The Embedded I/O Company

TEWS TECHNOLOGIES

TPMC680-SW-42

VxWorks Device Driver

8 x 8 Bit Digital I/O

Version 3.0.x

User Manual

Issue 3.0.1

March 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

TPMC680-SW-42

VxWorks Device Driver

8 x 8 Bit Digital I/O

Supported Modules:
TPMC680-10

Issue	Description	Date
1.0	First Issue	November 16, 2002
1.1	Read() parameter description changed	December 11, 2002
1.2	Flags “empty / not full” corrected	December 13, 2002
2.0.0	New driver start and device creation functions, new file list	May 24, 2006
3.0.0	Support for VxBus and API description added, general revision read() and write() functions replaced by ioctl() functions	February 3, 2010
3.0.1	Legacy vs. VxBus Driver modified	March 26, 2010

Table of Contents

1 Introduction

The TPMC680-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 releases 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.

The TPMC680-SW-42 device driver supports the following features:

- direct reading for input ports (8 bit / synchronous mode)
- direct writing for output ports (8 bit / synchronous mode)
- buffered read for input ports (16/32 bit handshake mode)
- buffered write for output ports (16/32 bit handshake mode)
- configuring ports
- connecting a function with parameter to an input event for input ports (8 bit / synchronous mode)

The TPMC680-SW-42 supports the modules listed below:

TPMC680-10	8 x 8 Bit Digital I/O	PMC

To get more information about the features and use of TPMC680 devices it is recommended to read the manuals listed below.

TPMC680 User manual

TPMC680 Engineering Manual

2 Installation

Following files are located on the distribution media:

Directory path 'TPMC680-SW-42':

TPMC680-SW-42-3.0.1.pdf	PDF copy of this manual
TPMC680-SW-42-VXBUS.zip	Zip compressed archive with VxBus driver sources
TPMC680-SW-42-LEGACY.zip	Zip compressed archive with legacy driver sources
ChangeLog.txt	Release history
Release.txt	Release information

The archive TPMC680-SW-42-VXBUS.zip contains the following files and directories:

Directory path './tews/tpmc680':

tpmc680drv.c	TPMC680 device driver source
tpmc680def.h	TPMC680 driver include file
tpmc680.h	TPMC680 include file for driver and application
tpmc680api.c	TPMC680 API file
Makefile	Driver Makefile
40tpmc680.cdf	Component description file for VxWorks development tools
tpmc680.dc	Configuration stub file for direct BSP builds
tpmc680.dr	Configuration stub file for direct BSP builds
include/tvxbHal.h	Hardware dependent interface functions and definitions
apps/tpmc680exa.c	Example application

The archive TPMC680-SW-42-LEGACY.zip contains the following files and directories:

Directory path './tpmc680':

tpmc680drv.c	TPMC680 device driver source
tpmc680def.h	TPMC680 driver include file
tpmc680.h	TPMC680 include file for driver and application
tpmc680api.c	TPMC680 API file
tpmc680pci.c	TPMC680 PCI MMU mapping for Intel x86 based targets
tpmc680exa.c	Example application
tpmc680init.c	Legacy driver initialization
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 TPMC680-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 TPMC680 device driver is located in the vendor and driver-specific directory *installDir/vxworks-6.x/target/3rdparty/tews/tpmc680.*

At this point the TPMC680 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/tpmc680

(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\tpmc680
```

```
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 TPMC680 driver with the VxWorks development tools (Workbench), the component configuration file *40tpmc680.cdf* must be copied to the directory *installDir/vxworks-6.x/target/config/comps/VxWorks*.

```
C:> cd \WindRiver\vxworks-6.7\target\3rdparty\tews\tpmc680
C:> copy 40tpmc680.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 TPMC680 driver can be included in VxWorks projects by selecting the *"TEWS TPMC680 Driver"* component 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 TPMC680 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\tpmc680
C:> copy tpmc680.dc \Windriver\vxworks-6.7\target\config\comps\src\hwif
C:> copy tpmc680.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 TPMC680-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 TPMC680-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 tpmc680 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 TPMC680 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 TPMC680 PCI memory spaces prior the MMU initialization (*usrMmuInit()*) is done.

The C source file **tpmc680pci.c** contains the function *tp680PciInit().* This routine finds out all TPMC680 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 *tp680PciInit()* 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 TPMC680 PCI spaces remains unmapped and an access fault occurs during driver initialization.

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

```
tp680PciInit();
```

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.

2.4 Configuration of FIFO depth

The depth of the FIFOs can be configured with the define *TP680_IOBUFSIZE* in tpmc680.h. The value defines the number of values that can be stored in each of the FIFOs. Changing this value will change the size of the used system memory for each devices.

After changing the definition of *TP680_IOBUFSIZE* the driver must be rebuilt to take the changes effect.

2.5 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	depends on FIFO size
Stack	< 1 KB	---

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 tpmc680Open()

Name

tpmc680Open() – opens a device.

Synopsis

```
TPMC680_DEV tpmc680Open
(
      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 first TPMC680 device is named “/tpmc680/0”, the second device is named “/tpmc680/1” and so on.

Example

```
#include “tpmc680.h”

TPMC680_DEV   pDev;

/*
** open file descriptor to device
*/
pDev = tpmc680Open(“/tpmc680/0”);
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 tpmc680Close()

Name

tpmc680Close() – closes a device.

Synopsis

```
int tpmc680Close
(
     TPMC680_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 “tpmc680.h”

TPMC680_DEV   pDev;
int           result;

/*
** close file descriptor to device
*/
result = tpmc680Close(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 tpmc680Read

Name

tpmc680Read – Read data from input port

Synopsis

```
STATUS tpmc680Read
(
    TPMC680_DEV          pDev,
    TP680_IO_STRUCT      *pReadBuf
)
```

Description

This function reads data from the device. Depending on the configured port mode, the data will be read directly or buffered.

