/tdrv011/0/0”);
if (pDev == NULL)
{
     /* handle open error */
}
```

RETURNS

A device descriptor pointer, or NULL if the function fails. An error code will be stored in *errno*.

ERROR CODES

The error codes are stored in *errno.*

The error code is a standard error code set by the I/O system.

3.1.2 tdrv011Close()

Name

tdrv011Close() – closes a device.

Synopsis

```
int tdrv011Close
(
     TDRV011_DEV   pDev
)
```

Description

This function closes previously opened devices.

Parameters

pDev

This value specifies the file descriptor pointer to the hardware module retrieved by a call to the corresponding open-function.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
int           result;

/*
** close file descriptor to device
*/
result = tdrv011Close(pDev);
if (result < 0)
{
     /* handle close error */
}
```

RETURNS

Zero, or -1 if the function fails. An error code will be stored in *errno*.

ERROR CODES

The error codes are stored in *errno*.

The error code is a standard error code set by the I/O system.

3.2 Device Access Functions

3.2.1 tdrv011Read

Name

tdrv011Read – read a CAN message

Synopsis

```
STATUS tdrv011Read
(
    TDRV011_DEV     pDev,
    Int             flushBeforeRead,
    int             timeout,
    unsigned long   *pIdentifier,
    int             *pExtended,
    int             *pLength,
    unsigned char   *pData
);
```

Description

This function reads a CAN message from the specified channel. If no message is available the call is blocked until a message is received or the request times out.

Before the driver can receive CAN messages it's necessary to define at least one receive message object.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

flushBeforeRead

If the message FIFO should be flushed before initiating the read sequence this parameter must be set to TRUE otherwise to FALSE.

timeout

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

pIdentifier

This parameter is a pointer to an unsigned long (32bit) value where the received CAN message identifier is stored.

pExtended

This parameter is a pointer to an int value. The value is set to 0 if a standard message frame was received and >0 if an extended message frame was received.

pLength

This parameter is a pointer to an int value where the length of the received CAN message (number of bytes) is stored. Possible values are 0..8.

pData

This parameter is a pointer to an unsigned char array where the received CAN message is stored. This buffer receives up to 8 data bytes. pData[0] receives message Data 0, pData[1] receives message Data 1 and so on.

Example

```
#include “tdrv011.h”

TDRV011_DEV        pDev;
STATUS             result;
unsigned long      msgIdentifier;
int                msgExtended;
int                msgLength;
unsigned char      msgData[TDRV011_MSG_LEN];

/*
** Read a CAN message from the device. Don’t flush before reading.
** If no message is available the read request times out after 120 ticks.
*/
result = tdrv011Read(   pDev,
                        FALSE,
                        120,
                        &msgIdentifier,
                        &msgExtended,
                        &msgLength,
                        &msgData[0]
                        );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_BUSOFF	The device is in BUS OFF state
S_tdrv011Drv_TIMEOUT	The read request timed out

3.2.2 tdrv011ReadNoWait

Name

tdrv011ReadNoWait – read a CAN message (non-blocked)

Synopsis

```
STATUS tdrv011ReadNoWait
(
        TDRV011_DEV     pDev,
        unsigned long   *pIdentifier,
        int             *pExtended,
        int             *pLength,
        unsigned char   *pData
);
```

Description

This function reads a CAN message from the specified channel. If no message is available the call returns immediately.

Before the driver can receive CAN messages it's necessary to define at least one receive message object.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

pIdentifier

This parameter is a pointer to an unsigned long (32bit) value where the received CAN message identifier is stored.

pExtended

This parameter is a pointer to an int value. The value is set to 0 if a standard message frame was received and >0 if an extended message frame was received.

pLength

This parameter is a pointer to an int value where the length of the received CAN message (number of bytes) is stored. Possible values are 0..8.

pData

This parameter is a pointer to an unsigned char array where the received CAN message is stored. This buffer receives up to 8 data bytes. pData[0] receives message Data 0, pData[1] receives message Data 1 and so on.

Example

