The Embedded I/O Company

TEWS TECHNOLOGIES

CARRIER-SW-82

Linux Device Driver

IPAC Carrier

Version 1.3.x

User Manual

Issue 1.3.3

April 2010

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

CARRIER-SW-82

Linux Device Driver

IPAC Carrier

Issue	Description	Date
1.0	First Issue	October 2, 2002
1.1	New driver naming convention	January 30, 2003
1.2	VMEbus IPAC carrier support	March 5, 2004
1.3	Reject specific Carrier Boards	September 23, 2004
1.1.4	Issue format changed	November 3, 2004
1.2.0	Kernel 2.6.x support	November 30, 2004
1.2.1	Modified install description and file list	February 13, 2006
1.3.0	Kernel 2.6.x VMEbus support, /proc interface	June 7, 2006
1.3.1	Distribution file list and address of TEWS LLC modified	December 21, 2006
1.3.2	Drivers for Tundra Universe and Generic IPAC added	April 25, 2008
1.3.3	Address TEWS LLC removed	April 13,2010

Table of Contents

1 INTRODUCTION 5
2 INSTALLATION 6
2.1 Build and install carrier drivers 7
2.2 Uninstall the device driver 7
2.3 Install the device driver in the running kernel 7
2.4 Remove device driver from the running kernel 8
3 VMEBUS IPAC CARRIER 9
3.1 Universe Device Driver 9
3.2 UniSDK Patch 9
3.2.1 UniSDK 3.4 - RedHat Linux 7.1 9
3.2.2 UniSDK 4.1 - SUSE 10.0 9
3.3 Configuration 10
4 UNIVERSE DEVICE DRIVER 11
4.1 Installation 11
4.1.1 Build and install the device driver 11
4.1.2 Uninstall the device driver 11
4.1.3 Install device driver into the running kernel 11
4.1.4 Remove device driver from the running kernel 12
4.2 User Interface Device Input/Output functions 13
4.2.1 open() 13
4.2.2 close() 15
4.2.3 ioctl() 16
4.2.3.1 UNIVERSE_IOCTL_ALLOCATE_REGION 18
4.2.3.2 UNIVERSE_IOCTL_FREE_REGION 20
4.3 Possible problems 21
5 CUSTOMER IPAC CARRIER SUPPORT 22
6 GENERIC IPAC DRIVER 23
6.1 Installation 23
6.1.1 Build and install the device driver 23
6.1.2 Uninstall the device driver 23
6.1.3 Install device driver into the running kernel 23
6.1.4 Remove device driver from the running kernel 24
6.2 User Interface Device Input/Output functions 25
6.2.1 open() 25
6.2.2 close() 27
6.2.3 ioctl() 28
6.2.3.1 GEN_IPAC_IOCTL_READ_UCHAR 30
6.2.3.2 GEN_IPAC_IOCTL_READ_USHORT 32
6.2.3.3 GEN_IPAC_IOCTL_READ_ULONG 34
6.2.3.4 GEN_IPAC_IOCTL_WRITE_UCHAR 36
6.2.3.5 GEN_IPAC_IOCTL_WRITE_USHORT 38
6.2.3.6 GEN_IPAC_IOCTL_WRITE_ULONG 40
6.2.3.7 GEN_IPAC_IOCTL_MOD_INFO 42
6.2.3.8 GEN_IPAC_IOCTL_RESET_SLOT 44

7 APPENDIX 45
7.1 Supported IPAC Carrier Boards 45
7.2 Enumeration of IPAC slots 45
7.3 Exclude specific PCI Devices 46
7.4 Diagnostic 47
7.4.1 /proc file system entry 47
7.4.2 Debug Statements (printk()) 48

1 Introduction

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 software architecture that hides all of these carrier board differences under a well defined interface.

Basically the IPAC and carrier device drivers are implemented with a three level module stacking. The carrier port driver is the lowest level. It handles the implementation details of the IPAC carrier board. The carrier class driver at the second level includes the management of IPAC slots and modules and provides a common interface between the IPAC driver and the carrier board driver. At the highest level resides the IPAC port driver.

Other benefits of this software architecture are the hot-plugging and Plug and Play facility. After installation of the required device drivers and loading the carrier class driver, this driver will recognize supported carrier boards by itself. He will start the required carrier port drivers; collect information about plugged IPAC modules and starts appropriate IPAC port drivers.

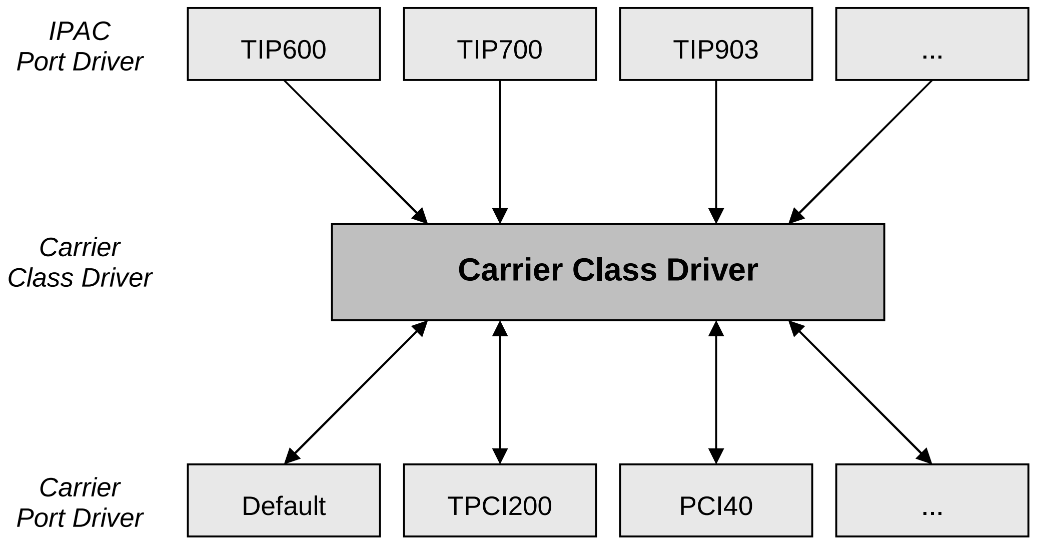

Figure 1: Stacked Driver Architecture

2 Installation

Usually the software is delivered together with the IPAC port driver.

