*The Embedded I/O Company*

TEWS TECHNOLOGIES

# TIP675-SW-95

## QNX-Neutrino Device Driver

48 TTL I/O Lines with Interrupts

Version 1.0.x

## User Manual

Issue 1.0.0

April 2009

| | | | |
|---|---|---|---|
| **TEWS TECHNOLOGIES GmbH** | | **TEWS TECHNOLOGIES LLC** | |
| Am Bahnhof 7 | Phone: +49 (0) 4101 4058 0 | 9190 Double Diamond Parkway, | Phone: +1 (775) 850 5830 |
| 25469 Halstenbek, Germany | Fax: +49 (0) 4101 4058 19 | Suite 127, Reno, NV  89521, USA | Fax: +1 (775) 201 0347 |
| www.tews.com | e-mail: info@tews.com | www.tews.com | e-mail: usasales@tews.com |

# TIP675-SW-95

QNX-Neutrino Device Driver

48 TTL I/O Lines with Interrupts

Supported Modules:
TIP675

| Issue | Description | Date |
|---|---|---|
| 1.0.0 | First Issue | April 1, 2009 |

# Table of Contents

# 1 Introduction

## 1.1 Device Driver

The TIP675-SW-95 QNX-Neutrino device driver allows the operation of the TIP675 Digital I/O IndustryPack® on QNX-Neutrino systems and requires the TEWS TECHNOLOGIES QNX-Neutrino IPAC Carrier Driver Software (CARRIER-SW-95).

The TIP675 device driver is basically implemented as a user installable Resource Manager. The standard file (I/O) functions (*open()*, *close()* and *devctl()*) provide the basic interface for opening and closing a file descriptor and for performing device I/O and control operations.

The TIP675 device driver supports the following features:

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

<u>The TIP675-SW-95 device driver supports the modules listed below:</u>

| | | |
|---|---|---|
| TIP675 | 48 TTL I/O Lines with Interrupts | IndustryPack® |

To get more information about the features and use of TIP675 devices it is recommended to read the manuals listed below.

TIP675 User manual

TIP675 Engineering Manual

CARRIER-SW-95 User Manual

## 1.2 IPAC Carrier Driver

IndustryPack (IPAC) carrier boards have different implementations of the system to IndustryPack bus bridge logic, different implementations of interrupt and error handling and so on. Also the different byte ordering (big-endian versus little-endian) of CPU boards will cause problems on accessing the IndustryPack I/O and memory spaces.

To simplify the implementation of IPAC device drivers which work with any supported carrier board, TEWS TECHNOLOGIES has designed a so called Carrier Driver that hides all differences of different carrier boards under a well defined interface.

The TEWS TECHNOLOGIES IPAC Carrier Driver CARRIER-SW-95 is part of this TIP675-SW-95 distribution. It is located in directory CARRIER-SW-95 on the corresponding distribution media.

This IPAC Device Driver requires a properly installed IPAC Carrier Driver. Due to the design of the Carrier Driver, it is sufficient to install the IPAC Carrier Driver once, even if multiple IPAC Device Drivers are used.

Please refer to the CARRIER-SW-95 User Manual for a detailed description how to install and setup the CARRIER-SW-95 device driver, and for a description of the TEWS TECHNOLOGIES IPAC Carrier Driver concept.

# 2 Installation

Following files are located on the distribution media:

Directory path 'TIP675-SW-95':

| | |
|---|---|
| TIP675-SW-95-SRC.tar.gz | GZIP compressed archive with driver source code |
| TIP675-SW-95-1.0.0.pdf | PDF copy of this manual |
| ChangeLog.txt | Release history |
| Release.txt | Release information |

For installation the files have to be copied to the desired target directory.

The GZIP compressed archive TIP675-SW-95-SRC.tar.gz contains the following files and directories:

Directory path 'tip675':

| | |
|---|---|
| driver/tip675.c | TIP675 device driver source |
| driver/tip675def.h | TIP675 driver include file |
| driver/tip675.h | TIP675 include file for driver and application |
| driver/Makefile | TIP675 driver makefile |
| api/tip675api.h | TIP675 API include file for application |
| api/tip675api.c | TIP675 API source file |
| example/exampleapp/ | Directory containing Interactive Example application (using driver functions) |
| example/read/ | Directory containing Console application for reading data (using API) |

Each example directory contains the following structure:

| | |
|---|---|
| tip675*example*.c | Example application source file |
| common.mk | make parameters |
| Makefile | recursive makefile |
| nto/Makefile | recursive makefile |
| nto/x86/Makefile | recursive makefile |
| nto/x86/o/Makefile | recursive makefile |

In order to perform an installation, copy TIP675-SW-95-SRC.tar.gz to /usr/src and extract all files of the archive. The command 'tar -xzvf TIP675-SW-95-SRC.tar.gz' will extract the files into the local directory.

**Before building a new device driver, the TEWS TECHNOLOGIES IPAC carrier driver must be installed properly, because this driver includes the header files *ipac_*.h*, which is part of the IPAC carrier driver distribution. Please refer to the IPAC carrier driver user manual in the directory path *CARRIER-SW-95* on the distribution media.**

**Its absolute important to extract the TIP675-SW-95-SRC.tar.gz in the /usr/src directory otherwise the automatic build with make will fail.**

## 2.1 Build the device driver

Change to the */usr/src/tip675/driver* directory

Execute the Makefile

```
# make install
```

After successful completion the driver binary will be installed in the */bin* directory.

## 2.2 Build the example application

Change to the desired example application sub-directory beneath */usr/src/tip675/example/*. Make sure that either symbolic links to the TIP675 Application Programming Interface source and header file (tip675api.c and tip675api.h) located in */usr/src/tip675/api/* are available in the example's directory, or simply copy the API files into the directory. The API must be compiled together with the provided example applications.

