The Embedded I/O Company

TEWS TECHNOLOGIES

TIP610-SW-82

Linux Device Driver

Digital I/O

Version 1.1.x

User Manual

Issue 1.2

February 2004

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

TIP610-SW-82

Digital I/O

Linux Device Driver

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.

©2004 by TEWS TECHNOLOGIES GmbH

Issue	Description	Date
1.0	First Issue	July 20, 2001
1.1	Changes for driver installation	August 23, 2001
1.2	Support for IPAC CARRIER DRIVER, DEVFS and SMP	February 10, 2004

Table of Contents

1 Introduction

The TIP610-SW-82 Linux device driver allows the operation of a TIP610 IPAC module on Linux operating systems with kernel version 2.4.4 or higher installed.

Because the TIP610 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 TIP610 device driver includes the following features:

- reading the actual port values
- writing new port values
- configure port directions
- configure port polarity
- wait for selectable input events (match, high-, low-, any-transition on the input line(s) of port A and port B)

2 Installation

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

The directory A:\TIP610-SW-82 contains the following files:

TIP610-SW-82.pdf	This manual in PDF format
TIP610-SW-82.tar.gz	GZIP compressed archive with driver source code

The GZIP compressed archive TIP610-SW-82.tar.gz contains the following files and directories:

tip610/tip610drv.c	Driver source code
tip610/tip610def.h	Driver include file
tip610/tip610.h	Driver include file for application program
tip610/makenode	Script to create device nodes on the file system
tip610/makefile	Device driver make file
tip610/example/example.c	Example application
tip610/example/makefile	Example application make file

In order to perform an installation, extract all files of the archive TIP610-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 tip610drv

- 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 TIP610 module found. The first TIP610 can be accessed with device node /dev/tip610_0, the second TIP610 or the second channel of the first TIP610 with device node /dev/tip610_1 and so on.

The allocation of device nodes to physical TIP610 modules depends on the search order of the IPAC carrier driver. Please refer to the IPAC carrier user manual.

Loading of the TIP610 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 tip610drv –r

If your kernel has enabled devfs, all /dev/tip610_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 "*tip610drv: 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 TIP610 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 TIP610_MAJOR.

To change the major number edit the file tip610drv.c, change the following symbol to appropriate value and enter **make install** to create a new driver.

TIP610_MAJOR Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TIP610_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/tip610_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 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 read the port registers of the TIP610 associated with the open file descriptor, *filedes*, into the read buffer pointed to by *buffer.* Remember the values depend on the port configuration, polarity and direction.

A pointer to the callers read buffer (*T610_RW_BUFFER*) and the size of this structure is passed by the parameters *buffer* and *size* to the device.

```
typedef struct
{
  unsigned char  portA;
  unsigned char  portB;
  unsigned char  portC;
  unsigned char  wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

portA, portB, portC

These parameters receive the actual state of the corresponding port registers.

wrenaPort

Is not used for the read function

EXAMPLE

```
{
  int fd;
  ssize_t  num_bytes;
  T610_RW_BUFFER  io_buf;

  ...

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

  /*
  **  Check the result of the last device I/O operation
  */
  if (num_bytes > 0)
  {
    printf("\nRead input lines successful\n");
    printf("Port A: %02Xh\n", io_buf.portA);
    printf("Port B: %02Xh\n", io_buf.portB);
    printf("Port C: %02Xh\n", io_buf.portC);
  }
  else
  {
    printf("\nRead failed -->  Error = %d\n", errno );
  }

  ...
}
```

RETURNS

On success read returns the size of the structure T610_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.
EFAULT	Invalid pointer to the read buffer.

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

The **write** function attempts to write to the port registers of the TIP610 associated with the open file descriptor, *filedes*, from the buffer pointed to by *buffer*

A pointer to the callers write buffer (*T610_RW_BUFFER*) and the size of this structure is passed by the parameters *buffer* and *size* to the device.

```
typedef struct
{
  unsigned char  portA;
  unsigned char  portB;
  unsigned char  portC;
  unsigned char  wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

portA, portB, portC

These parameters receive the actual state of the corresponding port registers.

wrenaPort

Set of bit flags that controls the write operation. If the corresponding port flag is set the port register will be written otherwise the port is inhibit from write.

The following flags could be OR'ed

T610_ENABLE_PORTA	The contents of the member *portA* will be written to the corresponding PORTA register (lines 9..16).
T610_ENABLE_PORTB	The contents of the member *portC* will be written to the corresponding PORTB register (lines 1..8).
T610_ENABLE_PORTC	The contents of the member *portC* will be written to the corresponding PORTC register (lines 17..20).