Parameters

pDev

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

pReadBuf

This argument points to a *TP680_IO_STRUCT* buffer where the value will be returned. The data structure is defined as follows:

```
typedef struct
{
    int                 port;                   /* 0.. 7 */
    union
    {
        unsigned char   val8;                   /* 8 bit wide port */
        struct
        {
            unsigned short  *buf;
            unsigned long   size;
        } val16;                                /* 16 bit wide port (mode 1) */
        struct
        {
            unsigned long   *buf;
            unsigned long   size;
        } val32;                                /* 32 bit wide port (mode 2) */
        struct
        {
            unsigned long   val64_l;
            unsigned long   val64_h;
        } val64;                                /* 64 bit wide port (synchon) */
    } u;
} TP680_IO_STRUCT, *PTP680_IO_STRUCT;
```

port

This parameter specifies the port where data shall be read from. The port number is always between 0 and 7. Dependent from the module configuration some ports are connected to others and only the first port is accessible with this function. If the port is connected or configured for output, *read()* will return *ERROR*. The data order is always 'big endian' (e.g. a 32 bit access on port 0 will return the input from port 0 / line 0 at the LSB and the input from port 4 / line 7 at the MSB.)

Port Mode Configuration [1)]		**Connected to Port**							
Port 0	**Port 2**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BYTE	BYTE	7	6	5	4	3	2	1	0
HS16BIT	BYTE	7	6	5[2)]	4[2)]	3	2	0	0
BYTE	HS16BIT	7	6	5[2)]	4[2)]	2	2	1	0
HS16BIT	HS16BIT	7	6	5[2)]	4[2)]	2	2	0	0
HS32BIT	---	7	6	5[2)]	4[2)]	0	0	0	0
SYNCHRON	---	0	0	0	0	0	0	0	0

[1)] The port mode configurations are assigned to the following driver configurations values:

Port Mode Configuration	Driver Configuration Value (tpmc680def.h)
BYTE	*TP680_IO_MODE_BYTE*
HS16BIT	*TP680_IO_MODE_HS16BIT*
HS32BIT	*TP680_IO_MODE_HS32BIT*
SYNCHRON	*TP680_IO_MODE_SYNCHRON*

[2)] Bits 0/1 may be used for HS and are unreadable.

u

This union allows handling the different input sizes and modes. Description follows below.

val8

This selection will be used for direct byte input. The value of the specified port will be returned in this parameter.

val16

This selection will be used for buffered 16bit input. The structure contains a pointer to a buffer (***buf***) of *unsigned short* and the size (***size*)** of this buffer. ***size*** will be specified in words. The read function will fill the buffer until the buffer is filled or there are no more data in the FIFO of the selected channel.

val32

This selection will be used for buffered 32bit input. The structure contains a pointer to a buffer (***buf***) of *unsigned long* and the size (***size*)** of this buffer. ***size*** will be specified in long words. The read function will fill the buffer until the buffer is filled or there are no more data in the FIFO of the selected channel.

val64

This selection will be used for direct 64bit synchronous input. The structure has two elements. The value of port 3..0 will be returned in ***val64_l*** where the LSB is the value of port 0 and the MSB is the value of port 3. The value of port 7..4 will be returned in ***val64_h*** where the LSB is the value of port 4 and the MSB is the value of port 7.

Example

```
#include “tpmc680.h”

TPMC680_DEV        pDev;
int                i;
int                retval;
TP680_IO_STRUCT    ioBuf;
unsigned short     usBuf[10];
unsigned long      ulBuf[10];

/*-----------------------------
  Read a byte value from port 5
  -----------------------------*/
ioBuf.port = 5;

retval = tpmc680Read( pDev, (int)&ioBuf );
if (retval != ERROR)
{
     printf("Port %d: %02Xh\n", ioBuf.port, ioBuf.u.val8);
} else {
     /* Handle Error */
}
…
/*-----------------------------------------
  Fill buffer with values (16Bit) from port 2
  -----------------------------------------*/
ioBuf.port = 2;
ioBuf.u.val16.size = 10;
ioBuf.u.val16.buf = usBuf;

retval = tpmc680Read( pDev, (int)&ioBuf );
if (retval != ERROR)
{
     for (i = 0; i < 10; i++)
     {
          printf("Val[%d]: %04Xh\n", i, usBuf[i]);
     }
} else {
     /* Handle Error */
}

…
```

```
/*----------------------------------------
  Fill buffer with values (32Bit) from port 0
  ----------------------------------------*/
ioBuf.port = 0;
ioBuf.u.val32.size = 10;
ioBuf.u.val32.buf = ulBuf;

retval = tpmc680Read( pDev, (int)&ioBuf );
if (retval != ERROR)
{
     for (i = 0; i < 10; i++)
     {
          printf("Val[%d]: %08lXh\n", i, ulBuf[i]);
     }
} else {
     /* Handle Error */
}

…

/*-----------------------------------
  Read port value (64Bit) from port 0
  -----------------------------------*/
ioBuf.port = 0;

retval = tpmc680Read( pDev, (int)&ioBuf );
if (retval != ERROR)
{
     printf("Val[%d]: %08lX%08lXh \n",
               ioBuf.u.val64.val64_h,
               ioBuf.u.val64.val64_l);
} else {
     /* Handle Error */
}
```

RETURNS

Number of bytes read 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) or a driver set error code described below.

Error code	Description
EBADF	The device handle is invalid
S_tp680Dev_IPORT	Illegal port number specified
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_IDIRECTION	Illegal direction specified

3.2.2 tpmc680Write

Name

tpmc680Write – Write data to output port

Synopsis

```
STATUS tpmc680Write
(
    TPMC680_DEV          pDev,
    TP680_IO_STRUCT      *pReadBuf
)
```

Description

This function writes data to the device. Depending on the configured port mode, the data will be written directly or buffered.

Parameters

pDev

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

pReadBuf

This argument points to a *TP680_IO_STRUCT* buffer where the data is stored. The data structure is defined as follows:

```
typedef struct
{
    int                 port;                   /* 0.. 7 */
    union
    {
        unsigned char   val8;                   /* 8 bit wide port */
        struct
        {
            unsigned short  *buf;
            unsigned long   size;
        } val16;                                /* 16 bit wide port (mode 1) */
        struct
        {
            unsigned long   *buf;
            unsigned long   size;
        } val32;                                /* 32 bit wide port (mode 2) */
        struct
        {
            unsigned long   val64_l;
            unsigned long   val64_h;
        } val64;                                /* 64 bit wide port (synchon) */
    } u;
} TP680_IO_STRUCT, *PTP680_IO_STRUCT;
```

port

This parameter specifies the port where data shall be written to. The port number is always between 0 and 7. Dependent from the module configuration some ports are connected to others and only the first port is accessible with this function. If the port is connected or configured as input, *write()* will return *ERROR*. The data order is always 'big endian' (e.g. a 32 bit access on port 0 will write the output of port 0 / line 0 with the LSB and the output of port 4 / line 7 with the MSB.)