The directory CARRIER-SW-82 on the distribution media contains the following files and directories:

CARRIER-SW-82-1.3.3.pdf	This manual in PDF format
CARRIER-SW-82-SRC.tar.gz	GZIP compressed archive with driver source code
unisdk-patch.tar.gz	GZIP compressed archive with SBS UniSDK patches
ChangeLog.txt	Release history
Release.txt	Release information

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

Directory path './ipac_carrier/':

ipac_carrier.h	Common used include file
Makefile	Makefile to build the complete carrier driver distribution
class	Sub-directory with carrier class driver sources
default	Sub-directory with default carrier port driver sources
tews_pci	Sub-directory with TEWS PCI carrier port driver sources
sbs_pci	Sub-directory with SBS PCI carrier port driver sources
universe	Sub-directory with Tundra Universe® driver sources
vme	Sub-directory with VME carrier port driver files
generic_ipac	Sub-directory with generic IPAC driver sources
include	Sub-directory with driver independent library functions

In order to perform an installation, extract all files of the archive CARRIER-SW-82-SRC.tar.gz to the desired target directory. The command 'tar -xzvf CARRIER-SW-82-SRC.tar.gz' will extract the files into the local directory.

The common used include file *ipac_carrier.h* must be copied also into the standard Linux include directory to */lib/modules/<kernel-version>/build/include* and */usr/include* to be available for installed IPAC port drivers. This file also provides the list of rejected PCI devices.

2.1 Build and install carrier drivers

- Login as *root*
- Change to the sub-directory ./class in the installation directory
- To create and install the driver in the module directory */lib/modules/<kernel-version>/misc* enter:

  # make install

- Change to the appropriate sub-directory for your carrier board (e.g. ./tews_pci if the IPAC modules are plugged on a TEWS TPCI200 carrier board)
- To create and install the driver in the module directory */lib/modules/<kernel-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 dependent kernel modules.

  # depmod –aq

2.2 Uninstall the device driver

- Login as *root*
- Change to the desired carrier driver sub-directory.
- To remove the driver from the module directory */lib/modules/<kernel-version>/misc* enter:

  # make uninstall

- Update kernel module dependency description file

  # depmod –aq

2.3 Install the device driver in the running kernel

If KMOD support is available (should be standard for most of all Linux distributions) and all module dependencies are known (depmod) it's only necessary to load the carrier class driver with:

# modprobe carrier_class

The carrier class driver will check the entire PCI bus for known IPAC carrier boards and start the appropriate carrier port drivers (e.g. carrier_tews_pci). Loaded carrier port drivers will announce their resources (IPAC slots) to the carrier class driver. The carrier class driver checks each IPAC slot for plugged modules and starts the appropriate IPAC port drivers (e.g. tip903drv) if necessary.

In this scenario, it's not necessary to start any other device driver manually except the carrier class driver.

If this automatic starting mechanism isn't desired the macro *CARRIER_PnP* in ./class/carrier_class.c must be removed (#undef).

Because all driver (module) dependencies are known, it's also possible to start the IPAC port driver (e.g. tip903drv) or the carrier port driver (e.g. carrier_tews_pci) first. All dependent drivers will be started automatically by modprobe or the carrier class driver.

The following screen shot shows the installed drivers and their dependencies:

```
# cat /proc/modules
tip903drv               8936   0 (autoclean) (unused)
carrier_tews_pci        5544   1 (autoclean)
carrier_class          10692   3 [tip903drv carrier_tews_pci]
```

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.

Assuming a TIP903 is plugged on a TPCI200 carrier board, you have to install necessary driver in the following order:

# modprobe carrier_class

# modprobe carrier_tews_pci

# modprobe tip903drv

The carrier class driver must be always the first. The order of all other drivers doesn't matter.

2.4 Remove device driver from the running kernel

Removing of IPAC port, carrier class and carrier port drivers must be done in the following order:

- IPAC port driver (e.g. tip816drv)

  # modprobe tip816drv -r

- Carrier port driver (e.g. carrier_tews_pci)

  # modprobe carrier_tews_pci -r

- Carrier class driver (carrier_class)

  # modprobe carrier_class -r

Be sure that the driver isn't opened by any application program. If opened you will got the response "*tip816drv: Device or resource busy*" and the driver will still remain in the system until you close all opened files and execute *modprobe <module> –r* again.

3 VMEbus IPAC Carrier

The VMEbus IPAC carrier driver supports VMEbus access either via the provided Tundra Universe® device driver or via a modified SBS UniSDK driver for Intel x86 platforms.

3.1 Universe Device Driver

The Universe device driver provides a simple kernel interface for VMEbus access, used by the VMEbus IPAC carrier driver. Additionally, a simple user interface is provided to map VMEbus address regions into an user application for direct access. See also chapter 4 for further information.