Execute the Makefile:

```
# make install
```

After successful completion the example binary (*tip675exa, tip675read*) will be installed in the */bin* directory.

## 2.3 Start the driver process

To start the TIP675 Resource Manager (Device Driver) you only have to start the TEWS TECHNOLOGIES IPAC Carrier Driver. The Carrier Driver automatically detects installed TEWS IPACs and dynamically loads the concerning module(s).

```
# ipac_class &
```

The TIP675 Resource Manager registers a device for each TIP675 in the QNX-Neutrinos pathname space under following name, where *n* specifies the used IPAC slot number (please refer to the IPAC Carrier Driver Manual):

```
/dev/tip675_n
```

This pathname must be used in the application program to open a path to the desired TIP675 device.

For debugging purposes, you can start the IPAC Carrier Driver with the –V (verbose) option. Now the Resource Manager will print versatile information about TIP675 configuration and command execution on the terminal window. For further details about debugging see IPAC Carrier Driver Manual.

# 3 I/O Functions

This chapter describes the interface to the device driver I/O system.

## 3.1 open()

### NAME

open() - open a file descriptor

### SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open
(
        const char   *pathname,
        int          flags
)
```

### DESCRIPTION

The *open()* function creates and returns a new file descriptor for a TIP675 device.

### PARAMETER

*pathname*

Specifies the device to open.

*flags*

Controls how the file is to be opened. TIP675 devices must be opened *O_RDWR*.

### EXAMPLE

```
int  fd;

fd = open( “/dev/tip675_0”, O_RDWR );
if (fd == -1)
{
     /* Handle error */
}
```

## RETURNS

The normal return value is a non-negative integer file descriptor. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

## ERROR CODES

Returns only Neutrino specific error codes, see Neutrino Library Reference.

## SEE ALSO

Library Reference - open()

## 3.2 close()

### NAME

close() – close a file descriptor

### SYNOPSIS

```
#include <unistd.h>

int close
(
    int         filedes
)
```

### DESCRIPTION

The *close()* function closes a file.

### PARAMETER

*filedes*

Specifies the file to close.

### EXAMPLE

```
int  fd;

if (close(fd) != 0)
{
     /* handle close error conditions */
}
```

### RETURNS

The normal return value is 0. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

## ERROR CODES

Returns only Neutrino specific error code, see Neutrino Library Reference.

## SEE ALSO

Library Reference - close()

## 3.3 devctl()

### NAME

devctl() – device control functions

### SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>
#include <tip675.h>

int devctl
(
        int         filedes,
        int         dcmd,
        void        *data_ptr,
        size_t      n_bytes,
        int         *dev_info_ptr
)
```

### DESCRIPTION

The *devctl()* function sends a control code directly to a device.

### PARAMETER

*filedes*

Specifies the device to perform the requested operation.

*dcmd*

Specifies the control code for the operation. The following commands are defined (*tip675.h*):

| **Command** | **Description** |
|---|---|
| DCMD_TIP675_READ | Read current digital input value |
| DCMD_TIP675_EVENT_READ | Wait for an event, and return input port (blocked read) |
| DCMD_TIP675_WRITE | Write new digital output value |
| DCMD_TIP675_WRITE_MASK | Write new output value with bitmask |
| DCMD_TIP675_OUTPUTBIT_SET | Set individual output line |
| DCMD_TIP675_OUTPUTBIT_CLEAR | Clear individual output line |
| DCMD_TIP675_CONFIGURE_DIRECTION | Configure direction of I/O lines |
| DCMD_TIP675_ENABLE_EXTCLK | Enable use of external clock signal and specify the active clock transition. |
| DCMD_TIP675_DISABLE_EXTCLK | Disable use of external clock |

*data_ptr*

Depends on the command and will be described for each command in detail later in this chapter. Usually points to a buffer that passes data between the user task and the driver

*n_bytes*

Depends on the command and will be described for each command in detail later in this chapter. Usually defines the size of the buffer pointed by *data_ptr*.

*dev_info_ptr*

Is unused for the TIP675 driver and should be set to *NULL*.

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERRORS

| | |
|---|---|
| ENOTTY | Inappropriate I/O control operation. This error code is returned if the requested *devctl()* function is unknown. Please check the argument *dcmd*. |

Other function dependent error codes will be described for each *devctl()* code separately. Note, the TIP675 driver always returns standard QNX error codes.

## SEE ALSO

Library Reference - devctl()

## 3.3.1 DCMD_TIP675_READ

### DESCRIPTION

This function reads the input value of the I/O lines. Depending on the configured mode for the external clock the returned value may not be the current state at the I/O lines.

The function specific argument *data_ptr* points to a data buffer (*TIP675_RW_BUFFER*) and *n_bytes* specifies its length in bytes.

```
typedef struct
{
        unsigned short          value_1_16;
        unsigned short          value_17_32;
        unsigned short          value_33_48;
} TIP675_RW_BUFFER;
```

*value_1_16*

This value returns the state of the I/O lines 1...16. Bit 0 contains the state of I/O line 1, bit 1 contains the state of line 2 and so on.

*value_17_32*

This value returns the state of the I/O lines 17...32. Bit 0 contains the state of I/O line 17, bit 1 contains the state of line 18 and so on.

*value_33_48*

This value returns the state of the I/O lines 33...48. Bit 0 contains the state of I/O line 33, bit 1 contains the state of line 34 and so on.

### EXAMPLE

```
#include <tip675.h>

int                 fd;
int                 result;
TIP675_RW_BUFFER    inBuf;

…
```

```
…

/* read input value */
result = devctl(   fd,
                   DCMD_TIP675_READ,
                   &inBuf,
                   sizeof(inBuf),
                   NULL);
if (result == EOK)
{
     printf( “Value: %04X %04X %04X\n”,
                   inBuf.value_33_48,
                   inBuf.value_17_32,
                   inBuf.value_1_16);
}
else
{
     /* Handle error */
}
```

### ERRORS

| | |
|---|---|
| EINVAL | Invalid argument specified, or buffer too small. |

## 3.3.2 DCMD_TIP675_EVENT_READ

### DESCRIPTION

This function reads the state of the I/O lines after a specified event has occurred. Events can be generated on the rising, falling edge or both (any transition) of a specified I/O line.