EXAMPLE

```
{
  int fd;
  ssize_t  NumBytes;
  T610_RW_BUFFER  io_buf;

  ...

  /*
  **  Write 0xBB to PORTB and 0x04 to PORTC. Inhibit
  **  PORTA from writing.
  */
  io_buf.portA = 0x00;
  io_buf.portB = 0xBB;
  io_buf.portC = 0x04;
  io_buf.wrenaPort = T610_ENABLE_PORTB | T610_ENABLE_PORTC;

  NumBytes = write(fd, &io_buf, sizeof(io_buf));

  if (NumBytes > 0)
  {
    /* Data successful written */
  }

  ...
}
```

RETURNS

On success write returns the size of the structure T610_RW_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 write buffer is too small.
EFAULT	Invalid pointer to the write buffer.

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 *TIP610.h*:

Value	Meaning
T610_IOCG_READ_DIR	Read current port direction configuration
T610_IOCS_WRITE_DIR	Write new port direction configuration
T610_IOCG_READ_POL	Read current port polarity configuration
T610_IOCS_WRITE_POL	Write new port polarity configuration
T610_IOCX_EVENT_READ	Read port after specified input event occur

See behind for more detailed information on each control code.

To use these TIP610 specific control codes the header file TIP610.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 TIP610 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.5.1 T610_IOCG_READ_DIR

NAME

T610_IOCG_READ_DIR - Read current port direction configuration

DESCRIPTION

This ioctl function attempts to read the contents of all port direction registers of the TIP610 associated with the open file descriptor, *filedes*, into the read buffer pointed to by *argp.*

The read buffer (*T610_RW_BUFFER*) has the following layout:

```
typedef struct
{
  unsigned char  portA;
  unsigned char  portB;
  unsigned char  portC;
  unsigned char  wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

portA, portB, portC

These parameters receive the contents of the corresponding port direction register. A 0 in bit position specifies the corresponding bit of the port as an output bit, while a 1 specifies it as an input.

wrenaPort

It's not used for this ioctl function.

EXAMPLE

```
{
  int fd;
  int result;
  T610_RW_BUFFER  io_buf;

  ...

  result = ioctl(fd, T610_IOCG_READ_DIR, &io_buf);

  /*
  **  Check the result of the last device I/O control operation
  */
  if (result >= 0)
  {
    printf("    Direction Port A: %02Xh\n", io_buf.portA);
    printf("    Direction Port B: %02Xh\n", io_buf.portB);
    printf("    Direction Port C:  %1Xh\n", io_buf.portC);
  }
  else
  {
    printf("Read direction failed --> Error = %d\n", errno);
  }
  ...
}
```

SEE ALSO

ioctl man pages

3.5.2 T610_IOCS_WRITE_DIR

NAME

T610_IOCS_WRITE_DIR - Write new port direction configuration

DESCRIPTION

This ioctl function attempts to write to the port direction registers of the TIP610 associated with the open file descriptor, *filedes*, from the write buffer pointed to by *argp.*

The write buffer (*T610_RW_BUFFER*) has the following layout:

```
typedef struct
{
  unsigned char  portA;
  unsigned char  portB;
  unsigned char  portC;
  unsigned char  wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

portA, portB, portC

These parameters contain the new values for the corresponding port direction register. A 0 in bit position specifies the corresponding bit of the port as an output bit, while a 1 specifies it as an input. All bits of port A and B must have the same direction. The direction of port C can be setup individually for each bit. A reset forces all bits to 0 (output) but this causes no problems because the port isn't enabled at this moment.

wrenaPort

Set of bit flags that control the write port direction operation. If the corresponding port flag is set the port direction register will be written otherwise the port direction register is inhibit from write.

The following flags could be OR'ed

T610_ENABLE_PORTA	The contents of the member *portA* will be written to the corresponding PORTA direction register (lines 9..16).
T610_ENABLE_PORTB	The contents of the member *portC* will be written to the corresponding PORTB direction register (lines 1..8).
T610_ENABLE_PORTC	The contents of the member *portC* will be written to the corresponding PORTC direction register (lines 17..20).