3.2 UniSDK Patch

The symbiosis with the SBS UniSDK driver allows concurrent access to the VMEbus from UniSDK based applications and IPAC carrier based drivers.

Due to the fact that the UniSDK driver is distributed as binary, the already installed driver must be replaced by our modified UniSDK driver. The GZIP compressed tar archive unisdk-patch.tar.gz on the distribution media contains driver patches for UniSDK V3.4 (RedHat Linux 7.1) and UniSDK V4.1 (SUSE 10.0). For installation extract the tar archive to a temporary folder and copy the driver files to the appropriate target directory and restart the UniSDK driver or the entire system to make the changes current.

3.2.1 UniSDK 3.4 - RedHat Linux 7.1

Search for the file vmedrv.o in the lib/modules directory path (usually /lib/modules/2.4.18-/misc) and replace this file with ./UniSDK/3.4/vmedrv.o from the tar archive.

3.2.2 UniSDK 4.1 - SUSE 10.0

Search for the file sbs-unisdk.ko in the /lib/modules directory path (usually /lib/modules/2.6.13-15-default/extra/) and replace this file with ./UniSDK/4.1/sbs-unisdk.ko from the tar archive.

3.3 Configuration

After loading either the Universe driver or the new UniSDK VMEbus driver into the kernel, the VMEbus IPAC carrier driver must be configured. Due to the fact that the VMEbus isn't a Plug&Play bus, VMEbus resources (memory, interrupts, etc.) must be configured manually. The header file *resource.h* in the *"/ipac_carrier/vme"* directory contains two tables for setting up required VMEbus memory windows (*image_desc[]*) and for declaring used IPAC carrier slots (*slot_desc[]*). All table entries must correspond to the real VMEbus carrier setup done by rotary switches or simple jumper configuration.

The default configuration in resource.h, setup two VMEbus windows (A16/D16 and A24/D16).

```
{  A16D16, 0x00000000, 0x00010000,  VME_A16,  VME_D16,  0,  -1 },
{  A24D16, 0x00D00000, 0x00100000,  VME_A24,  VME_D16,  0,  -1 },
```

The VMEbus window setup and the following IPAC slot setup are valid for the factory (default) setup of the TEWS TECHNOLOGIES VMEbus carrier TVME200.

```
{  0, 0x00006080,  0x80, A16D16, 0x00006000,  0x80, A16D16, 0x00D00000,
0x040000, A24D16, 64, 64,  1, 2 },
{  1, 0x00006180,  0x80, A16D16, 0x00006100,  0x80, A16D16, 0x00D40000,
0x040000, A24D16, 68, 68,  3, 4 },
{  2, 0x00006280,  0x80, A16D16, 0x00006200,  0x80, A16D16, 0x00D80000,
0x040000, A24D16, 72, 72,  5, 6 },
{  3, 0x00006380,  0x80, A16D16, 0x00006300,  0x80, A16D16, 0x00DC0000,
0x040000, A24D16, 76, 76,  7, 0 },
```

If the default configuration isn't suitable the existing entries can be modified as required. New entries can be added at the end of the list. Please refer to the comments in resource.h for detailed description of each parameter.

To make the new configuration current please rebuild the driver by entering *make install*. If the *vme* driver is already installed remove the driver from the running kernel first (see also 2.4) and install the new *vme* driver (see also 2.3).

4 Universe Device Driver

The Universe Device Driver is a kernel mode driver which allows access to the Tundra Universe® VMEbus controller chip. The driver provides a kernel mode interface used by the Carrier driver, and a simple user application interface.

4.1 Installation

4.1.1 Build and install the device driver

- Login as *root*
- Change to the target sub-directory *universe*
- To create and install the driver in the module directory */lib/modules/<version>/misc* enter:

  # make install

- To update the device driver's module dependencies, enter:

  # **depmod -aq**

4.1.2 Uninstall the device driver

- Login as *root*
- Change to the target sub-directory *universe*
- To remove the driver from the module directory */lib/modules/<version>/misc* enter:

  # make uninstall

4.1.3 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 universedrv

- After the first build or if you are using dynamic major device allocation it is necessary to create new device nodes on the file system. Please execute the script file *makenode* to do this. If your kernel has enabled a 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 the found Universe controller. This device can be accessed with device node /dev/universe.

4.1.4 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 -r universedrv
  ```

If your kernel has enabled a dynamic device file system like devfs or sysfs (udev), the /dev/universe node will be automatically removed from your file system after this.

Be sure that the driver is not opened by any application program. If opened you will get the response ``*universedrv: Device or resource busy*`` and the driver will still remain in the system until you close all opened files and execute *modprobe -r* again.

4.2 User Interface Device Input/Output functions

This chapter describes the user interface to the device driver I/O system.

4.2.1 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/universe", 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

4.2.2 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

4.2.3 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 *universe.h* :

Symbol	**Meaning**
UNIVERSE_IOCTL_ALLOCATE_REGION	Allocate a VME region
UNIVERSE_IOCTL_FREE_REGION	Free a previously allocated VME region

See behind for more detailed information on each control code.

To use these Universe specific control codes the header file universe.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*
EFAULT	Parameter data can not be copied to the drivers context

Other function dependent error codes will be described for each ioctl code separately. Note, the Universe driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

4.2.3.1 UNIVERSE_IOCTL_ALLOCATE_REGION

NAME

UNIVERSE_IOCTL_ALLOCATE_REGION – Allocate a VMEbus region

DESCRIPTION

This I/O control function allocates a VMEbus region, which can be mapped into the user application afterwards using mmap(). One filehandle can hold exactly one VMEbus region. If there is already a VME window configured which matches the specified requirements, the Universe VME Window will be reused. If this filehandle is already assigned to a VMEbus region, this former region will be freed automatically, i.e. if this region remains unused, it will be unmapped. To open multiple VMEbus regions, use multiple filehandles.

The function specific control parameter **argp** is a pointer to a *UNIVERSE_VME_WINDOW* structure.

