The Embedded I/O Company

TEWS TECHNOLOGIES

TPMC680-SW-72

LynxOS Device Driver

64 Digital Inputs/Outputs

Version 1.1.x

User Manual

Issue 1.1

November 2003

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

TEWS TECHNOLOGIES LLC
1 E. Liberty Street, Sixth Floor
Reno, Nevada 89504 / USA
Phone: +1 (775) 686 6077
Fax: +1 (775) 686 6024
e-mail: usasales@tews.com
www.tews.com

TPMC680-SW-72

64 Digital Inputs/Outputs

LynxOS Device Driver

Issue	Description	Date
1.0	First Issue	November 18, 2003
1.1	Error messages change (compatibility to older LynxOS versions)	November 25, 2003

Table of Content

1 Introduction

The TPMC680-SW-72 LynxOS device driver allows the operation of a TPMC680 64 Digital Inputs/Outputs PMC on a PowerPC platform with DRM based PCI interface.

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

The TPMC680 device driver includes the following functions:

- read digital input value (8 bit / 64 bit ports)
- write digital output value(8 bit / 64 bit ports)
- receive and transmit parallel data (16 bit / 32 bit handshake ports)
- configure port size, direction and handshake mode
- wait for a specified input event (8 bit / 64 bit ports)

2 Installation

The software is delivered on a PC formatted 3½" HD diskette.

Following files are located on the diskette:

tpmc680.c	Driver source code
tpmc680.h	Definitions and data structures for driver and application
tpmc680_info.c	Device information definition
tpmc680_info.h	Device information definition header
tpmc680.cfg	Driver configuration file include
tpmc680.import	Linker import file
Makefile	Device driver make file
Makefile.ppc.dldd	Make file for dynamic driver installation (for Power PC)
Makefile.x86.dldd	Make file for dynamic driver installation (for Intel x86)
tpmc680-sw-72.pdf	This Manual in PDF format

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.

In order to perform a static installation, copy the following files to the 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/ tpmc680

2. Copy the following files to this directory: tpmc680.c, Makefile

3. Copy *tpmc680.h* to */usr/include/*

4. Copy *tpmc680_info.c* to */sys/devices.xxx/* or */sys/devices* if */sys/devices.xxx* does not exist (xxx represents the BSP).

5. Copy *tpmc680_info.h* to */sys/dheaders/*

6. Copy *tpmc680.cfg* to */sys/cfg.ppc/*

2.1.1.1 Build the driver object

1. Change to the directory */sys/drivers.xxx/ tpmc680,* 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_prep = ...tpmc680_info.x
   ```

   And at the end of the Makefile

   ```
   ...
   tpmc680_info.o:$(DHEADERS)/tpmc680_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 in the file CONFIG.TBL

   Insert the entry after the console driver section

   ```
   # End of console devices
   I:tpmc680.cfg
   ```

2.1.1.4 Rebuild the Kernel

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

   ```
   make install
   ```

3. Reboot the newly-created operating system by the following command:

   ```
   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/tp680a, [/dev/tp680b, …]*

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.

Unlike the description of the dynamic installation in the manual “Writing Device Drivers for LynxOS”, the driver source must be placed in a directory under */sys/drivers.pp_drm/*

The following steps describe how to do a dynamic installation:

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

   For example: */sys/drivers.pp_drm/tpmc680*

2. Copy the following files to this directory:

   - - tpmc680.c
   - - tpmc680_info.c
   - - tpmc680_info.h
   - - tpmc680.import
   - - Makefile.ppc.dldd
   - - Makefile.x86.dldd

3. Copy *tpmc680.h* to */usr/include*

4. Change to the directory */sys/drivers.xxx /tpmc680*

5. To make the dynamic link-able driver enter :

   ```
   make –f Makefile.ppc.dldd      (Makefile.x86.dldd on x86 sytems!)
   ```

6. Create a device definition file for one major device

   ```
   gcc –DDLDD –o tpmc680_info tpmc680_info.c
   ./tpmc680_info > tp680a
   ```

7. To install the driver enter:

   ```
   drinstall –c tpmc680.obj
   ```

   If successful *drinstall* returns a unique *<driver-ID>*

8. To install the major device enter:

   ```
   devinstall –c –d <driver-ID> tp680a
   ```

   The <driver-ID> is returned by the drinstall command

   If successful *devinstall* returns the *<major_no>*

9. To create nodes for the devices enter:

   ```
   mknod /dev/tp680a c <major_no> 0
   ...
   ```

If all steps are successful completed the TPMC680 is ready to use.

To uninstall the TPMC680 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 TPMC680 major device.

The implementation of the device information definition is done through a C structure which is defined in the header file *tpmc680_info.h*.

This structure contains following parameter:

PCIBusNumber

Contains the PCI bus number at which the TPMC680 is connected. Valid bus numbers are in range from 0 to 255.

PCIDeviceNumber

Contains the device number (slot) at which the TPMC680 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 TPMC680 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 you have more than one TMPC680 major device.

A device information definition is unique for every TPMC680 major device. The file *tpmc680_info.c* on the distribution disk contains two device information declarations, **tp680a_info** for the first major device and **tp680b_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 unique name for example **tp680c_info**, **tp680d_info** and so on.

It is also necessary to modify the device and driver configuration file respectively the configuration include file *tpmc680.cfg*.

The following device declaration information uses the auto find method to detect the TPMC680 module on PCI bus.

