The Embedded I/O Company

TEWS TECHNOLOGIES

TIP600-SW-82

Linux Device Driver

16 Channel Digital Inputs

Version 1.2.x

User Manual

Issue 1.2.3

September 2009

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

TIP600-SW-82

Linux Device Driver

16 Channel Digital Inputs

Supported Modules:
TIP600

Issue	Description	Date
1.0	First Issue	March 8, 2002
1.1	Support for IPAC CARRIER DRIVER, DEVFS and SMP	February 9, 2004
1.2.0	Linux Kernel 2.6.x Revision	May 4, 2005
1.2.1	General revision	February 11, 2008
1.2.2	Carrier Driver description added	July 17, 2008
1.2.3	Address TEWS LLC removed	September 14, 2009

Table of Contents

1 Introduction

1.1 Device Driver

The TIP600-SW-82 Linux device driver allows the operation of a TIP600 IPAC module on Linux operating systems.

Because the TIP600 device driver is stacked on the TEWS TECHNOLOGIES IPAC Carrier Driver, it is necessary to install also the appropriate IPAC carrier driver. Please refer to the IPAC Carrier Driver User Manual for further information.

The TIP600-SW-82 device driver supports the following features:

- reading the actual port value
- configure port polarity
- waiting for selectable input events (match, high-, low-, any-transition on the input port)
- setting programmable debounce time for digital filtering

The TIP600-SW-82 device driver supports the modules listed below:

TIP600	16 Channel Digital Inputs	(IndustryPack ®)

To get more information about the features and use of TIP600 devices it is recommended to read the manuals listed below.

TIP600 User manual

TIP600 Engineering Manual

IPAC Carrier Driver 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 TIP600-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 'TIP600-SW-82':

TIP600-SW-82-SRC.tar.gz	GZIP compressed archive with driver source code
TIP600-SW-82-1.2.3.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 TIP600-SW-82-SRC.tar.gz contains the following files and directories:

Directory path 'tip600':

tip600.c	Driver source code
tip600def.h	Driver include file
tip600.h	Driver include file for application program
makenode	Script to create device nodes on the file system
Makefile	Device driver make file
example/tip600exa.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 TIP600-SW-82-SRC.tar.gz to the desired target directory. The command 'tar -xzvf TIP600-SW-82-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 tip600drv

- After the first build or if you are using dynamic major device allocation it is 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 TIP600 module found. The first TIP600 can be accessed with device node /dev/tip600_0, the second TIP600 with device node /dev/tip600_1, the third TIP600 with device node /dev/tip600_2 and so on.

The allocation of device nodes to physical TIP600 modules depends on the search order of the IPAC carrier driver. Please refer to the IPAC carrier user manual.

Loading of the TIP600 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 tip600drv –r

If your kernel has enabled devfs or sysfs (udev), all /dev/tip600_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 "*tip600drv: 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 TIP600 driver uses dynamic allocation of major device numbers per default. If this isn't suitable for the application it's possible to define a major number for the driver.

To change the major number edit the file tip600.c, change the following symbol to appropriate value and enter make install to create a new driver.

TIP600_MAJOR	Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TIP600_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

The open 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/tip600_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

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 open. For more information about open error codes, see the *GNU C Library description – Low-Level Input/Output*.

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

The close 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 filedes, void *buffer, size_t size*)

DESCRIPTION

The read function reads to current state of the input lines of the specified device and returns it in a read buffer to the caller.

A pointer to the callers read buffer (*T600_READ_BUFFER*) and the size of this structure is passed by the parameters *buffer* and *size* to the device.

```
typedef struct
{
        unsigned short      value;
} T600_READ_BUFFER;
```

value

This parameter receives the current state of all 16 input lines.

