The Embedded I/O Company

TEWS TECHNOLOGIES

TIP670-SW-72

LynxOS Device Driver

Digital I/O

Version 1.1.x

User Manual

Issue 1.1.1

July 2008

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

TIP670-SW-72

LynxOS Device Driver

Digital I/O

Issue	Description	Date
1.0	First Issue	March 27, 2003
1.1.0	IPAC CARRIER support	February 27, 2006
1.1.1	New Address TEWS LLC, Carrier Driver description added	July 17, 2008

Table of Contents

1 Introduction

1.1 Device Driver

The TIP670-SW-72 LynxOS device driver allows the operation of a TIP670 IPAC module on LynxOS operating systems.

Because the TIP670 device driver is stacked on the TEWS TECHNOLOGIES IPAC carrier driver, it's necessary to install also the IPAC carrier driver. Please refer to the IPAC carrier driver user manual for further information.

The standard file (I/O) functions (open, close, read) provide the basic interface for opening and closing a file descriptor and for performing device input operations.

The TIP670 device driver includes the following functions:

- reading the input register
- writing the output register
- programming polarity of inputs and outputs
- programming watchdog timeout
- waiting for a transition at a single input line or a group of input lines (OR'ed)

The TIP670-SW-72 supports the modules listed below:

TIP670-10	8 isolated digital inputs, 8 isolated digital outputs	(IndustryPack ®)
TIP670-20	4 isolated digital inputs, 4 isolated digital outputs	(IndustryPack ®)

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

TIP670 User manual

TIP670 Engineering Manual

CARRIER-SW-72 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-72 is part of this TIP670-SW-72 distribution. It is located in directory CARRIER-SW-72 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-72 User Manual for a detailed description how to install and setup the CARRIER-SW-72 device driver, and for a description of the TEWS TECHNOLOGIES IPAC Carrier Driver concept.

2 Installation

The directory TIP670-SW-72 on the distribution media contains the following files:

TIP670-SW-72-1.1.1.pdf	This manual in PDF format
TIP670-SW-72-SRC.tar	Device Driver and Example sources
ChangeLog.txt	Release history
Release.txt	Release information

The TAR archive TIP670-SW-72-SRC.tar contains the following files and directories:

Directory path 'tip670':

tip670.c	Driver source code
tip670.h	Definitions and data structures for driver and application
tip670def.h	Definitions and data structures for the driver
tip670_info.c	Device information definition
tip670_info.h	Device information definition header
tip670.cfg	Driver configuration file include
tip670.import	Linker imports file for PowerPC platforms
Makefile	Device driver make file
example/tip670exa.c	Example application source
example/Makefile	Example application make file

In order to perform a driver installation first extract the TAR file to a temporary directory then copy the following files to their target directories:

1. Create a new directory in the system drivers directory path /sys/drivers.xxx, where xxx represents the BSP that supports the target hardware.

   For example: /sys/drivers.pp_drm/tip670 or /sys/drivers.cpci_x86/tip670

2. Copy the following files to this directory:
   - tip670.c
   - tip670def.h
   - tip670.import
   - Makefile

3. Copy tip670.h to /usr/include/

4. Copy tip670_info.c to /sys/devices.xxx/ or /sys/devices if /sys/devices.xxx does not exist (xxx represents the BSP).

5. Copy tip670_info.h to /sys/dheaders/

6. Copy tip670.cfg to */sys/cfg.xxx/*, where xxx represents the BSP for the target platform

For example: /sys/cfg.ppc or /sys/cfg.x86 ....

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-72* on the separate distribution media.

2.1 Device Driver Installation

The two methods of driver installation are as follows:

- Static Installation
- Dynamic Installation (only native LynxOS systems)

Both installation methods require the TEWS TECHNOLOGIES IPAC Carrier Driver. Please refer to the IPAC Carrier Driver User Manual for detailed information.

2.1.1 Static Installation

With this method, the driver object code is linked with the kernel routines and is installed during system start-up.

