The Embedded I/O Company

TEWS TECHNOLOGIES

TDRV009-SW-72

LynxOS Device Driver

High Speed Synch/Asynch Serial Interface

Version 1.0.x

User Manual

Issue 1.0.0

January 2009

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

TDRV009-SW-72

LynxOS Device Driver

High Speed Synch/Asynch Serial Interface

Supported Modules:
TPMC363
TPMC863
TAMC863
TCP863

Issue	Description	Date
1.0.0	First Issue	January 19, 2009

Table of Contents

1 Introduction

The TDRV009-SW-72 LynxOS device driver allows the operation of the TDRV009 compatible devices on LynxOS platforms with DRM based PCI interface.

This driver was successfully tested on a TEWS TECHNOLOGIES TVME8240 PowerPC board and on an Intel x86 native system.

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

The TDRV009 device driver includes the following functions:

- setup and configure serial channels
- send and receive data buffers (character oriented)
- read and write onboard registers directly
- read and write access to onboard EEPROM
- control handshake lines
- wait for interrupts

The TDRV009-SW-72 supports the modules listed below:

TPMC863	4 Channel High Speed Synch/Asynch Serial Interface	PMC
TPMC363	4 Channel High Speed Synch/Asynch Serial Interface	PMC, Conduction Cooled
TAMC863	4 Channel High Speed Synch/Asynch Serial Interface	AMC
TCP863	4 Channel High Speed Synch/Asynch Serial Interface	CompactPCI

In this document all supported modules and devices will be called TDRV009. Specials for a certain device will be advised.

To get more information about the features and use of TDRV009 devices it is recommended to read the manuals listed below.

TPMC863 (or compatible) User manual

TPMC863 (or compatible) Engineering Manual

2 Installation

Following files are located in the directory TDRV009-SW-72 on the distribution media:

TDRV009-SW-72-1.0.0.pdf	This manual in PDF format
TDRV009-SW-72-SRC.tar.gz	Device Driver and Example sources
ChangeLog.txt	Release history
Release.txt	Information about the Device Driver Release

The GZIP compressed archive TDRV009-SW-72-SRC.tar.gz contains the following files and directories:

tdrv009.c	Driver source code
tdrv009.h	Definitions and data structures for driver and application
commCtrl.h	Include file for controller chip
tdrv009def.h	Definitions and data structures for the driver
tdrv009_info.c	Device information definition
tdrv009_info.h	Device information definition header
tdrv009.cfg	Driver configuration file include
tdrv009.import	Linker import file
Makefile	Device driver make file
example/tdrv009exa.c	Example application source

In order to perform a driver installation first extract the TAR file to a temporary directory, then copy the following files to their target directories:

1. Create a new directory in the system drivers directory path /sys/drivers.xxx, where xxx represents the BSP that supports the target hardware.

   For example: /sys/drivers.pp_drm/tdrv009 or /sys/drivers.cpci_x86/tdrv009

2. Copy the following files to this directory:
   - tdrv009.c
   - tdrv009def.h
   - commCtrl.h
   - tdrv009.import
   - Makefile

3. Copy tdrv009.h to /usr/include/

4. Copy tdrv009_info.c to /sys/devices.xxx/ or /sys/devices if /sys/devices.xxx does not exist (xxx represents the BSP).

5. Copy tdrv009_info.h to /sys/dheaders/

6. Copy tdrv009.cfg to */sys/cfg.xxx/*, where xxx represents the BSP for the target platform. For example: /sys/cfg.ppc or /sys/cfg.x86 ....

2.1 Device Driver Installation

The two methods of driver installation are as follows:

- Static Installation
- Dynamic Installation (only native LynxOS systems)

2.1.1 Static Installation

With this method, the driver object code is linked with the kernel routines and is installed during system start-up.

2.1.1.1 Build the driver object

1. Change to the directory /sys/drivers.xxx/tdrv009, where xxx represents the BSP that supports the target hardware.
2. To update the library /sys/lib/libdrivers.a enter:

   ```
   make install
   ```

2.1.1.2 Create Device Information Declaration

1. Change to the directory /sys/devices.xxx/ or /sys/devices if /sys/devices.xxx does not exist (xxx represents the BSP).
2. Add the following dependencies to the Makefile

   ```
   DEVICE_FILES_all = ... tdrv009_info.x
   ```

   And at the end of the Makefile

   ```
   tdrv009_info.o:$(DHEADERS)/tdrv009_info.h
   ```

3. To update the library /sys/lib/libdevices.a enter:

   ```
   make install
   ```

2.1.1.3 Modify the Device and Driver Configuration File

In order to insert the driver object code into the kernel image, an appropriate entry in file CONFIG.TBL must be created.

1. Change to the directory /sys/lynx.os/ respective /sys/bsp.xxx, where xxx represents the BSP that supports the target hardware.
2. Create an entry at the end of the file CONFIG.TBL

   Insert the following entry at the end of this file.

   ```
   I:tdrv009.cfg
   ```

2.1.1.4 Rebuild the Kernel

1. Change to the directory /sys/lynx.os/ (/sys/bsp.xxx)
2. Enter the following command to rebuild the kernel:

   ```
   make install
   ```

3. Reboot the newly created operating system by the following command (not necessary for KDIs):

   ```
   reboot –aN
   ```

   The N flag instructs init to run mknod and create all the nodes mentioned in the new nodetab.

4. After reboot you should find the following new devices (depends on the device configuration): /dev/tdrv009a_0, /dev/tdrv009a_1, /dev/tdrv009a_2, /dev/tdrv009a_3, /dev/tdrv009b_0, /dev/tdrv009b_1, …

2.1.2 Dynamic Installation

This method allows you to install the driver after the operating system is booted. The driver object code is attached to the end of the kernel image and the operating system dynamically adds this driver to its internal structures. The driver can also be removed dynamically.

2.1.2.1 Build the driver object

1. Change to the directory /sys/drivers.xxx/tdrv009, where xxx represents the BSP that supports the target hardware.
2. To make the dynamic link-able driver enter :

   ```
   make dldd
   ```

2.1.2.2 Create Device Information Declaration

1. Change to the directory /sys/drivers.xxx/tdrv009, where xxx represents the BSP that supports the target hardware.
2. To create a device definition file for the major device (this works only on native system)

   ```
   make t009info
   ```

3. To install the driver enter:

   ```
   drinstall –c tdrv009.obj
   ```

   If successful, drinstall returns a unique <driver-ID>
4. To install the major device enter:

   ```
   devinstall –c –d <driver-ID> t009info
   ```

   The <driver-ID> is returned by the drinstall command
5. To create a node for the device enter:

   ```
   mknod /dev/tdrv009a_0 c <major_no> 0
   mknod /dev/tdrv009a_1 c <major_no> 1
   mknod /dev/tdrv009a_2 c <major_no> 2
   mknod /dev/tdrv009a_3 c <major_no> 3
   ...
   ```

   The <major_no> is returned by the devinstall command.

