*The Embedded I/O Company*

TEWS TECHNOLOGIES

# TIP610-SW-72

## LynxOS Device Driver

Digital I/O

Version 1.0.x

## User Manual

Issue 1.0

March 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

**TIP610-SW-72**

Digital I/O

LynxOS Device Driver



| Issue | Description | Date |
|---|---|---|
| 1.0 | First Issue | March 24, 2003 |

# Table of Contents

# 1 Introduction

The TIP610-SW-72 LynxOS device driver allows the operation of a TIP610 IPAC module on PowerPC platforms.

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

The TIP610 device driver includes the following functions:

- reading the input register
- writing the output register
- programming direction of every I/O port
- waiting for a transition at a single input line or a group of input lines (OR'ed)

To understand all features of this device driver, it is recommended to read the TIP610 User Manual.

# 2 Installation

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

Following files are located on the diskette:

| | |
|---|---|
| tip610.c | Driver source code |
| tip610.h | Definitions and data structures for driver and application |
| tip610def.h | Definitions and data structures for the driver |
| tip610_info.c | Device information definition |
| tip610_info.h | Device information definition header |
| tip610.cfg | Driver configuration file include |
| tip610.import | Linker import file |
| Makefile | Device driver make file |
| Makefile.dldd | Make file for dynamic driver installation |
| example/example.c | Example application source |
| TIP610-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 their target directories:

1. Create a new directory in the system drivers directory path */sys/drivers.xxx*, where *xxx* represents the BSP that supports the target hardware.
   For example: */sys/drivers.pp_drm/tip610*

2. Copy the following files to this directory:
   *tip610.c, tip610def.h, Makefile*

3. Copy *tip610.h* to */usr/include/*

4. Copy *tip610_info.c* to */sys/devices.xxx/* or */sys/devices* if */sys/devices.xxx* does not exist (xxx represents the BSP).

5. Copy tip610_info.h to /sys/dheaders/

6. Copy tip610.cfg to /sys/cfg.ppc/

#### 2.1.1.1 Build the driver object

1. Change to the directory */sys/drivers.xxx/TIP610,* where *xxx* represents the BSP that supports the target hardware.

2. To update the library */sys/lib/libdrivers.a* enter:

```
make install
```

#### 2.1.1.2 Create Device Information Declaration

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

   DEVICE_FILES_all = ... tip610_info.x

   And at the end of the Makefile

   tip610_info.o:$(DHEADERS)/tip610_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

1. In order to insert the driver object code into the kernel image, an appropriate entry in file CONFIG.TBL must be created.
2. Change to the directory */sys/lynx.os/* respective */sys/bsp.xxx*, where *xxx* represents the BSP that supports the target hardware.
3. Create an entry at the end of the file CONFIG.TBL

   I:tip610.cfg

#### 2.1.1.4 Rebuild the Kernel

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

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

   ```
   reboot –aN
   ```

   The **N** flag instructs *init* to run *mknod* and create all the nodes mentioned in the new *nodetab*.
4. After reboot you should find the following new devices (depends on the device configuration): */dev/T610a, …]*

## 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.

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/TIP610*
2. Copy the following files to this directory:

   tip610.c
   tip610def.h
   tip610_info.c
   tip610_info.h
   tip610.import
   Makefile.dldd

3. Copy tip610.h to /usr/include
4. Change to the directory */sys/drivers.xxx/TIP610*

   To make the dynamic link-able driver enter:

   ```
   make –f Makefile.dldd
   ```

5. Create a device definition file for one major device

   ```
   gcc –DDLDD –o tip610_info tip610_info.c
   ./tip610_info > t610a_info
   ```

6. To install the driver enter:

   ```
   drinstall –c tip610.obj
   ```

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

7. To install the major device enter:

   ```
   devinstall –c –d <driver-ID> t610a_info
   ```

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

8. To create nodes for the devices enter:

   ```
   mknod /dev/T610a c <major_no> 0...
   ```

If all steps are successful completed the TIP610 is ready to use.

To uninstall the TIP610 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 TIP610 major device.

The implementation of the device information definition is done through a C structure, which is defined in the header file *tip610_info.h*.