```
TP680_INFO tp680a_info = {
    -1,                 /*  auto find the TPMC680 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 TPMC680 driver and devices into the LynxOS system, the configuration include file *tpmc680.cfg* must be included in the *CONFIG.TBL*.

The file *tpmc680.cfg* on the distribution disk contains the driver entry (C:tpmc680:\....) and one enabled major device entry (D:TPMC680 1:tp680a_info::) with one minor device entry (N: tp680a:0).

If the driver should support more than one major device the following entries for major and minor devices must be enabled by removing the comment character (#). By copy and paste an existing major and minor entry and renaming the new entries, it is possible to add any number of additional TPMC680 device.

The name of the device information declaration (info-block-name) must match to an existing C structure in the file *tpmc680_info.c*.

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:tpmc680:\
     :tp680open:tp680close:tp680read:tp680write:\
     ::tp680ioctl:tp680install:tp680uninstall
D:TPMC680 1:tp680a_info::
N:tp680a:0
```

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

```
/dev/tp680a
```

3 TPMC680 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 (TPMC680 device) named in ***path*** for reading and writing. The value of ***oflags*** indicates the intended use of the file. In case of a TPMC680 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 TPMC680 device already exists this argument is ignored.

EXAMPLE

```
int  fd

...

/*
**  open the device named "/dev/tp680a" for I/O
*/

fd = open ("/dev/tp680a", O_RDWR);

...
```

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);

...
```

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

The read function reads data from the TPMC680 device. How the data is read, differs dependent on the selected port size.

A pointer to the callers read buffer (*TP680_IOBUF*) and the size of the buffer are passed by the parameters *buff* and *count* to the device.

The *TP680_IOBUF* structure has the following layout:

```
typedef struct
{
        int             port;       /* 0.. 7                          */
        int             dataWidth;  /* size of data value             */
        int             dataSize;   /* size of data buffer in bytes   */
        char            buf[1];
} TP680_IOBUF, *PTP680_IOBUF;
```

port

Specifies the port number, allowed values are between 0 and 7. Dependent on the selected configuration, some port numbers will not be allowed or some bits will not be available for data. The table below gives an overview.

Port Mode Configuration [1]		Connected to Port							
Port 0	Port 2	7	6	5	4	3	2	1	0
8 bit	8 bit	7	6	5	4	3	2	1	0
16 bit	8 bit	7	6	5 [2]	4 [2]	3	2	0	0
8 bit	16 bit	7	6	5 [2]	4 [2]	2	2	1	0
16 bit	16 bit	7	6	5 [2]	4 [2]	2	2	0	0
32 bit	---	7	6	5 [2]	4 [2]	0	0	0	0
64 bit	---	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 (tp680def.h)
8 bit	TP680_MODE_SIZE_8BIT
16 bit	TP680_MODE_SIZE_16BIT
32 bit	TP680_MODE_SIZE_32BIT
64 bit	TP680_MODE_SIZE_64BIT

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

portWidth

Specifies the size of the port, the allowed port size depends on the module configuration.

dataSize

Specifies the size of the data buffer and returns the number of bytes returned in it. The *dataSize* must be specified in bytes and specifies the size of *buf*.

