The Embedded I/O Company

TEWS TECHNOLOGIES

TIP111-SW-82

Linux Device Driver

Motion Controller with Absolute Encoder Interface

Version 1.1.x

User Manual

Issue 1.1

September 2003

TEWS TECHNOLOGIES GmbH
Am Bahnhof 7
25469 Halstenbek / Germany
Phone: +49-(0)4101-4058-0
Fax: +49-(0)4101-4058-19
e-mail: info@tews.com
www.tews.com

TEWS TECHNOLOGIES LLC
1 E. Liberty Street, Sixth Floor
Reno, Nevada 89504 / USA
Phone: +1 (775) 686 6077
Fax: +1 (775) 686 6024
e-mail: usasales@tews.com
www.tews.com

TIP111-SW-82

Motion Controller with Absolute Encoder Interface

Linux Device Driver

Issue	Description	Date
1.0	First Issue	February 21, 2003
1.1	Support for DEVFS and SMP	September 16, 2003

Table of Contents

1 Introduction

The TIP111-SW-82 Linux device driver allows the operation of a TIP111 IPAC module on Linux operating systems with kernel version 2.4.4 or higher installed.

Because the TIP111 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 TIP111 device driver includes the following features:

- reading encoder value
- reading ADC value
- writing new DAC value
- reading input and status register
- setting and clearing output control bits
- configure encoder interface

2 Installation

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

The directory A:\TIP111-SW-82 contains the following files:

TIP111-SW-82.pdf	This manual in PDF format
TIP111-SW-82.tar.gz	GZIP compressed archive with driver source code

The GZIP compressed archive TIP111-SW-82.tar.gz contains the following files and directories:

tip111/tip111drv.c	Driver source code
tip111/tip111def.h	Driver include file
tip111/tip111.h	Driver include file for application program
tip111/makenode	Script to create device nodes on the file system
tip111/makefile	Device driver make file
tip111/example/example.c	Example application
tip111/example/makefile	Example application make file

In order to perform an installation, extract all files of the archive TIP111-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 *A:\CARRIER-SW-82* on the distribution diskette.

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

- 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 tip111drv

- 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 device file system (devfs) 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 TIP111 module found. The first TIP111 can be accessed with device node /dev/tip111_0, the second TIP111 or the second channel of the first TIP111 with device node /dev/tip111_1 and so on.

The allocation of device nodes to physical TIP111 modules depends on the search order of the IPAC carrier driver. Please refer to the IPAC carrier user manual.

Loading of the TIP111 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 tip111drv –r

If your kernel has enabled devfs, all /dev/tip111_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 "*tip111drv: 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 TIP111 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 TIP111_MAJOR.

To change the major number edit the file tip111drv.c, change the following symbol to appropriate value and enter **make install** to create a new driver.

TIP111_MAJOR Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TIP111_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/tip111_0”, O_RDWR);
}
```

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 *TIP111.h*:

Symbol	Meaning
T111_IOCG_READ_ENCODER	Read encoder value
T111_IOCG_READ_ADC	Read ADC value
T111_IOCT_WRITE_DAC	Write DAC value
T111_IOCG_READ_STATUS	Read input status register
T111_IOCT_SET_OUTPUT	Set output control bit(s)
T111_IOCT_CLR_OUTPUT	Clear output control bit(s)
T111_IOCS_CONFIG	Configure encoder interface

See behind for more detailed information on each control code.

To use these TIP111 specific control codes the header file TIP111.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 TIP111 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.3.1 T111_IOCG_READ_ENCODER

NAME

T111_IOCG_READ_ENCODER - Read encoder value

DESCRIPTION

This ioctl function reads the current position value from the connected SSI encoder. The data transfer (serial) could take up to 480 microseconds depending on the number of data bits and the selected clock rate. During the data transfer the driver is in a busy loop (no interrupt notification) and blocks other tasks. To minimize the blocking time please use always the maximal possible transfer rate (clock rate).

Be aware about the fact that a position jump occurs if the position bounds will be crossed (e.g. 24-bit SSI encoder will jump for 0 to 16777215 and vise versa from 16777215 to 0).

A pointer to an unsigned long variable which returns the encoder value is passed by the parameter *argp* to the driver.

EXAMPLE

```
int  fd;
int  result;
unsigned long  encoderValue;


result = ioctl(fd, T111_IOCG_READ_ENCODER, &encoderValue);

if (result < 0) {
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid argument pointer. Please check the argument argp.
EIO	Parity error occurred during transmission.
ETIME	Timeout occurred during data transfer from the encoder.

SEE ALSO

ioctl man pages

3.3.2 T111_IOCG_READ_ADC

NAME

T111_IOCG_READ_ADC - Read ADC value

DESCRIPTION

This ioctl function starts an AD conversion and returns the converted analog value (2's complement) to the caller.

Although the resolution of the ADC is 12-bit a 16-bit value is returned. The 12-bit value is shifted left by hardware and the lower bits were set to 0.

A pointer to a short variable which returns the converted analog value is passed by the parameter *argp* to the driver.

