The Embedded I/O Company

TEWS TECHNOLOGIES

TIP605-SW-82

Linux Device Driver

16 Digital Inputs

Version 1.0.x

User Manual

Issue 1.0.0

March 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

TIP605-SW-82

Linux Device Driver

16 Digital inputs

Supported Modules:
TIP605

Issue	Description	Date
1.0.0	First Issue	March 3, 2009

Table of Contents

1 Introduction

1.1 Device Driver

The TIP605-SW-82 Linux device driver allows the operation of TIP605 IPAC modules on Linux operating systems.

Because the TIP605 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 TIP605 device driver includes the following features:

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

The TIP605-SW-82 supports the modules listed below:

TIP605-10	16 Digital Input (Optically Isolated)	(IndustryPack®)
TIP605-50	16 Digital Input (Optically Isolated) *1	(IndustryPack®)
TIP605-51	16 Digital Input (Optically Isolated) *1	(IndustryPack®)
TIP605-52	16 Digital Input (Optically Isolated) *1	(IndustryPack®)

*1 Special User Input Wiring

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

TIP605 User manual

TIP605 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 TIP605-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

The directory TIP605-SW-82 on the distribution media contains the following files:

TIP605-SW-82-1.0.0.pdf	This manual in PDF format
TIP605-SW-82.tar.gz	GZIP compressed archive with driver source code
Release.txt	Release information
ChangeLog	Release History

The GZIP compressed archive TIP605-SW-82.tar.gz contains the following files and directories:

Directory path './tip605/':

tip605.c	Driver source code
tip605def.h	Driver include file
tip605.h	Driver include file for application program
makenode	Script to create device nodes on the file system
Makefile	Device driver make file
example/tip605exa.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 TIP605-SW-82.tar.gz to the desired target 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 tip605drv

- 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 the dynamic 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 TIP605 module found. The first TIP605 can be accessed with device node /dev/tip605_0, the second TIP605 with device node /dev/tip605_1 and so on.

The allocation of device nodes to physical TIP605 modules depends on the search order of the IPAC carrier driver. Please refer to the IPAC carrier user manual.

Loading of the TIP605 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 tip605drv –r

If your kernel has enabled a dynamic device file system all /dev/tip605_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 "*tip605drv: 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 TIP605 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 dynamic device file system the driver will not use the symbol TIP605_MAJOR.

To change the major number edit the file tip605.c, change the following symbol to appropriate value and enter **make install** to create a new driver.

TIP605_MAJOR	Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TIP605_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/tip605_0", O_RDWR);
if (fd < 0)
{
   /* 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 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 *tip605.h*:

Symbol	Meaning
TIP605_IOCG_READ	Read current state of Input Port
TIP605_IOCX_EVENT_READ	Read state of Input Port after a specified input event occurred
TIP605_IOCT_SET_DEBOUNCE_TIME	Setup or disable programmable debounce time

See behind for more detailed information on each control code.

To use these TIP605 specific control codes the header file tip605.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 TIP605 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.3.1 TIP605_IOCG_READ

NAME

TIP605_IOCG_READ – Read current state of input port

DESCRIPTION

This ioctl function reads the current state of the input port. A pointer to the caller's read buffer (*TIP605_RD_BUFFER*) is passed by the argument *argp* to the driver

```
typedef struct
{
        unsigned short          value;
} TIP605_RD_BUFFER;
```

value

Returns the current state of the input port.

EXAMPLE

```
#include "tip605.h"

int                 fd;
int                 result;
TIP605_RD_BUFFER    rdBuf;

/* Read current state of the input port */
result = ioctl(fd, TIP605_IOCG_READ, &rdBuf);
if (result >= 0)
{
  printf("\IOCTL successfull.\n");
  printf("      Port: %04Xh\n", rdBuf.value);

}
else
{
  printf("\nIOCTL failed --> Error = %d\n", errno );
}
```

3.3.2 TIP605_IOCX_EVENT_READ

NAME

TIP605_IOCX_EVENT_READ - Read state of input 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. A pointer to the caller's event read buffer (*TIP605_EVRD_BUFFER*) is passed by the argument *argp* to the driver.

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

```
typedef struct
{
        unsigned short          value;
        unsigned char           mask;
        unsigned char           mode;
        unsigned long           timeout;
} TIP605_EVRD_BUFFER, *PTIP605_EVRD_BUFFER;
```

value

Returns the value of the corresponding input port.

mask

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

mode

Specifies the "event" mode for this read request

TIP605_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.
TIP605_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.
TIP605_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 system ticks) the caller is willing to wait for the specified event to occur. A value of 0 means wait indefinitely.

EXAMPLE

```
#include <tip605.h>

int fd;
int result;
TIP605_EVRD_BUFFER evBuf;

/* Read the input port after a high-transition at */
/* input line 8 (bit 7) occurred */
evBuf.mode = TIP605_HIGH_TR;
evBuf.mask = 1 << 7;    /* high-transition at bit 7 */
evBuf.timeout = 1000;   /* ticks */

result = ioctl(fd, TIP605_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 tip605def.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.

3.3.3 TIP605_IOCT_SET_DEBOUNCE_TIME

NAME

TIP605_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 pointer to the caller's debounce buffer (*TIP605_DEB_BUFFER*) is passed by the argument *argp* to the driver

```
typedef struct
{
        unsigned char           value;
} TIP605_DEB_BUFFER;
```

value

Specifies the new debounce value. The debounce time is specified in steps of 1024µs, a debounce value of 0 will set the debounce time to 8us. The maximum debounce value is 255.

EXAMPLE

```
#include "tip605.h"

int                 fd;
int                 result;
TIP605_DEB_BUFFER   debBuf;

/* Tell the driver the new debounce time (~10ms) */
debBuf.value = 10; /* 8µs + 10 * 1024µs */
result = ioctl(fd, TIP605_IOCT_SET_DEBOUNCE_TIME, &debBuf);
if (result >= 0)
{
  printf("\IOCTL successfull.\n");
}
else
{
  printf("\nIOCTL failed --> Error = %d\n", errno );
}
```

4 Debugging

For debugging output see tip605.c. You will find the two following symbols:

```
#undef TIP605_DEBUG_INTR
#undef TIP605_DEBUG_VIEW
```

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

The TIP605_DEBUG_INTR symbol controls debugging output from the ISR.

```
TIP605  :  interrupt entry
```

The TIP605_DEBUG_VIEW symbol controls debugging output from the remaining part of the driver. You can retrieve these messages from the /proc file system using the following command:

 # cat /proc/kmsg

```
TIP605 - 16 Digital Inputs -  version 1.0.0 (2009-03-03)
TIP605  :  Probe new TIP605 mounted on <TEWS TECHNOLOGIES - TVME8240 VME
IPAC Carrier Driver> at slot G

TIP605  :  IP I/O Memory Space
00000000 : 00 00 00 01 00 00 00 00 00 00 00 00 00 58 00 0A
00000010 : 00 00 00 01 00 00 00 00 00 00 00 00 00 58 00 0A
VME :  carrier_request_irq() slot=6, vec=88, lev=5,6
```

…