Port Mode Configuration	Allowed Parameter Values		
size	port	portWidth	dataSize
8 bit	0, 1, 2, 3, 4, 5, 6, 7	sizeof(unsigned char)	1
16 bit	0, 2	sizeof(unsigned short)	depends on allocated buffer size
32 bit	0	sizeof(unsigned long)	depends on allocated buffer size
64 bit	0	2 * sizeof(unsigned long)	2 * sizeof(unsigned long)

buf

Is defined as the first byte of the data buffer where the read data will be returned. By allocating a buffer and overlaying the structure it is possible to get a greater buffer space. The size of the buffer is specified with the parameter *dataSize*.

The needed size of the buffer can be calculated with the predefined MACRO: *TP680_NEEDEDBUFFERSIZE(elemNum, elemSize)* where *elemNum* specifies the maximum number of elements that shall be returned and *elemSize* specifies the maximum size in bytes of one element.

For example: for 20 Longwords use:

TP680_NEEDEDBUFFERSIZE(20, sizeof(unsigned long))

EXAMPLE

```
int                hCurrent = 0;
TP680_IOBUF        *rdBuf;
unsigned short     *usBuf;
int                NumBytes;
int                bufSize;

bufSize = TP680_NEEDEDBUFFERSIZE(5,sizeof(unsigned short));
rdBuf = (TP680_IOBUF*)malloc(bufSize);
hCurrent = open(...);

...

/*
** Read a value from port 7 using the 8-bit mode
*/
rdBuf->port = 7;
rdBuf->dataWidth = sizeof(unsigned char);
rdBuf->dataSize = sizeof(unsigned char);

NumBytes = read(hCurrent, rdBuf, sizeof(TP680_IOBUF));
if (NumBytes > 0)
{
     /* Input data in rdBuf->buf[0] */
}
else
{
     /* read error */
}

...
```

```
...

/*
** Read a maximum of 5 values from port 2 using the 16-bit mode
*/
rdBuf->port = 2;
rdBuf->dataWidth = sizeof(unsigned short);
rdBuf->dataSize = 5 * sizeof(unsigned short);

NumBytes = read(hCurrent, rdBuf, bufSize);
if (NumBytes > 0)
{
     /* pointer to result buffer */
     usBuf = (unsigned short*)rdBuf->buf;
     /* Input data in array usBuf[] */
}
else
{
     /* read error */
}
```

RETURNS

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

ERRORS

EINVAL	Invalid argument. This error code is returned if the size of the read or data buffer is too small, or a specified parameter is out of range.
EFAULT	Invalid pointer to the read or data buffer.
EACCES	Access is not allowed, port has a false configuration.

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

The write function writes data to the TPMC680 device. How the data is written, differs dependent on the selected port size.

A pointer to the callers write buffer (*TP680_IOBUF*) and the size of the buffer are passed by the parameters *buff* and *count* to the device.

The *TP680_IOBUF* structure has the following layout:

```
typedef struct
{
        int             port;       /* 0.. 7                          */
        int             dataWidth;  /* size of data value             */
        int             dataSize;   /* size of data buffer in bytes   */
        char            buf[1];
} TP680_IOBUF, *PTP680_IOBUF;
```

port

Specifies the port number, allowed values are between 0 and 7. Dependent on the selected configuration, some port numbers will not be allowed or some bits will not be available for data. The table below gives an overview.

Port Mode Configuration [1]		Connected to Port							
Port 0	Port 2	7	6	5	4	3	2	1	0
8 bit	8 bit	7	6	5	4	3	2	1	0
16 bit	8 bit	7	6	5 [2]	4 [2]	3	2	0	0
8 bit	16 bit	7	6	5 [2]	4 [2]	2	2	1	0
16 bit	16 bit	7	6	5 [2]	4 [2]	2	2	0	0
32 bit	---	7	6	5 [2]	4 [2]	0	0	0	0
64 bit	---	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 (tp680def.h)**
8 bit	*TP680_MODE_SIZE_8BIT*
16 bit	*TP680_MODE_SIZE_16BIT*
32 bit	*TP680_MODE_SIZE_32BIT*
64 bit	*TP680_MODE_SIZE_64BIT*

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

portWidth

Specifies the size of the port, the allowed port size depends on the module configuration.

dataSize

Specifies the size of the data buffer. The *dataSize* must be specified in bytes and specifies the size of *buf*.