**Transition detection and value reading is done by an interrupt service routine, and is affected by the system's interrupt latency, so this function should not be used for fast changing signals.**

The function specific argument *data_ptr* points to a data structure (*TIP675_EVRD_BUFFER*) and *n_bytes* specifies its length in bytes.

```
typedef struct
{
        unsigned short      value_1_16;
        unsigned short      value_17_32;
        unsigned short      value_33_48;
        unsigned short      mask_1_16;
        unsigned short      mask_17_32;
        unsigned short      mask_33_48;
        unsigned char       mode;
        long                timeout;
} TIP675_EVRD_BUFFER;
```

*value_1_16*

This value returns the state of the I/O lines 1...16. Bit 0 contains the state of I/O line 1, bit 1 contains the state of line 2 and so on.

*value_17_32*

This value returns the state of the I/O lines 17...32. Bit 0 contains the state of I/O line 17, bit 1 contains the state of line 18 and so on.

*value_33_48*

This value returns the state of the I/O lines 33...48. Bit 0 contains the state of I/O line 33, bit 1 contains the state of line 34 and so on.

*mask_1_16*

This mask specifies the I/O lines which shall be observed for the specified I/O event. This value specifies the mask for I/O lines 1...16. Bit 0 contains the mask for I/O line 1, bit 1 contains the mask for line 2 and so on.

*mask_17_32*

This mask specifies the I/O lines which shall be observed for the specified I/O event. This value specifies the mask for I/O lines 17...32. Bit 0 contains the mask for I/O line 17, bit 1 contains the mask for line 18 and so on.

*mask_33_48*

This mask specifies the I/O lines which shall be observed for the specified I/O event. This value specifies the mask for I/O lines 33...48. Bit 0 contains the mask for I/O line 33, bit 1 contains the mask for line 33 and so on.

*mode*

This parameter specifies the "event" mode for this read request. Following values are possible:

| Value | Description |
|---|---|
| TIP675_HIGH_TR | The driver reads the input registers if a high-transition at the specified bit position occurs. A set bit in the *mask_x_y* values specifies the bit position of the input register. 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. |
| TIP675_LOW_TR | The driver reads the input port registers if a low-transition at the specified bit position occurs. A set bit in the *mask_x_y* values 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. |
| TIP675_ANY_TR | The driver reads the input port registers if a transition transition (high or low) at the specified bit position occurs. A set bit in the *mask_x_y* values 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. |

*timeout*

Specifies the amount of time (in seconds) the caller is at least willing to wait for the specified event to occur. A value of -1 means wait indefinitely or no timeout.

## EXAMPLE

```
#include <tip675.h>

int                     fd;
int                     result;
TIP675_EVRD_BUFFER      EventBuf;

…
```

```
…

/*
** Wait for a rising edge (high transition) at first input line
*/
EventBuf.mode            = TIP675_HIGH_TR;
EventBuf.mask_1_16       = 0x0001;
EventBuf.mask_17_32      = 0x0000;
EventBuf.mask_33_48      = 0x0000;
EventBuf.timeout         = 10;         /* 10 seconds */

result = devctl(   fd,
                   DCMD_TIP675_EVENT_READ,
                   &EventBuf,
                   sizeof(EventBuf),
                   NULL);
if (result == EOK)
{
     printf( “Value: %04X %04X %04X\n”,
                   EventBuf.value_33_48,
                   EventBuf.value_17_32,
                   EventBuf.value_1_16);
}
else
{
     /* Handle error */
}
```

## ERRORS

| | |
|---|---|
| EINVAL | Invalid argument specified, or buffer too small. |
| ETIMEDOUT | The event has not occurred before the specified timeout. |

## 3.3.3 DCMD_TIP675_WRITE

### DESCRIPTION

This function sets the new output register value.

The function specific argument *data_ptr* points to a data buffer (*TIP675_RW_BUFFER*), and *n_bytes* specifies its length in bytes.

```
typedef struct
{
        unsigned short          value_1_16;
        unsigned short          value_17_32;
        unsigned short          value_33_48;
} TIP675_RW_BUFFER;
```

*value_1_16*

This value specifies the output value for the lines 1...16. Bit 0 specifies the value for line 1, bit 1 contains the value for line 2 and so on.

*value_17_32*

This value specifies the output value for the lines 17...32. Bit 0 specifies the value for line 17, bit 1 specifies the value for line 18 and so on.

*value_33_48*

This value specifies the output value for the lines 33...48. Bit 0 specifies the value for line 33, bit 1 specifies the value for line 34 and so on.

### EXAMPLE

```
#include <tip675.h>

int                 fd;
int                 result;
TIP675_RW_BUFFER    outBuf;

…
```

```
…

/*
** Write new output value
*/
outBuf.value_33_48 = 0xBA98;
outBuf.value_17_32 = 0x7654;
outBuf.value_1_16  = 0x3210;
result = devctl(   fd,
                   DCMD_TIP675_WRITE,
                   &outBuf,
                   sizeof(outBuf),
                   NULL);
if (result == EOK)
{
     /* OK */
}
else
{
     /* Handle error */
}
```

## ERRORS

| | |
|---|---|
| EINVAL | Invalid argument specified, or buffer too small. |

### 3.3.4 DCMD_TIP675_WRITE_MASK

#### DESCRIPTION

This function writes a specified set of data bits to the digital output register.