This structure contains the following parameter:

**ipIOVirtualAddress** This parameter contains the kernel virtual address of the IP I/O space (device registers). This address depends on the configuration of the IP carrier board. In case of a VMEbus carrier this space usually appears in the VMEbus short I/O space A16/D16.

**ipIdVirtualAddress** This parameter contains the kernel virtual address of the IP ID space (ID-PROM). This address depends on the configuration of the IP carrier board. In case of a VMEbus carrier this space usually appears in the VMEbus short I/O space A16/D16.

**ipInterruptVector** Contains the vector at which the TIP610 generate interrupts. If the TIP610 is plugged on a VMEbus carrier any free vector from 64 to 255 can be used. The drivers setup the vector register in the TIP610 with this vector during driver initialization.

**If the TIP610 is plugged on a VMEbus carrier be sure that the appropriate VMEbus driver *uvme* or *vme* is started (CONFIG.TBL). See also *Chapter 5 – Accessing Hardware* in the "Writing Device Drivers for LynxOS" manual and the man pages *uvmedrvr* and *vmedrvr* for information about the VMEbus configuration and mapping.**

**Be sure that the used VME address windows (A16/D32) are enabled. If the *uvmedrvr* driver is responsible for the VMEbus access please check the file *../dheaders/uvmeinfo.h* (pci slave A16 D32).**

A device information definition is unique for every TIP610 major device. The file *tip610_info.c* on the distribution disk contains a device information declaration.

If the driver should support more major devices it is necessary to copy and paste an existing declaration and rename it with unique name for example **t610b_info**, **t610c_info** and so on.

**It is also necessary to modify the device and driver configuration file, respectively the configuration include file *tip610.cfg*.**

The following device declaration information expected that the IP spaces appear at virtual address 0xCFFF8000 for IP I/O and at virtual address 0xCFFF8080 for IP ID space. The interrupt vector 64 is the first usable vector on the VMEbus.

```
T610_INFO t610a_info =
{
     0xCfff8000          /*  IP I/O Space     */
     0xCfff8080          /*  IP ID Space      */
     64                  /*  Interrupt Vector */
};
```

## 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 reads the file and produces a new set of driver and device configuration tables and a corresponding *nodetab*.

To install the TIP610 driver and devices into the LynxOS system, the configuration include file *tip610.cfg* must be included in the *CONFIG.TBL*.

The file *tip610.*cfg on the distribution disk contains the driver entry (C:TIP610:\....) and one enabled major device entry ( D:TIP610 1:t610a_info:: ) with one minor device entry (N: t610a:0).

If the driver should support more than one major device (TIP610) 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 TIP610 devices.

**The name of the device information declaration (info-block-name) must match to an existing C structure in the file *tip610_info.c*.**

This example shows a driver entry with one major 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:tip610:\
     :t610open:t610close:t610read:t610write:\
     ::t610ioctl:t610install:t610uninstall
D:TIP610 1:t610a_info::
N:t610a:0
```

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

```
/dev/t610a
```

# 3 TIP610 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 (TIP610 device) named in ***path*** for reading and writing. The value of ***oflags*** indicates the intended use of the file. In case of a TIP610 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 TIP610 device already exists this argument is ignored.

### EXAMPLE

```
int  fd

/*  open the device named "/dev/t610a" for I/O */
fd = open ("/dev/t610a", 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 file

## SYNOPSIS

#include <tip610.h>

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

## DESCRIPTION

This function attempts to read the input registers of the TIP610 associated with the file descriptor *fd* into a structure (*T610_RW_BUFFER*) pointed by *buff*. The argument *count* specifies the length of the buffer and must be set to the length of the structure *T610_RW_BUFFER*.