2.1.1.1 Build the driver object

1. Change to the directory /sys/drivers.xxx/tip670, where xxx represents the BSP that supports the target hardware.
2. To update the library /sys/lib/libdrivers.a enter:

```
make install
```

2.1.1.2 Create Device Information Declaration

1. Change to the directory /sys/devices.xxx/ or /sys/devices if /sys/devices.xxx does not exist (xxx represents the BSP).
2. Add the following dependencies to the Makefile

```
DEVICE_FILES_all = ... tip670_info.x
```

   And at the end of the Makefile

```
tip670_info.o:$(DHEADERS)/tip670_info.h
```

3. To update the library /sys/lib/libdevices.a enter:

```
make install
```

2.1.1.3 Modify the Device and Driver Configuration File

In order to insert the driver object code into the kernel image, an appropriate entry in file CONFIG.TBL must be created.

1. Change to the directory /sys/lynx.os/ respective /sys/bsp.xxx, where xxx represents the BSP that supports the target hardware.
2. Create an entry at the end of the file CONFIG.TBL

   Insert the following entry at the end of this file. Be sure that the necessary TEWS TECHNOLOGIES IPAC carrier driver is included **before** this entry.

```
I:tip670.cfg
```

2.1.1.4 Rebuild the Kernel

1. Change to the directory /sys/lynx.os/ (/sys/bsp.xxx)
2. Enter the following command to rebuild the kernel:

   ```
   make install
   ```

3. Reboot the newly created operating system by the following command (not necessary for KDIs):

   ```
   reboot –aN
   ```

   The N flag instructs init to run mknod and create all the nodes mentioned in the new nodetab.

4. After reboot you should find the following new devices (depends on the device configuration): /dev/tip670_0, /dev/tip670_1, /dev/tip670_2 and so on.

2.1.2 Dynamic Installation

This method allows you to install the driver after the operating system is booted. The driver object code is attached to the end of the kernel image and the operating system dynamically adds this driver to its internal structures. The driver can also be removed dynamically.

2.1.2.1 Build the driver object

1. Change to the directory /sys/drivers.xxx/tip670, where xxx represents the BSP that supports the target hardware.
2. To make the dynamic link-able driver enter :

```
make
```

2.1.2.2 Create Device Information Declaration

1. Change to the directory /sys/drivers.xxx/tip670, where xxx represents the BSP that supports the target hardware.
2. To create a device definition file for the major device (this work only on native system)

```
make t670info
```

3. To install the driver enter:

```
drinstall –c tip670.obj
```

If successful, drinstall returns a unique <driver-ID>

4. To install the major device enter:

```
devinstall –c –d <driver-ID> t670info
```

The <driver-ID> is returned by the drinstall command

5. To create nodes for the devices enter:

```
mknod /dev/tip670_0 c <major_no> 0
mknod /dev/tip670_1 c <major_no> 1
mknod /dev/tip670_2 c <major_no> 2
...
```

The <major_no> is returned by the devinstall command.

If all steps are successful completed the TIP670 is ready to use.

2.1.2.3 Uninstall dynamic loaded driver

To uninstall the TIP670 device driver enter the following commands:

```
devinstall –u –c <device-ID>
drinstall –u <driver-ID>
```

2.1.3 Configuration File: CONFIG.TBL

The device and driver configuration file CONFIG.TBL contains entries for device drivers and its major and minor device declarations. Each time the system is rebuild, the config utility read this file and produces a new set of driver and device configuration tables and a corresponding nodetab.

To install the TIP670 driver and devices into the LynxOS system, the configuration include file tip670.cfg must be included in the CONFIG.TBL (see also 2.1.1.3).

The file tip670.cfg on the distribution disk contains the driver entry (*C:tip670:\....*) and a major device entry (*D:TIP670:t670info::*) with 9 minor device entries (*"N: tip670_0:0", ..., "N: tip670_8:8"*).