Port Mode Configuration [1]		**Connected to Port**							
Port 0	**Port 2**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BYTE	BYTE	7	6	5	4	3	2	1	0
HS16BIT	BYTE	7	6	5[2]	4[2]	3	2	0	0
BYTE	HS16BIT	7	6	5[2]	4[2]	2	2	1	0
HS16BIT	HS16BIT	7	6	5[2]	4[2]	2	2	0	0
HS32BIT	---	7	6	5[2]	4[2]	0	0	0	0
SYNCHRON	---	0	0	0	0	0	0	0	0

[1] The port mode configurations are assigned to the following driver configurations values:

Port Mode Configuration	Driver Configuration Value (tpmc680def.h)
BYTE	*TP680_IO_MODE_BYTE*
HS16BIT	*TP680_IO_MODE_HS16BIT*
HS32BIT	*TP680_IO_MODE_HS32BIT*
SYNCHRON	*TP680_IO_MODE_SYNCHRON*

[2)] Bits 0/1 may be used for HS and are unwriteable.

u

This union allows handling the different output sizes and modes. Description follows below.

val8

This selection will be used for direct byte output. The specified value will be written to the specified port.

val16

This selection will be used for buffered 16bit output. The structure contains a pointer to a buffer (***buf***) of *unsigned short* and the size (***size*)** of this buffer. ***size*** will be specified in words. The write function will copy the buffer into a driver internal FIFO and output this value on the specified port. The function will return immediately.

val32

This selection will be used for buffered 32bit output. The structure contains a pointer to a buffer (***buf***) of *unsigned long* and the size (***size*)** of this buffer. ***size*** will be specified in words. The write function will copy the buffer into a driver internal FIFO and output this value on the specified port. The function will return immediately.

Example

```
#include “tpmc680.h”

TPMC680_DEV        pDev;
int                i;
int                retval;
TP680_IO_STRUCT    ioBuf;
unsigned short     usBuf[10];
unsigned long      ulBuf[10];

/*-------------------------
  Write 0x12 to byte port 5
  -------------------------*/
ioBuf.port = 5;
ioBuf.u.val8 = 0x12;

retval = tpmc680Write( pDev, (int)&ioBuf );
if (retval != ERROR)
{
     /* Output OK */;
}
else
{
     /* Handle Error */
}

…
```

```
…

/*----------------------------------------
  Write the buffered values to 16bit port 2
  ----------------------------------------*/
usBuf[0] = 0x1111;
usBuf[1] = 0x2222;
usBuf[2] = 0x3333;
ioBuf.port = 2;
ioBuf.u.val16.size = 3;
ioBuf.u.val16.buf = usBuf;

retval = tpmc680Write( pDev, (int)&ioBuf );
if (retval != ERROR)
{
     /* Output OK */;
}
else
{
     /* Handle Error */
}

…

/*----------------------------------------
  Write the buffered values to 32bit port 0
  ----------------------------------------*/
ulBuf[0] = 0x11111111;
ulBuf[1] = 0x22222222;
ulBuf[2] = 0x33333333;
ioBuf.port = 0;
ioBuf.u.val16.size = 3;
ioBuf.u.val32.buf = ulBuf;

retval = tpmc680Write( pDev, (int)&ioBuf );
if (retval != ERROR)
{
     /* Output OK */;
}
else
{
     /* Handle Error */
}

…
```

…

```
/*-----------------------------------
  Write port value (64Bit) to port 0
  -----------------------------------*/
ioBuf.port = 0;
ioBuf.u.val64.val64_h = 0x12345678;
ioBuf.u.val64.val64_l = 0x87654321;

retval = tpmc680Write( pDev, (int)&ioBuf );
if (retval != ERROR)
{
     /* Output OK */;
}
else
{
     /* Handle Error */
}
```

RETURNS

Number of bytes read 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) or a driver set error code described below.

Error code	Description
EBADF	The device handle is invalid
S_tp680Dev_IPORT	Illegal port number specified
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_IDIRECTION	Illegal direction specified

3.2.3 tpmc680SetPortDirection

Name

tpmc680SetPortDirection – Setup port direction

Synopsis

```
STATUS tpmc680SetPortDirection
(
    TPMC680_DEV         pDev,
    int                 port,
    unsigned long       direction
)
```

Description

This function sets the port direction of the specified port. The mode of the port will be left untouched. If the port is configured in handshake mode, the FIFO will be flushed.

Parameters

pDev

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

port

This parameter specifies the port which shall be configured. The port number is always between 0 and 7. Dependent from the module configuration some ports are connected to others and only the first port is accessible with this function. If the port is connected, the function will return *ERROR*.

Port Mode Configuration [1)]		**Connected to Port**							
Port 0	**Port 2**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BYTE	BYTE	7	6	5	4	3	2	1	0
HS16BIT	BYTE	7	6	5[2)]	4[2)]	3	2	0	0
BYTE	HS16BIT	7	6	5[2)]	4[2)]	2	2	1	0
HS16BIT	HS16BIT	7	6	5[2)]	4[2)]	2	2	0	0
HS32BIT	---	7	6	5[2)]	4[2)]	0	0	0	0
SYNCHRON	---	0	0	0	0	0	0	0	0

[1)] The port mode configurations are assigned to the following driver configurations values:

Port Mode Configuration	Driver Configuration Value (tpmc680def.h)
BYTE	*TP680_IO_MODE_BYTE*
HS16BIT	*TP680_IO_MODE_HS16BIT*
HS32BIT	*TP680_IO_MODE_HS32BIT*
SYNCHRON	*TP680_IO_MODE_SYNCHRON*

[2] Bits 0/1 may be used for HS and are unconfigurable.

direction

This parameter specifies the new port direction. The following values are valid.

