



TIP830-SW-82
Linux Device Driver
TIP830 – 8 Simultaneous Channel
16/12 Bit ADC

Version 1.0.x

Reference Manual
Issue 1.0

October 2001

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

TIP830-SW-82

8 Simultaneous Channel

16/12-Bit ADC

Linux Device Driver

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

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

This product has been designed to operate with IndustryPack® compatible carriers. Connection to incompatible hardware is likely to cause serious damage.

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

IndustryPack is a registered trademark of GreenSpring Computers, Inc

©2001 by TEWS TECHNOLOGIES GmbH

Issue	Description	Date
1.0	First Issue	November 20, 2001

Table of Contents

1	INTRODUCTION	4
2	INSTALLATION.....	5
2.1	Build the device driver	5
2.2	Install the device driver in the running kernel.....	5
2.3	Change Major Device Number.....	7
3	DEVICE INPUT/OUTPUT FUNCTIONS	8
3.1	open()	8
3.2	close()	9
3.3	read().....	10
3.4	ioctl()	12
3.4.1	T830_IOCTL_READ_PARAM	13

1 Introduction

The TIP830-SW-82 Linux device driver allows the operation of a TIP830 8 simultaneous channel 16/12 bit ADC IP on Linux operating systems with Intel and Intel-compatible x86 CPU.

The device driver was developed and tested on a Compact PCI system with Linux kernel Version 2.2.13 / 2.4.4 installed (SuSE Linux 6.3 / 7.2 Distribution).

The TIP830 device driver includes the following functions:

- ☞ reading converted AD values from the eight input channels with or without data correction
- ☞ reading module type and correction values out of the ID PROM

2 Installation

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

Following files are located on the diskette:

tip830drv.c	Driver source code
tip830def.h	Driver include file
tip830.h	Driver include file for application program
load830	Script for driver loading
unload830	Script for driver unloading
makefile	Device driver make file
TIP830-SW-82.pdf	This manual in PDF format
example/example.c	Example application
example/makefile	Example application makefile

In order to perform an installation, copy all files on the distribution diskette to the desired target directory.

2.1 Build the device driver

1. Change to the target directory
2. To create and install the driver in the module directory */lib/modules/<version>* enter:

```
make install
```

Note

For Linux distributions with the new style include directory path the Makefile must be modified. Enable the line *INCLUDEDIR = /lib/modules/`uname -r`/build/include* instead of *INCLUDEDIR = /usr/include*

2.2 Install the device driver in the running kernel

To install the driver in the running kernel execute the shell script *load830*.

```
sh load830 io=<ioaddr1>,<ioaddr2>,... id=<ioaddr1>,<ioaddr2>,...
```

This shell script removes a previously installed TIP830 driver, installs the new one into the kernel and creates nodes for up to eight TIP830 devices.

The arguments *io* and *id* defines the physical IP I/O and ID address of the TIP830 to initialize. The parameter *ioaddr1*, *idaddr1* belong to the first TIP830 the parameter *ioaddr1*, *idaddr2* belong to the second device and so on. Up to 8 TIP830 devices can be initialized without modifying the driver source.

Example:

```
sh load830 io=0xE0001000,0xE0002000 id=0xE0001100,0xE0002100
```

The example above initializes a TIP830 at physical I/O address 0xE0001000 and ID address 0xE0001100 and a second TIP830 at physical address 0xE0002000 and ID address 0xE0002100.

Created device nodes are:

```
/dev/tip830_0, /dev/tip830_1, ..., /dev/tip830_7
```

Note

The unmodified driver uses dynamic allocation of major device numbers. To get the current used major number the script extracts the major number of the TIP830 driver from */proc/devices* to create the correct device nodes.

For example the TIP830 I/O addresses can be extracted from the Linux */proc* file system.

The following dump shows a SBS PCI40 IP carrier board

```
# cat /proc/pci
.
.
.
Bus 0, device 9, function 0:
  Class 0680: PCI device 124b:0040 (rev 11).
  IRQ 12.
  Non-prefetchable 32 bit memory at 0xe7000000 [0xe70000ff].
  I/O at 0xe000 [0xe0ff].
  Non-prefetchable 32 bit memory at 0xe0000000 [0xe3fffffff].
```

The TIP830 I/O and ID space are located in the second memory space (BAR2) at PCI bus address 0xE0000000. The first IP appears at offset 0x1000/0x1100 from the beginning. The second IP appears at offset 0x2000/0x2100. Because the PCI bus address is equal to the physical address on i386 systems we got the following addresses for installation. The associated IRQ is also extracted from the dump.

```
sh load830 io=0xE0001000,0xE0002000 id=0xE0001100,0xE0002100
```

Note

On PowerPC systems the PCI bus address are different to the physical addresses used for driver installation. In this case you have to convert the bus address to an appropriate physical address. Please refer to related documentation from the CPU board supplier and Linux distribution.

The example above is only suitable for SBS PCI 40 IP carriers on a specific CPU board and PCI slot. If you use other IP carrier boards or the same carrier on other CPU boards (probably) or PCI slots you have to adapt the memory mapping. Please refer to related documentation.