If the driver should support more than nine TIP670, additional minor device entries must be added. To create the device node */dev/tip670_9* the line *N:tip670_9:9* must be added at the end of the file tip670.cfg. For the next node a minor device entry with 10 must be added and so on.

This example shows the predefined driver entry:

```
#    Format:
#    C:driver-name:open:close:read:write:select:control:install:uninstall
#    D:device-name:info-block-name:raw-partner-name
#    N:node-name:minor-dev

C:TIP670:\
     :t670open:t670close:t670read:t670write:\
     ::t670ioctl:t670install:t670uninstall
D:TIP670:t670info::
N:tip670_0:0
N:tip670_1:1
N:tip670_2:2
N:tip670_3:3
N:tip670_4:4
N:tip670_5:5
N:tip670_6:6
N:tip670_7:7
N:tip670_8:8
```

The configuration above creates the following node in the /dev directory.

/dev/tip670_0 ... /dev/tip670_8

3 TIP670 Device Driver Programming

LynxOS system calls are all available directly to any C program. They are implemented as ordinary function calls to "glue" routines in the system library, which trap to the OS code.

Note that many system calls use data structures, which should be obtained in a program from appropriate header files. Necessary header files are listed with the system call synopsis.

3.1 open()

NAME

open() - open a file

SYNOPSIS

#include <sys/file.h>
#include <sys/types.h>
#include <fcntl.h>

int open (char *path, int oflags[, mode_t mode])

DESCRIPTION

Opens a file (TIP670 device) named in *path* for reading and writing. The value of *oflags* indicates the intended use of the file. In case of a TIP670 devices *oflags* must be set to *O_RDWR* to open the file for both reading and writing. The *mode* argument is required only when a file is created. Because a TIP670 device already exists this argument is ignored.

EXAMPLE

```
int  fd

/*  open the device named "/dev/tip670_0" for I/O */
fd = open ("/dev/tip670_0", O_RDWR);
```

RETURNS

open returns a file descriptor number if successful, or –1 on error.

SEE ALSO

LynxOS System Call - open()

3.2 close()

NAME

close() – close a file

SYNOPSIS

int close(int fd)

DESCRIPTION

This function closes an opened device.

EXAMPLE

```
int  result;

result = close(fd);
```

RETURNS

close returns 0 (OK) if successful, or –1 on error

SEE ALSO

LynxOS System Call - close()

3.3 read()

NAME

read() - read from a file

SYNOPSIS

#include <tip670.h>

int read (int fd, char *buff, int count)

DESCRIPTION

This function attempts to read the input registers of the TIP670 associated with the file descriptor *fd* into a structure (*T670_RW_BUFFER*) pointed by *buff*. The argument *count* specifies the length of the buffer and must be set to the length of the structure *T670_RW_BUFFER*.

The *T670_RW_BUFFER* structure has the following layout:

```
typedef struct {
   unsigned char   portA;
   unsigned char   portB;
   unsigned char   wrenaPort;
} T670_RW_BUFFER, *PT670_RW_BUFFER;
```

portA

This parameter returns the status of outputs 1 to 4(8). Where bit 0 corresponds to output 1, bit 1 to output 2 and so on.

portB

This parameter returns the status of inputs 1 to 4(8). Where bit 0 corresponds to input 1, bit 1 to input 2 and so on.

wrenaPort

Not used.

EXAMPLE

```
int  fd;
int  result;
T670_RW_BUFFER  ioBuf;


result = read(fd, (char*)&ioBuf, sizeof(ioBuf));

if (result != sizeof(T670_RW_BUFFER)) {
     // process error;
}
```

RETURNS

When read succeeds, the size of the read buffer (*T670_RW_BUFFER)* is returned. If read fails, -1 (SYSERR) is returned.

On error, errno will contain a standard read error code (see also LynxOS System Call – read) or one of the following TIP670 specific error codes:

ENXIO	Illegal device
EINVAL	Invalid argument. This error code is returned if the size of the read buffer is too small.

