

TIP672-SW-82

Linux Device Driver

24 Differential I/O Lines with Interrupts

Version 1.3<.x

User Manual

Issue 1.3.0

February 2009

TEWS TECHNOLOGIES GmbH

Am Bahnhof 7
25469 Halstenbek, Germany
www.tews.com

Phone: +49 (0) 4101 4058 0
Fax: +49 (0) 4101 4058 19
e-mail: info@tews.com

TEWS TECHNOLOGIES LLC

9190 Double Diamond Parkway,
Suite 127, Reno, NV 89521, USA
www.tews.com

Phone: +1 (775) 850 5830
Fax: +1 (775) 201 0347
e-mail: usasales@tews.com

TIP672-SW-82

Linux Device Driver

24 Differential I/O Lines with Interrupts

Supported Modules:

TIP672-10

This document contains information, which is proprietary to TEWS TECHNOLOGIES GmbH. Any reproduction without written permission is forbidden.

TEWS TECHNOLOGIES GmbH has made any effort to ensure that this manual is accurate and complete. However TEWS TECHNOLOGIES GmbH reserves the right to change the product described in this document at any time without notice.

TEWS TECHNOLOGIES GmbH is not liable for any damage arising out of the application or use of the device described herein.

©2009 by TEWS TECHNOLOGIES GmbH

Issue	Description	Date
1.0	First Issue	December 1, 2002
1.1	Support for DEVFS and SMP	February 16, 2004
1.2.0	Linux Kernel 2.6.x Revision	July 12, 2005
1.3.0	General Revision, new File List	February 4, 2009

Table of Contents

1	INTRODUCTION.....	4
	1.1 Device Driver	4
	1.2 IPAC Carrier Driver	4
2	INSTALLATION.....	5
	2.1 Build and install the device driver.....	5
	2.2 Uninstall the device driver	6
	2.3 Install device driver into the running kernel	6
	2.4 Remove device driver from the running kernel	7
	2.5 Change Major Device Number	7
3	DEVICE INPUT/OUTPUT FUNCTIONS	8
	3.1 open()	8
	3.2 close().....	10
	3.3 read()	11
	3.4 write()	13
	3.5 ioctl()	15
	3.5.1 T672_IOCSET_SET_DIR.....	17
	3.5.2 T672_IOCSET_ENA_EXCLK.....	18
	3.5.3 T672_IOCSET_DIS_EXCLK	19
	3.5.4 T672_IOCSET_WAIT_TRANS.....	20
4	DEBUGGING	22

1 Introduction

1.1 Device Driver

The TIP672-SW-82 Linux device driver allows the operation of the TIP672 IPAC module conforming to the Linux I/O system specification. This includes a device-independent basic I/O interface with *open()*, *close()*, *read()*, *write()* and *ioctl()* functions.

Because the TIP672 device driver is stacked on the TEWS TECHNOLOGIES IPAC carrier driver, it's necessary to install also the appropriate IPAC carrier driver. Please refer to the IPAC carrier driver user manual for further information.

The TIP672-SW-82 device driver supports the following features:

- Reading from input register
- Setting output register
- Programming direction of every I/O line
- configure simultaneous update feature
- waiting for a transition at a single input line or a group of input lines (OR'ed)

The TIP672-SW-82 device driver supports the modules listed below:

TIP672-10	24 differential I/O lines	(IndustryPack)
-----------	---------------------------	----------------

To get more information about the features and use of TIP672 devices it is recommended to read the manuals listed below.

- TIP672 User manual
- TIP672 Engineering Manual
- CARRIER-SW-82 IPAC Carrier 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-82 is part of this TIP672-SW-82 distribution. It is located in directory CARRIER-SW-82 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-82 User Manual for a detailed description how to install and setup the CARRIER-SW-82 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 ‘.\TIP672-SW-82\’:

TIP672-SW-82-SRC.tar.gz	GZIP compressed archive with driver source code
TIP672-SW-82-1.3.0.pdf	PDF copy of this manual
ChangeLog.txt	Release history
Release.txt	Release information

The GZIP compressed archive TIP672-SW-82-SRC.tar.gz contains the following files and directories:

Directory path ‘./tip672/’:

tip672.c	Driver source code
tip672def.h	Driver include file
tip672.h	Driver include file for application program
makenode	Script to create device nodes on the file system
Makefile	Device driver make file
example/tip672exa.c	Example application
example/Makefile	Example application make file
include/config.h	Driver independent library header file
include/tpmodule.h	Kernel independent library header file
include/tpmodule.c	Kernel independent library source code file

In order to perform an installation, extract all files of the archive TIP672-SW-82-SRC.tar.gz to the desired target directory. The command ‘tar -xzvf TIP672-SW-72-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 file *ipac_carrier.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-82* on the separate distribution media.