```
#include “tdrv011.h”

TDRV011_DEV        pDev;
STATUS             result;
unsigned long      msgIdentifier;
int                msgExtended;
int                msgLength;
unsigned char      msgData[TDRV011_MSG_LEN];

/*
** Read a CAN message from the device.
*/
result = tdrv011ReadNoWait(  pDev,
                             &msgIdentifier,
                             &msgExtended,
                             &msgLength,
                             &msgData[0]
                             );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_BUSOFF	The device is in BUS OFF state
S_tdrv011Drv_ NODATA	No data available

3.2.3 tdrv011Write

Name

tdrv011Write – write a CAN message

Synopsis

```
STATUS tdrv011Write
(
        TDRV011_DEV     pDev,
        int             timeout,
        unsigned long   identifier,
        int             extended,
        int             length,
        unsigned char   *pData
);
```

Description

This function writes a messages to the specified device for subsequent transmission on the CAN bus. The request will be blocked until the message was send or an error occurs.

By default the write function uses message object 1 for transmit. If this isn't suitable the default transmit message object can be changed by redefining the macro TX_MSG_OBJ in tdrv011def.h. Valid message objects are in the range between 1 and 14.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

timeout

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

identifier

Contains the message identifier of the CAN message to write.

extended

Set this parameter to TRUE to send an extended message frame or to FALSE to send a standard message frame.

length

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

pData

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

Example

```
#include “tdrv011.h”

TDRV011_DEV        pDev;
STATUS             result;
unsigned long      msgIdentifier;
int                msgExtended;
int                msgLength;
unsigned char      msgData[TDRV011_MSG_LEN];

/*
** Write a CAN message
*/
msgIdentifier = 1234;
msgExtended   = TRUE;
msgLength     = 2;
msgData[0]    = 0xaa;
msgData[1]    = 0xbb;

result = tdrv011Write(  pDev,
                        120,
                        msgIdentifier,
                        msgExtended,
                        msgLength,
                        &msgData[0]
                        );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

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

3.2.4 tdrv011WriteNoWait

Name

tdrv011WriteNoWait – write a CAN message (non-blocked)

Synopsis

```
STATUS tdrv011WriteNoWait
(
        TDRV011_DEV     pDev,
        unsigned long   identifier,
        int             extended,
        int             length,
        unsigned char   *pData
);
```

Description

This function writes a messages to the specified device for subsequent transmission on the CAN bus. The call will return immediately after the message was written to the CAN controller or an error occurred.

By default the write function use message object 1 for transmit. If this isn't suitable the default transmit message object can be changed by redefining the macro TX_MSG_OBJ in tdrv011def.h. Valid message object are in the range between 1 and 14.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

identifier

Contains the message identifier of the CAN message to write.

extended

Set this parameter to TRUE to send an extended message frame or to FALSE to send a standard message frame.

length

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

pData

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

Example

```
#include “tdrv011.h”

TDRV011_DEV        pDev;
STATUS             result;
unsigned long      msgIdentifier;
int                msgExtended;
int                msgLength;
unsigned char      msgData[TDRV011_MSG_LEN];

/*
** Write a CAN message to the device.
*/
msgIdentifier = 1234;
msgExtended   = TRUE;
msgLength     = 2;
msgData[0]    = 0xaa;
msgData[1]    = 0xbb;

result = tdrv011WriteNoWait( pDev,
                             msgIdentifier,
                             msgExtended,
                             msgLength,
                             &msgData[0]
                             );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_BUSOFF	The device is in BUS OFF state
S_tdrv011Drv_IPARAM	Illegal parameter value specified
S_tdrv011Drv_TIMEOUT	The device is busy and cannot send the message immediatly