EXAMPLE

```
{
  int fd;
  int result;
  T610_RW_BUFFER  io_buf;
  ...
  /*
  ** Set direction for
  ** Port A : input
  ** Port B : output
  ** Port C : bit 2 input, bit 0,1,3 output
  */
  io_buf.portA = 0xFF;
  io_buf.portB = 0x00;
  io_buf.portC = 0x04;

  io_buf.wrenaPort = T610_ENABLE_PORTA | T610_ENABLE_PORTB|
                     T610_ENABLE_PORTC;

  result = ioctl(fd, T610_IOCS_WRITE_DIR, &io_buf);

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

ERRORS

EFAULT	Invalid pointer to the write buffer.

SEE ALSO

ioctl man pages

3.5.3 T610_IOCG_READ_POL

NAME

T610_IOCG_READ_POL - Read current port polarity configuration

DESCRIPTION

This ioctl function attempts to read the contents of all port polarity registers of the TIP610 associated with the open file descriptor, *filedes*, into the read buffer pointed to by *argp.*

The read buffer (*T610_RW_BUFFER*) has the following layout:

```
typedef struct
{
  unsigned char  portA;
  unsigned char  portB;
  unsigned char  portC;
  unsigned char  wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

portA, portB, portC

These parameters receive the contents of the corresponding port polarity register. A 0 in a particular bit position specifies the corresponding bit path of the port as non-inverting (that is, a HIGH level at the I/O connector is a 1). If a bit is written with 1, the data path is programmed inverting.
After reset the data path is non-inverting.

wrenaPort

It's not used for this ioctl function.

EXAMPLE

```
{
  int fd;
  int result;
  T610_RW_BUFFER  io_buf;
  ...

  result = ioctl(fd, T610_IOCG_READ_POL, &io_buf);
  if (result >= 0)
  {
    printf("Polarity Port A: %02Xh\n", io_buf.portA);
    printf("Polarity Port B: %02Xh\n", io_buf.portB);
    printf("Polarity Port C:  %1Xh\n", io_buf.portC);
  }
  else
  {
    printf("Read polarity failed -->  Error = %d\n", errno);
  }
  ...
}
```

ERRORS

EFAULT	Invalid pointer to the read buffer.

SEE ALSO

ioctl man pages

3.5.4 T610_IOCS_WRITE_POL

NAME

T610_IOCS_WRITE_POL - Write new port polarity configuration

DESCRIPTION

This ioctl function attempts to write to the port polarity registers of the TIP610 associated with the open file descriptor, *filedes*, from the write buffer pointed to by *argp.*

The write buffer (*T610_RW_BUFFER*) has the following layout:

```
typedef struct
{
  unsigned char  portA;
  unsigned char  portB;
  unsigned char  portC;
  unsigned char  wrenaPort;
} T610_RW_BUFFER, *PT610_RW_BUFFER;
```

portA, portB, portC

These parameters contain the new values for the corresponding port polarity register. A 0 in a particular bit position specifies the corresponding bit path of the port as non-inverting (that is, a HIGH level at the I/O connector is a 1). If a bit is written with 1, the data path is programmed inverting.
After reset the data path is non-inverting.

wrenaPort

Set of bit flags that control the write port polarity operation. If the corresponding port flag is set the port polarity register will be written otherwise the port polarity register is inhibit from write.

The following flags could be OR'ed

T610_ENABLE_PORTA	The contents of the member *portA* will be written to the corresponding PORTA polarity register (lines 9..16).
T610_ENABLE_PORTB	The contents of the member *portC* will be written to the corresponding PORTB polarity register (lines 1..8).
T610_ENABLE_PORTC	The contents of the member *portC* will be written to the corresponding PORTC polarity register (lines 17..20).

EXAMPLE

```
{
  int fd;
  int result;
  T610_RW_BUFFER  io_buf;

  ...

  /*
  ** Port A : bit 0..3 non-inverting, bit 4..7 inverting
  ** Port B : non-inverting
  ** Port C : unchanged
  */
  io_buf.portA = 0xF0;
  io_buf.portB = 0x00;
  io_buf.portC = 0x00;

  io_buf.wrenaPort = T610_ENABLE_PORTA | T610_ENABLE_PORTB;

  result = ioctl(fd, T610_IOCS_WRITE_POL, &io_buf);

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

ERRORS

EFAULT	Invalid pointer to the write buffer.

SEE ALSO

ioctl man pages

3.5.5 T610_IOCX_EVENT_READ

NAME

T610_IOCX_EVENT_READ - Read port after specified input event occur

DESCRIPTION

The ioctl function reads the contents of the input ports after a specified event occur. Possible events are rising or falling edge or both, at a specified input bit or a pattern match of masked input bits.

A pointer to the callers read buffer (*T610_EVRD_BUFFER*) is passed by the argument *argp* to the driver. The *T610_EVRD_BUFFER* structure has the following layout:

```
typedef struct
{
  unsigned char  portA;
  unsigned char  portB;
  unsigned char  portC;
  unsigned char  maskA;
  unsigned char  maskB;
  unsigned char  matchA;
  unsigned char  matchB;
  unsigned char  mode;
  unsigned long  timeout;
} T610_EVRD_BUFFER, *PT610_EVRD_BUFFER;
```

portA, portB, portC

These parameters receive the contents of the corresponding port registers.

maskA, maskB

These parameters specify a bit mask. A 1 value marks the corresponding bit position as relevant.

matchA, matchB

These parameters specify a pattern that must match to the contents of the input port. Only the bit positions specified by *maskA/maskB* must compare to the input port.

mode

It specifies the “event” mode for this read request. Possible is one of the following modes:

T610_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 *maskA/maskB*. A 1 value in *maskA/maskB* means than the input bit value “must-match” identically to the corresponding bit in the *matchA/matchB* parameter.

T610_HIGH_TR	If a high-transition at the specified input bit position occurs, the driver reads the input port. A 1 value in *maskA/maskB* 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 occur.
T610_LOW_TR	If a low-transition at the specified input bit position occurs, the driver reads the input port. A 1 value in *maskA/maskB* 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 occur.
T610_ANY_TR	If a high- or low-transition at the specified input bit position occurs, the driver reads the input port. A 1 value in *maskA/maskB* 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 occur.

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.

EXAMPLE

```
{
  int fd;
  int result;
  T610_EVRD_BUFFER ev_buf;
  ...
  /*
  ** Read the input port after..
  ** bit 0 = 0
  ** bit 1 = 1
  ** bit 6 = 0
  ** bit 7 = 1
  */
  ev_buf.mode = T610_MATCH;
  ev_buf.maskA = 0xC3;    /* bit 0,1,6,7 are relevant */
  ev_buf.matchA = 0x82;
  ev_buf.maskB = 0;       /* port B isn't relvant */
  ev_buf.matchB = 0;
  ev_buf.timeout = 100;   /* ticks */

  result = ioctl(fd, T610_IOCX_EVENT_READ, &ev_buf);

  if (result >= 0)
  {
    printf("Port A: %02Xh\n", ev_buf.portA);
```

```
    printf("Port B: %02Xh\n", ev_buf.portB);
    printf("Port C: %02Xh\n", ev_buf.portC);
  }
  else
  {
    /* handle read error */
  }

  /*
  **  Read the input port after a high-transition at
  **  input line 8 occured (Port B bit 7)
  */
  ev_buf.mode = T610_HIGH_TR;
  ev_buf.maskB = 1<<7;       /* high-transition at bit 7 */
  ev_buf.maskA = 0;
 ev_buf.timeout = 100;       /* ticks */

  result = ioctl(fd, T610_IOCX_EVENT_READ, &ev_buf);
  if (result >= 0)
  {
    printf("Port A: %02Xh\n", ev_buf.portA);
    printf("Port B: %02Xh\n", ev_buf.portB);
    printf("Port C: %02Xh\n", ev_buf.portC);
  }
  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 the symbol MAX_REQUESTS in tip610def.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.

SEE ALSO

ioctl man pages

4 Debugging

For debugging output see tip610drv.c. You will find the two following symbols:

```
#undef TIP610_DEBUG_INTR
#undef TIP610_DEBUG_VIEW
```

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

The TIP610_DEBUG_INTR symbol controls debugging output from the ISR.

```
TIP610  :  interrupt entry
TIP610  :  IACK[0] vector = 0004
```

The TIP610_DEBUG_VIEW symbol controls debugging output from the remaining part of the driver.

```
TIP610  :  Probe new TIP610 mounted on <TEWS TECHNOLOGIES - (Compact)PCI
IPAC Carrier> at slot B

TIP610  : Create minor node /dev/tip610_0 (devfs).

TIP610  :  IP I/O Memory Space
00000000 : FF 01 FF 01 FF 01 FF 01 FF 01 FF 01 FF 01 FF 01
00000010 : FF 01 FF 01 FF 01 FF 01 FF 01 FF 01 FF 01 FF 01

IP I/O Memory Space after initialization
00000000 : FF F9 FF 00 FF 00 FF 84 FF FB FF 00 FF 00 FF 84
00000010 : FF F9 FF 00 FF 00 FF 84 FF FB FF 00 FF 00 FF 84

...
```