2.1 Build and install the device driver

- Login as *root*
- Change to the target directory
- To create and install the driver in the module directory */lib/modules/<version>/misc* enter:

make install

For Linux kernel 2.6.x, there may be compiler warnings claiming some undefined *ipac_ symbols. These warnings are caused by the IPAC carrier driver, which is unknown during compilation of this TIP driver. The warnings can be ignored.**

- Also after the first build we have to execute *depmod* to create a new dependency description for loadable kernel modules. This dependency file is later used by *modprobe* to automatically load the correct IPAC carrier driver modules.

```
# depmod -aq
```

2.2 Uninstall the device driver

- Login as *root*
- Change to the target directory
- To remove the driver from the module directory */lib/modules/<version>/misc* enter:

```
# make uninstall
```

- Update kernel module dependency description file

```
# depmod -aq
```

2.3 Install device driver into the running kernel

- To load the device driver into the running kernel, login as *root* and execute the following commands:

```
# modprobe tip672drv
```

- After the first build or if you are using dynamic major device allocation it's necessary to create new device nodes on the file system. Please execute the script file *makenode* to do this. If your kernel has enabled a device file system (*devfs* or *sysfs* with *udev*) then you have to skip running the *makenode* script. Instead of creating device nodes from the script the driver itself takes creating and destroying of device nodes in its responsibility.

```
# sh makenode
```

On success the device driver will create a minor device for each TIP672 module found. The first TIP672 can be accessed with device node */dev/tip672_0*, the second TIP672 with device node */dev/tip672_1*, the third TIP672 with device node */dev/tip672_2* and so on.

The allocation of device nodes to physical TIP672 modules depends on the search order of the IPAC carrier driver. Please refer to the IPAC carrier user manual.

Loading of the TIP672 device driver will only work if kernel KMOD support is installed, necessary carrier board drivers already installed and the kernel dependency file is up to date. If KMOD support isn't available you have to build either a new kernel with KMOD installed or you have to install the IPAC carrier kernel modules manually in the correct order (please refer to the IPAC carrier driver user manual).

2.4 Remove device driver from the running kernel

- To remove the device driver from the running kernel login as root and execute the following command:

```
# modprobe tip672drv -r
```

If your kernel has enabled dynamic device file system, all /dev/tip672_x nodes will be automatically removed from your file system after this.

Be sure that the driver isn't opened by any application program. If opened you will get the response "*tip672drv: Device or resource busy*" and the driver will still remain in the system until you close all opened files and execute *modprobe -r* again.

2.5 Change Major Device Number

The TIP672 driver use dynamic allocation of major device numbers by default. If this isn't suitable for the application it's possible to define a major number for the driver. If the kernel has enabled devfs the driver will not use the symbol TIP672_MAJOR.

To change the major number edit the file tip672drv.c, change the following symbol to appropriate value and enter **make install** to create a new driver.

TIP672_MAJOR

Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TIP672_MAJOR      122
```

3 Device Input/Output functions

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

3.1 open()

NAME

open() - open a file descriptor

SYNOPSIS

```
#include <fcntl.h>

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

DESCRIPTION

This function creates and returns a new file descriptor for the file named by *filename*. The *flags* argument controls how the file is to be opened. This is a bit mask; you create the value by the bitwise OR of the appropriate parameters (using the | operator in C). See also the GNU C Library documentation for more information about the open function and open flags.

EXAMPLE

```
int fd;

fd = open("/dev/tip672_0", O_RDWR);
if (fd == -1)
{
    /* handle error condition */
}
```

RETURNS

The normal return value from `open` 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.

ERRORS

<code>ENODEV</code>	The requested minor device does not exist.
---------------------	--

This is the only error code returned by the driver, other codes may be returned by the I/O system during open.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.2 close()

NAME

close() – close a file descriptor

SYNOPSIS

```
#include <unistd.h>
```

```
int close  
(  
    int    filedes  
)
```

DESCRIPTION

This function closes the file descriptor *filedes*.

EXAMPLE

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

RETURNS

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

ERRORS

ENODEV	The requested minor device does not exist.
--------	--

This is the only error code returned by the driver, other codes may be returned by the I/O system during close. For more information about close error codes, see the *GNU C Library description – Low-Level Input/Output*.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.3 read()

NAME

read() – read from a device

SYNOPSIS

```
#include <unistd.h>
```

```
ssize_t read  
(  
    int          fildes,  
    void         *buffer,  
    size_t       size  
)
```

DESCRIPTION

This function reads the input register of the TIP672 associated with the file descriptor *fildes* into an unsigned long variable pointed by *buffer*. The argument *size* specifies the length of the buffer and must be set to the length of an unsigned long.