The function specific argument *data_ptr* points to a data structure (*TIP675_RW_MASKED_BUFFER*) and *n_bytes* specifies its length in bytes.

```
typedef struct
{
        unsigned short      value_1_16;
        unsigned short      value_17_32;
        unsigned short      value_33_48;
        unsigned short      mask_1_16;
        unsigned short      mask_17_32;
        unsigned short      mask_33_48;
} TIP675_RW_MASKED_BUFFER;
```

*value_1_16*

This value specifies the output value for the lines 1...16. Bit 0 specifies the value for line 1, bit 1 contains the value for line 2 and so on.

*value_17_32*

This value specifies the output value for the lines 17...32. Bit 0 specifies the value for line 17, bit 1 specifies the value for line 18 and so on.

*value_33_48*

This value specifies the output value for the lines 33...48. Bit 0 specifies the value for line 33, bit 1 specifies the value for line 34 and so on.

*mask_1_16*

This parameter specifies the bitmask for the lines 1...16. Only active bits (1) will be written to the TIP675 output register. Bit 0 masks the value of line 1, bit 1 masks the value of line 2 and so on.

*mask_17_32*

This parameter specifies the bitmask for the lines 17...32. Only active bits (1) will be written to the TIP675 output register. Bit 0 masks the value of line 17, bit 1 masks the value of line 18 and so on.

*mask_33_48*

This parameter specifies the bitmask for the lines 33...48. Only active bits (1) will be written to the TIP675 output register. Bit 0 masks the value of line 33, bit 1 masks the value of line 34 and so on.

## EXAMPLE

```
#include <tip675.h>

int                             fd;
int                             result;
TIP675_RW_MASKED_BUFFER         WriteBuf;

/*
** Write new output value
**   Change only lines 1..8, 16, 17 and 32
*/
WriteBuf.value_33_48    = 0x1234;
WriteBuf.value_17_32    = 0xFFFF;
WriteBuf.value_1_16     = 0x0000;
WriteBuf.mask_33_48     = 0x0000; /* no lines to modify */
WriteBuf.mask_17_32     = 0x8001; /* modify line 17 and 32 */
WriteBuf.mask_1_16      = 0x80FF;     /* modify lines 1..8 and 16 */
result = devctl(   fd,
                   DCMD_TIP675_WRITE_MASK,
                   &WriteBuf,
                   sizeof(WriteBuf),
                   NULL);
if (result == EOK)
{
     /* OK */
}
else
{
     /* Handle error */
}
```

## ERRORS

| | |
|---|---|
| EINVAL | Invalid argument specified, or buffer too small. |

## 3.3.5 DCMD_TIP675_OUTPUTBIT_SET

### DESCRIPTION

This function sets a single output line leaving all other output lines in the current state.

The function specific argument *data_ptr* points to an *int* buffer, and *n_bytes* specifies its length in bytes. The value specifies the output line that shall be set. Allowed values are 1 for I/O line 1 up to 48 for I/O line 48.

### EXAMPLE

```
#include <tip675.h>

int             fd;
int             result;
int             lineNo;

/*
** set output line 8
*/
lineNo = 8;

result = devctl(   fd,
                   DCMD_TIP675_OUTPUTBIT_SET,
                   &lineNo,
                   sizeof(lineNo),
                   NULL);
if (result == EOK)
{
     /* OK */
}
else
{
     /* Handle error ot timeout */
}
```

### ERRORS

| | |
|---|---|
| EINVAL | Invalid argument specified, or buffer too small. |

## 3.3.6 DCMD_TIP675_OUTPUTBIT_CLEAR

### DESCRIPTION

This function clears a single output line leaving all other output lines in the current state.

The function specific argument *data_ptr* points to an *int* buffer, and *n_bytes* specifies its length in bytes. The value specifies the output line that shall be cleared. Allowed values are 1 for I/O line 1 up to 48 for I/O line 48.

### EXAMPLE

```
#include <tip675.h>

int             fd;
int             result;
int             lineNo;

/*
** clear output line 42
*/
lineNo = 42;

result = devctl(   fd,
                   DCMD_TIP675_OUTPUTBIT_SET,
                   &lineNo,
                   sizeof(lineNo),
                   NULL);
if (result == EOK)
{
     /* OK */
}
else
{
     /* Handle error ot timeout */
}
```

### ERRORS

| | |
|---|---|
| EINVAL | Invalid argument specified, or buffer too small. |

## 3.3.7 DCMD_TIP675_CONFIGURE_DIRECTION

### DESCRIPTION

This function configures the I/O line direction for input or output.

The function specific argument *data_ptr* points to a data buffer (*TIP675_RW_BUFFER*), and *n_bytes* specifies its length in bytes.

```
typedef struct
{
        unsigned short          value_1_16;
        unsigned short          value_17_32;
        unsigned short          value_33_48;
} TIP675_RW_BUFFER;
```

*value_1_16*

This value specifies the direction for lines 1...16. Bit 0 specifies the direction for line 1, bit 1 contains the direction for line 2 and so on.
A set bit (1) will enable output. A cleared bit (0) configures the I/O line for input.

*value_17_32*

This value specifies the direction for lines 17...32. Bit 0 specifies the direction for line 17, bit 1 specifies the direction for line 18 and so on.
A set bit (1) will enable output. A cleared bit (0) configures the I/O line for input.

*value_33_48*

This value specifies the direction for lines 33...48. Bit 0 specifies the direction for line 33, bit 1 specifies the direction for line 34 and so on.
A set bit (1) will enable output. A cleared bit (0) configures the I/O line for input.