```
typedef struct
{
    unsigned long  addr;
    unsigned long  size;
    unsigned long  offs;
    int            space;
    int            width;
    int            windownr;
} UNIVERSE_VME_WINDOW;
```

addr

This parameter describes the physical VMEbus address.

size

This parameter describes the size in bytes for this VMEbus area.

offs

This parameter returns the memory offset which was calculated by the driver, due to the required VMEbus alignment. This offset must be added to the virtual address returned by mmap() later on.

space

This parameter specifies the VMEbus address space. Valid values are:

Value	Description
UNIVERSE_A16	VMEbus A16 address space
UNIVERSE_A24	VMEbus A24 address space
UNIVERSE_A32	VMEbus A32 address space

width

This parameter specifies the data width of the VMEbus address space. Valid values are:

Value	**Description**
UNIVERSE_D8	VMEbus D8 address space
UNIVERSE_D16	VMEbus D16 address space
UNIVERSE_D24	VMEbus D24 address space
UNIVERSE_D32	VMEbus D32 address space
UNIVERSE_D64	VMEbus D64 address space

EXAMPLE

```
#include “universe.h”

int                     fd;
UNIVERSE_VME_WINDOW     VmeWindow;
int                     retval;

/*------------------------------------------------
  Configure and allocate an A16/D16 VMEbus Region
  ------------------------------------------------*/
VmeWindow.addr     = 0x6000;
VmeWindow.size     = 0x1000;
VmeWindow.offs     = 0x6000;
VmeWindow.space    = UNIVERSE_A16;
VmeWindow.width    = UNIVERSE_D16;

retval = ioctl(fd, UNIVERSE_IOCTL_ALLOCATE_REGION, (int)&VmeWindow);
if (retval >= 0)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
```

ERROR CODES

Error code	**Description**
EINVAL	Invalid parameter specified. At least one parameter inside the structure is invalid.
ENOMEM	Not enough resources to allocate the area.

4.2.3.2 UNIVERSE_IOCTL_FREE_REGION

NAME

UNIVERSE_IOCTL_FREE_REGION – Free a previously allocated VMEbus region

DESCRIPTION

This I/O control function frees a previously allocated VMEbus region. If this VMEbus region remains unused, it will be unmapped from the system. The function specific control parameter is not required and can be omitted.

EXAMPLE

```
#include “universe.h”

int       fd;
int       retval;

/*-------------------------------------------------
  Free a previously allocated VMEbus Region
  -------------------------------------------------*/
retval = ioctl(fd, UNIVERSE_IOCTL_FREE_REGION);
if (retval >= 0)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
```

ERROR CODES

Error code	Description
EINVAL	There is no VMEbus region assigned to this specific filehandle.

4.3 Possible problems

If the Universe controller chip is located on a higher PCI bus than 0, there might be some problems regarding resource allocation. The VMEbus areas are accessed over the PCI bus, and must be mapped to specific PCI memory areas dynamically during runtime. PCI memory resources which are unused during system startup are often assigned to PCI bus 0. This may result in problems, if the Universe controller is located on e.g. PCI bus 1.

Each PCI-to-PCI bridge has assigned a default amount of PCI memory, which may not be enough to map the required VMEbus resources into the PCI area.

This is a general problem, which cannot be solved by the CARRIER-SW-82 device driver. This problem is specific to the PCI setup of the system, which is mostly done by the system BIOS, or the BSP. Please contact the corresponding vendor for further help. A possible solution to this problem is to setup the affected PCI bridges differently, and claim enough memory by default.

5 Customer IPAC Carrier Support

If your IPAC carrier isn't supported by the carrier port drivers on the distribution media and your carrier board is a PCI bus carrier please contact TEWS TECHNOLOGIES.

Usually we will implement the carrier port driver without any charge within a few days.

If your carrier board doesn't require any initialization or special interrupt or error handling you can create IPAC slot entries in the default carrier port driver. The default carrier port driver will be loaded automatically by the carrier class driver.

To add IPAC slots you must change to the sub-directory */default* in the installation directory. Open the source file carrier_default.c in an appropriate editor and add a new entry in the array slot_info[] after the comment /* Please add slot entries here! */.

The creation of a new slot entry is very easy. Please copy and paste an entry from the example and change address and interrupt parameter as necessary. Be sure using always physical addresses! All fields are described in detail in the structure definition above.

You must create a slot entry for each slot. If you have a carrier board with four slots you have to create four slot entries.

After modification you have to build and install the default driver like any other carrier port driver (see also 2.1).

6 Generic IPAC Driver

The Generic IPAC Device Driver is a generic driver for IPAC module access. It can be used as a basis to develop custom device drivers for specific IPAC modules. It demonstrates the hardware access using the Carrier Driver Interface, ioctl() access functions, as well as the usage of interrupts. Please refer to the corresponding driver source files for further information.

6.1 Installation

6.1.1 Build and install the device driver

- Login as *root*
- Change to the target sub-directory *generic_ipac*
- To create and install the driver in the module directory */lib/modules/<version>/misc* enter:

```
# make install
```

- To update the device driver's module dependencies, enter:

```
# depmod -aq
```

6.1.2 Uninstall the device driver

- Login as *root*
- Change to the target sub-directory *universe*
- To remove the driver from the module directory */lib/modules/<version>/misc* enter:

```
# make uninstall
```

6.1.3 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 gen_ipacdrv
```

- After the first build or if you are using dynamic major device allocation it is necessary to create new device nodes on the file system. Please execute the script file *makenode* to do this. If your kernel has enabled a 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 found supported IPAC module. The first device can be accessed with device node /dev/gen_ipac_0, the second with /dev/gen_ipac_1 and so on.