If all steps are successfully completed, the TDRV009 is ready to use.

2.1.2.3 Uninstall dynamic loaded driver

To uninstall the TDRV009 device enter the following commands:

```
devinstall –u –c <device-ID>
drinstall –u <driver-ID>
```

2.1.3 Device Information Definition File

The device information definition contains information necessary to install the TDRV009 major device.

The implementation of the device information definition is done through a C structure, which is defined in the header file *tdrv009_info.h*.

This structure contains the following parameter:

PCIBusNumber Contains the PCI bus number at which the TDRV009 compatible device is connected. Valid bus numbers are in range from 0 to 255.

PCIDeviceNumber Contains the device number (slot) at which the TDRV009 compatible device is connected. Valid device numbers are in range from 0 to 31.

If both PCIBusNumber and PCIDeviceNumber are –1 then the driver will auto scan for the TDRV009 compatible device. The first device found in the scan order will be allocated by the driver for this major device.

Already allocated devices can't be allocated twice. This is important to know if there are more than one TDRV009 major devices.

A device information definition is unique for every TDRV009 major device. The file *tdrv009_info.c* on the distribution disk contains two device information declarations, **tdrv009a_info** for the first major device and **tdrv009b_info** for the second major device.

If the driver should support more than two major devices it is necessary to copy and paste an existing declaration and rename it with a unique name, for example **tdrv009c_info**, **tdrv009d_info** and so on.

It is also necessary to modify the device and driver configuration file, respectively the configuration include file *tdrv009.cfg*.

The following device declaration information uses the auto find method to detect a TDRV009 compatible device on the PCI bus.

```
TDRV009_INFO tdrv009a_info = {

    -1,                 /*  Auto find the device on any PCI bus    */
    -1,

};
```

2.1.4 Configuration File: CONFIG.TBL

The device and driver configuration file CONFIG.TBL contains entries for device drivers and its major and minor device declarations. Each time the system is rebuild, the config utility read this file and produces a new set of driver and device configuration tables and a corresponding nodetab.

To install the TDRV009 driver and devices into the LynxOS system, the configuration include file tdrv009.cfg must be included in the CONFIG.TBL (see also chapter 2.1.1.3).

The file tdrv009.cfg on the distribution disk contains the driver entry (*C:tdrv009:*\....) and a major device entry (*D:TDRV009 1:tdrv009a_info::*).

If the driver should support more than one major device, the following entries for major devices must be enabled by removing the comment character (#). By copy and paste an existing major and minor entries and renaming the new entries, it is possible to add any number of additional TDRV009 devices.

This example shows a driver entry with one major device and one minor device:

```
#    Format:
#    C:driver-name:open:close:read:write:select:control:install:uninstall
#    D:device-name:info-block-name:raw-partner-name
#    N:node-name:minor-dev

C:tdrv009:\
     :tdrv009open:tdrv009close:tdrv009read:tdrv009write:\
     ::tdrv009ioctl:tdrv009install:tdrv009uninstall
D:TDRV009 1:tdrv009a_info::
N:tdrv009a_0:0
N:tdrv009a_1:1
N:tdrv009a_2:2
N:tdrv009a_3:3
#D:TDRV009 2:tdrv009b_info::
#N:tdrv009b_0:0
#N:tdrv009b_1:1
#N:tdrv009b_2:2
#N:tdrv009b_3:3
```

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

```
/dev/tdrv009a
```

3 TDRV009 Device Driver Programming

LynxOS system calls are all available directly to any C program. They are implemented as ordinary function calls to "glue" routines in the system library, which trap to the OS code.

Note that many system calls use data structures, which should be obtained in a program from appropriate header files. Necessary header files are listed with the system call synopsis.

3.1 open()

NAME

open() - open a file

SYNOPSIS

#include <sys/file.h>
#include <sys/types.h>
#include <fcntl.h>

int open (char *path, int oflags[, mode_t mode])

DESCRIPTION

Opens a file (TDRV009 device) named in ***path*** for reading and writing. The value of ***oflags*** indicates the intended use of the file. In case of a TDRV009 devices ***oflags*** must be set to **O_RDWR** to open the file for both reading and writing.

The ***mode*** argument is required only when a file is created. Because a TDRV009 device already exists this argument is ignored.

EXAMPLE

```
int fd

/* open the device named "/dev/tdrv009a_0" for I/O */
fd = open ("/dev/tdrv009a_0", O_RDWR);
if (!fd)
{
     /* handle error */
}
```

RETURNS

open returns a file descriptor number if successful, or –1 on error.

SEE ALSO

LynxOS System Call - open()

3.2 close()

NAME

close() – close a file

SYNOPSIS

int close(int fd)

DESCRIPTION

This function closes an opened device.

EXAMPLE

```
int  result;

/*
**   close the device
*/
result = close(fd);
if (result < 0)
{
     /* handle error */
}
```

RETURNS

close returns 0 (OK) if successful, or –1 on error

SEE ALSO

LynxOS System Call - close()

3.3 read()

NAME

read() – read from a device

SYNOPSIS

int read(*int fd, char *buff, int count*)

DESCRIPTION

This function attempts to read an input buffer from a TDRV009 compatible channel associated with the file descriptor *fd*. The argument *count* specifies the length of the buffer. Available data (up to *count* bytes) is copied into the user's buffer pointed to by *buff*. The function performs a blocking read operation, i.e. the functions waits until data is available or the specified timeout expires (see *SET_READ_TIMEOUT*). In synchronous channel mode, data is returned up to frame end.

EXAMPLE

```
int       fd;
int       num_bytes;
char      buffer[100];

num_bytes = read(fd, buffer, 100);
if (num_bytes > 0)
{
     // process received data
}
```

RETURNS

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

ERRORS

EFAULT	Invalid pointer to the read buffer.
EINVAL	Data buffer is too small.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.4 write()

NAME

write() – write to a device

SYNOPSIS

int write(*int fd, char *buff, int count*)

DESCRIPTION

This function attempts to write an output buffer to a TDRV009 compatible channel associated with the file descriptor *fd*. The argument *count* specifies the length of the buffer. The function performs a blocking write operation, i.e. the functions waits until all data is transferred into the hardware FIFO or the specified timeout expires (see *SET_WRITE_TIMEOUT*).

EXAMPLE

```
int       fd;
int       num_bytes;
char      buffer[100];

/* send some text */
sprintf( buffer, “Hello World” );
num_bytes = write(fd, buffer, strlen(buffer));
if ( num_bytes != strlen(buffer) )
{
      // process error;
}
```

RETURNS

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

ERRORS

EFAULT	Invalid pointer to the read buffer.
EINVAL	Data buffer is too small.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.5 ioctl()

NAME

ioctl() – I/O device control

SYNOPSIS

#include <ioctl.h>
#include <tdrv009.h>

int ioctl (int fd, int request, char *arg)

DESCRIPTION

ioctl provides a way of sending special commands to a device driver. The call sends the value of request and the pointer arg to the device associated with the descriptor fd.