### EXAMPLE

```
#include <tip675.h>

int                 fd;
int                 result;
TIP675_RW_BUFFER    dirBuf;

…
```

```
…

/*
** Configure I/O line 1..20 for input and 21..48 for output
*/
dirBuf.value_33_48 = 0xFFFF;
dirBuf.value_17_32 = 0xFFF0;
dirBuf.value_1_16  = 0x0000;
result = devctl(   fd,
                   DCMD_TIP675_CONFIGURE_DIRECTION,
                   &dirBuf,
                   sizeof(dirBuf),
                   NULL);
if (result == EOK)
{
     /* OK */
}
else
{
     /* Handle error */
}
```

## ERRORS

| | |
|---|---|
| EINVAL | Invalid argument specified, or buffer too small. |

## 3.3.8 DCMD_TIP675_ENABLE_EXTCLK

### DESCRIPTION

This function configures and enables the external clock input, which allows synchronized update of data output and input. The active edge of the external clock input can be configured by a parameter. A value of *TIP675_POS_EDGE_CLK* will set the low to high transition active, a value *TIP675_POS_EDGE_CLK* the high to low transition.

The function specific argument *data_ptr* points to an *int* data buffer holding the configuration parameter, and *n_bytes* species its length in bytes.

### EXAMPLE

```
#include <tip675.h>

int             fd;
int             result;
int             extClkCfg;

/*
** Use external clock with the positive edge
*/
extClkCfg = TIP675_POS_EDGE_CLK;
result = devctl(   fd,
                   DCMD_TIP675_ENABLE_EXTCLK,
                   &extClkCfg,
                   sizeof(extClkCfg),
                   NULL);
if (result == EOK)
{
     /* OK */
}
else
{
     /* Handle error */
}
```

### ERRORS

| | |
|---|---|
| EINVAL | Invalid argument specified, or buffer too small. |

## 3.3.9 DCMD_TIP675_DISABLE_EXTCLK

### DESCRIPTION

This function disables external clock input.

The function does not use the parameters *data_ptr* and *n_bytes*, they must be set to 0.

### EXAMPLE

```
#include <tip675.h>

int  fd;
int  result;

/*
** Disable external clock input
*/
result = devctl(   fd,
                   DCMD_TIP675_DISABLE_EXTCLK,
                   NULL,
                   0,
                   NULL);
if (result == EOK)
{
    /* OK */
}
else
{
    /* Handle error */
}
```

# 4 API Documentation

## 4.1 General Functions

### 4.1.1 tip675open()

#### Name

tip675open() – opens a device.

#### Synopsis

```
int tip675open
(
     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.

#### Example

```
#include “tip675api.h”
int FileDescriptor;

/*
** open file descriptor to device
*/
FileDescriptor = tip675open( “/dev/tip675_0” );
if (FileDescriptor < 0)
{
          /* handle open error */
}
```

### RETURNS

A device descriptor number, 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.

### 4.1.2 tip675close()

#### Name

tip675close() – closes a device.

#### Synopsis

```
int tip675close
(
     int  FileDescriptor
);
```

#### Description

This function closes previously opened devices.

#### Parameters

*FileDescriptor*

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

#### Example

```
#include “tip675api.h”
int FileDescriptor;
int result;

/*
** close file descriptor to device
*/
result = tip675close( FileDescriptor );
if (result < 0)
{
     /* handle close error */
}
```

## RETURNS

EOK, 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.

## 4.2 Device Access Functions

### 4.2.1 tip675read()

#### Name

tip675read() – read current input value.

#### Synopsis

```
int tip675read
(
     int                 FileDescriptor,
     unsigned short*     pInputValue_1_16,
     unsigned short*     pInputValue_17_32,
     unsigned short*     pInputValue_33_48
);
```

#### Description

This function reads the current value of the input register of the specified device.

#### Parameters

*FileDescriptor*

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

*pInputValue_1_16*

This value is a pointer to an unsigned short buffer which receives the current value of the input register for I/O lines 1…16. Bit 0 of this value corresponds to I/O line 1, bit 1 corresponds to I/O line 2 and so on. A NULL pointer can be specified, if the input value is not needed.

*pInputValue_17_32*

This value is a pointer to an unsigned short buffer which receives the current value of the input register for I/O lines 17…32. Bit 0 of this value corresponds to I/O line 17, bit 1 corresponds to I/O line 18 and so on. A NULL pointer can be specified, if the input value is not needed.

*pInputValue_33_48*

This value is a pointer to an unsigned short buffer which receives the current value of the input register for I/O lines 33…48. Bit 0 of this value corresponds to I/O line 33, bit 1 corresponds to I/O line 34 and so on. A NULL pointer can be specified, if the input value is not needed.

## Example

```
#include “tip675api.h”

int                 FileDescriptor;
int                 result;
unsigned short      inVal_1_16;
unsigned short      inVal_17_32;
unsigned short      inVal_33_48;

/*
** read current input value
*/
result = tip675read(    FileDescriptor,
                        &inVal_1_16,
                        &inVal_17_32,
                        &inVal_33_48 );
if (result == EOK)
{
     printf( “Input Value: %04X  %04X  %04X\n”,
              inVal_33_48, inVal_17_32, inVal_1_16);
}
else
{
     /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

All error codes a standard error codes set by the I/O system.

### 4.2.2 tip675eventRead()

#### Name

tip675eventRead() – wait for an event, and read input value

#### Synopsis

```
int tip675eventRead
(
     int                   EventMode,
     unsigned char         Mask_1_16,
     unsigned char         Mask_17_32,
     unsigned char         Mask_33_48,
     long                  Timeout,
     unsigned short*       pInputValue_1_16,
     unsigned short*       pInputValue_17_32,
     unsigned short*       pInputValue_33_48
);
```

#### Description

This function waits for an event, and reads the value of the input register of the specified device after this event has occurred. This function should not be used for fast changing devices, because event detection and reading the input value is highly affected by the system's interrupt latency.