SEE ALSO

LynxOS System Call - read()

TIP670 example application

3.4 write()

NAME

write() – write to a file

SYNOPSIS

#include <tip670.h>

int write (int fd, char *buff, int count)

DESCRIPTION

This function attempts to write to the output registers of the TIP670 associated with the file descriptor *fd* from a structure (T670_RW_BUFFER) pointed by buff. The argument count specifies the length of the buffer and must be set to the length of the structure T670_RW_BUFFER.

The T670_RW_BUFFER structure has the following layout:

```
typedef struct {
   unsigned char   portA;
   unsigned char   portB;
   unsigned char   wrenaPort;
} T670_RW_BUFFER, *PT670_RW_BUFFER;
```

portA

This parameter holds the new value for outputs 1 to 4(8). Where bit 0 corresponds to output 1, bit 1 to output 2 and so on.

portB

Not used.

wrenaPort

Not used.

EXAMPLE

```
int  fd;
int  result;
T670_RW_BUFFER     ioBuf;


// set outputs(not pin) 3, 5 of a TIP670-20 to logic high if
// polarity(positive) was set before
ioBuf.portA     = 0x14;

result = write(fd, (char*)&ioBuf, sizeof(ioBuf));

if (result != sizeof(T670_RW_BUFFER)) {
     // process error;
}
```

RETURNS

When write succeeds, the size of the write buffer (*T670_RW_BUFFER)* is returned. If read fails, -1 (SYSERR) is returned.

On error, errno will contain a standard write error code (see also LynxOS System Call – read) or one of the following TIP670 specific error codes:

ENXIO	Illegal device
EINVAL	Invalid argument. This error code is returned if the size of the write buffer is too small.

SEE ALSO

LynxOS System Call - write()

TIP670 example application

3.5 ioctl()

NAME

ioctl() – I/O device control

SYNOPSIS

#include <ioctl.h>
#include <tip670.h>

int ioctl (int fd, int request, char *arg)

DESCRIPTION

ioctl provides a way of sending special commands to a device driver. The call sends the value of request and the pointer arg to the device associated with the descriptor fd.

The following ioctl codes are supported by the driver and are defined in tip670.h :

Value	Meaning
T670_READ_POL	Read current port polarity configuration
T670_WRITE_POL	Set new port polarity configuration
T670_EVENT_READ	Read port after specified input event occurred
T670_SET_WATCHDOG	Sets a new value for the watchdog timeout

See behind for more detailed information on each control code.

RETURNS

ioctl returns 0 if successful, or –1 on error.

The TIP670 ioctl function returns always standard error codes. See LynxOS system call ioctl of a detailed description of possible error codes.

SEE ALSO

LynxOS System Call - ioctl().

3.5.1 T670_READ_POL

NAME

T670_READ_POL – Read the current port polarity configuration

DESCRIPTION

This ioctl function read the current port polarity configuration of the TIP670. The argument *arg* passes a pointer to a polarity mask structure (T670_RW_BUFFER) to the driver. After the call the driver will have filled the current port polarity configuration into the passed buffer.

The *T670_RW_BUFFER* structure has the following layout:

```
typedef struct {
    unsigned char   portA;
    unsigned char   portB;
    unsigned char   wrenaPort;
} T670_RW_BUFFER, *PT670_RW_BUFFER;
```

portA (outputs[1..4(8)]), portB (inputs[1..4(8)])

This parameter contains the current values for the corresponding port polarity register. For port A 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. For port B it's vice versa. This is caused by AC optical coupler, they do also invert the input signal. In this way, after reset the data path for port A is non-inverting and for port B it's inverting. So you have to regard port A (non-inverting) and port B (inverting) as normal port operation (0=LOW VOLTAGE LEVEL, 1=HIGH VOLTAGE LEVEL).

wrenaPort

Not used.