The following ioctl codes are supported by the driver and are defined in *tdrv009.h*:

Symbol	Meaning
TDRV009_C_SET_OPERATION_MODE	set a channel's operation mode
TDRV009_C_GET_OPERATION_MODE	get a channel's current operation mode
TDRV009_C_SET_BAUDRATE	set baudrate without other changes
TDRV009_C_SET_RECEIVER_STATE	set operation state of the channel's receiver
TDRV009_C_CLEAR_RX_BUFFER	clear the internal receive buffer
TDRV009_C_SET_EXT_XTAL	specify an externally supplied frequency
TDRV009_C_SET_READ_TIMEOUT	specify a timeout for read operations
TDRV009_C_REG_WRITE	directly write to a register
TDRV009_C_REG_READ	directly read from a register
TDRV009_C_SCC_REG_WRITE	directly write to an SCC register
TDRV009_C_SCC_REG_READ	directly read from an SCC register
TDRV009_C_EEPROM_WRITE	write value to EEPROM
TDRV009_C_EEPROM_READ	read value from EEPROM
TDRV009_C_RTS_SET	Assert RTS handshake line
TDRV009_C_RTS_CLEAR	De-Assert RTS handshake line
TDRV009_C_CTS_GET	Read state of CTS
TDRV009_C_DTR_SET	Set DTR signal line (only channel 3)
TDRV009_C_DTR_CLEAR	Clear DTR signal line (only channel 3)
TDRV009_C_DSR_GET	Read state of DSR (only channel 3)
TDRV009_C_WAITFORINTERRUPT	Wait for specific channel interrupt

See behind for more detailed information on each control code.

RETURNS

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

On error, *errno* will contain a standard error code (see also LynxOS System Call – ioctl).

SEE ALSO

LynxOS System Call - ioctl().

3.5.1 TDRV009_C_SET_OPERATION_MODE

NAME

TDRV009_C_SET_OPERATION_MODE – set a channel's operation mode

DESCRIPTION

This I/O control function configures the channel's operation mode. The function specific control parameter **arg** is a pointer to a *TDRV009_OPERATION_MODE_STRUCT* structure. It is necessary to completely initialize the structure. This can be done by calling the I/O control function TDRV009_IOCG_GET_OPERATION_MODE described below.

A call to this function must be done prior to any communication operation, because after driver startup, the channel's transceivers are disabled.

```
typedef struct
{
   TDRV009_COMM_TYPE            CommType;
   TDRV009_TRANSCEIVER_MODE     TransceiverMode;
   TDRV009_ENABLE_DISABLE       Oversampling;
   TDRV009_BRGSOURCE            BrgSource;
   TDRV009_TXCSOURCE            TxClkSource;
   unsigned long                TxClkOutput;
   TDRV009_RXCSOURCE            RxClkSource;
   TDRV009_CLKMULTIPLIER        ClockMultiplier;
   unsigned long                Baudrate;
   unsigned char                ClockInversion;
   unsigned char                Encoding;
   TDRV009_PARITY               Parity;
   int                          Stopbits;
   int                          Databits;
   TDRV009_ENABLE_DISABLE       UseTermChar;
   char                         TermChar;
   TDRV009_ENABLE_DISABLE       HwHs;
   TDRV009_CRC                  Crc;
} TDRV009_OPERATION_MODE_STRUCT;
```

CommType

This parameter describes the general communication type for the specific channel. Possible values are:

Value	Description
TDRV009_COMMTYPE_ASYNC	Asynchronous communication
TDRV009_COMMTYPE_HDLC_ADDR0	Standard HDLC communication without address recognition. Used for synchronous communication.
TDRV009_COMMTYPE_HDLC_TRANSP	Extended Transparent mode. No protocol processing, channel works as simple bit collector.

TransceiverMode

This parameter describes the transceiver mode of the programmable multi-protocol transceivers. Possible values are:

Value	Description
TDRV009_TRNSCVR_NOT_USED	Default V.11
TDRV009_TRNSCVR_RS530A	EIA-530A (V.11 / V.10)
TDRV009_TRNSCVR_RS530	EIA-530 (V.11), also suitable for RS422
TDRV009_TRNSCVR_X21	X.21 (V.11)
TDRV009_TRNSCVR_V35	V.35 (V.35 / V.28)
TDRV009_TRNSCVR_RS449	EIA-449 (V.11)
TDRV009_TRNSCVR_V36	V.36 (V.11)
TDRV009_TRNSCVR_RS232	EIA-232 (V.28)
TDRV009_TRNSCVR_V28	V.28 (V.28)
TDRV009_TRNSCVR_NO_CABLE	High impedance

Oversampling

This parameter enables or disables 16times oversampling, used for asynchronous communication. For communication with standard UARTs it is recommended to enable this feature. Valid values are:

Value	Description
TDRV009_DISABLED	The 16 times oversampling is not used.
TDRV009_ENABLED	The 16 times oversampling is used.

BrgSource

This parameter specifies the frequency source used as input to the BRG (Baud Rate Generator). Valid values are:

Value	Description
TDRV009_BRGSRC_XTAL1	XTAL1 oscillator is used for BRG input
TDRV009_BRGSRC_XTAL2	XTAL2 oscillator is used for BRG input
TDRV009_BRGSRC_XTAL3	XTAL3 oscillator is used for BRG input
TDRV009_BRGSRC_RXCEXTERN	External clock at RxC input used for BRG input
TDRV009_BRGSRC_TXCEXTERN	External clock at TxC input used for BRG input

TxClkSource

This parameter specifies the frequency source used as input to the transmit engine. Valid values are:

Value	Description
TDRV009_TXCSRC_BRG	Baud Rate Generator output used for Tx clock
TDRV009_TXCSRC_BRGDIV16	BRG output divided by 16 used for Tx clock
TDRV009_TXCSRC_RXCEXTERN	External clock at RxC input used for Tx clock
TDRV009_TXCSRC_TXCEXTERN	External clock at TxC input used for Tx clock
TDRV009_TXCSRC_DPLL	DPLL output used for Tx clock

TxClkOutput

This parameter specifies which output lines are used to output the transmit clock, e.g. for synchronous communication. The given values can be binary OR'ed. Valid values are:

Value	Description
TDRV009_TXCOUT_TXC	Transmit clock available at TxC output line
TDRV009_TXCOUT_RTS	Transmit clock available at RTS output line

RxClkSource

This parameter specifies the frequency source used as input to the receive engine. Valid values are:

Value	Description
TDRV009_RXCSRC_BRG	Baud Rate Generator output used for Rx clock
TDRV009_RXCSRC_RXCEXTERN	External clock at RxC input used for Rx clock
TDRV009_RXCSRC_DPLL	DPLL output used for Rx clock

ClockMultiplier

This parameter specifies the multiplier used for BRG clock input. Valid values are:

Value	Description
TDRV009_CLKMULT_X1	Clock multiplier disabled
TDRV009_CLKMULT_X4	Selected input clock is multiplied by 4

Baudrate

This parameter specifies the desired frequency to be generated by the Baud Rate Generator (BRG), which can be used as clock input signal. The value is derived from the selected clock source. Please note that only specific values depending on the selected oscillator are valid. This frequency is internally multiplied by 16, if oversampling shall be used.