6.1.4 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 -r gen_ipacdrv

If your kernel has enabled a dynamic device file system like devfs or sysfs (udev), the device nodes will be automatically removed from your file system after this.

Be sure that the driver is not opened by any application program. If opened you will get the response ``*gen_ipacdrv: Device or resource busy*`` and the driver will still remain in the system until you close all opened files and execute *modprobe -r* again.

6.2 User Interface Device Input/Output functions

This chapter describes the user interface to the device driver I/O system.

6.2.1 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/gen_ipac_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

6.2.2 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

6.2.3 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 *gen_ipac.h* :

Symbol	**Meaning**
GEN_IPAC_IOCTL_READ_UCHAR	Read byte (8bit) values from the IPAC module
GEN_IPAC_IOCTL_READ_USHORT	Read word (16bit) values from the IPAC module
GEN_IPAC_IOCTL_READ_ULONG	Read dword (32bit) values from the IPAC module
GEN_IPAC_IOCTL_WRITE_UCHAR	Write byte (8bit) values to the IPAC module
GEN_IPAC_IOCTL_WRITE_USHORT	Write word (16bit) values to the IPAC module
GEN_IPAC_IOCTL_WRITE_ULONG	Write dword (32bit) values to the IPAC module
GEN_IPAC_IOCTL_MOD_INFO	Return IPAC module information
GEN_IPAC_IOCTL_RESET_SLOT	Perform IPAC reset (if supported by carrier board)

See behind for more detailed information on each control code.

To use these Universe specific control codes the header file gen_ipac.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*
EFAULT	Parameter data can not be copied to the drivers context

Other function dependent error codes will be described for each ioctl code separately. Note, the Universe driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

6.2.3.1 GEN_IPAC_IOCTL_READ_UCHAR

NAME

GEN_IPAC_IOCTL_READ_UCHAR – Read byte (8bit) values from the IPAC module

DESCRIPTION

This I/O control function reads a number of byte (8bit) values from a specified IPAC area by using 8bit accesses. The Carrier Driver interface is used for hardware access.

The function specific control parameter **argp** is a pointer to a *TGEN_IPAC_RWBUFFER* structure.

```
typedef struct
{
        int   NumItems;
        unsigned char ipac_space;
        unsigned long offset;
        union {
                unsigned char   ucBuf[1];
                unsigned short  usBuf[1];
                unsigned long   ulBuf[1];
        } u;
} TGEN_IPAC_RWBUFFER;
```

NumItems

This parameter describes the number of items to read.

ipac_space

This parameter describes the IPAC space type. Possible values are:

Value	**Description**
TGEN_IPAC_IO_SPACE	IPAC I/O Space
TGEN_IPAC_ID_SPACE	IPAC ID Space
TGEN_IPAC_MEM_SPACE	IPAC MEM Space

offset

This parameter specifies the starting offset to read from. This value is a byte offset relative to the beginning of the specified IPAC space.

u

This union contains the dynamically expandable data section. Use the union member *ucBuf* to treat the data as *byte* values.

EXAMPLE

```
#include “gen_ipac.h”

int                     fd;
int                     retval, i;
TGEN_IPAC_RWBUFFER      *pRWbuf;

/*------------------------------------------------
  Read 10 Bytes from IPAC ID space (IDPROM)
  ------------------------------------------------*/
pRWbuf = (TGEN_IPAC_RWBUFFER*)malloc( sizeof(TGEN_IPAC_RWBUFFER) +
                                       10*sizeof(unsigned char) );
pRWbuf->NumItems   = 10;
pRWbuf->ipac_space = TGEN_IPAC_ID_SPACE;
pRWbuf->offset     = 0;

retval = ioctl(fd, GEN_IPAC_IOCTL_READ_UCHAR, (int)pRWbuf);
if (retval >= 0)
{
     /* function succeeded */
     for (i=0; i<10; i++)
     {
          printf( “%02X “, pRWbuf->u.ucBuf[i] );
     }
}
else
{
     /* handle the error */
}
free( pRWbuf );
```

ERROR CODES

Error code	Description
EFAULT	Error copying data between kernel and user space. Check parameter pointer.
EINVAL	Invalid parameter specified. At least one parameter inside the structure is invalid.
ENOMEM	Not enough resources to allocate internal memory.

6.2.3.2 GEN_IPAC_IOCTL_READ_USHORT

NAME

GEN_IPAC_IOCTL_READ_USHORT – Read word (16bit) values from the IPAC module

DESCRIPTION

This I/O control function reads a number of word (16bit) values from a specified IPAC area by using 16bit accesses. The Carrier Driver interface is used for hardware access.