3.2.5 tdrv011SetFilter

Name

tdrv011SetFilter – set acceptance filter masks

Synopsis

```
STATUS tdrv011SetFilter
(
    TDRV011_DEV     pDev,
    unsigned short  globalMaskStandard,
    unsigned long   globalMaskExtended,
    unsigned long   message15Mask
);
```

Description

This function modifies the acceptance filter masks of the specified 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 value of ‘1’ means that the incoming bit value "must-match" identically to the corresponding bit in the message identifier.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

globalMaskStandard

This parameter contains 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 appears in bit position 5..15.

globalMaskExtended

This parameter contains the value for the Global Mask-Extended Register. The Global Mask-Extended Register applies only to messages using the extended CAN identifier. The 29 bit identifier appears in bit position 3..31.

message15Mask

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

A detailed description of the acceptance filter can be found in the Intel 82527 Architectural Overview - Acceptance Filtering Implications.

Example

```
#include “tdrv011.h”

TDRV011_DEV        pDev;
STATUS             result;
unsigned short     globalMaskStandard;
unsigned long      globalMaskExtended;
unsigned long      message15Mask;

/*
** Set acceptance filter.
*/
globalMaskStandard = 0xffff;
globalMaskExtended = 0xffffffff;
message15Mask      = 0;

result = tdrv011SetFilter(   pDev,
                             globalMaskStandard,
                             globalMaskExtended,
                             message15Mask
                             );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

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

This function returns no driver specific error codes.

3.2.6 tdrv011GetFilter

Name

tdrv011GetFilter – get acceptance filter masks

Synopsis

```
STATUS tdrv011GetFilter
(
        TDRV011_DEV     pDev,
        unsigned short  *pGlobalMaskStandard,
        unsigned long   *pGlobalMaskExtended,
        unsigned long   *pMessage15Mask
);
```

Description

This function reads the acceptance filter masks from the specified 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 value of ‘1’ means that the incoming bit value "must-match" identically to the corresponding bit in the message identifier.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

pGlobalMaskStandard

This parameter is a pointer to an unsigned short (16bit) value where the contents of the Global Mask-Standard Register is stored. The 11 bit identifier appears in bit position 5..15.

pGlobalMaskExtended

This parameter is a pointer to an unsigned long (32bit) value where the contents of the Global Mask-Extended Register is stored. The 29 bit identifier appears in bit position 3..31.

pMessage15Mask

This parameter is a pointer to an unsigned long (32bit) value where the contents of the Message 15 Mask Register is stored. The 29 bit identifier appears in bit position 3..31.

A detailed description of the acceptance filter can be found in the Intel 82527 Architectural Overview - Acceptance Filtering Implications.

Example

```
#include “tdrv011.h”

TDRV011_DEV        pDev;
STATUS             result;
unsigned short     globalMaskStandard;
unsigned long      globalMaskExtended;
unsigned long      message15Mask;

/*
** Set acceptance filter.
*/

result = tdrv011GetFilter(   pDev,
                             &globalMaskStandard,
                             &globalMaskExtended,
                             &message15Mask
                             );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

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

This function returns no driver specific error codes.

3.2.7 tdrv011SetBitTiming

Name

tdrv011SetBitTiming – set bit timing register

Synopsis

```
STATUS tdrv011SetBitTiming
(
        TDRV011_DEV     pDev,
        unsigned short  timingValue,
        int             useThreeSamples
);
```

Description

This function modifies the bit timing register of the CAN controller to setup a new CAN bus transfer speed.
Keep in mind to setup a valid bit timing before entering the BUSON state.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

timingValue

This parameter contains the new value for the bit timing register 0 (bit 8…15) and 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).

useThreeSamples

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

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

Example

```
#include “tdrv011.h”

TDRV011_DEV        pDev;
STATUS             result;

/*
** Set CAN bus bit timing
*/

result = tdrv011SetBitTiming(pDev,
                             TDRV011_100KBIT,
                             FALSE
                             );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

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

This function returns no driver specific error codes.

3.2.8 tdrv011DefineReceiveMsgObj

Name

tdrv011DefineReceiveMsgObj – setup a message object for receive

Synopsis