ClockInversion

This parameter specifies the inversion of the transmit and/or the receive clock. This value can be binary OR'ed. Possible values are:

Value	Description
TDRV009_CLKINV_NONE	no clock inversion
TDRV009_CLKINV_TXC	transmit clock is inverted
TDRV009_CLKINV_RXC	receive clock is inverted

Encoding

This parameter specifies the data encoding used for communication. Valid values are:

Value	Description
TDRV009_ENC_NRZ	NRZ data encoding
TDRV009_ENC_NRZI	NRZI data encoding
TDRV009_ENC_FM0	FM0 data encoding
TDRV009_ENC_FM1	FM1 data encoding
TDRV009_ENC_MANCHESTER	Manchester data encoding

Parity

This parameter specifies the parity bit generation used for asynchronous communication. Valid values are:

Value	Description
TDRV009_PAR_DISABLED	No parity generation is used.
TDRV009_PAR_EVEN	EVEN parity bit
TDRV009_PAR_ODD	ODD parity bit
TDRV009_PAR_SPACE	SPACE parity bit (always insert ‘0’)
TDRV009_PAR_MARK	MARK parity bit (always insert ‘1’)

Stopbits

This parameter specifies the number of stop bits to use for asynchronous communication. Possible values are 1 or 2.

Databits

This parameter specifies the number of data bits to use for asynchronous communication. Possible values are 5 to 8.

UseTermChar

This parameter enables or disables the usage of a termination character for asynchronous communication. Valid values are:

Value	Description
TDRV009_DISABLED	A termination character is not used.
TDRV009_ENABLED	A termination character is used.

TermChar

This parameter specifies the termination character. After receiving this termination character, the communication controller will forward the received data packet immediately to the host system and use a new data packet for further received data. Any 8bit value may be used for this parameter.

HwHs

This parameter enables or disables the hardware handshaking mechanism using RTS/CTS. Valid values are:

Value	Description
TDRV009_DISABLED	Hardware handshaking is not used.
TDRV009_ENABLED	Hardware handshaking is used.

Crc

This parameter is a structure describing the CRC checking configuration.

```
typedef struct
{
   TDRV009_CRC_TYPE          Type;
   TDRV009_ENABLE_DISABLE    RxChecking;
   TDRV009_ENABLE_DISABLE    TxGeneration;
   TDRV009_CRC_RESET         ResetValue;
} TDRV009_CRC;
```

Type

This parameter describes the CRC type to be used. Possible values are:

Value	Description
TDRV009_CRC_16	16bit CRC algorithm is used for checksum
TDRV009_CRC_32	32bit CRC algorithm is used for checksum

RxChecking

This parameter enables or disables the receive CRC checking. Possible values are:

Value	Description
TDRV009_DISABLED	CRC checking will not be used
TDRV009_ENABLED	CRC checking will be used

TxGeneration

This parameter enables or disables the transmit CRC generation. Possible values are:

Value	Description
TDRV009_DISABLED	A CRC checksum will be generated
TDRV009_ENABLED	A CRC checksum will not be generated

ResetValue

This parameter describes the reset value for the CRC algorithm. Possible values are:

Value	Description
TDRV009_CRC_RST_FFFF	CRC reset value will be 0xFFFF
TDRV009_CRC_RST_0000	CRC reset value will be 0x0000

EXAMPLE

```
#include “tdrv009.h”

int                                fd;
int                                result;
TDRV009_OPERATION_MODE_STRUCT      OperationMode;

/*-----------------------------------------------
  Configure channel for Async / RS232 / 115200bps
  -----------------------------------------------*/
OperationMode.CommType             = TDRV009_COMMTYPE_ASYNC;
OperationMode.TransceiverMode      = TDRV009_TRNSCVR _RS232;
OperationMode.Oversampling         = TDRV009_ENABLED;
OperationMode.BrgSource            = TDRV009_BRGSRC_XTAL1;
OperationMode.TxClkSource          = TDRV009_TXCSRC_BRG;
OperationMode.TxClkOutput          = 0;
OperationMode.RxClkSource          = TDRV009_RXCSRC_BRG;
OperationMode.ClockMultiplier      = TDRV009_CLKMULT_X1;
OperationMode.Baudrate             = 115200;
OperationMode.ClockInversion       = TDRV009_CLKINV_NONE;
OperationMode.Encoding             = TDRV009_ENC_NRZ;
OperationMode.Parity               = TDRV009_PAR_DISABLED;
OperationMode.Stopbits             = 1;
OperationMode.Databits             = 8;
OperationMode.UseTermChar          = TDRV009_DISABLED;
OperationMode.TermChar             = 0;
OperationMode.HwHs                 = TDRV009_DISABLED;
OperationMode.Crc.Type             = TDRV009_CRC_16;
OperationMode.Crc.RxChecking       = TDRV009_DISABLED;
OperationMode.Crc.TxGeneration     = TDRV009_DISABLED;
OperationMode.Crc.ResetValue       = TDRV009_CRC_RST_FFFF;

result = ioctl(fd, TDRV009_C_SET_OPERATION_MODE, (char*)&OperationMode);
if (result != OK) {
       /* handle ioctl error */
}
```

ERRORS

EINVAL	At least one specified parameter is invalid.

Other returned error codes are system error conditions.

3.5.2 TDRV009_C_GET_OPERATION_MODE

NAME

TDRV009_C_GET_OPERATION_MODE – get a channel's current operation mode

DESCRIPTION

This I/O control function returns the channel's current operation mode. The function specific control parameter **arg** is a pointer to a *TDRV009_OPERATION_MODE_STRUCT* structure.

```
typedef struct
{
   TDRV009_COMM_TYPE            CommType;
   TDRV009_TRANSCEIVER_MODE     TransceiverMode;
   TDRV009_ENABLE_DISABLE       Oversampling;
   TDRV009_BRGSOURCE            BrgSource;
   TDRV009_TXCSOURCE            TxClkSource;
   unsigned long                TxClkOutput;
   TDRV009_RXCSOURCE            RxClkSource;
   TDRV009_CLKMULTIPLIER        ClockMultiplier;
   unsigned long                Baudrate;
   unsigned char                ClockInversion;
   unsigned char                Encoding;
   TDRV009_PARITY               Parity;
   int                          Stopbits;
   int                          Databits;
   TDRV009_ENABLE_DISABLE       UseTermChar;
   char                         TermChar;
   TDRV009_ENABLE_DISABLE       HwHs;
   TDRV009_CRC                  Crc;
} TDRV009_OPERATION_MODE_STRUCT;
```

CommType

This parameter describes the general communication type for the specific channel. Possible values are:

Value	Description
TDRV009_COMMTYPE_ASYNC	Asynchronous communication
TDRV009_COMMTYPE_HDLC_ADDR0	Standard HDLC communication without address recognition. Used for synchronous communication.
TDRV009_COMMTYPE_HDLC_TRANSP	Extended Transparent mode. No protocol processing, channel works as simple bit collector.