Value	Description
TP680_IO_DIR_IN	Configure specified port for input
TP680_IO_DIR_OUT	Configure specified port for output

Example

```
#include “tpmc680.h”

TPMC680_DEV        pDev;
int                retval;

/*----------------------------
  configure port 3 for output
  ----------------------------*/
retval = tpmc680SetPortDirection( pDev, 3, TP680_IO_DIR_OUT );
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the 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) or a driver set error code described below.

Error code	Description
EBADF	The device handle is invalid
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_IPORTMODE	Illegal port mode specified
S_tp680Dev_IPORT	Illegal port number specified

3.2.4 tpmc680SetPortMode

Name

tpmc680SetPortMode– Setup port mode

Synopsis

```
STATUS tpmc680SetPortMode
(
    TPMC680_DEV         pDev,
    int                 port,
    unsigned long       mode,
    unsigned long       hsFlags
)
```

Description

This function sets up the mode of a specified port. Calling this function will flush the FIFO if the previous mode has been a handshake (buffered) mode.

For a detailed description of the modes refer to the TPMC680 User Manual

Parameters

pDev

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

port

This parameter specifies the port which shall be configured. The port number is always between 0 and 7. Dependent from the actual module configuration some ports are connected to others and only the first port is accessible with this function. If the new mode connects less ports then the mode before, the unconfigured ports will be set to BYTE mode. The previous directions of the ports will be kept.

Port Mode Configuration [1]		**Connected to Port**							
Port 0	**Port 2**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BYTE	BYTE	7	6	5	4	3	2	1	0
HS16BIT	BYTE	7	6	5[2]	4[2]	3	2	0	0
BYTE	HS16BIT	7	6	5[2]	4[2]	2	2	1	0
HS16BIT	HS16BIT	7	6	5[2]	4[2]	2	2	0	0
HS32BIT	---	7	6	5[2]	4[2]	0	0	0	0
SYNCHRON	---	0	0	0	0	0	0	0	0

[1] The port mode configurations are assigned to the following driver configurations values:

Port Mode Configuration	**Driver Configuration Value (tpmc680def.h)**
BYTE	*TP680_IO_MODE_BYTE*
HS16BIT	*TP680_IO_MODE_HS16BIT*
HS32BIT	*TP680_IO_MODE_HS32BIT*
SYNCHRON	*TP680_IO_MODE_SYNCHRON*

[2] Bits 0/1 may be used for HS and are unconfigurable.

mode

This parameter specifies the new port mode.

Value	**Description**
TP680_IO_MODE_BYTE	Configures a port as byte I/O, no other ports will be connected. Valid for port 0..7. (The driver will support direct I/O)
TP680_IO_MODE_HS16BIT	Configures a port as 16 bit handshake port. This mode is valid for port 0 (port 1 will be connected) and port 2 (port 3 will be connected). Port 4 will be set to input (Handshake). (The driver will support buffered I/O)
TP680_IO_MODE_HS32BIT	Configures a port as 32 bit handshake port. This mode is valid for port 0 (port 1, 2 and 3 will be connected). Port 4 will be set to input (Handshake). (The driver will support buffered I/O)
TP680_IO_MODE_SYNCHRON	Configures the module for synchronous byte I/O. All ports will be connected. The only valid port is port 0. This mode will be used for 64 bit parallel I/O. (The driver will support direct I/O)

hsFlags

This flags will specify the output handshake mode.

Value	Description
TP680_IO_HSFLAG_NO	The handshake output will not be used. (Port 5 will be left in the previous mode)
TP680_IO_HSFLAG_INTERLOCKED	The handshake output will be used in interlocked mode. Port 5 will be used for output.
TP680_IO_HSFLAG_PULSED	The handshake output will be used in pulsed mode. Port 5 will be used for output.
TP680_IO_HSFIFOEV_NOTFULL	The FIFO event will be set to “Not Full”. This value must be ORed with *TP680_IO_HSFLAG_INTERLOCKED or TP680_IO_HSFLAG_PULSED.*
TP680_IO_HSFIFOEV_EMPTY	The FIFO event will be set to “Empty”. This value must be ORed with *TP680_IO_HSFLAG_INTERLOCKED or TP680_IO_HSFLAG_PULSED.*

Example

```
#include “tpmc680.h”

TPMC680_DEV        pDev;
int                retval;

/*-------------------------------------------------------------
  configure port 2 for 16bit handshake mode
  the output handshake shall interlocked and the event shall
  occur on “not full”
  -------------------------------------------------------------*/
retval = tpmc680SetPortDirection( pDev, 2, TP680_IO_MODE_HS16BIT,
TP680_IO_HSFLAG_INTERLOCKED | TP680_IO_HSFIFOEV_NOTFULL );
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the 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) or a driver set error code described below.

Error code	Description
EBADF	The device handle is invalid
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_IPORTMODE	Illegal port mode specified
S_tp680Dev_IPORT	Illegal port number specified

3.2.5 tpmc680SetInterruptRoutine

Name

tpmc680SetInterruptRoutine – Connect application interrupt function to input event

Synopsis

```
STATUS tpmc680SetInterruptRoutine
(
        TPMC680_DEV             pDev,
        int                     port,
        int                     line,
        unsigned long           edge,
        void                    *function,
        unsigned long           parameter
)
```

Description

This function connects a user specified function to a specified interrupt. The selected I/O line must be part of a port configured for byte wise (8/64bit) input. The function will be disconnected if it is specified with the below argument or if the port mode is no longer input and byte wise.

The function will be called while the interrupt is handled. Keep this function short and do not use any operations that will block the task.

Parameters

pDev

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

port

This parameter specifies the port where the interrupt function shall be installed to.

line

This parameter specifies the I/O line where the interrupt function shall be installed to.

edge

This parameter specifies the event on which the interrupt shall be generated.

Value	Description
TP680_IO_EDGE_NO	Disconnect installed function from interrupt.
TP680_IO_EDGE_HI	Call specified function on a low to high event of the specified input line.
TP680_IO_EDGE_LO	Call specified function on a high to low event of the specified input line.
TP680_IO_EDGE_ANY	Call specified function on any event of the specified input line.

function

This parameter specifies the entry point of the interrupt callback function.

parameter

The function will be called with this parameter.