The *T610_RW_BUFFER* structure has the following layout:

```
typedef struct {
   unsigned char   portA;
   unsigned char   portB;
   unsigned char   portC;
   unsigned char   wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

**unsigned char portA**

Returns the status of I/O-lines 1 to 8. Where bit 0 corresponds to I/O-line 1, bit 1 to input line 2, and so on.

**unsigned char portB**

Returns the status of I/O-lines 9 to 16. Where bit 0 corresponds to I/O-line 9, bit 1 to I/O-line 10, and so on.

**unsigned char portC**

Returns the status of I/O-lines 17 to 20. Where bit 0 corresponds to I/O-line 17, bit 1 to I/O-line 18, and so on.

**unsigned char wrenaPort**

not used

## EXAMPLE

```
int                fd;
int                result;
T610_RW_BUFFER     ioBuf;


result = read(fd, (char*)&ioBuf, sizeof(ioBuf));

if (result != sizeof(T610_RW_BUFFER)) {
     // process error;
}
```

## RETURNS

When *read* succeeds, the size of the read buffer (*T610_RW_BUFFER)* is returned. If read fails, -1 (SYSERR) is returned.

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

## SEE ALSO

LynxOS System Call - read()

TIP610 example application

# 3.4 write()

## NAME

write() – write to a file

## SYNOPSIS

#include <tip610.h>

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

## DESCRIPTION

This function attempts to write to the output registers of the TIP610 associated with the file descriptor *fd* from a structure (T610_RW_BUFFER) pointed by buff. The argument count specifies the length of the buffer and must be set to the length of the structure T610_RW_BUFFER.

The T610_RW_BUFFER structure has the following layout :

```
typedef struct {
   unsigned char   portA;
   unsigned char   portB;
   unsigned char   portC;
   unsigned char   wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

**unsigned char portA**

Holds the new value for I/O-lines 1 to 8. Where bit 0 corresponds to I/O-line 1, bit 1 to I/O-line 2 and so on.

**unsigned char portB**

Holds the new value for I/O-lines 9 to 16. Where bit 0 corresponds to I/O-line 9, bit 1 to I/O-line 10 and so on.

**unsigned char portC**

Holds the new value for I/O-lines 17 to 20. Where bit 0 corresponds to output line 17, bit 1 to I/O-line 18 and so on.

**unsigned char wrenaPort**

Holds the new value for write enable of port A - C.

The following flags could be OR'ed

| | |
|---|---|
| T610_ENABLE_PORTA | The contents of the member portA will be written to the corresponding port A data register (I/O-lines 1..8). |
| T610_ENABLE_PORTB | The contents of the member portB will be written to the corresponding port B data register (I/O-lines 9..16). |
| T610_ENABLE_PORTC | The contents of the member portC will be written to the corresponding port C data register (I/O-lines 17..20). |

## EXAMPLE

```
int                fd;
int                result;
T610_RW_BUFFER     ioBuf;

...
// set I/O-line(not pin) 1, 8, 12 and 18 to logic high if
// polarity(positive) and direction(output) was set before
ioBuf.portA    = 0x81;
ioBuf.portB    = 0x08;
ioBuf.portC    = 0x02;
ioBuf.wrenaPort = 0x07;

result = write(fd, (char*)&ioBuf, sizeof(ioBuf));

if (result != sizeof(T610_RW_BUFFER)) {
     // process error;
}...
```

## RETURNS

When *write* succeeds, the size of the write buffer (*T610_RW_BUFFER)* is returned. If write fails, -1 (SYSERR) is returned.

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

## SEE ALSO

LynxOS System Call - write()

TIP610 example application

# 3.5 ioctl()

## NAME

ioctl() – I/O device control

## SYNOPSIS

#include <ioctl.h>
#include <tip610.h>

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

## DESCRIPTION

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

The following ioctl codes are supported by the driver and are defined in tip610.h :

| Symbol | Meaning |
|---|---|
| *T610_READ_DIR* | Read current port direction of I/O lines |
| *T610_WRITE_DIR* | Set port direction of I/O lines |
| *T610_READ_POL* | Read current port polarity configuration |
| *T610_WRITE_POL* | Set new port polarity configuration |
| *T610_EVENT_READ* | Read port after specified input event occurred |

See behind for more detailed information on each control code.

## RETURNS

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

The TIP610 ioctl functions returns always standard error codes in errno . See LynxOS system call ioctl of a detailed description of possible error codes.

## SEE ALSO

LynxOS System Call - ioctl().