TransceiverMode

This parameter describes the transceiver mode of the programmable multi-protocol transceivers. Possible values are:

Value	Description
TDRV009_TRNSCVR_NOT_USED	Default V.11
TDRV009_TRNSCVR_RS530A	EIA-530A (V.11 / V.10)
TDRV009_TRNSCVR_RS530	EIA-530 (V.11), also suitable for RS422
TDRV009_TRNSCVR_X21	X.21 (V.11)
TDRV009_TRNSCVR_V35	V.35 (V.35 / V.28)
TDRV009_TRNSCVR_RS449	EIA-449 (V.11)
TDRV009_TRNSCVR_V36	V.36 (V.11)
TDRV009_TRNSCVR_RS232	EIA-232 (V.28)
TDRV009_TRNSCVR_V28	V.28 (V.28)
TDRV009_TRNSCVR_NO_CABLE	High impedance

Oversampling

This parameter enables or disables 16times oversampling, used for asynchronous communication. For communication with standard UARTs it is recommended to enable this feature. Valid values are:

Value	Description
TDRV009_DISABLED	The 16 times oversampling is not used.
TDRV009_ENABLED	The 16 times oversampling is used.

BrgSource

This parameter specifies the frequency source used as input to the BRG (Baud Rate Generator). Valid values are:

Value	Description
TDRV009_BRGSRC_XTAL1	XTAL1 oscillator is used for BRG input
TDRV009_BRGSRC_XTAL2	XTAL2 oscillator is used for BRG input
TDRV009_BRGSRC_XTAL3	XTAL3 oscillator is used for BRG input
TDRV009_BRGSRC_RXCEXTERN	External clock at RxC input used for BRG input
TDRV009_BRGSRC_TXCEXTERN	External clock at TxC input used for BRG input

TxClkSource

This parameter specifies the frequency source used as input to the transmit engine. Valid values are:

Value	Description
TDRV009_TXCSRC_BRG	Baud Rate Generator output used for Tx clock
TDRV009_TXCSRC_BRGDIV16	BRG output divided by 16 used for Tx clock
TDRV009_TXCSRC_RXCEXTERN	External clock at RxC input used for Tx clock
TDRV009_TXCSRC_TXCEXTERN	External clock at TxC input used for Tx clock
TDRV009_TXCSRC_DPLL	DPLL output used for Tx clock

TxClkOutput

This parameter specifies which output lines are used to output the transmit clock, e.g. for synchronous communication. The given values can be binary OR'ed. Valid values are:

Value	Description
TDRV009_TXCOUT_TXC	Transmit clock available at TxC output line
TDRV009_TXCOUT_RTS	Transmit clock available at RTS output line

RxClkSource

This parameter specifies the frequency source used as input to the receive engine. Valid values are:

Value	Description
TDRV009_RXCSRC_BRG	Baud Rate Generator output used for Rx clock
TDRV009_RXCSRC_RXCEXTERN	External clock at RxC input used for Rx clock
TDRV009_RXCSRC_DPLL	DPLL output used for Rx clock

ClockMultiplier

This parameter specifies the multiplier used for BRG clock input. Valid values are:

Value	Description
TDRV009_CLKMULT_X1	Clock multiplier disabled
TDRV009_CLKMULT_X4	Selected input clock is multiplied by 4

Baudrate

This parameter specifies the desired frequency to be generated by the Baud Rate Generator (BRG), which can be used as clock input signal. The value is derived from the selected clocksource. Please note that only specific values depending on the selected oscillator are valid. This frequency is internally multiplied by 16, if oversampling shall be used.

ClockInversion

This parameter specifies the inversion of the transmit and/or the receive clock. This value can be binary OR'ed. Possible values are:

Value	Description
TDRV009_CLKINV_NONE	no clock inversion
TDRV009_CLKINV_TXC	transmit clock is inverted
TDRV009_CLKINV_RXC	receive clock is inverted

Encoding

This parameter specifies the data encoding used for communication. Valid values are:

Value	Description
TDRV009_ENC_NRZ	NRZ data encoding
TDRV009_ENC_NRZI	NRZI data encoding
TDRV009_ENC_FM0	FM0 data encoding
TDRV009_ENC_FM1	FM1 data encoding
TDRV009_ENC_MANCHESTER	Manchester data encoding

Parity

This parameter specifies the parity bit generation used for asynchronous communication. Valid values are:

Value	Description
TDRV009_PAR_DISABLED	No parity generation is used.
TDRV009_PAR_EVEN	EVEN parity bit
TDRV009_PAR_ODD	ODD parity bit
TDRV009_PAR_SPACE	SPACE parity bit (always insert ‘0’)
TDRV009_PAR_MARK	MARK parity bit (always insert ‘1’)

Stopbits

This parameter specifies the number of stop bits to use for asynchronous communication. Possible values are 1 or 2.

Databits

This parameter specifies the number of data bits to use for asynchronous communication. Possible values are 5 to 8.

UseTermChar

This parameter enables or disables the usage of a termination character for asynchronous communication. Valid values are:

Value	Description
TDRV009_DISABLED	A termination character is not used.
TDRV009_ENABLED	A termination character is used.

TermChar

This parameter specifies the termination character. After receiving this termination character, the communication controller will forward the received data packet immediately to the host system and use a new data packet for further received data. Any 8bit value may be used for this parameter.

HwHs

This parameter enables or disables the hardware handshaking mechanism using RTS/CTS. Valid values are:

Value	Description
TDRV009_DISABLED	Hardware handshaking is not used.
TDRV009_ENABLED	Hardware handshaking is used.

Crc

This parameter is a structure describing the CRC checking configuration.

```
typedef struct
{
   TDRV009_CRC_TYPE           Type;
   TDRV009_ENABLE_DISABLE     RxChecking;
   TDRV009_ENABLE_DISABLE     TxGeneration;
   TDRV009_CRC_RESET          ResetValue;
} TDRV009_CRC;
```

Type

This parameter describes the CRC type to be used. Possible values are:

Value	Description
TDRV009_CRC_16	16bit CRC algorithm is used for checksum
TDRV009_CRC_32	32bit CRC algorithm is used for checksum

RxChecking

This parameter enables or disables the receive CRC checking. Possible values are:

Value	Description
TDRV009_DISABLED	CRC checking will not be used
TDRV009_ENABLED	CRC checking will be used

TxGeneration

This parameter enables or disables the transmit CRC generation. Possible values are:

Value	Description
TDRV009_DISABLED	A CRC checksum will be generated
TDRV009_ENABLED	A CRC checksum will not be generated

ResetValue

This parameter describes the reset value for the CRC algorithm. Possible values are:

Value	Description
TDRV009_CRC_RST_FFFF	CRC reset value will be 0xFFFF
TDRV009_CRC_RST_0000	CRC reset value will be 0x0000

EXAMPLE

```
#include “tdrv009.h”

int                                fd;
int                                result;
TDRV009_OPERATION_MODE_STRUCT      OperationMode;

/*-------------------------------------------------
  Retrieve current configuration
  -------------------------------------------------*/
result = ioctl(fd, TDRV009_C_GET_OPERATION_MODE, (char*)&OperationMode);
if (result = OK) {
     printf( “Current speed: %d bps\n”, OperationMode.Baudrate );
} else {
     /* handle ioctl error */
}
```

3.5.3 TDRV009_C_SET_BAUDRATE

NAME

TDRV009_C_SET_BAUDRATE – set baudrate without other changes

DESCRIPTION

This I/O control function sets up the transmission rate for the specific channel. This is done without all the other configuration stuff contained in TDRV009_IOCS_SET_OPERATION_MODE. If async oversampling is enabled, the desired baudrate is internally multiplied by 16. It is important that this result can be derived from the selected clocksource. This function specifies the desired frequency which should be generated by the Baud Rate Generator (BRG). The function dependent parameter **arg** passes an *unsigned long* value containing the desired baudrate to the device driver.

EXAMPLE

```
#include “tdrv009.h”

int       fd;
int       result;

/*-------------------------------------------------
  Set baudrate to 14400bps
  -------------------------------------------------*/
result = ioctl(fd, TDRV009_C_SET_BAUDRATE, (char*)14400);

if (result != OK) {
       /* handle ioctl error */
}
```

ERRORS

EINVAL	Invalid parameter specified.

Other returned error codes are system error conditions.

3.5.4 TDRV009_C_SET_RECEIVER_STATE

NAME

TDRV009_C_SET_RECEIVER_STATE – set operation state of the channel's receiver

DESCRIPTION

This command sets the channel's receiver either to active or inactive.

The parameter **arg** passed to this I/O control function defines the new state of the receiver. Possible values are:

Value	Description
TDRV009_RCVR_ON	The receiver is enabled.
TDRV009_RCVR_OFF	The receiver is disabled.

EXAMPLE

```
#include "tdrv009.h"

int           fd;
int           result;

/*
**  enable the receiver
*/
result = ioctl(fd, TDRV009_C_SET_RECEIVER_STATE, (char*)TDRV009_RCVR_ON);

if (result != OK) {
       /* handle ioctl error */
}
```

ERRORS

EINVAL	Invalid parameter specified.

Other returned error codes are system error conditions.

3.5.5 TDRV009_C_CLEAR_RX_BUFFER

NAME

TDRV009_C_CLEAR_RX_BUFFER – clear the internal receive buffer

DESCRIPTION

This I/O control function removes all received data from the channels receive buffer, and flushes the hardware FIFO as well. The function dependent argument **arg** is not used for this function.

EXAMPLE

```
#include “tdrv009.h”

int                         fd;
int                         result;

/*-------------------------------------------------------------
  Clear receive buffer
  -------------------------------------------------------------*/
result = ioctl(fd, TDRV009_C_CLEAR_RX_BUFFER, 0);

if (result != OK) {
       /* handle ioctl error */
}
```

ERRORS

EIO	Error during reset or init of the receiver’s hardware DMA engine.

Other returned error codes are system error conditions.

3.5.6 TDRV009_C_SET_EXT_XTAL

NAME

TDRV009_C_SET_EXT_XTAL – specify an externally supplied frequency

DESCRIPTION

This I/O control function specifies the frequency of an externally provided clock. This frequency is used for baudrate calculation, and describes the input frequency to the Baud Rate Generator (BRG). The external frequency may be supplied either at input line TxC or RxC. The function dependent argument **arg** passes the clock frequency in Hz.

EXAMPLE

```
#include “tdrv009.h”

int       fd;
int       result;

/*-------------------------------------------------------------
  Specify 1000000Hz (1MHz) as external clock frequency
  -------------------------------------------------------------*/
result = ioctl(fd, TDRV009_C_SET_EXT_XTAL, (char*)1000000);

if (result != OK) {
   /* handle ioctl error */
}
```

ERRORS

EINVAL	Invalid parameter. The specified frequency must be larger than 0.

Other returned error codes are system error conditions.

3.5.7 TDRV009_C_SET_READ_TIMEOUT

NAME

TDRV009_C_SET_READ_TIMEOUT – specify a timeout for read operations

DESCRIPTION

This I/O control function specifies the timeout used for read operations. The parameter **arg** passes an unsigned long value containing the new timeout value in system-ticks to the device driver.

EXAMPLE

```
#include “tdrv009.h”

int           fd;
int           result;

/*-------------------------------------------------------------
  specify the read-timeout (100 clock ticks)
  -------------------------------------------------------------*/
result = ioctl(fd, TDRV009_C_SET_READ_TIMEOUT, (char*)100);

if (result != OK) {
   /* handle ioctl error */
}
```

3.5.8 TDRV009_C_REG_WRITE

NAME

TDRV009_C_REG_WRITE – directly write to a register

DESCRIPTION

This I/O control function writes one 32bit word to the communication controller's register space, relative to the beginning of the register set. The function specific control parameter **arg** is a pointer to a *TDRV009_ADDR_STRUCT* structure.

```
typedef struct
{
   unsigned long    Offset;
   unsigned long    Value;
} TDRV009_ADDR_STRUCT;
```

Offset

This parameter specifies a byte offset into the communication controller's register space. Please refer to the hardware user manual for further information.

Value

This 32bit word will be written to the communication controller's register space.

Modifying register contents may result in communication problems, system crash or other unexpected behavior.

EXAMPLE

```
#include “tdrv009.h”

int                     fd;
int                     result;
TDRV009_ADDR_STRUCT     AddrBuf;

/*-------------------------------------------------
  Write a 32bit value (FIFO Control Register 4)
  -------------------------------------------------*/
AddrBuf.Offset = 0x0034;
AddrBuf.Value  = 0xffffffff;

result = ioctl(fd, TDRV009_C_REG_WRITE, (char*)&AddrBuf);

if (result != OK) {
   /* handle ioctl error */
}
```

ERRORS

EINVAL	The specified offset is invalid.

Other returned error codes are system error conditions.

3.5.9 TDRV009_C_REG_READ

NAME

TDRV009_C_REG_READ – directly read from a register

DESCRIPTION

This I/O control function reads one 32bit word from the communication controller's register space, relative to the beginning of the register set. The function specific control parameter **arg** is a pointer to a *TDRV009_ADDR_STRUCT* structure.

```
typedef struct
{
   unsigned long    Offset;
   unsigned long    Value;
} TDRV009_ADDR_STRUCT;
```

Offset

This parameter specifies a byte offset into the communication controller's register space. Please refer to the hardware user manual for further information.

Value

This parameter returns the 32bit word from the communication controller's register space.