#### Parameters

*FileDescriptor*

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

*EventMode*

This parameter specifies the “event” mode for this read request

| Value | Description |
|---|---|
| TIP675_HIGH_TR | The driver reads the value of the input register if a high-transition at the specified bit position occurs. A 1 value in *mask* 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. |
| TIP675_LOW_TR | The driver reads the value of the input register if a low-transition at the specified bit position occurs. A 1 value in *mask* 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. |
| TIP675_ANY_TR | The driver reads the value of the input register if a transition (high or low) at the specified bit position occurs. A 1 value in *mask* 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. |

*Mask_1_16*

This mask specifies the I/O lines which shall be observed for the specified I/O event. This value specifies the mask for I/O lines 1...16. Bit 0 contains the mask for I/O line 1, bit 1 contains the mask for line 2 and so on.

*Mask_17_32*

This mask specifies the I/O lines which shall be observed for the specified I/O event. This value specifies the mask for I/O lines 17...32. Bit 0 contains the mask for I/O line 17, bit 1 contains the mask for line 18 and so on.

*Mask_33_48*

This mask specifies the I/O lines which shall be observed for the specified I/O event. This value specifies the mask for I/O lines 33...48. Bit 0 contains the mask for I/O line 33, bit 1 contains the mask for line 33 and so on.

*Timeout*

Specifies the amount of time (in seconds) the caller is at least willing to wait for the specified event to occur. A value of -1 means wait indefinitely or no timeout.

*pInputValue_1_16*

This value is a pointer to an unsigned short buffer which receives the current value of the input register for I/O lines 1…16. Bit 0 of this value corresponds to I/O line 1, bit 1 corresponds to I/O line 2 and so on. A NULL pointer can be specified, if the input value is not needed.

*pInputValue_17_32*

This value is a pointer to an unsigned short buffer which receives the current value of the input register for I/O lines 17…32. Bit 0 of this value corresponds to I/O line 17, bit 1 corresponds to I/O line 18 and so on. A NULL pointer can be specified, if the input value is not needed.

*pInputValue_33_48*

This value is a pointer to an unsigned short buffer which receives the current value of the input register for I/O lines 33…48. Bit 0 of this value corresponds to I/O line 33, bit 1 corresponds to I/O line 34 and so on. A NULL pointer can be specified, if the input value is not needed.

## Example

```
#include “tip675api.h”

int                 FileDescriptor;
int                 result;
unsigned short      inVal_1_16;
unsigned short      inVal_17_32;
unsigned short      inVal_33_48;

/*
** wait at least 5 seconds for any transition on input line 5
*/
result = tip675eventRead(
              FileDescriptor,
              TIP675_ANY_TR,      /* EventMode                */
              0x0010,             /* Mask 1..16               */
              0x0000,             /* Mask 17..32              */
              0x0000,             /* Mask 33..48              */
              5,                  /* Timeout                  */
              &inVal_1_16,
              &inVal_17_32,
              &inVal_33_48 );
if (result == EOK)
{
     printf( “Event occurred: Input Value: %04X %04X %04X\n”,
              &inVal_33_48, &inVal_17_32, &inVal_1_16);
}
else
{
     /* handle error */
}

…
```

…

```
/*
** wait at least 30 seconds for high transition on input line 40
*/
result = tip675eventRead(
              FileDescriptor,
              TIP675_HIGH_TR,     /* EventMode                  */
              0x0000,             /* Mask 1..16                 */
              0x0000,             /* Mask 17..32                */
              0x0080,             /* Mask 33..48                */
              30,                 /* Timeout                    */
              &inVal_1_16,
              &inVal_17_32,
              &inVal_33_48 );
if (result == EOK)
{
     printf( “Event occurred: Input Value: %04X %04X %04X\n”,
              &inVal_33_48, &inVal_17_32, &inVal_1_16);
}
else
{
     /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

| | |
|---|---|
| ETIMEDOUT | The event has not occurred before the specified timeout. |
| EINVAL | Invalid event mode specifed. |

All other error codes a standard error codes set by the I/O system.

## 4.2.3 tip675write()

### Name

tip675write() – write new output value.

### Synopsis

```
int tip675write
(
     int                 FileDescriptor,
     unsigned short      OutputValue_1_16,
     unsigned short      OutputValue_17_32,
     unsigned short      OutputValue_33_48
);
```

### Description

This function writes a new output value for the specified device. All output lines are affected, and will be set according to the specified output value. If external clock is enabled, the output values will not be updated before the clock transition occurs.

### Parameters

*FileDescriptor*

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

*OutputValue_1_16*

This value specifies the new output value for I/O lines 1...16. Bit 0 of this value corresponds to I/O line 1, bit 1 corresponds to I/O line 2 and so on.

*OutputValue_17_32*

This value specifies the new output value for I/O lines 17...32. Bit 0 of this value corresponds to I/O line 17, bit 1 corresponds to I/O line 18 and so on.

*OutputValue_33_48*

This value specifies the new output value for I/O lines 33...48. Bit 0 of this value corresponds to I/O line 33, bit 1 corresponds to I/O line 34 and so on.

## Example

```
#include “tip675api.h”

int             FileDescriptor;
int             result;

/*
** write new output value:
** set I/O line 2 and I/O line 17 to ON, all other lines to OFF.
*/
result = tip675write(
                FileDescriptor,
                0x0002,       /* OutputValue 1..16         */
                0x0001,       /* OutputValue 17..32        */
                0x0000);      /* OutputValue 33..48        */
if (result == EOK)
{
     /* OK */
}
else
{
     /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

All error codes a standard error codes set by the I/O system.

### 4.2.4 tip675writeMask()

#### Name

tip675writeMask() – write relevant bits of output value.

#### Synopsis

```
int tip675writeMask
(
     int                  FileDescriptor,
     unsigned short       OutputValue_1_16,
     unsigned short       OutputValue_17_32,
     unsigned short       OutputValue_33_48,
     unsigned short       Mask_1_16,
     unsigned short       Mask_17_32,
     unsigned short       Mask_33_48
);
```

#### Description

This function writes relevant bits of a new output value for the specified device.

#### Parameters

*FileDescriptor*

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

*OutputValue_1_16*

This value specifies the new output value for I/O lines 1...16. Bit 0 of this value corresponds to I/O line 1, bit 1 corresponds to I/O line 2 and so on.