The unsigned long variable returns the content of the line input register in the lower 24 bit, where bit 2^0 corresponds to INPUT 1 and bit 2^{23} corresponds to INPUT 24. The upper 8 bit of the unsigned long variable are always read as 0.

EXAMPLE

```
int          fd;  
ssize_t     num_bytes;  
unsigned long value;  
  
num_bytes = read(fd, &value, sizeof(unsigned long));  
if (num_bytes != sizeof(unsigned long))  
{  
    // process error;  
}
```

RETURNS

On success read returns the number of byte read (always size of unsigned long). In the case of an error, a value of -1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL This error code is returned if the size of the buffer is wrong.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.4 write()

NAME

write() – write to a device

SYNOPSIS

```
#include <unistd.h>
```

```
ssize_t write  
(  
    int          filedes,  
    void         *buffer,  
    size_t       size  
)
```

DESCRIPTION

This function sets the output register of the TIP672 associated with the file descriptor *filedes*. The output value is specified in an unsigned long variable pointed by *buffer*. The argument *size* specifies the length of the buffer and must be set to the length of an unsigned long.

The lower 24 bit of the unsigned long variable corresponds to the 24 output lines.

EXAMPLE

```
int          fd;  
ssize_t      num_bytes;  
unsigned long value;  
  
/* set OUTPUT 1,15 and 24 to logic high */  
value = 0x804001;  
num_bytes = write(fd, &value, sizeof(unsigned long));  
if (num_bytes != sizeof(unsigned long))  
{  
    // process error;  
}
```

RETURNS

On success write returns the size of bytes written (always the size of an unsigned long). In the case of an error, a value of `-1` is returned. The global variable `errno` contains the detailed error code.

ERRORS

`EINVAL` This error code is returned if the size of the buffer is wrong.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.5 ioctl()

NAME

ioctl() – device control functions

SYNOPSIS

```
#include <sys/ioctl.h>
```

```
int ioctl
(
    int          filedes,
    int          request
    [, void      *argp]
)
```

DESCRIPTION

The **ioctl** function sends a control code directly to a device, specified by *filedes*, causing the corresponding device to perform the requested operation.

The argument *request* specifies the control code for the operation. The optional argument *argp* depends on the selected request and is described for each request in detail later in this chapter.

The following ioctl codes are defined in *TIP672.h*:

ioctl code	Meaning
T672_IOCS_SET_DIR	Set direction of I/O lines
T672_IOCS_ENA_EXCLK	Enable simultaneous update feature
T672_IOC_DIS_EXCLK	Disable simultaneous update feature
T672_IOCX_WAIT_TRANS	Waiting for a transition at a single input line or a group of input lines (OR'ed)

See behind for more detailed information on each control code.

To use these TIP672 specific control codes the header file TIP672.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 <code>ioctl</code> function is unknown. Please check the argument <i>request</i> .
--------	---

Other function dependent error codes will be described for each `ioctl` code separately. Note, the TIP672 driver always returns standard Linux error codes.

SEE ALSO

`ioctl` man pages

3.5.1 T672_IOCS_SET_DIR

NAME

T672_IOCS_SET_DIR - Set direction of I/O lines

DESCRIPTION

This function allows setting each of the 24 I/O line individually as input or output. Bit 0 of the direction mask corresponds to line 1, bit 1 to line 2 and so on. To set a line to be input, set the corresponding bit in the mask to 1.

A pointer to the direction mask (unsigned long) is passed by the parameter *argp* to the driver.

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

EXAMPLE

```
#include <tip672.h>

int          fd;
int          result;
unsigned long direction;

// Set line 1..12 as input and line 13..24 as output
direction = 0x000FFF;

result = ioctl(fd, T672_IOCS_SET_DIR, &direction);
if (result < 0)
{
    /* handle ioctl error */
}
```

ERRORS

EFAULT Invalid pointer to the direction mask. Please check the argument *argp*.

SEE ALSO

ioctl man pages

3.5.2 T672_IOCS_ENA_EXCLK

NAME

T672_IOCS_ENA_EXCLK - Enable simultaneous update feature

DESCRIPTION

This function enables the simultaneous update feature of the TIP672. The argument *argp* passes a pointer to an unsigned long value to the driver. Setting this variable to a 0 will cause the inputs and outputs to be latched on the rising edge of the external clock, while setting the value to 1 it will latch the inputs and outputs on the falling edge.

After driver startup the simultaneous update feature is disabled.