EXAMPLE

```
int   fd;
int   result;
short   adcValue;


result = ioctl(fd, T111_IOCG_READ_ADC, &adcValue);

if (result < 0) {
   /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid argument pointer. Please check the argument argp.
ETIME	Timeout occurred during AD conversion.

SEE ALSO

ioctl man pages

3.3.3 T111_IOCT_WRITE_DAC

NAME

T111_IOCT_WRITE_DAC – Write DAC value

DESCRIPTION

This ioctl function writes a new DAC value and starts DA conversion.

The new DAC value is passed by the argument *argp* to the driver.

EXAMPLE

```
int  fd;
int  result;
short  dacValue;

/*
**   write +full-scale (near 10V) to DAC
*/
dacValue = 0x7fff;

result = ioctl(fd, T111_IOCT_WRITE_DAC, dacValue);

if (result < 0) {
   /* handle ioctl error */
}
```

ERRORS

No special driver errors will be returned.

SEE ALSO

ioctl man pages

3.3.4 T111_IOCG_READ_STATUS

NAME

T111_IOCG_READ_STATUS – Read Input Status Register

DESCRIPTION

This ioctl function reads the input status register (INPSR).

A pointer to an unsigned char variable which returns the contents of the input status register (INPSR) is passed by the parameter *argp* to the driver.

Macros for all input status register bits are defined in tip111.h.

EXAMPLE

```
int  fd;
int  result;
unsigned char  inputPort;


result = ioctl(fd, T111_IOCG_READ_STATUS, &inputPort);

if (result < 0) {
   /* handle ioctl error */
}
else {
   if ((inputPort & T111_LOW_LIMIT) != 0) /* do anything */;
}
```

ERRORS

No special driver errors will be returned.

SEE ALSO

ioctl man pages

3.3.5 T111_IOCT_SET_OUTPUT

NAME

T111_IOCT_SET_OUTPUT – Set output control bit(s)

DESCRIPTION

This ioctl function sets specified bits in the output control register. Set more than one bit at the same time by a bitwise OR using the | operator in C.

The new output control register port mask is passed by the argument *argp* to the driver.

Macros for all output control register bits are defined in tip111.h.

EXAMPLE

```
int  fd;
int  result;
unsigned char  outputMask;

/*
**   enable the servo amplifier and the status LED
*/
outputMask = T111_SERVO_ENA | T111_STATUS_LED;

result = ioctl(fd, T111_IOCT_SET_OUTPUT, outputMask);

if (result < 0) {
   /* handle ioctl error */
}
```

ERRORS

No special driver errors will be returned.

SEE ALSO

ioctl man pages

3.3.6 T111_IOCT_CLR_OUTPUT

NAME

T111_IOCT_CLR_OUTPUT – Clear output control bit(s)

DESCRIPTION

This ioctl function clears specified bits in the output control register. Clear more than one bit at the same time by a bitwise OR using the | operator in C.

To clear the specified bits the driver uses a statement like this: *OUTCR &= ~outputMask.*

The new output control register port mask is passed by the argument *argp* to the driver.

Macros for all output control register bits are defined in tip111.h.

EXAMPLE

```
int  fd;
int  result;
unsigned char  outputMask;

/*
**   disbale the servo amplifier and the status LED
*/
outputMask = T111_SERVO_ENA | T111_STATUS_LED;

result = ioctl(fd, T111_IOCT_CLR_OUTPUT, outputMask);

if (result < 0) {
   /* handle ioctl error */
}
```

ERRORS

No special driver errors will be returned.

SEE ALSO

ioctl man pages

3.3.7 T111_IOCS_CONFIG

NAME

T111_IOCS_CONFIG - *Configure* encoder interface

DESCRIPTION

This ioctl function configures the encoder interface of the TIP111.

A pointer to the structure *T111_CONFIG* is passed by the argument *argp* to the driver.

The *T111_CONFIG* structure has the following layout:

```
typedef struct
{
        int     numDataBits;
        int     clockRate;
        int     enableParity;
        int     enableGrayCode;
} T111_CONFIG;
```

numDataBits

Specifies the number of data bits of the connected absolute encoder. Valid values are between 1 and 32 data bits.

clockRate

Specifies the clock speed of the serial data transfer. The clock speed can be programmed in steps of 1 microsecond in the range from 1 to 15.

enableParity

Set TRUE (1) to enable parity or FALSE (0) to disable parity. Enable parity only if this feature is supported by the encoder, otherwise *T111_IOCG_READ_ENCODER* will return a parity error (EIO).

enableGrayCode

Set TRUE (0) to enable Gray Code for serial data transfer or FALSE (0) for Dual Code. This configuration must match to the encoder properties; otherwise strange position values will be transferred. Please not the driver cannot make any plausibility check and therefore no errors will be returned if the configuration do not match.

EXAMPLE

```
int  fd;
int  result;
T111_CONFIG  configBuf;


configBuf.numDataBits       = 24;
configBuf.clockRate         = 1;    /* fastest */
configBuf.enableParity      = FALSE;
configBuf.enableGrayCode    = TRUE;

result = ioctl(fd, T111_IOCS_CONFIG, &configBuf);

if (result < 0) {
   /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid argument pointer. Please check the argument argp.
EINVAL	Either the number of data bits or the clock rate is out of range.

SEE ALSO

ioctl man pages