EXAMPLE

```
int  fd;
int  result;
T670_RW_BUFFER  polBuf;

result = ioctl(fd, T670_READ_POL, (char*)&polBuf);

if (result != OK) {
     // process error;
}
```

ERRORS

EINVAL	Invalid parameter.

3.5.2 T670_WRITE_POL

NAME

T670_WRITE_POL – Set port polarity configuration

DESCRIPTION

This ioctl function sets the port polarity configuration. The argument *arg* passes a pointer to an polarity mask structure (T670_RW_BUFFER) to the driver.

The *T670_RW_BUFFER* structure has the following layout:

```
typedef struct {
    unsigned char   portA;
    unsigned char   portB;
    unsigned char   wrenaPort;
} T670_RW_BUFFER, *PT670_RW_BUFFER;
```

portA (outputs[1..4(8)]), portB (inputs[1..4(8)])

These parameters contain the new values for the corresponding port polarity register. For portA 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. For portB it's vice versa.

For state after reset, see also ioctl function *T670_READ_POL*.

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

T670_ENABLE_PORTA The contents of the member *portA* will be written to the corresponding port A polarity register (Output-lines 1...4(8)).

T670_ENABLE_PORTB The contents of the member *portB* will be written to the corresponding port B polarity register (Input-lines 1...4(8)).

EXAMPLE

```
int                fd;
int                result;
T670_RW_BUFFER     polBuf;

// Set polarity for output 1 and 3 “inverting”
// Set polarity for input 2 “non-inverting”
polBuf.portA  = 0x05;
polBuf.portB  = ~0x02;
polBuf.wrenaPort   = T670_ENABLE_PORTB | T670_ENABLE_PORTA;

result = ioctl(fd, T670_WRITE_POL, (char*)polBuf);

if (result != OK) {
     // process error;
}
```

3.5.3 T670_EVENT_READ

NAME

T670_ 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 (T670_EVRD_BUFFER) is passed by the argument arg to the driver.

The T670_EVRD_BUFFER structure has the following layout:

```
typedef struct {
   unsigned char  portA;     /* value on lines of Port A */
   unsigned char  portB;     /* value on lines of Port B */
   unsigned char  maskB;     /* mask for lines of Port B */
   unsigned char  matchB;    /* pattern creating event for Port B */
   unsigned char  mode;      /* event mode */
   long           timeout;   /* timeout in ticks */
} T670_EVRD_BUFFER, *PT670_EVRD_BUFFER;
```

portA, portB

These parameters hold the contents of the corresponding port data registers when the event occurred. Please note that port line states aren't latched by hardware interrupt. So inconsistency may occur between these parameters and the real port line states. So use them with care.

maskB

This parameter specifies a bit mask. A 1 value marks the corresponding bit position as relevant.

matchB

This parameter specifies a pattern that must match to the contents of the input port. Only the bit positions specified by *maskB* must compare to the input port.

mode

Specifies the "event" mode for this read request

T670_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 *maskB*. A 1 value in *maskB* means that the input bit value "must-match" identically to the corresponding bit in the *matchB* parameter.

T670_HIGH_TR — The driver reads the input port if a high-transition at the specified bit position occurs. A 1 value in *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.

T670_LOW_TR — The driver reads the input port if a low-transition at the specified bit position occurs. A 1 value in *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.

T670_ANY_TR — The driver reads the input port if a transition (high or low) at the specified bit position occurs. A 1 value in *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 (-1) means wait indefinitely or no timeout.