Example

```
#include “tpmc680.h”

void intfunction (unsigned long param)
{
     /* Handle interrupt */
}

…

TPMC680_DEV        pDev;
int                retval;

/*-------------------------------------------------------
  connect interrupt function to port 5 line 6
  the function shall be called on the high to low transition
  the parameter shall be 0x1234
  -------------------------------------------------------*/
retval = tpmc680SetInterruptRoutine( pDev,
              5,
              6,
              TP680_IO_EDGE_LO,
              intfunction,
              0x1234);

…
```

…

```
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the 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) or a driver set error code described below.

Error code	Description
EBADF	The device handle is invalid
S_tp680Dev_IPORT	Illegal port number specified
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_ILINE	Illegal line number specified
S_tp680Dev_IEDGE	Illegal interrupt event specified
S_tp680Dev_IRQBUSY	The specified input line is already connected with a callback function

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 TPMC680 driver. For the VxBus-enabled TPMC680 driver, the driver will be installed automatically in the I/O system and devices will be created as needed for detected modules.

4.1 tp680Drv()

NAME

tp680Drv() - installs the TPMC680 driver in the I/O system

SYNOPSIS

#include “tpmc680.h”

STATUS tp680Drv(void)

DESCRIPTION

This function searches for devices on the PCI bus, allocates driver and device resources, initializes them and installs the TPMC680 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 “tpmc680.h”

STATUS            result;

/*-------------------
  Initialize Driver
  -------------------*/
result = tp680Drv();
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
ENXIO	No TPMC680-10 found

SEE ALSO

VxWorks Programmer's Guide: I/O System

4.2 tp680DevCreate()

NAME

tp680DevCreate() – Add a TPMC680 device to the VxWorks system

SYNOPSIS

```
#include “tpmc680.h”

STATUS tp680DevCreate
(
        char        *name,
        int         devIdx,
        int         funcType,
        void        *pParam
)
```

DESCRIPTION

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

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

PARAMETER

name

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

devIdx

This index number specifies the device to add to the system. Channel numbers will be assigned in the order the VxWorks *pciFindDevice()* function will find the devices.

Example: (A system with 2 TPMC680-10) will assign the following device indices:

Module	Device Index
TPMC680-10 (1st)	0
TPMC680-10 (2nd)	1

funcType

This parameter is unused and should be set to *0*.

pParam

This parameter is unused and should be set to *NULL*.

EXAMPLE

```
#include "tpmc680.h"

STATUS            result;

/*-----------------------------------------------------
  Create the device "/tpmc680/0" for the first device
  -----------------------------------------------------*/
result = tp680DevCreate(    "/tpmc680/0",
                            0,
                            0,
                            NULL);
if (result == OK)
{
     /* Device successfully created */
}
else
{
     /* Error occurred when creating the device */
}
```

RETURNS

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

ERROR CODES

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

Error code	Description
S_ioLib_NO_DRIVER	The TPMC680 driver has not been started
ENXIO	Specified device not found
EBUSY	The device has already been initialized

SEE ALSO

VxWorks Programmer's Guide: I/O System

4.3 tp680PciInit()

NAME

tp680PciInit() – Generic PCI device initialization

SYNOPSIS

void tp680PciInit()

DESCRIPTION

This function is required only for Intel x86 VxWorks platforms. The purpose is to setup the MMU mapping for all required TPMC680 PCI spaces (base address register) and to enable the TPMC680 device for access.

The global variable *tp680Status* 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 tp680Status is equal to the number of mapped PCI spaces
0	No TPMC680 device found
< 0	Initialization failed. The value of (tp680Status & 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 tp680PciInit();

…

tp680PciInit();

…
```

5 Basic I/O Functions

The VxWorks basic I/O interface functions are useable with the TPMC680 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 TPMC680 device, a file descriptor must be opened by invoking the basic I/O function *open().*

PARAMETER

name

Specifies the device which shall be opened, the name specified in tp680DevCreate() must be used

flags

Not used

mode

Not used

EXAMPLE

```
int      fd;

/*----------------------------------------
  Open the device named "/tpmc680/0" for I/O
  ----------------------------------------*/
fd = open("/tpmc680/0", 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 “tpmc680.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
FIO_TP680_READ	Read data from input port
FIO_TP680_WRITE	Write data to output port
FIO_TP680_SET_DIR	Setup port direction
FIO_TP680_SET_IRQ	Connect application interrupt function to input event
FIO_TP680_SET_MODE	Setup port mode
FIO_TP680_GET_DEBUG	Undocumented debug function

arg

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

RETURNS

OK or ERROR, otherwise the return value is described below with the function. 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_tp680Dev_ICMD	Illegal Ioctl() command

SEE ALSO

ioLib, basic I/O routine - ioctl()

5.3.1 FIO_TP680_READ

This I/O control function reads data from the specified device. Depending on the configured port mode, the data will be read directly or buffered. The data size depends on the selected mode. The function specific control parameter **arg** is a pointer on a *TP680_IO_STRUCT* structure.

```
typedef struct
{
        int                     port;                           /* 0.. 7 */
        union
        {
                unsigned char       val8;                       /* 8 bit wide port */
                struct
                {
                        unsigned short      *buf;
                        unsigned long       size;
                } val16;                                        /* 16 bit wide port (mode 1) */
                struct
                {
                        unsigned long       *buf;
                        unsigned long       size;
                } val32;                                        /* 32 bit wide port (mode 2) */
                struct
                {
                        unsigned long       val64_l;
                        unsigned long       val64_h;
                } val64;                                        /* 64 bit wide port (synchon) */
        } u;
} TP680_IO_STRUCT, *PTP680_IO_STRUCT;
```

port

This parameter specifies the port where data shall be read from. The port number is always between 0 and 7. Dependent from the module configuration some ports are connected to others and only the first port is accessible with this function. If the port is connected or configured for output, *read()* will return *ERROR*. The data order is always 'big endian' (e.g. a 32 bit access on port 0 will return the input from port 0 / line 0 at the LSB and the input from port 4 / line 7 at the MSB.)

Port Mode Configuration [1]		**Connected to Port**							
Port 0	**Port 2**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BYTE	BYTE	7	6	5	4	3	2	1	0
HS16BIT	BYTE	7	6	5[2]	4[2]	3	2	0	0
BYTE	HS16BIT	7	6	5[2]	4[2]	2	2	1	0
HS16BIT	HS16BIT	7	6	5[2]	4[2]	2	2	0	0
HS32BIT	---	7	6	5[2]	4[2]	0	0	0	0
SYNCHRON	---	0	0	0	0	0	0	0	0

[1] The port mode configurations are assigned to the following driver configurations values:

Port Mode Configuration	**Driver Configuration Value (tpmc680def.h)**
BYTE	*TP680_IO_MODE_BYTE*
HS16BIT	*TP680_IO_MODE_HS16BIT*
HS32BIT	*TP680_IO_MODE_HS32BIT*
SYNCHRON	*TP680_IO_MODE_SYNCHRON*

[2] Bits 0/1 may be used for HS and are unreadable.

u

This union allows handling the different input sizes and modes. Description follows below.

val8

This selection will be used for direct byte input. The value of the specified port will be returned in this parameter.

val16

This selection will be used for buffered 16bit input. The structure contains a pointer to a buffer (***buf***) of *unsigned short* and the size (***size*)** of this buffer. ***size*** will be specified in words. The read function will fill the buffer until the buffer is filled or there are no more data in the FIFO of the selected channel.

val32

This selection will be used for buffered 32bit input. The structure contains a pointer to a buffer (***buf***) of *unsigned long* and the size (***size*)** of this buffer. ***size*** will be specified in long words. The read function will fill the buffer until the buffer is filled or there are no more data in the FIFO of the selected channel.

val64

This selection will be used for direct 64bit synchronous input. The structure has two elements. The value of port 3..0 will be returned in ***val64_l*** where the LSB is the value of port 0 and the MSB is the value of port 3. The value of port 7..4 will be returned in ***val64_h*** where the LSB is the value of port 4 and the MSB is the value of port 7.

EXAMPLE