Port Mode Configuration	Allowed Parameter Values		
size	port	portWidth	dataSize
8 bit	0, 1, 2, 3, 4, 5, 6, 7	sizeof(unsigned char)	1
16 bit	0, 2	sizeof(unsigned short)	depends on allocated buffer size
32 bit	0	sizeof(unsigned long)	depends on allocated buffer size
64 bit	0	2 * sizeof(unsigned long)	2 * sizeof(unsigned long)

buf

Is defined as the first byte of the data buffer where the write data must be placed. By allocating a buffer and overlaying the structure it is possible to get a greater buffer space. The size of the buffer is specified with the parameter *dataSize*.

The needed size of the buffer can be calculated with the predefined MACRO: *TP680_NEEDEDBUFFERSIZE(elemNum, elemSize)* where *elemNum* specifies the maximum number of elements that shall be returned and *elemSize* specifies the maximum size in bytes of one element.

For example: for 20 Longwords use:

TP680_NEEDEDBUFFERSIZE(20, sizeof(unsigned long))

EXAMPLE

```
int                hCurrent = 0;
TP680_IOBUF        *wrBuf;
int                NumBytes;
int                bufSize;
unsigned short     *usVal;

bufSize = TP680_NEEDEDBUFFERSIZE(5,sizeof(unsigned short));;
wrBuf = (TP680_IOBUF*)malloc(bufSize);

hCurrent = open(...);

...

/*
** Write a value (0x55) to port 7 using the 8-bit mode
*/
wrBuf->port = 7;
wrBuf->dataWidth = sizeof(unsigned char);
wrBuf->dataSize = sizeof(unsigned char);

wrBuf->buf[0] = 0x55;

NumBytes = read(hCurrent, wrBuf, sizeof(TP680_IOBUF));
if (NumBytes > 0)
{
     /* 8-bit value written */
}
else
{
     /* write error */
}

...
```

```
...

/*
** Write 5 values to port 2 using the 16-bit mode
*/
wrBuf->port = 2;
wrBuf->dataWidth = sizeof(unsigned short);
wrBuf->dataSize = 5 * sizeof(unsigned short);

usVal = (unsigned short*) wrBuf->buf;

usVal [0] = 0x1111;
usVal [1] = 0x1112;
usVal [2] = 0x1122;
usVal [3] = 0x1222;
usVal [4] = 0x2222;

NumBytes = write(hCurrent, wrBuf, bufSize);
if (NumBytes > 0)
{
     /* data written to 16-bit port */
}
else
{
     /* write error */
}
```

RETURNS

On success write returns the size of written data. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL	Invalid argument. This error code is returned if the size of the read or data buffer is too small, or a specified parameter is out of range.
EFAULT	Invalid pointer to the write or data buffer.
EACCES	Access is not allowed, port has a false configuration.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.5 ioctl()

NAME

ioctl() – device control functions

SYNOPSIS

#include <ioctl.h>
#include <tpmc680.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 argument *request* specifies the control code for the operation. The optional argument *arg* depends on the selected request and is described for each request in detail later in this chapter.

The following ioctl codes are defined in *tpmc680.h* :

Value	Meaning
TP680_SETMODE	Configure ports
TP680_EVENTWAIT	Wait for a specified event

See behind for more detailed information on each control code.

To use these TPMC680 specific control codes the header file TPMC680.h must be included in the application

RETURNS

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

ERRORS

EINVAL	Invalid argument. This error code is returned if the requested ioctl function is unknown. Please check the argument *request*.

Other function dependant error codes will be described for each ioctl code separately. Note, the TPMC680 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.5.1 TP680_SETMODE

NAME

TP680_SETMODE – Configure a port

DESCRIPTION

This *ioctl()* function configures the specified port of the TPMC680.

A pointer to the callers configuration buffer (*TP680_SETMODEBUF*) is passed by the parameter *arg* to the driver.

The *TP680_SETMODEBUF* structure has the following layout:

```
typedef struct
{
        unsigned long           port;           /* Port number to handle */
        unsigned long           Size;           /* Port size */
        unsigned long           Direction;      /* Port direction */
        unsigned long           HSMode;         /* Handshake Output Mode */
        unsigned long           HSFifo;         /* Handshake Output Fifo Mode */
} TP680_SETMODEBUF, *PTP680_SETMODEBUF;
```

port

This member specifies the port that shall be configured. Valid values are between 0 and 7.

Size

This argument specifies the port size. The following table describes the allowed port sizes and for which ports they are allowed.