```
STATUS tdrv011DefineReceiveMsgObj
(
    TDRV011_DEV     pDev,
    int             messageNumber,
    unsigned long   identifier,
    int             extended
);
```

Description

This function setup a free controller message object to receive CAN messages with the specified identifier.
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.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

messageNumber

This parameter contains the number of the message object to define. Valid numbers are in the range between 1 and 15 with exception of the number of the default transmit object (usually 1).

identifier

This parameter specifies the message identifier that should be received by this message object. Depending on the acceptance filter configuration this initial message identifier value may be changed by other accepted messages with different identifiers. This may cause confusion after changing the acceptance filter masks without redefining the receive message objects.

extended

Set to TRUE to receive extended message frames or to FALSE to receive standard message frames.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
int           msgNum;
unsigned long msgIdentifier;

/*
** Define message object 15 to receive extended messages with the
** specified identifier
*/
msgNum = 15;
msgIdentifier = 1;

result = tdrv011DefineReceiveMsgObj(  pDev,
                                      msgNum,
                                      msgIdentifier,
                                      TRUE
                                      );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified (e.g. transmit message object number
S_tdrv011Drv_MSGBUSY	The specified message is already in use

3.2.9 tdrv011DefineTransmitMsgObj

Name

tdrv011DefineTransmitMsgObj – setup a message object for transmit

Synopsis

```
STATUS tdrv011DefineTransmitMsgObj
(
    TDRV011_DEV     pDev,
    int             messageNumber,
    unsigned long   identifier,
    int             extended,
    int             length,
    unsigned char   *pData
);
```

Description

This function setup a free controller message object to transmit CAN messages immediately without waiting for an acknowledge.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

messageNumber

This parameter contains the number of the message object to define. Valid numbers are in the range between 1 and 14.

identifier

Contains the message identifier of the CAN message to write.

extended

Set this parameter to TRUE to send an extended message frame or to FALSE to send a standard message frame.

length

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

pData

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

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
int           msgNum;
unsigned long msgIdentifier;
int           msgExtended;
int           msgLength;
unsigned char msgData[TDRV011_MSG_LEN];

/*
** Define a transmit buffer object to send the CAN message
** immediately
*/
msgNum        = 2;
msgIdentifier = 1234;
msgExtended   = TRUE;
msgLength     = 2;
msgData[0]    = 0xaa;
msgData[1]    = 0xbb;

result = tdrv011DefineTransmitMsgObj( pDev,
                                      msgNum,
                                      msgIdentifier,
                                      msgExtended,
                                      msgLength,
                                      &msgData[0]
                                      );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified (e.g. transmit message object number
S_tdrv011Drv_MSGBUSY	The specified message is already in use

3.2.10 tdrv011DefineRemoteMsgObj

Name

tdrv011DefineRemoteMsgObj – setup a remote transmit buffer message object

Synopsis

```
STATUS tdrv011DefineRemoteMsgObj
(
    TDRV011_DEV     pDev,
    int             messageNumber,
    unsigned long   identifier,
    int             extended,
    int             length,
    unsigned char   *pData
);
```

Description

This function setup a free controller message object as remote transmission buffer.

A remote transmit buffer 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 tdrv011UpdateRemoteMsgObj or released with tdrv011ReleaseMsgObj.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

messageNumber

This parameter contains the number of the message object to define. Valid numbers are in the range between 1 and 14.

identifier

Contains the message identifier of the CAN message to write.

extended

Set this parameter to TRUE to send extended message frames or to FALSE to send standard message frames.

length

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

pData

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

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
int           msgNum;
unsigned long msgIdentifier;
int           msgExtended;
int           msgLength;
unsigned char msgData[TDRV011_MSG_LEN];

/*
** Define a remote buffer message object
*/
msgNum        = 4;
msgIdentifier = 1234;
msgExtended   = TRUE;
msgLength     = 2;
msgData[0]    = 0xaa;
msgData[1]    = 0xbb;

result = tdrv011DefineRemoteMsgObj(   pDev,
                                      msgNum,
                                      msgIdentifier,
                                      msgExtended,
                                      msgLength,
                                      &msgData[0]
                                      );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified (e.g. transmit message object number
S_tdrv011Drv_MSGBUSY	The specified message is already in use

3.2.11 tdrv011UpdateTransmitMsgObj

Name

tdrv011UpdateTransmitMsgObj – update transmit message object data

Synopsis