EXAMPLE

```
int fd;
int result;
T670_EVRD_BUFFER evBuf;

/*
**  Read the input port after..
**   input1 = 0
**   input2 = 1
**   ..
**   input7 = 0
**   input8 = 1
*/
evBuf.mode = T670_MATCH;
evBuf.maskB = 0xC3;          /* bit 0,1,6,7 are relevant */
evBuf.matchB = 0x82;
evBuf.timeout = 100;         /* ticks */

result = ioctl(fd, T670_EVENT_READ, &evBuf);

if (result >= 0) {
     printf("Port A: %02Xh\n", evBuf.portA);
     printf("Port B: %02Xh\n", evBuf.portB);
}
else {
```

```
/* handle read error */
}


/*
**  Read the input port after a high-transition at
**  input line 8 occured (Port B bit 7)
*/
evBuf.mode = T670_HIGH_TR;
evBuf.maskB = 1 << 7;            /* high-transition at bit 7 */
evBuf.timeout = 100;             /* ticks */

result = ioctl(fd, T670_EVENT_READ, &evBuf);

if (result >= 0) {
     printf("Port A: %02Xh\n", evBuf.portA);
     printf("Port B: %02Xh\n", evBuf.portB);
}
else {
  /* handle read error */
}
```

ERRORS

EINVAL	Invalid parameter.
EAGAIN	You've set a timeout value, but there are no timeouts available. Do it again without a timeout.
EBUSY	The maximum number of concurrent requests was exceeded. Increase the value of *T670_MAX_REQUESTS* in "tip670def.h".
ETIMEDOUT	The allowed time to finish the request is elapsed.
EINTR	Interrupted system call (probably by a signal).

3.5.4 T670_SET_WATCHDOG

NAME

T670_SET_WATCHDOG – Sets a new value for the watchdog timeout

DESCRIPTION

The ioctl function sets a new value for the watchdog timeout. A pointer to the callers watchdog buffer (T670_WDG_BUFFER) is passed by the argument arg to the driver.

The T670_WDG_BUFFER structure has the following layout:

```
typedef struct {
        unsigned short  timeout;    /* new timeout for watchdog timer, 0 disables the watchdog */
} T670_WDG_BUFFER, *PT670_WDG_BUFFER;
```

timeout

A value greater than 0 sets a new value for the watchdog timeout and enables the watchdog timer. A value of 0 disables the watchdog function. The timeout value has to be in the range from 0 to 30000 [us]. For further information see the TIP670 hardware manual.

Example

```
int  fd;
int  result;
T670_WDG_BUFFER  wdgBuf;

/*
** Enable the watchdog and set timeout to 100 us
*/
wdgBuf.timeout = 100;

result = ioctl(fd, T670_SET_WATCHDOG, &wdgBuf);

/* Check the result of the last device I/O control operation */

if (result >= 0) {
     printf("\nSet watchdog successful\n");
}
else {
     printf("\nSet watchdog failed -->  Error = %d\n", errno);
     PrintErrorMessage();
}
```

4 Debugging and Diagnostic

If your installed IPAC port driver (e.g. tip670) doesn't find any devices although the IPAC is properly plugged on a carrier port, it's interesting to know what's going on in the system.

Usually all TEWS TECHNOLOGIES device driver announced significant event or errors via the device driver routine kkprintf(). To enable the debug output you must define the macro DEBUG in the device driver source files (e.g. carrier_class.c, carrier_tews_pci.c, tip670.c,...).

The debug output should appear on the console. If not please check the symbol KKPF_PORT in uparam.h. This symbol should be configured to a valid COM port (e.g. SKDB_COM1).

The following output appears at the LynxOS debug console if the carrier and IPAC driver starts:

```
TEWS TECHNOLOGIES - IPAC Carrier Class Driver version 1.2.0 (2005-04-05)
IPAC_CC  :  IPAC (Manuf-ID=B3, Model#=02) recognized @ slot=0 carrier=<TEWS TECHNOLOGIES -
(Compact)PCI IPAC Carrier>
TIP670 - Digital I/O version 1.1.0 (2006-02-27)
TIP670  :  Probe new TIP670 mounted on <TEWS TECHNOLOGIES - (Compact)PCI IPAC Carrier>
 at slot A
```

If you can't solve the problem by yourself, please contact TEWS TECHNOLOGIES with a detailed description of the error condition, your system configuration and the debug outputs.