The function specific control parameter **argp** is a pointer to a *TGEN_IPAC_RWBUFFER* structure.

```
typedef struct
{
        int   NumItems;
        unsigned char ipac_space;
        unsigned long offset;
        union {
                unsigned char   ucBuf[1];
                unsigned short  usBuf[1];
                unsigned long   ulBuf[1];
        } u;
} TGEN_IPAC_RWBUFFER;
```

NumItems

This parameter describes the number of items to read.

ipac_space

This parameter describes the IPAC space type. Possible values are:

Value	Description
TGEN_IPAC_IO_SPACE	IPAC I/O Space
TGEN_IPAC_ID_SPACE	IPAC ID Space
TGEN_IPAC_MEM_SPACE	IPAC MEM Space

offset

This parameter specifies the starting offset to read from. This value is a byte offset relative to the beginning of the specified IPAC space.

u

This union contains the dynamically expandable data section. Use the union member *usBuf* to treat the data as *word* values.

EXAMPLE

```
#include “gen_ipac.h”

int                     fd;
int                     retval, i;
TGEN_IPAC_RWBUFFER      *pRWbuf;

/*-------------------------------------------------
  Read 10 Words from IPAC IO space
  -------------------------------------------------*/
pRWbuf = (TGEN_IPAC_RWBUFFER*)malloc( sizeof(TGEN_IPAC_RWBUFFER) +
                                       10*sizeof(unsigned short) );
pRWbuf->NumItems   = 10;
pRWbuf->ipac_space = TGEN_IPAC_IO_SPACE;
pRWbuf->offset     = 0;

retval = ioctl(fd, GEN_IPAC_IOCTL_READ_USHORT, (int)pRWbuf);
if (retval >= 0)
{
     /* function succeeded */
     for (i=0; i<10; i++)
     {
          printf( “%04X “, pRWbuf->u.usBuf[i] );
     }
}
else
{
     /* handle the error */
}
free( pRWbuf );
```

ERROR CODES

Error code	Description
EFAULT	Error copying data between kernel and user space. Check parameter pointer.
EINVAL	Invalid parameter specified. At least one parameter inside the structure is invalid.
ENOMEM	Not enough resources to allocate internal memory.

6.2.3.3 GEN_IPAC_IOCTL_READ_ULONG

NAME

GEN_IPAC_IOCTL_READ_ULONG – Read dword (32bit) values from the IPAC module

DESCRIPTION

This I/O control function reads a number of dword (32bit) values from a specified IPAC area by using 32bit accesses. The Carrier Driver interface is used for hardware access.

The function specific control parameter **argp** is a pointer to a *TGEN_IPAC_RWBUFFER* structure.

```
typedef struct
{
        int   NumItems;
        unsigned char ipac_space;
        unsigned long offset;
        union {
                unsigned char   ucBuf[1];
                unsigned short  usBuf[1];
                unsigned long   ulBuf[1];
        } u;
} TGEN_IPAC_RWBUFFER;
```

NumItems

This parameter describes the number of items to read.

ipac_space

This parameter describes the IPAC space type. Possible values are:

Value	Description
TGEN_IPAC_IO_SPACE	IPAC I/O Space
TGEN_IPAC_ID_SPACE	IPAC ID Space
TGEN_IPAC_MEM_SPACE	IPAC MEM Space

offset

This parameter specifies the starting offset to read from. This value is a byte offset relative to the beginning of the specified IPAC space.

u

This union contains the dynamically expandable data section. Use the union member *ulBuf* to treat the data as *dword* values.

EXAMPLE

```
#include “gen_ipac.h”

int                     fd;
int                     retval, i;
TGEN_IPAC_RWBUFFER      *pRWbuf;

/*------------------------------------------------
  Read 10 DWords from IPAC MEM space
  ------------------------------------------------*/
pRWbuf = (TGEN_IPAC_RWBUFFER*)malloc( sizeof(TGEN_IPAC_RWBUFFER) +
                                       10*sizeof(unsigned long) );
pRWbuf->NumItems   = 10;
pRWbuf->ipac_space = TGEN_IPAC_MEM_SPACE;
pRWbuf->offset     = 0;

retval = ioctl(fd, GEN_IPAC_IOCTL_READ_ULONG, (int)pRWbuf);
if (retval >= 0)
{
     /* function succeeded */
     for (i=0; i<10; i++)
     {
          printf( “%08lX “, pRWbuf->u.ulBuf[i] );
     }
}
else
{
     /* handle the error */
}
free( pRWbuf );
```

ERROR CODES

Error code	Description
EFAULT	Error copying data between kernel and user space. Check parameter pointer.
EINVAL	Invalid parameter specified. At least one parameter inside the structure is invalid.
ENOMEM	Not enough resources to allocate internal memory.

6.2.3.4 GEN_IPAC_IOCTL_WRITE_UCHAR

NAME

GEN_IPAC_IOCTL_WRITE_UCHAR – Write byte (8bit) values to the IPAC module

DESCRIPTION

This I/O control function writes a number of byte (8bit) values to a specified IPAC area by using 8bit accesses. The Carrier Driver interface is used for hardware access.

The function specific control parameter **argp** is a pointer to a *TGEN_IPAC_RWBUFFER* structure.

```
typedef struct
{
        int   NumItems;
        unsigned char ipac_space;
        unsigned long offset;
        union {
                unsigned char   ucBuf[1];
                unsigned short  usBuf[1];
                unsigned long   ulBuf[1];
        } u;
} TGEN_IPAC_RWBUFFER;
```

NumItems

This parameter describes the number of items to write.

ipac_space

This parameter describes the IPAC space type. Possible values are:

Value	**Description**
TGEN_IPAC_IO_SPACE	IPAC I/O Space
TGEN_IPAC_ID_SPACE	IPAC ID Space
TGEN_IPAC_MEM_SPACE	IPAC MEM Space

offset

This parameter specifies the starting offset to write to. This value is a byte offset relative to the beginning of the specified IPAC space.

u

This union contains the dynamically expandable data section. Use the union member *ucBuf* to treat the data as *byte* values.