```
STATUS tdrv011UpdateTransmitMsgObj
(
    TDRV011_DEV     pDev,
    int             messageNumber,
    int             length,
    unsigned char   *pData
);
```

Description

This function writes new data into the previous allocated transmit message object and starts the transmission immediately.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

messageNumber

This parameter contains the number of the previous defined transmit message object. Valid numbers are in the range between 1 and 14.

length

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

pData

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

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
int           msgNum;
int           msgLength;
unsigned char msgData[TDRV011_MSG_LEN];

/*
** Update a transmit buffer object and start transmission
** immediately
*/
msgNum        = 2;
msgLength     = 2;
msgData[0]    = 0x12;
msgData[1]    = 0x34;

result = tdrv011UpdateTransmitMsgObj( pDev,
                                      msgNum,
                                      msgLength,
                                      &msgData[0]
                                      );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified
S_tdrv011Drv_MSGNOTDEF	The message is not properly setup for transmit
S_tdrv011Drv_MSGBUSY	The specified message is already in use

3.2.12 tdrv011UpdateRemoteMsgObj

Name

tdrv011UpdateRemoteMsgObj – update remote buffer message object data

Synopsis

```
STATUS tdrv011UpdateRemoteMsgObj
(
    TDRV011_DEV     pDev,
    int             messageNumber,
    int             length,
    unsigned char   *pData
);
```

Description

This function writes only new data into the previous allocated remote buffer message object without starting transmission. The data can be requested by other CAN nodes by sending a RTR message with the same identifier.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

messageNumber

This parameter contains the number of the previous defined remote buffer message object. Valid numbers are in the range between 1 and 14.

length

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

pData

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

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
int           msgNum;
int           msgLength;
unsigned char msgData[TDRV011_MSG_LEN];

/*
** Update a remote buffer message object
*/
msgNum        = 2;
msgLength     = 2;
msgData[0]    = 0x12;
msgData[1]    = 0x34;

result = tdrv011UpdateRemoteMsgObj(   pDev,
                                      msgNum,
                                      msgLength,
                                      &msgData[0]
                                      );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified
S_tdrv011Drv_MSGNOTDEF	The message is not properly setup for transmit
S_tdrv011Drv_MSGBUSY	The specified message is already in use

3.2.13 tdrv011RequestRemoteData

Name

tdrv011RequestRemoteData – request a remote CAN message

Synopsis

```
STATUS tdrv011RequestRemoteData
(
    TDRV011_DEV    pDev,
    int            messageNumber
);
```

Description

This function transmits a CAN message with the RTR bit set to request a remote message from other CAN nodes. The requesting receive message object must be previously defined with tdrv011DefineReceiveMsgObj.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

messageNumber

This parameter contains the number of the previous defined receive message object. Valid numbers are in the range between 1 and 15.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
int           msgNum;

/*
** Request a remote CAN message via message object 2
*/
msgNum        = 2;

result = tdrv011RequestRemoteData(    pDev,
                                      msgNum
                                      );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified
S_tdrv011Drv_MSGNOTDEF	The message is not properly setup for transmit

3.2.14 tdrv011ReleaseMsgObj

Name

tdrv011ReleaseMsgObj – release a previous defined message object

Synopsis