Value	Ports	Description
TP680_MODE_SIZE_8BIT	0, 1, 2, 3, 4, 5, 6, 7	The port has a width of 8 bit. Each port can be accessed separately.
TP680_MODE_SIZE_16BIT	0,2	The port has a width of 16 bit and the output is controlled by the handshake signals. Two ports are used together. If port 0 is selected port 1 is used also. If port 2 is selected also port 3 will be used. The configuration of the connected ports is always adapted. **If this mode is selected for any port the handshake port 4 will be configured as an 8-bit input port.**
TP680_MODE_SIZE_32BIT	0	The port has a width of 32 bit and the output is controlled by the handshake signals. The ports 0, 1, 2 and 3 will be used together. The configuration of the connected ports is always set together. **If this mode is selected the handshake port 4 will be configured as an 8-bit input port.**
TP680_MODE_SIZE_64BIT	0	All ports are connected and can be used as simple 64 bit input or output port. All ports get the same configuration.

Direction

This member specifies direction of the port all connected ports will get the same direction. Allowed values are:

Value	Description
TP680_MODE_DIR_INPUT	The port will be used as an input port.
TP680_MODE_DIR_OUTPUT	The port will be used as an output port.

HSMode

This value specifies the handshake mode and is only valid if the port shall be configured in 16 or 32 bit handshake mode (*TP680_MODE_SIZE_16BIT, TP680_MODE_SIZE_32BIT*). Using an output handshake, will change the direction of port 5 to output. The allowed values are:

Value	Description
TP680_MODE_HSFLAG_NO	No output handshake will be used.
TP680_MODE_HSFLAG_INTERLOCKED	The interlocked output handshake mode will be used.
TP680_MODE_HSFLAG_PULSED	The pulsed output handshake mode will be used.

HSFifo

This value specifies the handshake event depending on the handshake FIFO fill level. This value is only used if an output handshake is configured. The values are:

Value	Description
TP680_MODE_HSFIFOEV_NOTFULL	The event announces FIFO is not full.
TP680_MODE_HSFIFOEV_EMPTY	The event announces FIFO is empty.

When setting up ports other that depends on the selected, may change direction or mode. (Please refer to the TPMC680 User Manual.

Changing a port size from big to small will also change the mode of the previously connected ports. The ports will be set into 8 bit mode and they will keep their direction.

EXAMPLE

```
int                hCurrent = 0;
int                result;
TP680_SETMODEBUF   modeBuf;

hCurrent = open(...);

...

/* Configure port (2) for 16-bit output handshake mode */

modeBuf.port = 2;
modeBuf.Size = TP680_MODE_SIZE_16BIT;
modeBuf.Direction = TP680_MODE_DIR_OUTPUT;
modeBuf.HSMode = TP680_MODE_HSFLAG_INTERLOCKED;
                            /* interlocked output HS mode */
modeBuf.HSFifo = TP680_MODE_HSFIFOEV_EMPTY;
                            /* ouput event on FIFO empty */

result = ioctl(hCurrent, TP680_SETMODE, &modeBuf);
if(result >= 0)
{
     /* Setting port mode successful */
}
else
{
     /* Setting portmode failed */
}
```

ERRORS

EINVAL	Invalid argument. This error code is returned if the size of the read or data buffer is too small, or a specified parameter is out of range, or if a specified parameter value is out of range.
EFAULT	Invalid pointer to the configuration buffer.
EACCES	Access is not allowed, port has a false configuration

SEE ALSO

ioctl man pages

3.5.2 TP680_EVENTWAIT

NAME

TP680_EVENTWAIT – Wait for a specified input event

DESCRIPTION

This *ioctl()* function waits for a specified event on a specified input line of the TPMC680.

A pointer to the callers event buffer (*TP680_EVENTWAITBUF*) is passed by the parameter *arg* to the driver.

The *TP680_EVENTWAITBUF* structure has the following layout:

```
typedef struct
{
    unsigned long    port;        /* Port number to handle */
    unsigned long    lineNo;      /* Input Line, event shall occur on */
    unsigned long    transition;  /* Specify transition */
    unsigned long    timeout;     /* Timeout in seconds */
} TP680_EVENTWAITBUF, *PTP680_EVENTWAITBUF;
```

port

This member specifies the port to wait for. Valid values are between 0 and 7.

lineNo

This member specified the line to wait for. Valid values are between 0 and 7.

transition

This member specifies the event to wait for. The following events are supported:

Value	Description
TP680_IO_EDGE_LO	The event will occur if the specified input line changes from High to Low.
TP680_IO_EDGE_ANY	The event will occur if the specified input line changes its value.

timeout

This argument specifies the timeout in ticks. If the specified event does not occur in the specified time, the function will return with an error code.