```
#include “tpmc680.h”

int                fd;
int                i;
int                retval;
TP680_IO_STRUCT    ioBuf;
unsigned short     usBuf[10];
unsigned long      ulBuf[10];

/*-----------------------------
  Read a byte value from port 5
  -----------------------------*/
ioBuf.port = 5;

retval = ioctl(fd, FIO_TP680_READ, (int)&ioBuf);
if (retval != ERROR)
{
     printf("Port %d: %02Xh\n", ioBuf.port, ioBuf.u.val8);
}
else
{
     /* Handle Error */
}
…
/*-------------------------------------------
  Fill buffer with values (16Bit) from port 2
  -------------------------------------------*/
ioBuf.port = 2;
ioBuf.u.val16.size = 10;
ioBuf.u.val16.buf = usBuf;

retval = ioctl(fd, FIO_TP680_READ, (int)&ioBuf);
if (retval != ERROR)
{
     for (i = 0; i < 10; i++)
     {
          printf("Val[%d]: %04Xh\n", i, usBuf[i]);
     }
}
else
{
     /* Handle Error */
}
```

```
…

/*----------------------------------------
  Fill buffer with values (32Bit) from port 0
  ----------------------------------------*/
ioBuf.port = 0;
ioBuf.u.val32.size = 10;
ioBuf.u.val32.buf = ulBuf;

retval = ioctl(fd, FIO_TP680_READ, (int)&ioBuf);
if (retval != ERROR)
{
     for (i = 0; i < 10; i++)
     {
          printf("Val[%d]: %08lXh\n", i, ulBuf[i]);
     }
}
else
{
     /* Handle Error */
}

…

/*-----------------------------------
  Read port value (64Bit) from port 0
  -----------------------------------*/
ioBuf.port = 0;

retval = ioctl(fd, FIO_TP680_READ, (int)&ioBuf);
if (retval != ERROR)
{
     printf("Val[%d]: %08lX%08lXh \n",
               ioBuf.u.val64.val64_h,
               ioBuf.u.val64.val64_l);
}
else
{
     /* Handle Error */
}
```

RETURNS

Number of bytes read 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) or a driver set error code described below.

Error code	Description
S_tp680Dev_IPORT	Illegal port number specified
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_IDIRECTION	Illegal direction specified

5.3.2 FIO_TP680_WRITE

This I/O control function writes data to the specified device. Depending on the configured port mode, the data will be written direct or buffered. The data size depends on the selected mode. The function specific control parameter **arg** is a pointer on a *TP680_IO_STRUCT* structure.

```
typedef struct
{
        int                     port;                           /* 0.. 7 */
        union
        {
                unsigned char       val8;                       /* 8 bit wide port */
                struct
                {
                        unsigned short      *buf;
                        unsigned long       size;
                } val16;                                        /* 16 bit wide port (mode 1) */
                struct
                {
                        unsigned long       *buf;
                        unsigned long       size;
                } val32;                                        /* 32 bit wide port (mode 2) */
                struct
                {
                        unsigned long       val64_l;
                        unsigned long       val64_h;
                } val64;                                        /* 64 bit wide port (synchon) */
        } u;
} TP680_IO_STRUCT, *PTP680_IO_STRUCT;
```

port

This parameter specifies the port where data shall be written to. The port number is always between 0 and 7. Dependent from the module configuration some ports are connected to others and only the first port is accessible with this function. If the port is connected or configured as input, *write()* will return *ERROR*. The data order is always 'big endian' (e.g. a 32 bit access on port 0 will write the output of port 0 / line 0 with the LSB and the output of port 4 / line 7 with the MSB.)

Port Mode Configuration [1]		**Connected to Port**							
Port 0	**Port 2**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BYTE	BYTE	7	6	5	4	3	2	1	0
HS16BIT	BYTE	7	6	5[2]	4[2]	3	2	0	0
BYTE	HS16BIT	7	6	5[2]	4[2]	2	2	1	0
HS16BIT	HS16BIT	7	6	5[2]	4[2]	2	2	0	0
HS32BIT	---	7	6	5[2]	4[2]	0	0	0	0
SYNCHRON	---	0	0	0	0	0	0	0	0

[1] The port mode configurations are assigned to the following driver configurations values:

Port Mode Configuration	**Driver Configuration Value (tpmc680def.h)**
BYTE	*TP680_IO_MODE_BYTE*
HS16BIT	*TP680_IO_MODE_HS16BIT*
HS32BIT	*TP680_IO_MODE_HS32BIT*
SYNCHRON	*TP680_IO_MODE_SYNCHRON*

[2] Bits 0/1 may be used for HS and are unwriteable.

u

This union allows handling the different output sizes and modes. Description follows below.

val8

This selection will be used for direct byte output. The specified value will be written to the specified port.

val16

This selection will be used for buffered 16bit output. The structure contains a pointer to a buffer (***buf***) of *unsigned short* and the size (***size*)** of this buffer. ***size*** will be specified in words. The write function will copy the buffer into a driver internal FIFO and output this value on the specified port. The function will return immediately.

val32

This selection will be used for buffered 32bit output. The structure contains a pointer to a buffer (***buf***) of *unsigned long* and the size (***size*)** of this buffer. ***size*** will be specified in words. The write function will copy the buffer into a driver internal FIFO and output this value on the specified port. The function will return immediately.

val64

This selection will be used for direct 64bit synchronous input. The structure has two elements. The value of port 3..0 will be specified in ***val64_l*** where the LSB is the value of port 0 and the MSB is the value of port 3. The value of port 7..4 will be specified in ***val64_h*** where the LSB is the value of port 4 and the MSB is the value of port 7.

EXAMPLE