EXAMPLE

```
#include <tip672.h>

int          fd;
int          result;
unsigned long value;

// Enable simultaneous update on falling edge
value = 1;

result = ioctl(fd, T672_IOCS_ENA_EXCLK, &value);
if (result < 0)
{
    /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer. Please check the argument <i>argp</i> .
--------	--

SEE ALSO

ioctl man pages

3.5.3 T672_IOCS_DIS_EXCLK

NAME

T672_IOCS_DIS_EXCLK - Disable simultaneous update feature

DESCRIPTION

This function disables the simultaneous update feature of the TIP672. The argument *argp* is not needed and can be omitted.

After driver startup the simultaneous update feature is disabled.

EXAMPLE

```
#include <tip672.h>

int      fd;
int      result;

result = ioctl(fd, T672_IOCS_DIS_EXCLK);
if (result < 0)
{
    /* handle ioctl error */
}
```

SEE ALSO

ioctl man pages

3.5.4 T672_IOCX_WAIT_TRANS

NAME

T672_IOCX_WAIT_TRANS - Waiting for a transition on a specified input line

DESCRIPTION

This function will block until a specified transition on a selected input line occurs or the maximum allowed time elapses. If more than one input line is selected, at least one specified transition must occur (logical OR) to finish this function. On success this function returns the content of the input register to the caller.

A pointer to the callers parameter buffer (*T672_TRANS_PAR*) is passed by the parameter *argp* to the driver.

```
typedef struct
{
    unsigned long    select_mask;
    unsigned long    polarity_mask;
    long            timeout;
    unsigned long    status_mask;
    unsigned long    input_reg
} T672_TRANS_PAR;
```

select_mask

This parameter contains a bit mask to select a certain bit position or a group of bits for an input transition detection. Bits 2^0 to 2^{23} correspond to INPUT 1 to INPUT 24. A certain input line can be selected by setting the corresponding bit position to 1. A certain input line can be occupied only once. If you try to select an input line which is already occupied by other requests you will get the error code *EBADSLT*.

polarity_mask

This parameter defines the active transition for the selected input lines. Bits 2^0 ... 2^{23} correspond to INPUT 1 ... INPUT 24. To generate an event for a positive transition (0->1) set the corresponding bit to 1. For a negative transition (1->0) set the corresponding bit to 0. Only selected bits (*select_mask*) are relevant for the polarity setting.

timeout

Specifies the amount of time (in ticks) the caller is willing to wait for the occurrence of the requested transition. A value of 0 means wait indefinitely.

status_mask

This parameter returns the content of the TIP672 interrupt status register and can be used to determine the source of the event (interrupt). This is useful if more than one bit is selected. If the selected transition has occurred, the corresponding bit position contains a 1.

input_reg

This parameter returns the contents of the line input register after the requested event has occurred. Please note that the input register isn't latched with the interrupt and depending on the interrupt latency the read to the input register is delayed.

EXAMPLE

```
#include <tip672.h>

int          fd;
int          result;
T672_TRANS_PAR  par;

// Wait for a (0->1) transition at INPUT 24 OR a (1->0) transition
// at INPUT 1.
// The request times out after 1000 ticks.
par.select_mask    = 0x800001;
par.polarity_mask  = 0x800000;
par.timeout        = 1000;

result = ioctl(fd, T672_IOCX_WAIT_TRANS, &par);
if (result < 0)
{
    /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer. Please check the argument argp.
EBUSY	The maximum number of concurrent read requests was exceeded. Increase the value of MAX_REQUESTS in tip672def.h.
EBADSLT	At least one requested input line was already occupied by other T672_IOCX_WAIT_TRANS requests. Please check the select_mask of other ioctl function calls.
ETIME	The allowed time to finish the read request is elapsed.
EINTR	Interrupted function call; an asynchronous signal occurred and prevented completion of the call. When this happens, you should try the call again.

SEE ALSO

ioctl man pages

4 Debugging

For debugging output see tip672drv.c. You will find the two following symbols:

```
#undef TIP672_DEBUG_INTR
```

```
#undef TIP672_DEBUG_VIEW
```

To enable a debug output replace “undef” with “define”.

The TIP672_DEBUG_INTR symbol controls debugging output from the ISR.

```
TIP672 : interrupt entry
```

```
TIP672 : IACK[0] vector = 0005
```

The TIP672_DEBUG_VIEW symbol controls debugging output from the remaining part of the driver.

```
TIP672 - 24 Differential I/O Lines with Interrupts version 1.3.0 (2009-02-04)
```

```
TIP672 : Probe new TIP672 mounted on <TEWS TECHNOLOGIES - (Compact)PCI  
IPAC Carrier> at slot B
```

```
TIP672 : Create minor node /dev/tip672_0 (devfs).
```

```
TIP672 : IP I/O Memory Space
```

```
00000000 : 00 00 00 00 8F DF 00 E1 FF FF 00 FF 00 00 FF FF
```

```
00000010 : 00 05 00 00 00 00 00 00 00 00 00 00 00 00
```

```
TIP672 : Remove /dev/tip672_0 node (devfs).
```