```
STATUS tdrv011ReleaseMsgObj
(
    TDRV011_DEV    pDev,
    int            messageNumber
);
```

Description

This function marks the specified message object invalid and stops any transaction with this message object.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

messageNumber

This parameter contains the number of the previous defined message object. Valid numbers are in the range between 1 and 15.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
int           msgNum;

/*
** Release a previous defined message object
*/
msgNum        = 2;

result = tdrv011ReleaseMsgObj(   pDev,
                                 msgNum
                                 );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified

3.2.15 tdrv011GetMsgObjStatus

Name

tdrv011GetMsgObjStatus – get the status of a transmit message object

Synopsis

```
STATUS tdrv011GetMsgObjStatus
(
    TDRV011_DEV     pDev,
    int             messageNumber,
    unsigned long   *pMsgStatus
);
```

Description

This function returns the state of the specified transmit message object.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

messageNumber

This parameter contains the number of the previous defined transmit message object. Valid numbers are in the range between 1 and 14.

pMsgStatus

This parameter is a pointer to an unsigned long (32bit) value which returns the current state of the specified message object.

Status code	Description
S_tdrv011Drv_IDLE	The transmit message object is idle and ready to transmit new data
S_tdrv011Drv_BUSY	The transmit message object is busy transmitting data
S_tdrv011Drv_MSGNOTDEF	The message object has not been defined
S_tdrv011Drv_IMSGNUM	This is not a transmit message object

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
int           msgNum;
unsigned long msgStatus

/*
** Get the message object status
*/
msgNum        = 2;

result = tdrv011GetMsgObjStatus( pDev,
                                 msgNum,
                                 &msgStatus
                                 );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_IMSGNUM	Illegal message number specified

3.2.16 tdrv011Start

Name

tdrv011Start – set the CAN controller online

Synopsis

```
STATUS tdrv011Start
(
        TDRV011_DEV    pDev
);
```

Description

This function sets the CAN controller associated with the specified device online to enter the BUSON state.
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 bus 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

Before the CAN controller can communicate over the CAN after driver start-up or a previous BUSOFF error condition this control function must be executed.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;

/*
** Enter the BUSON state
*/

result = tdrv011Start( pDev );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

The error codes are stored in *errno* and 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) or a driver set code described below. Function specific error codes will be described with the function.

Error code	Description
S_tdrv011Drv_BUSOFF	Unable to enter the BUSON state, the CAN controller is still offline and unable to communicate over the CAN bus

3.2.17 tdrv011Stop

Name

tdrv011Stop – set the CAN controller offline

Synopsis

```
STATUS tdrv011Stop
(
        TDRV011_DEV    pDev
);
```

Description

This function sets the specified CAN controller associated with the specified device into the bus OFF state. After execution of this control function the CAN controller is completely removed from the CAN bus and cannot communicate until the control function tdrv010Start() is executed.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;

/*
** Enter the BUSOFF state
*/

result = tdrv011Stop( pDev );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

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

This function returns no driver specific error codes.

3.2.18 tdrv011FlushReceiveFifo

Name

tdrv011FlushReceiveFifo – discard all messages in the receive FIFO

Synopsis

```
STATUS tdrv011FlushReceiveFifo
(
        TDRV011_DEV    pDev
);
```

Description

This function flushes the CAN message receive FIFO. All messages will be discarded.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;

/*
** Flush the CAN message receive FIFO
*/

result = tdrv011FlushReceiveFifo( pDev );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

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

This function returns no driver specific error codes.

3.2.19 tdrv011GetControllerStatus

Name

tdrv011GetControllerStatus – get contents of the CAN controller status register

Synopsis

```
STATUS tdrv011GetControllerStatus
(
        TDRV011_DEV     pDev,
        unsigned long   *pControllerStatus
);
```

Description

This function returns the contents of the CAN controller status register for diagnostic purposes.

Parameters

pDev

This parameter specifies the device descriptor to the hardware module retrieved by a call to the corresponding open-function.

pControllerStatus

This parameter is a pointer to an unsigned long (32bit) value which returns the contents of the CAN controller status register.

Example

```
#include “tdrv011.h”

TDRV011_DEV   pDev;
STATUS        result;
unsigned long controllerStatus

/*
** Read the CAN controller Status register
*/
result = tdrv011GetControllerStatus(  pDev,
                                      &controllerStatus
                                      );

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

RETURN VALUE

OK if function succeeds or ERROR.