EXAMPLE

```
#include “tdrv009.h”

int                     fd;
int                     result;
TDRV009_ADDR_STRUCT     AddrBuf;

/*-----------------------------------------------
  Read a 32bit value (Version Register)
  -----------------------------------------------*/
AddrBuf.Offset = 0x00F0;

result = ioctl(fd, TDRV009_C_REG_READ, (char*)&AddrBuf);

if (result == OK) {
     printf( “Value = 0x%lX\n”, AddrBuf.Value );
} else {
     /* handle ioctl error */
}
```

ERRORS

EINVAL	The specified offset is invalid

Other returned error codes are system error conditions.

3.5.10 TDRV009_C_SCC_REG_WRITE

NAME

TDRV009_C_SCC_REG_WRITE – directly write to an SCC register

DESCRIPTION

This I/O control function writes one 32bit word to the communication controller's register space, relative to the beginning of the specific channel's SCC register set. The function specific control parameter **arg** is a pointer to a *TDRV009_ADDR_STRUCT* structure.

```
typedef struct
{
   unsigned long    Offset;
   unsigned long    Value;
} TDRV009_ADDR_STRUCT;
```

Offset

This parameter specifies a byte offset into the communication controller's channel SCC register space. Please refer to the hardware user manual for further information.

Value

This 32bit word will be written to the communication controller's channel SCC register space.

Modifying register contents may result in communication problems, system crash or other unexpected behavior.

EXAMPLE

```
#include “tdrv009.h”

int                     fd;
int                     result;
TDRV009_ADDR_STRUCT     AddrBuf;

/*-----------------------------------------------
  Write a 32bit value (Termination Character Register)
  -----------------------------------------------*/
AddrBuf.Offset = 0x0048;
AddrBuf.Value  = (1 << 15) | 0x42;

result = ioctl(fd, TDRV009_C_SCC_REG_WRITE, (char*)&AddrBuf);

if (result != OK) {
   /* handle ioctl error */
}
```

ERRORS

EINVAL	The specified offset is invalid

Other returned error codes are system error conditions.

3.5.11 TDRV009_C_SCC_REG_READ

NAME

TDRV009_C_SCC_REG_READ – directly read from an SCC register

DESCRIPTION

This I/O control function reads one 32bit word from the communication controller's register space, relative to the beginning of the specific channel's SCC register set. The function specific control parameter **arg** is a pointer to a *TDRV009_ADDR_STRUCT* structure.

```
typedef struct
{
   unsigned long    Offset;
   unsigned long    Value;
} TDRV009_ADDR_STRUCT;
```

Offset

This parameter specifies a byte offset into the communication controller's channel SCC register space. Please refer to the hardware user manual for further information.

Value

This parameter returns the 32bit word from the communication controller's channel SCC register space.

EXAMPLE

```
#include “tdrv009.h”

int                     fd;
int                     result;
TDRV009_ADDR_STRUCT     AddrBuf;

/*-----------------------------------------------
  Read a 32bit value (Status Register)
  -----------------------------------------------*/
AddrBuf.Offset = 0x0004;

result = ioctl(fd, TDRV009_C_SCC_REG_READ, (char*)&AddrBuf);

if (result == OK) {
   printf( “Value = 0x%lX\n”, AddrBuf.Value );
} else {
   /* handle ioctl error */
}
```

ERRORS

EINVAL	The specified offset is invalid

Other returned error codes are system error conditions.

3.5.12 TDRV009_C_EEPROM_ WRITE

NAME

TDRV009_C_EEPROM_ WRITE – write value to EEPROM

DESCRIPTION

This I/O control function writes one 16bit word into the onboard EEPROM. The first part of the EEPROM is reserved for factory usage, write accesses to this area will result in an error. The function specific control parameter **arg** is a pointer to a *TDRV009_EEPROM_BUFFER* structure.

```
typedef struct
{
   unsigned long    Offset;
   unsigned long    Value;
} TDRV009_EEPROM_BUFFER;
```

Offset

This parameter specifies a 16bit word offset into the EEPROM.
Following offsets are available:

Offset	Access
00h – 5Fh	R
60h – 7Fh	R / W

Value

This parameter specifies the 16bit word to be written into the EEPROM at the given offset.

EXAMPLE

```
#include “tdrv009.h”

int                     fd;
int                     result;
TDRV009_EEPROM_BUFFER   EepromBuf;

/*-------------------------------------------------
  Write a 16bit value into the EEPROM, offset 0x61
  -------------------------------------------------*/
EepromBuf.Offset = 0x61;
EepromBuf.Value  = 0x1234;

…
```

```
result = ioctl(fd, TDRV009_C_EEPROM_ WRITE, (char*)&EepromBuf);

if (result != OK) {
   /* handle ioctl error */
}
```

ERRORS

EIO	Error writing to the EEPROM.
EACCES	The specified offset address is invalid, or read-only.

Other returned error codes are system error conditions.

3.5.13 TDRV009_C_EEPROM_READ

NAME

TDRV009_C_EEPROM_READ – read value from EEPROM

DESCRIPTION

This I/O control function reads one 16bit word from the onboard EEPROM. The function specific control parameter **arg** is a pointer to a *TDRV009_EEPROM_BUFFER* structure.

```
typedef struct
{
   unsigned long    Offset;
   unsigned long    Value;
} TDRV009_EEPROM_BUFFER;
```

Offset

This parameter specifies a 16bit word offset into the EEPROM.
Following offsets are available:

Offset	Access
00h – 5Fh	R
60h – 7Fh	R / W

Value

This parameter returns the 16bit word from the EEPROM at the given offset.

EXAMPLE

```
#include “tdrv009.h”

int                     fd;
int                     result;
TDRV009_EEPROM_BUFFER   EepromBuf;

/*-------------------------------------------------
  Read a 16bit value from the EEPROM, offset 0
  -------------------------------------------------*/
EepromBuf.Offset = 0;
…
```

```
result = ioctl(fd, TDRV009_C_EEPROM_READ, (char*)&EepromBuf);
if (result == OK) {
     printf( “Value = 0x%X\n”, EepromBuf.Value );
} else {
   /* handle ioctl error */
}
```

ERRORS

EIO	Error reading from EEPROM.
EACCES	Specified offset is invalid, and may not be accessed.

Other returned error codes are system error conditions.

3.5.14 TDRV009_C_RTS_SET

NAME

TDRV009_C_RTS_SET – Assert RTS handshake line

DESCRIPTION

This I/O control function asserts the RTS handshake signal line of the specific channel. This function is not available if the channel is configured for hardware handshaking. The function dependent argument **arg** is not used for this function.

EXAMPLE

```
#include “tdrv009.h”

int           fd;
int           result;

/*-------------------------------------------------------------
  Assert RTS
  -------------------------------------------------------------*/
result = ioctl(fd, TDRV009_C_RTS_SET, (char*)0);

if (result != OK) {
   /* handle ioctl error */
}
```

ERRORS

EPERM	The channel is in handshake mode, so this function is not allowed.