```
#include “tpmc680.h”

int                fd;
int                i;
int                retval;
TP680_IO_STRUCT    ioBuf;
unsigned short     usBuf[10];
unsigned long      ulBuf[10];

/*-------------------------
  Write 0x12 to byte port 5
  -------------------------*/
ioBuf.port = 5;
ioBuf.u.val8 = 0x12;

retval = ioctl(fd, FIO_TP680_WRITE, (int)&ioBuf);
if (retval != ERROR)
{
     /* Output OK */;
}
else
{
     /* Handle Error */
}

…
```

```
…

/*----------------------------------------
  Write the buffered values to 16bit port 2
  ----------------------------------------*/
usBuf[0] = 0x1111;
usBuf[1] = 0x2222;
usBuf[2] = 0x3333;
ioBuf.port = 2;
ioBuf.u.val16.size = 3;
ioBuf.u.val16.buf = usBuf;

retval = ioctl(fd, FIO_TP680_WRITE, (int)&ioBuf);
if (retval != ERROR)
{
     /* Output OK */;
}
else
{
     /* Handle Error */
}

…

/*----------------------------------------
  Write the buffered values to 32bit port 0
  ----------------------------------------*/
ulBuf[0] = 0x11111111;
ulBuf[1] = 0x22222222;
ulBuf[2] = 0x33333333;
ioBuf.port = 0;
ioBuf.u.val16.size = 3;
ioBuf.u.val32.buf = ulBuf;

retval = ioctl(fd, FIO_TP680_WRITE, (int)&ioBuf);
if (retval != ERROR)
{
     /* Output OK */;
}
else
{
     /* Handle Error */
}

…
```

…

```
/*----------------------------------
  Write port value (64Bit) to port 0
  ----------------------------------*/
ioBuf.port = 0;
ioBuf.u.val64.val64_h = 0x12345678;
ioBuf.u.val64.val64_l = 0x87654321;

retval = ioctl(fd, FIO_TP680_WRITE, (int)&ioBuf);
if (retval != ERROR)
{
     /* Output OK */;
}
else
{
     /* Handle Error */
}
```

RETURNS

Number of bytes written 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) or a driver set error code described below.

Error code	Description
S_tp680Dev_IPORT	Illegal port number specified
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_IDIRECTION	Illegal direction specified

5.3.3 FIO_TP680_SET_DIR

This I/O control function sets the port direction of the specified port. The mode of the port will be left untouched. If the port is configured in handshake mode, the FIFO will be flushed. The function specific control parameter **arg** is a pointer on a *TP680_IOCTL_DIR_STRUCT* structure.

```
typedef struct
{
        int             port;
        unsigned long   direction;
} TP680_IOCTL_DIR_STRUCT, *PTP680_IOCTL_DIR_STRUCT;
```

port

This parameter specifies the port which shall be configured. The port number is always between 0 and 7. Dependent from the module configuration some ports are connected to others and only the first port is accessible with this function. If the port is connected, the function will return *ERROR*.

Port Mode Configuration [1]		**Connected to Port**							
Port 0	**Port 2**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BYTE	BYTE	7	6	5	4	3	2	1	0
HS16BIT	BYTE	7	6	5[2]	4[2]	3	2	0	0
BYTE	HS16BIT	7	6	5[2]	4[2]	2	2	1	0
HS16BIT	HS16BIT	7	6	5[2]	4[2]	2	2	0	0
HS32BIT	---	7	6	5[2]	4[2]	0	0	0	0
SYNCHRON	---	0	0	0	0	0	0	0	0

[1] The port mode configurations are assigned to the following driver configurations values:

Port Mode Configuration	**Driver Configuration Value (tpmc680def.h)**
BYTE	*TP680_IO_MODE_BYTE*
HS16BIT	*TP680_IO_MODE_HS16BIT*
HS32BIT	*TP680_IO_MODE_HS32BIT*
SYNCHRON	*TP680_IO_MODE_SYNCHRON*

[2] Bits 0/1 may be used for HS and are unconfigurable.

direction

This parameter specifies the new port direction. The following values are valid.

Value	**Description**
TP680_IO_DIR_IN	Configure specified port for input
TP680_IO_DIR_OUT	Configure specified port for output

EXAMPLE

```
#include “tpmc680.h”

int                     fd;
TP680_IOCTL_DIR_STRUCT  dirBuf;
int                     retval;

/*---------------------------
  configure port 3 for output
  ---------------------------*/
dirBuf.port        = 3;
dirBuf.direction   = TP680_IO_DIR_OUT;

retval = ioctl(fd, FIO_TP680_SET_DIR, (int)&dirBuf);
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
```

ERROR CODES

Error code	Description
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_IPORTMODE	Illegal port mode specified
S_tp680Dev_IPORT	Illegal port number specified

5.3.4 FIO_TP680_SET_IRQ

This I/O control function connects a user specified function to a specified interrupt. The selected I/O line must be part of a port configured for byte wise (8/64bit) input. The function will be disconnected if it is specified with this function or the port mode is no longer input and byte wise. The function specific control parameter **arg** is a pointer on a *TP680_IOCTL_IRQ_STRUCT* structure.

```
typedef struct
{
        int             port;
        int             line;
        unsigned long   edge;
        void            *function;
        unsigned long   parameter;
} TP680_IOCTL_IRQ_STRUCT, *PTP680_IOCTL_IRQ_STRUCT;
```

port

This parameter specifies the port where the interrupt function shall be installed to.

line

This parameter specifies the I/O line where the interrupt function shall be installed to.

edge

This parameter specifies the event on which the interrupt shall be generated.

Value	Description
TP680_IO_EDGE_NO	Disconnect installed function from interrupt.
TP680_IO_EDGE_HI	Call specified function on a low to high event of the specified input line.
TP680_IO_EDGE_LO	Call specified function on a high to low event of the specified input line.
TP680_IO_EDGE_ANY	Call specified function on any event of the specified input line.

function

This parameter specifies the entry point of the interrupt callback function.

parameter

The function will be called with this parameter.

The function will be called while the interrupt is handled. Keep this function short and do not use any operations that will block the task.

EXAMPLE

```
#include “tpmc680.h”

void intfunction (unsigned long param)
{
     /* Handle interrupt */
}

…

int                     fd;
TP680_IOCTL_IRQ_STRUCT  irqBuf;
int                     retval;