EXAMPLE

```
#include “gen_ipac.h”

int                     fd;
int                     retval, i;
TGEN_IPAC_RWBUFFER      *pRWbuf;

/*----------------------------------------------------
  Write 2 Bytes to IPAC IO space, starting at offset 2
  ----------------------------------------------------*/
pRWbuf = (TGEN_IPAC_RWBUFFER*)malloc( sizeof(TGEN_IPAC_RWBUFFER) +
                                      2*sizeof(unsigned char) );
pRWbuf->NumItems   = 2;
pRWbuf->ipac_space = TGEN_IPAC_IO_SPACE;
pRWbuf->offset     = 2;
pRWbuf->u.ucBuf[0] = 0x42;
pRWbuf->u.ucBuf[1] = 0x43;

retval = ioctl(fd, GEN_IPAC_IOCTL_WRITE_UCHAR, (int)pRWbuf);
if (retval >= 0)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
free( pRWbuf );
```

ERROR CODES

Error code	Description
EFAULT	Error copying data between kernel and user space. Check parameter pointer.
EINVAL	Invalid parameter specified. At least one parameter inside the structure is invalid.
ENOMEM	Not enough resources to allocate internal memory.

6.2.3.5 GEN_IPAC_IOCTL_WRITE_USHORT

NAME

GEN_IPAC_IOCTL_WRITE_USHORT – Write word (16bit) values to the IPAC module

DESCRIPTION

This I/O control function writes a number of word (16bit) values to a specified IPAC area by using 16bit accesses. The Carrier Driver interface is used for hardware access.

The function specific control parameter **argp** is a pointer to a *TGEN_IPAC_RWBUFFER* structure.

```
typedef struct
{
        int   NumItems;
        unsigned char ipac_space;
        unsigned long offset;
        union {
                unsigned char   ucBuf[1];
                unsigned short  usBuf[1];
                unsigned long   ulBuf[1];
        } u;
} TGEN_IPAC_RWBUFFER;
```

NumItems

This parameter describes the number of items to write.

ipac_space

This parameter describes the IPAC space type. Possible values are:

Value	Description
TGEN_IPAC_IO_SPACE	IPAC I/O Space
TGEN_IPAC_ID_SPACE	IPAC ID Space
TGEN_IPAC_MEM_SPACE	IPAC MEM Space

offset

This parameter specifies the starting offset to write to. This value is a byte offset relative to the beginning of the specified IPAC space.

u

This union contains the dynamically expandable data section. Use the union member *usBuf* to treat the data as *word* values.

EXAMPLE

```
#include “gen_ipac.h”

int                     fd;
int                     retval, i;
TGEN_IPAC_RWBUFFER      *pRWbuf;

/*-----------------------------------------------------
  Write 2 Words to IPAC IO space, starting at offset 0
  -----------------------------------------------------*/
pRWbuf = (TGEN_IPAC_RWBUFFER*)malloc( sizeof(TGEN_IPAC_RWBUFFER) +
                                      2*sizeof(unsigned short) );
pRWbuf->NumItems   = 2;
pRWbuf->ipac_space = TGEN_IPAC_IO_SPACE;
pRWbuf->offset     = 0;
pRWbuf->u.usBuf[0] = 0x4243;
pRWbuf->u.usBuf[1] = 0x4445;

retval = ioctl(fd, GEN_IPAC_IOCTL_WRITE_USHORT, (int)pRWbuf);
if (retval >= 0)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
free( pRWbuf );
```

ERROR CODES

Error code	Description
EFAULT	Error copying data between kernel and user space. Check parameter pointer.
EINVAL	Invalid parameter specified. At least one parameter inside the structure is invalid.
ENOMEM	Not enough resources to allocate internal memory.

6.2.3.6 GEN_IPAC_IOCTL_WRITE_ULONG

NAME

GEN_IPAC_IOCTL_WRITE_ULONG – Write dword (32bit) values to the IPAC module

DESCRIPTION

This I/O control function writes a number of dword (32bit) values to a specified IPAC area by using 16bit accesses. The Carrier Driver interface is used for hardware access.

The function specific control parameter **argp** is a pointer to a *TGEN_IPAC_RWBUFFER* structure.

```
typedef struct
{
        int   NumItems;
        unsigned char ipac_space;
        unsigned long offset;
        union {
                unsigned char   ucBuf[1];
                unsigned short  usBuf[1];
                unsigned long   ulBuf[1];
        } u;
} TGEN_IPAC_RWBUFFER;
```

NumItems

This parameter describes the number of items to write.

ipac_space

This parameter describes the IPAC space type. Possible values are:

Value	**Description**
TGEN_IPAC_IO_SPACE	IPAC I/O Space
TGEN_IPAC_ID_SPACE	IPAC ID Space
TGEN_IPAC_MEM_SPACE	IPAC MEM Space

offset

This parameter specifies the starting offset to write to. This value is a byte offset relative to the beginning of the specified IPAC space.

u

This union contains the dynamically expandable data section. Use the union member *ulBuf* to treat the data as *dword* values.