### 3.5.1 T610_READ_DIR

#### NAME

T610_READ_DIR - Read current port direction of I/O lines

#### DESCRIPTION

With this ioctl function the direction status of the three I/O ports can be read. A 0x00 in corresponding T610_RW_BUFFER member means current direction is output and 0xFF means input.

A pointer to the direction mask structure (T610_RW_BUFFER) is passed by the parameter *arg* to the driver. The driver will copy port direction status to the corresponding T610_RW_BUFFER member.

The *T610_RW_BUFFER* structure has the following layout:

```
typedef struct {
    unsigned char   portA;
    unsigned char   portB;
    unsigned char   portC;
    unsigned char   wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

**unsigned char portA**

Holds the current direction for I/O-lines 1 to 8. Where bit 20 corresponds to I/O-line 1, bit 21 to I/O-line 2 and so on.

**unsigned char portB**

Holds the current direction for I/O-lines 9 to 16. Where bit 20 corresponds to I/O-line 9, bit 21 to I/O-line 10 and so on.

**unsigned char portC**

Holds the current direction for I/O-lines 17 to 20. Where bit 20 corresponds to I/O-line 17, bit 21 to I/O-line 18 and so on. (Note: I/O-line 19 is input only)

**unsigned char wrenaPort**

not used

**After driver startup all I/O lines are set to be inputs.**

## 3.5.2 T610_WRITE_DIR

### NAME

T610_WRITE_DIR - Set direction of I/O lines

### DESCRIPTION

With this ioctl function each of the three I/O ports may be individually set as input or output. To set a port to be input, set the corresponding bits in the mask to 0. All I/O-lines of a port must have the same direction.

A pointer to the direction mask structure (T610_RW_BUFFER) is passed by the parameter *arg* to the driver.

The *T610_RW_BUFFER* structure has the following layout:

```
typedef struct {
    unsigned char   portA;
    unsigned char   portB;
    unsigned char   portC;
    unsigned char   wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

**unsigned char portA**

Holds the new direction for I/O-lines 1 to 8. Where bit 20 corresponds to I/O-line 1, bit 21 to I/O-line 2 and so on.

**unsigned char portB**

Holds the new direction for I/O-lines 9 to 16. Where bit 20 corresponds to I/O-line 9, bit 21 to I/O-line 10 and so on.

**unsigned char portC**

Holds the new direction for I/O-lines 17 to 20. Where bit 20 corresponds to I/O-line 17, bit 21 to I/O-line 18 and so on. (Note: I/O-line 19 is input only)

**unsigned char wrenaPort**

Determines whether changing of port direction (default is input) is allowed or not.

The following flags could be OR'ed :

*T610_ENABLE_PORTA* The contents of the member *portA* will be written to the corresponding port A direction register (I/O-lines 1..8).

*T610_ENABLE_PORTB* The contents of the member *portB* will be written to the corresponding port B direction register (I/O-lines 9..16).

*T610_ENABLE_PORTC* The contents of the member *portC* will be written to the corresponding port C direction register (I/O-lines 17..20).

**After driver startup all I/O lines are set to be inputs.**

## EXAMPLE

```
int                fd;
int                result;
T610_RW_BUFFER     dirBuf;


// Set line 1...8, 17, 18 and 20 to be output and line 9...16 to be input
dirBuf.portA  = 0x00;
dirBuf.portB  = 0xFF;
dirBuf.portC  = 0x00;

result = ioctl(fd, T610_SET_DIR, (char*)&dirBuf);

if (result != OK) {
     // process error;
}
```

## 3.5.3 T610_READ_POL

### NAME

T610_READ_POL – Read the current port polarity configuration

### DESCRIPTION

This ioctl function read the current port polarity configuration of the TIP610. The argument *arg* passes a pointer to a polarity mask structure (T610_RW_BUFFER) to the driver. After the call the driver will have filled the current port polarity configuration into the passed buffer.