/*-------------------------------------------------------------
  connect interrupt function to port 5 line 6
  the function shall be called on the high to low transition
  the parameter shall be 0x1234
  -------------------------------------------------------------*/
irqBuf.port        = 5;
irqBuf.line        = 6;
irqBuf.edge        = TP680_IO_EDGE_LO;
irqBuf.function    = (void*)intfunction;
irqBuf.parameter   = 0x1234;

retval = ioctl(fd, FIO_TP680_SET_IRQ, (int)&irqBuf);
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
```

ERROR CODES

Error code	Description
S_tp680Dev_IPORT	Illegal port number specified
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_ILINE	Illegal line number specified
S_tp680Dev_IEDGE	Illegal interrupt event specified
S_tp680Dev_IRQBUSY	The specified input line is already connected with a callback function

5.3.5 FIO_TP680_SET_MODE

This I/O control function sets up the mode of a specified port. Calling this function will flush the FIFO if the previous mode has been a handshake (buffered) mode. The function specific control parameter **arg** is a pointer on a *TP680_IOCTL_MODE_STRUCT* structure.

```
typedef struct
{
        int             port;
        unsigned long   mode;
        unsigned long   hsFlags;    /* only valid for HS modes */
} TP680_IOCTL_MODE_STRUCT, *PTP680_IOCTL_MODE_STRUCT;
```

port

This parameter specifies the port which shall be configured. The port number is always between 0 and 7. Dependent from the actual module configuration some ports are connected to others and only the first port is accessible with this function. If the new mode connects less ports then the mode before, the unconfigured ports will be set to BYTE mode. The previous directions of the ports will be kept.

Port Mode Configuration [1]		**Connected to Port**							
Port 0	**Port 2**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BYTE	BYTE	7	6	5	4	3	2	1	0
HS16BIT	BYTE	7	6	5[2]	4[2]	3	2	0	0
BYTE	HS16BIT	7	6	5[2]	4[2]	2	2	1	0
HS16BIT	HS16BIT	7	6	5[2]	4[2]	2	2	0	0
HS32BIT	---	7	6	5[2]	4[2]	0	0	0	0
SYNCHRON	---	0	0	0	0	0	0	0	0

[1] The port mode configurations are assigned to the following driver configurations values:

Port Mode Configuration	**Driver Configuration Value (tpmc680def.h)**
BYTE	*TP680_IO_MODE_BYTE*
HS16BIT	*TP680_IO_MODE_HS16BIT*
HS32BIT	*TP680_IO_MODE_HS32BIT*
SYNCHRON	*TP680_IO_MODE_SYNCHRON*

[2] Bits 0/1 may be used for HS and are unconfigurable.

mode

This parameter specifies the new port mode.

Value	Description
TP680_IO_MODE_BYTE	Configures a port as byte I/O, no other ports will be connected. Valid for port 0..7. (The driver will support direct I/O)
TP680_IO_MODE_HS16BIT	Configures a port as 16 bit handshake port. This mode is valid for port 0 (port 1 will be connected) and port 2 (port 3 will be connected). Port 4 will be set to input (Handshake). (The driver will support buffered I/O)
TP680_IO_MODE_HS32BIT	Configures a port as 32 bit handshake port. This mode is valid for port 0 (port 1, 2 and 3 will be connected). Port 4 will be set to input (Handshake). (The driver will support buffered I/O)
TP680_IO_MODE_SYNCHRON	Configures the module for synchronous byte I/O. All ports will be connected. The only valid port is port 0. This mode will be used for 64 bit parallel I/O. (The driver will support direct I/O)

hsFlags

This flags will specify the output handshake mode.

Value	Description
TP680_IO_HSFLAG_NO	The handshake output will not be used. (Port 5 will be left in the previous mode)
TP680_IO_HSFLAG_INTERLOCKED	The handshake output will be used in interlocked mode. Port 5 will be used for output.
TP680_IO_HSFLAG_PULSED	The handshake output will be used in pulsed mode. Port 5 will be used for output.
TP680_IO_HSFIFOEV_NOTFULL	The FIFO event will be set to “Not Full”. This value must be ORed with *TP680_IO_HSFLAG_INTERLOCKED or TP680_IO_HSFLAG_PULSED.*
TP680_IO_HSFIFOEV_EMPTY	The FIFO event will be set to “Empty”. This value must be ORed with *TP680_IO_HSFLAG_INTERLOCKED or TP680_IO_HSFLAG_PULSED.*

For a detailed description of the modes refer to the TPMC680 User Manual

EXAMPLE

```
#include “tpmc680.h”

int                     fd;
TP680_IOCTL_MODE_STRUCT modeBuf;
int                     retval;

/*-------------------------------------------------------------
  configure port 2 for 16bit handshake mode
  the output handshake shall interlocked and the event shall
  occur on “not full”
  -------------------------------------------------------------*/
modeBuf.port       = 2;
modeBuf.mode       = TP680_IO_MODE_HS16BIT;
modeBuf.hsflags    = TP680_IO_HSFLAG_INTERLOCKED |
                     TP680_IO_HSFIFOEV_NOTFULL;

retval = ioctl(fd, FIO_TP680_SET_MODE, (int)&modeBuf);
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
```

ERROR CODES

Error code	Description
S_tp680Dev_PORTBUSY	The port is connected to an other port and is not addressable
S_tp680Dev_IPORTMODE	Illegal port mode specified
S_tp680Dev_IPORT	Illegal port number specified

5.3.6 FIO_TP680_GET_DEBUG

The function code *FIO_TP680_GET_DEBUG* is a debug function and may be useful for error fixing. The function is undocumented. The switch *_TP680_DEBUG_MODE_ENABLE_* shall be undefined in “tpmc680.h”.

6 Appendix

6.1 Additional Error Codes

Error code	Error value	Description
S_tp680Dev_ICMD	0x06800000	Illegal Ioctl() command
S_tp680Dev_IPORT	0x06800001	Illegal port number specified
S_tp680Dev_ILINE	0x06800002	Illegal line number specified
S_tp680Dev_IDIRECTION	0x06800003	Illegal direction specified
S_tp680Dev_PORTBUSY	0x06800004	The port is connected to an other port and is not addressable
S_tp680Dev_IRQBUSY	0x06800005	The specified input line is already connected with a callback function
S_tp680Dev_IEDGE	0x06800006	Illegal interrupt event specified
S_tp680Dev_IPORTMODE	0x06800007	Illegal port mode specified