ERROR CODES

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

This function returns no driver specific error codes.

4 Legacy I/O system functions

This chapter describes the legacy 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.

The legacy I/O system functions are only relevant for the legacy TDRV011 driver. For the VxBus-enabled TDRV011 driver, the driver will be installed automatically in the I/O system and devices will be created as needed for detected CAN channels.

4.1 tdrv011Drv()

NAME

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

SYNOPSIS

#include “tdrv011.h”

STATUS tdrv011Drv(void)

DESCRIPTION

This function searches for devices on the PCI bus, installs the TDRV011 driver in the I/O system.

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

Error codes are only set by system functions. The error codes are stored in *errno* and can be read with the function *errnoGet()*.

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

4.2 tdrv011DevCreate()

NAME

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

SYNOPSIS

#include “tdrv011.h”

```
STATUS tdrv011DevCreate
(
        char        *name,
        int         devIdx,
                    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.
See also 3.1.1 tdrv011Open() for the preferred device naming convention for legacy devices.

devIdx

This index number specifies the device to add to the system. The 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, 7, 8, 9, 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 "/tdrv011/0/0" for the first CAN device
** Initial Bittiming: 250 KBps
*/
result = tdrv011DevCreate(   "/tdrv011/0/0",
                             0,
                             TDRV011_250KBIT);

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_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

4.3 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();
```

4.4 tdrv011Init()

NAME

tdrv011Init() – initialize TDRV011 driver and devices

SYNOPSIS

#include “tdrv011.h”

STATUS tdrv011Init(void)

DESCRIPTION

This function is used by the TDRV011 example application to install the driver and to add all available devices to the VxWorks system.

See also 3.1.1 tdrv011Open() for the device naming convention for legacy devices.

After calling this function it is not necessary to call tdrv011Drv() and tdrv011DevCreate() explicitly.

EXAMPLE

```
#include "tdrv011.h”

STATUS    result;


result = tdrv011Init();

if (result == ERROR)
{
     /* Error handling */
}
```

RETURNS

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

ERROR CODES

Error codes are only set by system functions. The error codes are stored in *errno* and can be read with the function *errnoGet()*.

See 4.1 and 4.2 for a description of possible error codes.

5 Basic I/O Functions

The VxWorks basic I/O interface functions are useable with the TDRV011 legacy and VxBus-enabled driver in a uniform manner.

5.1 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

This parameter points to a null-terminated string that specifies the name of the device See also 3.1.1 tdrv011Open() for the device naming convention for TDRV011 devices.

flags

Not used

mode

Not used

EXAMPLE

```
int       fd;

/*
** open the device named "/tdrv011/0/0" for I/O
*/
fd = open("/tdrv011/0/0", 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()*

5.2 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()

5.3 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 BUS ON
TDRV011_BUSOFF	Set device BUS OFF
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()

5.3.1 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 another 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

5.3.2 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 another 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

5.3.3 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 */
}
```

5.3.4 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;
acceptMasks.GlobalMaskExtended   = 0xffffff80;
acceptMasks.Message15Mask        = 0xfffff800;

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

5.3.5 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 */
}
```

5.3.6 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 */
}
```

5.3.7 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...15. One of the message objects is reserved for internal use of the device driver (write). By default, message object 1 is used for transmission. This can be changed as stated in chapter **Fehler! Verweisquelle konnte nicht gefunden werden. Fehler! Verweisquelle konnte nicht gefunden werden.**. 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).