The *T610_RW_BUFFER* structure has the following layout:

```
typedef struct {
   unsigned char   portA;
   unsigned char   portB;
   unsigned char   portC;
   unsigned char   wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

**unsigned char portA (I/O[1..8]), portB (I/O[9..16]), portC (I/O[17..20])**

Contains the current values for the corresponding port polarity register. A 0 in a particular bit position specifies the corresponding bit path of the port as non-inverting (that is, a HIGH level at the I/O connector is a 1). If a bit is written with 1, the data path is programmed inverting.

After reset the data path is non-inverting.

**unsigned char wrenaPort**

not used

### EXAMPLE

```
int                fd;
int                result;
T610_RW_BUFFER     polBuf;

result = ioctl(fd, T610_READ_POL, (char*)&polBuf);

if (result != OK) {
     // process error;
}
```

## 3.5.4 T610_WRITE_POL

### NAME

T610_WRITE_POL – Set port polarity configuration

### DESCRIPTION

This ioctl function sets the port polarity configuration. The argument *arg* passes a pointer to a polarity mask structure (T610_RW_BUFFER) to the driver.

The *T610_RW_BUFFER* structure has the following layout:

```
typedef struct {
    unsigned char   portA;
    unsigned char   portB;
    unsigned char   portC;
    unsigned char   wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

**unsigned char portA (I/O[1..8]), portB (I/O[9..16]), portC (I/O[17..20])**

Contains the new values for the corresponding port polarity register. A 0 in a particular bit position specifies the corresponding bit path of the port as non-inverting (that is, a HIGH level at the I/O connector is a 1). If a bit is written with 1, the data path is programmed inverting.

After reset the data path is non-inverting.

**unsigned char wrenaPort**

Set of bit flags that control the write port polarity operation. If the corresponding port flag is set the port polarity register will be written otherwise the port polarity register is inhibit from write.

The following flags could be OR'ed

| | |
|---|---|
| T610_ENABLE_PORTA | The contents of the member portA will be written to the corresponding port A polarity register (I/O-lines 1..8). |
| *T610_ENABLE_PORTB* | The contents of the member *portB* will be written to the corresponding port B polarity register (I/O-lines 9..16). |
| *T610_ENABLE_PORTC* | The contents of the member *portC* will be written to the corresponding port C polarity register (I/O-lines 17..20). |

### EXAMPLE

```
int                fd;
int                result;
T610_RW_BUFFER     polBuf;

// Set I/O-line 3, 9 and 12 to polarity “inverting”
polBuf.portA  = 0x04;
polBuf.portB  = 0x09;
polBuf.portC  = 0x00;
polBuf.wrenaPort   = 0x07;

result = ioctl(fd, T610_WRITE_POL, (char*)polBuf);

if (result != OK) {
     // process error;
}
```

## 3.5.5 T610_EVENT_READ

### NAME

T610_ EVENT_READ - Read port after specified input event occur

### DESCRIPTION

The ioctl function reads the contents of the input ports after a specified event occur.

Possible events are rising or falling edge or both, at a specified input bit or a pattern match of masked input bits.

A pointer to the callers read buffer (T610_EVRD_BUFFER) is passed by the argument arg to the driver.

The T610_EVRD_BUFFER structure has the following layout:

```
typedef struct {
   unsigned char   portA;      /* value on lines of Port A */
   unsigned char   portB;      /* value on lines of Port B */
   unsigned char   portC;      /* value on lines on Port C (Bit 0..3) */

   unsigned char   maskA;      /* mask for lines of Port A */
   unsigned char   maskB;      /* mask for lines of Port B */

   unsigned char   matchA;     /* pattern creating event for Port A */
   unsigned char   matchB;     /* pattern creating event for Port B */

   unsigned char   mode;       /* event mode */
   long            timeout;    /* timeout in ticks */
} T610_EVRD_BUFFER, *PT610_EVRD_BUFFER;
```

**unsigned char portA, portB, portC**

Holds the contents of the corresponding port data registers when the event occurred.

**unsigned char maskA, maskB**

Specifies a bit mask. A 1 value marks the corresponding bit position as relevant.

**unsigned matchA, matchB**

Specifies a pattern that must match to the contents of the input port. Only the bit positions specified by maskA / maskB must compare to the input port.