*OutputValue_17_32*

This value specifies the new output value for I/O lines 17...32. Bit 0 of this value corresponds to I/O line 17, bit 1 corresponds to I/O line 18 and so on.

*OutputValue_33_48*

This value specifies the new output value for I/O lines 33...48. Bit 0 of this value corresponds to I/O line 33, bit 1 corresponds to I/O line 34 and so on.

*Mask_1_16*

This parameter specifies the bitmask. Only active bits (1) will be written to the TIP675 output register of I/O lines 1...16, all other output lines will be left unchanged. Bit 0 of this value corresponds to I/O line 1, bit 1 corresponds to I/O line 2 and so on.

*Mask_17_32*

This parameter specifies the bitmask. Only active bits (1) will be written to the TIP675 output register of I/O lines 17...32, all other output lines will be left unchanged. Bit 0 of this value corresponds to I/O line 17, bit 1 corresponds to I/O line 18 and so on.

*Mask_33_48*

This parameter specifies the bitmask. Only active bits (1) will be written to the TIP675 output register of I/O lines 33...48, all other output lines will be left unchanged. Bit 0 of this value corresponds to I/O line 33, bit 1 corresponds to I/O line 48 and so on.

### Example

```
#include “tip675api.h”

int             FileDescriptor;
int             result;

/*
** write new output value:
** set I/O line 2 to OFF and I/O line 17 to ON,
** all other lines shall be unchanged.
*/
result = tip675writeMask(
              FileDescriptor,
              0x0000,           /* OutputValue 1..16           */
              0x0001,           /* OutputValue 17..32          */
              0x0000,           /* OutputValue 33..48          */
              0x0002,           /* Mask 1..16                  */
              0x0001,           /* Mask 17..32                 */
              0x0000);          /* Mask 33..48                 */
if (result == EOK)
{
     /* OK */
}
else
{
     /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

All error codes a standard error codes set by the I/O system.

### 4.2.5 tip675configDir()

#### Name

tip675configDir() – set I/O direction.

#### Synopsis

```
int tip675configDir
(
     int                  FileDescriptor,
     unsigned short       Direction_1_16,
     unsigned short       Direction_17_32,
     unsigned short       Direction_33_48
);
```

#### Description

This function configures the direction of the I/O lines for the specified device.

#### Parameters

*FileDescriptor*

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

*Direction_1_16*

This value specifies the new direction for I/O lines 1...16. Bit 0 of this value corresponds to I/O line 1, bit 1 corresponds to I/O line 2 and so on. A set bit (1) will configure the corresponding I/O line for output. A cleared bit (0) configures the I/O line for input.

*Direction_17_32*

This value specifies the new direction for I/O lines 17...32. Bit 0 of this value corresponds to I/O line 17, bit 1 corresponds to I/O line 18 and so on. A set bit (1) will configure the corresponding I/O line for output. A cleared bit (0) configures the I/O line for input.

*Direction_33_48*

This value specifies the new direction for I/O lines 33...48. Bit 0 of this value corresponds to I/O line 33, bit 1 corresponds to I/O line 34 and so on. A set bit (1) will configure the corresponding I/O line for output. A cleared bit (0) configures the I/O line for input.

### Example

```
#include “tip675api.h”

int             FileDescriptor;
int             result;

/*
** configure I/O line 1..26 for output and 27..48 for input
*/
result = tip675configDir(
              FileDescriptor,
              0xFFFF,             /* Direction 1..16               */
              0x03FF,             /* Direction 17..32              */
              0x0000);            /* Direction 33..48              */
if (result == EOK)
{
     /* OK */
}
else
{
     /* handle error */
}
```

### RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

### ERROR CODES

All error codes a standard error codes set by the I/O system.

### 4.2.6 tip675outputLinesSet()

#### Name

tip675outputLinesSet() – set I/O lines

#### Synopsis

```
int tip675outputLinesSet
(
     int                  FileDescriptor,
     unsigned short       Mask_1_16,
     unsigned short       Mask_17_32,
     unsigned short       Mask_33_48
);
```

#### Description

This function sets specified I/O lines.

#### Parameters

*FileDescriptor*

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

*Mask_1_16*

This value specifies which of the I/O lines 1...16 shall be set. Bit 0 of this value corresponds to I/O line 1, bit 1 corresponds to I/O line 2 and so on. A set bit (1) will set the corresponding I/O line, an unset bit (0) will not change the output state.

*Mask_17_32*

This value specifies which of the I/O lines 17...32 shall be set. Bit 0 of this value corresponds to I/O line 17, bit 1 corresponds to I/O line 18 and so on. A set bit (1) will set the corresponding I/O line, an unset bit (0) will not change the output state.

*Mask_33_48*

This value specifies which of the I/O lines 33...48 shall be set. Bit 0 of this value corresponds to I/O line 33, bit 1 corresponds to I/O line 34 and so on. A set bit (1) will set the corresponding I/O line, an unset bit (0) will not change the output state.

## Example

```
#include "tip675api.h"

int             FileDescriptor;
int             result;

/*
** set I/O line 41..48
*/
result = tip675outputLinesSet(
                FileDescriptor,
                0x0000,           /* Mask 1..16              */
                0x0000,           /* Mask 17..32             */
                0xFF00);          /* Mask 33..48             */
if (result == EOK)
{
     /* OK */
}
else
{
     /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

All error codes a standard error codes set by the I/O system.

## 4.2.7 tip675outputLinesClear()

### Name

tip675outputLinesSet() – clear I/O lines

### Synopsis

```
int tip675write
(
     int                   FileDescriptor,
     unsigned short        Mask_1_16,
     unsigned short        Mask_17_32,
     unsigned short        Mask_33_48
);
```

### Description

This function clears specified I/O lines.