Other returned error codes are system error conditions.

3.5.15 TDRV009_C_RTS_CLEAR

NAME

TDRV009_C_RTS_CLEAR – De-Assert RTS handshake line

DESCRIPTION

This I/O control function de-asserts the RTS handshake signal line of the specific channel. This function is not available if the channel is configured for hardware handshaking. The function dependent argument **arg** is not used for this function.

EXAMPLE

```
#include “tdrv009.h”

int           fd;
int           result;

/*-------------------------------------------------------------
  De-assert RTS
  -------------------------------------------------------------*/
result = ioctl(fd, TDRV009_C_RTS_CLEAR, (char*)0);

if (result != OK) {
   /* handle ioctl error */
}
```

ERRORS

EPERM	The channel is in handshake mode, so this function is not allowed.

Other returned error codes are system error conditions.

3.5.16 TDRV009_C_CTS_GET

NAME

TDRV009_C_CTS_GET – Read state of CTS

DESCRIPTION

This I/O control function returns the current state of the CTS handshake signal line of the specific channel. The function dependent argument **arg** passes a pointer to an *unsigned long* value to the driver. Depending on the state of CTS, either 0 (inactive) or 1 (active) is returned.

EXAMPLE

```
#include “tdrv009.h”

int           fd;
int           result;
unsigned long CtsState;

/*-------------------------------------------------------------
  Read CTS state
  -------------------------------------------------------------*/
result = ioctl(fd, TDRV009_C_CTS_GET, (char*)&CtsState);

if (result == OK) {
     printf( “CTS = %ld\n”, CtsState );
} else {
   /* handle ioctl error */
}
```

3.5.17 TDRV009_C_DTR_SET

NAME

TDRV009_C_DTR_SET – Set DTR signal line (only channel 3)

DESCRIPTION

This I/O control function sets the DTR signal line to HIGH. This function is only available for local module channel 3. The function dependent argument **arg** is not used for this function.

EXAMPLE

```
#include “tdrv009.h”

int           fd;
int           result;

/*-------------------------------------------------------------
  Set DTR to HIGH (only valid for channel 3)
  -------------------------------------------------------------*/

result = ioctl(fd, TDRV009_C_DTR_SET, (char*)0);

if (result != OK) {
   /* handle ioctl error */
}
```

ERRORS

EACCES	This function is not supported by the specific channel.

Other returned error codes are system error conditions.

3.5.18 TDRV009_C_DTR_CLEAR

NAME

TDRV009_C_DTR_CLEAR – Clear DTR signal line (only channel 3)

DESCRIPTION

This I/O control function sets the DTR signal line to LOW. This function is only available for local module channel 3. The function dependent argument **arg** is not used for this function.

EXAMPLE

```
#include “tdrv009.h”

int           fd;
int           result;

/*-------------------------------------------------------------
  Set DTR to LOW (only valid for channel 3)
  -------------------------------------------------------------*/
result = ioctl(fd, TDRV009_C_DTR_CLEAR, (char*)0);

if (result != OK) {
   /* handle ioctl error */
}
```

ERRORS

EACCES	This function is not supported by the specific channel.

Other returned error codes are system error conditions.

3.5.19 TDRV009_C_DSR_GET

NAME

TDRV009_C_DSR_GET – Read state of DSR (only channel 3)

DESCRIPTION

This I/O control function returns the current state of the DSR signal line of the specific channel. This function is only available for local module channel 3. The function dependent argument **arg** passes a pointer to an *unsigned long* value to the driver. Depending on the state of DSR, either 0 (inactive) or 1 (active) is returned.

EXAMPLE

```
#include “tdrv009.h”

int           fd;
int           result;
unsigned long DsrState;

/*-------------------------------------------------------------
  Read DSR state
  -------------------------------------------------------------*/
result = ioctl(fd, TDRV009_C_DSR_GET, (char*)&DsrState);

if (result == OK) {
     printf( “DSR = %ld\n”, DsrState );
} else {
     /* handle the error */
}
```

ERRORS

EACCES	This function is not supported by the specific channel.

Other returned error codes are system error conditions.

3.5.20 TDRV009_C_WAITFORINTERRUPT

NAME

TDRV009_C_WAITFORINTERRUPT – Wait for specific channel interrupt

DESCRIPTION

This I/O control function waits until a specified SCC-interrupt or the timeout occurs. The function specific control parameter **arg** is a pointer to a *TDRV009_WAIT_STRUCT* structure.

```
typedef struct
{
   unsigned long    Interrupts;
   int              Timeout;
} TDRV009_WAIT_STRUCT;
```

Interrupts

This parameter specifies specific interrupt bits to wait for. If one interrupt occurs, the value is returned in this parameter. Please refer to the hardware user manual for further information on the possible interrupt bits.

Timeout

This parameter specifies the time (in system ticks) to wait for an interrupt. If -1 is specified, the function will block indefinitely.

EXAMPLE

```
#include “tdrv009.h”

int                     fd;
int                     result;
TDRV009_WAIT_STRUCT     WaitStruct;

/*-------------------------------------------------
  Wait at least 500 system ticks for a
  CTS Staus Change (CSC) interrupt
  -------------------------------------------------*/
WaitStruct.Interrupts = (1 << 14);
WaitStruct.Timeout    = 500;

…
```

```
result = ioctl(fd, TDRV009_C_WAITFORINTERRUPT, (char*)&WaitStruct);

if (result == OK) {
     printf( “Interrupts = 0x%lX\n”, WaitStruct.Interrupts );
} else {
     /* handle the error */
}
```

ERRORS

EBUSY	Too many simultaneous wait jobs active.

Other returned error codes are system error conditions.

4 Debugging and Diagnostic

If the driver will not work properly, please enable debug outputs by defining the symbols *DEBUG*, *DEBUG_TPMC*, *DEBUG_PCI*, *DEBUG_INT_CFG*, *DEBUG_INT_RX* and *DEBUG_INT_TX*.

The debug output should appear on the console. If not, please check the symbol *KKPF_PORT* in *uparam.h*. This symbol should be configured to a valid COM port (e.g. *SKDB_COM1*).

The debug output displays the device information data for the current major device like this.

```
Bus = 1  Dev = 2  Func = 0
[00] = 035F1498
[04] = 02000000
[08] = 02800000
[0C] = 00005800
[10] = 84000000
[14] = 00000000
[18] = 00000000
[1C] = 00000000
[20] = 00000000
[24] = 00000000
[28] = 00000000
[2C] = 000A1498
[30] = 00000000
[34] = 00000000
[38] = 00000000
[3C] = 0A03010B
Found a TDRV009 compatible module, BusNo=1, DevNo=2
TDRV009: Board #0: ctrl_space = 0xB4600000
TDRV009: found TPMC863-10 (FPGA-Core version 0x4)
TDRV009:    XTAL1: 14745600
TDRV009:    XTAL2: 24000000
TDRV009:    XTAL3: 10000000
```

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