**unsigned char mode**

Specifies the “event” mode for this read request

| | |
|---|---|
| *T610_MATCH* | The driver reads the input port if the masked input bits match to the specified pattern. The input mask must be specified in the parameter *maskA/maskB*. A 1 value in *maskA/maskB* means than the input bit value “must-match” identically to the corresponding bit in the *matchA / matchB* parameter. |
| *T610_HIGH_TR* | The driver reads the input port if a high-transition at the specified bit position occur. A 1 value in *maskA / maskB* specifies the bit position of the input port. If you specify more than one bit position the events are OR’ed. That means the read is completed if a high-transition at least at one relevant bit position occur. |
| *T610_LOW_TR* | The driver reads the input port if a low-transition at the specified bit position occur. A 1 value in *maskA / maskB* specifies the bit position of the input port. If you specify more than one bit position the events are OR’ed. That means the read is completed if a low-transition at least at one relevant bit position occur. |
| *T610_ANY_TR* | The driver reads the input port if a transition (high or low) at the specified bit position occur. A 1 value in *maskA / maskB* specifies the bit position of the input port. If you specify more than one bit position the events are OR’ed. That means the read is completed if a transition at least at one relevant bit position occur. |

**long timeout**

Specifies the amount of time (in ticks) the caller is willing to wait for the specified event to occur. A value of (-1) means wait indefinitely.

## EXAMPLE

```
int fd;
int result;
T610_EVRD_BUFFER evBuf;


/*
**  Read the input port after..
**   bit 0 = 0
**   bit 1 = 1
**   bit 6 = 0
**   bit 7 = 1
*/
evBuf.mode = T610_MATCH;
evBuf.maskA = 0xC3;          /* bit 0,1,6,7 are relevant */
evBuf.matchA = 0x82;
evBuf.maskB = 0;             /* port B isn't relvant */
evBuf.matchB = 0;
evBuf.timeout = 100;         /* ticks */

result = ioctl(fd, T610_EVENT_READ, &evBuf);

if (result >= 0) {
```

```
     printf("Port A: %02Xh\n", evBuf.portA);
     printf("Port B: %02Xh\n", evBuf.portB);
     printf("Port C: %02Xh\n", evBuf.portC);
}
else {
     /* handle read error */
}


/*
**  Read the input port after a high-transition at
**  input line 8 occured (Port B bit 7)
*/
evBuf.mode = T610_HIGH_TR;
evBuf.maskB = 1<<7;              /* high-transition at bit 7 */
evBuf.maskA = 0;
evBuf.timeout = 100;             /* ticks */

result = ioctl(fd, T610_EVENT_READ, &evBuf);

if (result >= 0) {
     printf("Port A: %02Xh\n", evBuf.portA);
     printf("Port B: %02Xh\n", evBuf.portB);
     printf("Port C: %02Xh\n", evBuf.portC);
}
else {
     /* handle read error */
}
```

## ERRORS

| | |
|---|---|
| EBUSY | The maximum number of concurrent requests was exceeded. Increase the value of *T610_MAX_REQUESTS* in “tip610def.h”. |
| ETIMEDOUT | The allowed time to finish the request is elapsed. |
| EINTR | Interrupted system call (probably by a signal). |

# 4 Debugging and Diagnostic

This driver was successfully tested on a Motorola MVME2306-900 board, and developed on a Windows Cross Environment for LynxOS V4.0.0.

If the driver will not work properly please enable debug outputs by removing the comments around the symbol *DEBUG.*

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

The debug output displays the device information data for the current major device, a memory dump of the IP ID space (contents of the ID-PROM) and a memory dump of the IP I/O space (registers) like this.

```
TIP610 Device Driver Install
IP IO Virtual Address    = CFFF8000
IP ID  Virtual Address   = CFFF8080
Interrupt Vector         = 64

IP ID space (ID-PROM)...
CFFF8080 : FF 49 FF 50 FF 41 FF 43 FF B3 FF 2B FF 10 FF 00
CFFF8090 : FF 00 FF 00 FF 0D FF 10 FF 0A FF 00 FF 00 FF 00

IP IO space (CIO direct accessable registers)...
CFFF8000 : FF 01 FF 01 FF 01 FF 01
```

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