### Parameters

*FileDescriptor*

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

*Mask_1_16*

This value specifies which I/O lines 1...16 shall be cleared. Bit 0 of this value corresponds to I/O line 1, bit 1 corresponds to I/O line 2 and so on. A set bit (1) will clear the corresponding I/O line. An unset bit (0) will not change the output state.

*Mask_17_32*

This value specifies which I/O lines 17...32 shall be cleared. Bit 0 of this value corresponds to I/O line 17, bit 1 corresponds to I/O line 18 and so on. A set bit (1) will clear the corresponding I/O line. An unset bit (0) will not change the output state.

*Mask_33_48*

This value specifies which I/O lines 33...48 shall be cleared. Bit 0 of this value corresponds to I/O line 33, bit 1 corresponds to I/O line 34 and so on. A set bit will clear (1) the corresponding I/O line. An unset bit (0) will not change the output state.

## Example

```
#include “tip675api.h”

int             FileDescriptor;
int             result;

/*
** clear I/O line 5..16 and 45..48
*/
result = tip675outputLinesClear (
              FileDescriptor,
              0xFFF0,            /* Mask 1..16              */
              0x0000,            /* Mask 17..32             */
              0xF000);           /* Mask 33..48             */
if (result == EOK)
{
     /* OK */
}
else
{
     /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

All error codes a standard error codes set by the I/O system.

### 4.2.8 tip675setOutputLine()

#### Name

tip675setOutputLine() – set a single I/O line

#### Synopsis

```
int tip675setOutputLine
(
     int                  FileDescriptor,
     int                  outputLine
);
```

#### Description

This function sets a single specified I/O line.

#### Parameters

*FileDescriptor*

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

*outputLine*

This value specifies the I/O line which shall be set. Valid values are between 1 and 48.

#### Example

```
#include “tip675api.h”

int            FileDescriptor;
int            result;

/*
** set I/O line 41
*/
result = tip675setOutputLine(
                    FileDescriptor,
                    41);

…
```

```
…

if (result == EOK)
{
    /* OK */
}
else
{
    /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

| | |
|---|---|
| EINVAL | Invalid I/O line specified. |

All other error codes a standard error codes set by the I/O system.

### 4.2.9 tip675clearOutputLine()

#### Name

tip675clearOutputLine() – clear a single I/O line

#### Synopsis

```
int tip675clearOutputLine
(
     int                 FileDescriptor,
     int                 outputLine
);
```

#### Description

This function clears a single specified I/O line.

#### Parameters

*FileDescriptor*

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

*outputLine*

This value specifies the I/O line which shall be cleared. Valid values are between 1 and 48.

#### Example

```
#include “tip675api.h”

int            FileDescriptor;
int            result;

/*
** clear I/O line 41
*/
result = tip675clearOutputLine(
                    FileDescriptor,
                    41);

…
```

…

```
if (result == EOK)
{
     /* OK */
}
else
{
     /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

| | |
|---|---|
| EINVAL | Invalid I/O line specified. |

All other error codes a standard error codes set by the I/O system.

## 4.2.10 tip675configureExtClk()

### Name

tip675configureExtClk() – Configures the function of the external clock.

### Synopsis

```
int tip675configureExtClk
(
    int          FileDescriptor,
    int          mode
);
```

### Description

This function configures the function of the external clock. The external clock may be used to synchronize input data of different TIP675 or to use an external update trigger.

### Parameters

*FileDescriptor*

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

*mode*

This value specifies the function of the external clock. The following values are allowed:

| Value | Description |
|---|---|
| TIP675_EXT_CLK_DISABLED | This value will disable the external clock. Output values will appear at the I/O pin immediately. Changes on input lines will also be available in the input register immediately. |
| TIP675_EXT_CLK_POS_EDGE | This value specifies that the positive edge of the external clock line will update the state of the output lines and the content of the input register. |
| TIP675_EXT_CLK_NEG_EDGE | This value specifies that the negative edge of the external clock line will update the state of the output lines and the content of the input register. |

## Example

```
#include “tip675api.h”
int              FileDescriptor;
int              result;

/*
** use the positive egde of the external clock
*/
result = tip675configureExtClk(
              FileDescriptor,
              TIP675_EXT_CLK_POS_EDGE);
if (result == EOK)
{
     /* OK */
}
else
{
     /* handle error */
}
```

## RETURNS

On success, EOK is returned. In the case of an error, the appropriate error code is returned by the function (not in errno!).

## ERROR CODES

| | |
|---|---|
| EINVAL | Invalid external clock mode specified. |

All other error codes a standard error codes set by the I/O system.

# 5 Appendix

## 5.1 Initializing TIP675 Devices

The following steps are an example how to initialize a TIP675 device:

The example shows a setup where the I/O lines 1…32 are used for input and 33…48 are used for output. The startup value of the output lines is “0”.

1) Start the IPAC-Carrier driver. This will automatically find and create the TIP675 devices.
```
ipac_class &
```

2) Open the TIP675 device and get a device handle.
```
devHandle = tip675open(“/dev/tip675_x”);
```

3) Configure external clock mode
```
result = tip675configureExtClk(devHandle, TIP675_EXT_CLK_DISABLED);
```

4) Configure the default output value (if needed)
```
result = tip675write (devHandle, 0x0000, 0x0000, 0x0000);
```

5) Configure direction of the I/O lines
```
result = tip675configDir(devHandle, 0x0000, 0x0000, 0xFFFF);
```

Now the device should be ready for your application.

## 5.2 Example Applications

### 5.2.1 tip675exa (example/exampleapp)

This example allows to call and to test all driver functions of the TIP675. All functions can be individually selected and executed.

This example does not use the API, instead it directly calls the QNX-Neutrino I/O interface functions (*open()*, *close()*, and *devctl()*).

### 5.2.2 tip675read (example/read)

This example simply reads out the current value of the input register.

This example uses the API-functions. It shows how the API should be used.