For some IP carrier boards it also necessary to initialize some registers before the TIP830 device driver can access the IP or can use interrupts. This initialization can be done within the TIP830 device driver (function `init_PCI40()` in `tip830drv.c`) or in special initialization modules, which are started before the TIP830 driver. Usually this code initializes board interrupts and enables PCI spaces. Please refer to related documentation which initializations are necessary.

An other important point is alignment of the TIP830 controller registers in the IP I/O and ID space of the carrier board and the big/little endian problem on some systems. By default the driver assumes that the registers appears at even addresses with a gap of on byte and word registers appear with the same endian to the processor.

If you are not sure about the alignment of your carrier board enable the memory dump of controller registers by defining the symbol `TIP830_DEBUG_VIEW` in `tip830drv.c`. The memory dump will appear in the xconsole window.

With the defines `T830_ODD_MAP` and `T830_SWAP_BYTES` in `tip830def.h` its possible to adapt the driver to the hardware requirements of the based system.

2.3 Change Major Device Number

The TIP830 driver use 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 `tip830def.h`, change the following symbol to appropriate value and enter `make install` to create a new driver.

TIP830_MAJOR Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TIP830_MAJOR      122
```

Note

Be sure that the desired major number isn't used by an other driver. Please check `/proc/devices` to see which numbers are free.

Keep in mind that's necessary to create new device nodes if the major number for the TIP830 driver has changed and the `load830` script isn't used.

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 bit wise 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/tip830_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

E_NODEV 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.

ERR

E_NODEV 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 attempts to start an AD conversion and returns the converted values in a read buffer to the caller.

A pointer to the callers read buffer (*T830_RW_BUFFER*) and the size of this structure is passed by the parameters *buffer* and *size* to the device.

```
typedef struct
{
    unsigned int    correction[T830_MAX_CHAN];    /* use correction data? */
    int             data[T830_MAX_CHAN];         /* converted ADC data */
} T830_IO_BUFFER, *PT830_IO_BUFFER;
```

correction[]

If this corresponding entry of this array is *T830_CORR* the driver performs an automatic offset and gain correction with factory calibration data stored in the TIP830 ID-PROM, otherwise the value should be *T830_NOCORR*. Index 0 specifies channel 1, index 1 specifies channel 2 and so on.

data[]

The analog input values read from the ADC channels are returned in this array. The analog data is returned as sign extended two's complement integer value with 16-bit resolution. The lower four bits. Index 0 specifies channel 1, index 1 specifies channel 2 and so on.

EXAMPLE

```
{
    int          fd;
    ssize_t     NumBytes;
    int         i;
    /* use correction for channel 1, 3, 5, 7 */
    T830_IO_BUFFER ADCBuf =
        {{ T830_CORR , 0,
           T830_CORR , 0,
           T830_CORR , 0,
           T830_CORR , 0},
         {,,,,,,,,}};

    ...

    ADCBuf.correction= T830_CORR;

    NumBytes = read(fd, &ioBuf, sizeof(ioBuf));

    /*
    ** Check the result of the last device I/O operation
    */
    if (NumBytes > 0)
    {
        for (i = 0; i < ; i++)
            printf("ADC Value = %d\n", ADCBuf.data);
    }
    else
    {
        printf("Read failed --> Error = %d\n", errno);
    }
    ...
}
```

RETURNS

On success read returns the size of the structure T830_IO_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 50 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 fildes, int request [, void *argp])
```

DESCRIPTION

The **ioctl** function sends a control code directly to a device, specified by *fildes*, 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 *TIP830.h*:

Value	Meaning
<i>T830_IOCTL_READ_PARAM</i>	Read module parameter

See behind for more detailed information on each control code.

Note

To use these TIP830 specific control codes the header file *TIP830.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 TIP830 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.4.1 T830_IOCTL_READ_PARAM

NAME

T830_IOCTL_READ_PARAM - Read module parameter

DESCRIPTION

This ioctl function attempts to read the module type and calibration data of the TIP830 associated with the open file descriptor, *filedes*, into the parameter buffer pointed to by *argp*.

The parameter buffer (*T830_PARAM_BUFFER*) has the following layout:

```
typedef struct
{
    unsigned int    ModuleType;
    int             calGain[T830_MAX_CHAN];
    int             calOffs[T830_MAX_CHAN];
} T830_PARAM_BUFFER, *PT830_PARAM_BUFFER;
```

ModuleType

Receives the type code (10/20) of the associated TIP830.

calGain[]

Receives the gain error of the channel specified with the array index in the unit 1 LSB (see also Hardware User Manual). Index 0 specifies the gain error of channel 1, index 1 specifies the gain error of channel 2 and so on.

calOffs[]

Receives the offset (zero) error of the channel specified with the array index in the unit 1 LSB (see also Hardware User Manual). Index 0 specifies the offset error of channel 1, index 1 specifies the offset error of channel 2 and so on.

EXAMPLE

```
{
    int                fd;
    int                result;
    T830_PARAM_BUFFER ParamBuf;
    ...
    result = ioctl(fd, T830_IOCTL_READ_PARAM, &ParamBuf);

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

ERRORS

EFAULT Invalid pointer to the read buffer.

SEE ALSO

ioctl man pages