This function is only supported for 8 bit and 64 bit ports. Other configurations will return an error code.

EXAMPLE

```
int                hCurrent = 0;
int                result;
TP680_EVENTWAITBUF evantBuf;

hCurrent = open(...);

...

/*
** Wait for a high to low transition on line 3 of port 3, timeout after
** 10000 ticks.
*/
eventBuf.port = 3;
eventBuf.lineNo = 3;
eventBuf.transition = TP680_IO_EDGE_LO;
eventBuf.timeout = 10000;

result = ioctl(hCurrent, TP680_EVENTWAIT, &eventBuf);
if(result >= 0)
{
     /* Event occurred */
}
else
{
     /* Event did not occur or access failed */
}
```

ERRORS

EFAULT	Invalid pointer to the configuration buffer.
EINVAL	Invalid argument. This error code is returned if the size of the read or data buffer is too small, or a specified parameter is out of range, or if a specified parameter value is out of range.
EACCES	Access is not allowed, port has a false configuration
EBUSY	The input line is already connected to a waiting event

SEE ALSO

ioctl man pages

4 Debugging

This driver was successful tested on a MVME2305 board (Power PC) with a Windows Cross development and on a PC (Intel x86) system in a native LynxOS environment.

If the driver will not work properly, usually a PCI bus or interrupt problem, you can enable debug outputs by removing the comments around the symbols *DEBUG*, *DEBUG_PCI* and *DEBUG_TPMC*. The debug output will appear on the console.

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 PCI Header, the address of each base address register and a memory dump of all mapped memory and I/O spaces of the TPMC680 like this (see also *TPMC680 User Manual – PCI Configuration*).

```
TPMC680 Device Driver Install
Bus = 0  Dev = 16  Func = 0
[00] = 02A81498
[04] = 02800000
[08] = 11800000
[0C] = 00000008
[10] = 02042000
[14] = 0000C001
[18] = 02043000
[1C] = 00000000
[20] = 00000000
[24] = 00000000
[28] = 00000000
[2C] = 000A1498
[30] = 00000000
[34] = 00000040
[38] = 00000000
[3C] = 00000109
PCI Base Address 0 (PCI_RESID_BAR0)

B8142000 : 00 FF FF 0F 00 00 00 00 00 00 00 00 00 00 00 00
B8142010 : 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
B8142020 : 00 00 00 00 00 00 00 00 A0 20 81 15 00 00 00 00
B8142030 : 00 00 00 00 00 00 00 00 00 00 00 00 81 00 00 00
B8142040 : 00 00 00 00 00 00 00 00 00 00 00 00 41 00 30 00
B8142050 : 00 00 78 18 C0 B6 24 00 00 00 00 00 00 00 00 00
B8142060 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B8142070 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
PCI Base Address 1 (PCI_RESID_BAR1)

B0108000 : 00 FF FF 0F 00 00 00 00 00 00 00 00 00 00 00 00
B0108010 : 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
B0108020 : 00 00 00 00 00 00 00 00 A0 20 81 15 00 00 00 00
B0108030 : 00 00 00 00 00 00 00 00 00 00 00 00 81 00 00 00
B0108040 : 00 00 00 00 00 00 00 00 00 00 00 00 41 00 30 00
B0108050 : 00 00 78 18 C0 B6 24 00 00 00 00 00 00 00 00 00
B0108060 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B0108070 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCI Base Address 2 (PCI_RESID_BAR2)

B8143000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12
B8143010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B8143020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 32
B8143030 : 00 00 00 00 00 00 00 00 00 00 00 3E 00 00 00 42
B8143040 : 00 00 FF DF 00 00 00 00 00 00 00 4E 00 00 00 52
B8143050 : 00 00 00 56 00 00 00 5A 00 00 00 5E 00 00 00 62
B8143060 : 00 00 00 66 00 00 00 6A 00 00 00 6E 00 00 00 72
B8143070 : 00 00 00 76 00 00 00 7A 00 00 00 7E 00 00 00 82
```