EXAMPLE

```
int fd;
ssize_t num_bytes;
T600_READ_BUFFER read_buf;

/*
**  Send read request to the device driver
*/
num_bytes = read(fd, &read_buf, sizeof(read_buf));

/*
**  Check the result of the last device I/O operation
*/
if (num_bytes > 0){
     printf("\nRead input lines successful\n");
     printf("      Port: %04Xh\n", read_buf.value);
} else {
     printf("\nRead input lines failed -->  Error = %d\n", errno );
     PrintErrorMessage();
}
```

RETURNS

On success read returns the size of the structure T600_READ_BUFFER. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL	Invalid argument. This error code is returned if the size of the read buffer is too small or if the gain or channel parameter out of range.
ETIME	The conversion was not completed within 20 microseconds. The hardware seems to be faulty or the device mapping is incorrect.
EFAULT	Invalid pointer to the read buffer.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.4 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 *tip600.h*:

Symbol	Meaning
T600_IOCT_SET_DEBOUNCE_TIME	Setup or disable programmable debounce time
T600_IOCX_EVENT_READ	Read port after a specified input event occurred

See behind for more detailed information on each control code.

To use these TIP600 specific control codes the header file tip600.h must be included in the application.

RETURNS

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

ERRORS

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

Other function dependant error codes will be described for each ioctl code separately. Note, the TIP600 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.4.1 T600_IOCT_SET_DEBOUNCE_TIME

NAME

T600_IOCT_SET_DEBOUNCE_TIME - Setup or disable programmable debounce time

DESCRIPTION

This ioctl function sets the programmable debounce time for digital filtering of the input lines. A value less than US_PER_TICK (for more details see tip600.h) disables programmable debounce time and enables default TIP600 debounce period of 9 µs. The debounce time value is passed (call-by-value) by the parameter *argp* to the driver.

EXAMPLE

```
#include <tip600.h>

int fd;
int result;

/*
**  Tell the driver the new debounce time
*/
result = ioctl(fd, T600_IOCT_SET_DEBOUNCE_TIME, 123);

/*
**  Check the result of the last device I/O control operation
*/
if (result >= 0){
     printf("\IOCTL successfull.\n");
} else {
     printf("\nIOCTL failed --> Error = %d\n", errno );
     PrintErrorMessage();
}
```

ERRORS

EFAULT	Invalid pointer to the read buffer. Invalid channel number specified.

3.4.2 T600_IOCX_EVENT_READ

NAME

T600_IOCX_EVENT_READ - Read port after a specified input event occurred

DESCRIPTION

The ioctl function reads the content of the input port after a specified event occurred. Possible events are rising, falling or any edge of a specified input bit or a pattern match of masked input bits. A pointer to the callers event read buffer (*T600_EVRD_BUFFER*) is passed by the argument *argp* to the driver.

```
typedef struct
{
        unsigned short      value;
        unsigned char       mask;
        unsigned char       match;
        unsigned char       mode;
        unsigned long       timeout;
} T600_EVRD_BUFFER, *PT600_EVRD_BUFFER;
```

value

Receives the value of the corresponding port.

mask

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

match

Specifies a pattern that must match to the contents of the input port. Only the bit positions specified by *mask* must match to the input port.

mode

Specifies the “event” mode for this read request. Following values are possible:

Value	Description
T600_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 mask. A 1 value in mask means than the input bit value “must-match” identically to the corresponding bit in the match parameter. *Note that this mode is not recommended for use anymore, runtime problems may hide matching events*.
T600_HIGH_TR	The driver reads the input port 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 occurs.
T600_LOW_TR	The driver reads the input port 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 occurs.
T600_ANY_TR	The driver reads the input port 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 occurs.

timeout

Specifies the amount of time (in ticks) the caller is willing to wait for the specified event to occur. A value of 0 means wait indefinitely.

There is a delay between the specified event and the input value read which is based on the system and OS dependent interrupt latency.

EXAMPLE

```
#include <tip600.h>

int fd;
int result;
T600_EVRD_BUFFER evBuf;

/*
**  Read the input port after a high-transition at
**  input line 8 occured
*/
evBuf.mode = T600_HIGH_TR;
evBuf.mask = 1 << 7;        /* high-transition at bit 7 */
evBuf.timeout = 100;        /* ticks */

result = ioctl(fd, T600_IOCX_EVENT_READ, &evBuf);

if (result >= 0) {
     printf("Port value: %04Xh\n", evBuf.value);
} else {
     /* handle read error */
}
```

ERRORS

EFAULT	Invalid pointer to the read buffer.
EBUSY	The maximum number of concurrent read requests is exceeded. Increase the value of MAX_REQUESTS in tip600def.h.
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.