Message objects programmed to transmit are also affected by the Global Masks (standard and extended). The 82527 uses the Global Mask registers to identify which of its message objects transmitted a message. If two 82527 transmit message objects have message IDs that are non-distinct in all ``must-match'' bit locations, a successful transmission of the higher numbered message object will not be recognized by the 82527. The lower numbered message object will be falsely identified as the transmit message object and its transmit request bit will be reset and its interrupt pending bit set. The actual transmit message object will re-transmit without end because its transmit request bit will not be reset.

This could result in a catastrophic condition since the higher numbered message object may dominate the CAN bus by resending its message without end. To avoid this condition, applications should require all transmit message objects to use message IDs that are unique with respect to the ``must-match'' bits. If this is not possible, the application should disable lower numbered message objects with similar message IDs until the higher numbered message object has transmitted successfully.

For further information about this problem, please refer to the Intel 82527 Architectural Overview, chapter Acceptance Filtering Implications.

```
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.

Value	Description
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 2..5
  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      = 2;
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      = 3;
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      = 4;
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      = 5;
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 (e.g. transmit message object number)
S_tdrv011Drv_MSGBUSY	The specified message is already in use

5.3.8 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 (e.g. transmit message object)
S_tdrv011Drv_MSGBUSY	The specified message is in use
S_tdrv011Drv_MSGNOTDEF	The message object has not been defined

5.3.9 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…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

5.3.10 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

5.3.11 TDRV011_BUSOFF

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

EXAMPLE

```
#include “tdrv011.h”

int                fd;
int                retval;


retval = ioctl(fd, TDRV011_BUSOFF, 0);
if (retval == ERROR)
{
     /* handle error */
}
```

5.3.12 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

5.3.13 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);
}
```

6 Debugging and Diagnostic

Especially the TDRV011 VxBus device driver provides functions and debug statements to display versatile information of the driver installation and status on the debugging console.

By default the TDRV011 show routine is included in the driver and can be called from the VxWorks shell. If this function is not needed or program space is rare the function can be removed from the code by un-defining the macro INCLUDE_TDRV011_SHOW in tdrv011drv.c

The tdrv011Show function displays detailed information about probed modules, assignment of devices respective device names to probed TRDV011 modules and device statistics.

If TDRV011 modules were probed but no devices were created it may helpful to enable debugging code inside the driver code by defining the macro TDRV011_DEBUG in tdrv011drv.c. Certain debug information can be selected by assigning one or more (logical OR) TDRV_DBD_xxx values to variable tdrv011Debug.

```
-> tdrv011Show
Probed Modules:
    [0] TPMC901: Bus=4, Dev=1, DevId=0x9050, VenId=0x10b5, Init=OK, vxDev=0x4bb218
    [1] TPMC316: Bus=4, Dev=2, DevId=0x013c, VenId=0x1498, Init=OK, vxDev=0x4bb318

Associated Devices:
    [0] TPMC901: /tdrv011/0/0 /tdrv011/0/1 /tdrv011/0/2 /tdrv011/0/3 /tdrv011/0/4 /tdrv011/0/5
    [1] TPMC316: /tdrv011/1/0 /tdrv011/1/1

Device Statistics:
    /tdrv011/0/0:
        open count = 0
        interrupt count = 2
        bus off count = 0
        receive count = 0
        transmit count = 2
        object overrun = 0
        fifo overrun = 0
        timing value = 0x14
        global standard mask = 0xffff
        global extended mask = 0xffffffff
        local message 15 mask = 0x0
    /tdrv011/0/1:
        open count = 0
        interrupt count = 2
        bus off count = 0
        receive count = 2
        transmit count = 0
        object overrun = 0
        fifo overrun = 0
        timing value = 0x14
        global standard mask = 0xffff
        global extended mask = 0xffffffff
        local message 15 mask = 0x0
...
```

7 Appendix

7.1 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