EXAMPLE

```
#include “gen_ipac.h”

int                     fd;
int                     retval, i;
TGEN_IPAC_RWBUFFER      *pRWbuf;

/*-----------------------------------------------------
  Write 2 DWords to IPAC IO space, starting at offset 0
  -----------------------------------------------------*/
pRWbuf = (TGEN_IPAC_RWBUFFER*)malloc( sizeof(TGEN_IPAC_RWBUFFER) +
                                      2*sizeof(unsigned long) );
pRWbuf->NumItems   = 2;
pRWbuf->ipac_space = TGEN_IPAC_IO_SPACE;
pRWbuf->offset     = 0;
pRWbuf->u.ulBuf[0] = 0x42434445;
pRWbuf->u.ulBuf[1] = 0x01020304;

retval = ioctl(fd, GEN_IPAC_IOCTL_WRITE_ULONG, (int)pRWbuf);
if (retval >= 0)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
free( pRWbuf );
```

ERROR CODES

Error code	Description
EFAULT	Error copying data between kernel and user space. Check parameter pointer.
EINVAL	Invalid parameter specified. At least one parameter inside the structure is invalid.
ENOMEM	Not enough resources to allocate internal memory.

6.2.3.7 GEN_IPAC_IOCTL_MOD_INFO

NAME

GEN_IPAC_IOCTL_MOD_INFO – Return IPAC module information

DESCRIPTION

This I/O control function returns specific information about the IPAC module.

The function specific control parameter **argp** is a pointer to a *TGEN_IPAC_INFO* structure.

```
typedef struct
{
        int   ManufacturerID;
        int   ModelNumber;
        int   SlotIndex;
} TGEN_IPAC_INFO;
```

ManufacturerID

This parameter describes the Manufacturer ID of the IPAC module (0xB3 for TEWS TECHNOLOGIES).

ModelNumber

This parameter describes the Model Number of the IPAC module (0x36 for TEWS' TIP675)

SlotIndex

This parameter returns a zero-based slot index, where 0=A, 1=B etc.

EXAMPLE

```
#include “gen_ipac.h”

int                 fd;
int                 retval;
TGEN_IPAC_INFO      ModuleInfo;

/*----------------------------------------------------
  Read IPAC Module Information
  ----------------------------------------------------*/

retval = ioctl(fd, GEN_IPAC_IOCTL_MOD_INFO, (int)&ModuleInfo);
if (retval >= 0)
{
     /* function succeeded */
     printf(“Manufacturer: %02X\n”, ModuleInfo.ManufacturerID);
     printf(“Model Number: %02X\n”, ModuleInfo.ModelNumber);
     printf(“Slot Index  : %02X\n”, ModuleInfo.SlotIndex);
}
else
{
     /* handle the error */
}
```

ERROR CODES

Error code	Description
EFAULT	Error copying data between kernel and user space. Check parameter pointer.

6.2.3.8 GEN_IPAC_IOCTL_RESET_SLOT

NAME

GEN_IPAC_IOCTL_RESET_SLOT – Perform IPAC reset (if supported)

DESCRIPTION

This I/O control function performs a reset of the specific IPAC slot, if this feature is supported by the used carrier board.

The function specific control parameter **argp** is not used and can be omitted.

EXAMPLE

```
#include “gen_ipac.h”

int                 fd;
int                 retval;

/*----------------------------------------------------
  Reset IPAC slot
  -----------------------------------------------------*/

retval = ioctl(fd, GEN_IPAC_IOCTL_RESET_SLOT);
if (retval >= 0)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}
```

ERROR CODES

Error code	Description
EPERM	Function is not supported by the used carrier board.

7 Appendix

7.1 Supported IPAC Carrier Boards

The following TEWS TECHNOLOGIES and SBS IPAC carrier boards are supported:

Driver	Carrier Board	Descrition
carrier_tews_pci	TPCI100	PCI carrier for 2 IndustryPack modules
	TPCI200	PCI carrier for 4 IndustryPack modules
	TCP201	Compact PCI carrier for 4 IndustryPack modules
	TCP211	Compact PCI carrier for 2 IndustryPack modules
	TCP212	Compact PCI carrier for 2 IndustryPack modules
	TCP213	Compact PCI carrier for 2 IndustryPack modules
	TCP220	Compact PCI carrier for 4 IndustryPack modules
	TAMC100	AMC Carrier for 1 IndustryPack® module
	TVME230	PCI Expansion Card for 4 IndustryPack Modules
	TVME8240	Local IP slots of the TVME8240 CPU board
carrier_sbs_pci	PCI40	PCI carrier for 4 IndustryPack modules
	cPCI100	Compact PCI carrier for 2 IndustryPack modules
	cPCI200	Compact PCI carrier for 4 IndustryPack modules
carrier_vme (Universe Driver or SBS UniSDK)	TVME200	VMEbus carrier for 4 IndustryPack modules
	TVME201	VMEbus carrier for 4 IndustryPack modules
	TVME210	VMEbus carrier for 2 IndustryPack modules
	TVME211	VMEbus carrier for 2 IndustryPack modules
	TVME220	VMEbus carrier for 4 IndustryPack modules

7.2 Enumeration of IPAC slots

If more than one IPAC module is installed, maybe on different carrier boards, it is sometimes necessary to know which device node belongs to a certain slot on a carrier board.

The search and allocation order of the carrier class driver is always deterministic and never accidental. Usually the PCI bus will be searched from lower buses to higher buses and from lower devices to higher devices.

On carrier boards the slots will be enumerated from lower slots to higher slots.

If different carrier boards are installed in the system the order depends on the start order of the carrier port drivers. If the carrier port driver will be automatically started by the carrier class driver the start order depends on order of entries in the list *carrier_PnP_list* in the header file *./class/pnpinf.h*. If manually started, the order depends of course on the manually start order.

7.3 Exclude specific PCI Devices

To exclude some specific PCI devices, the exact location on the PCI bus can be specified in the structure *rejectedPciDevices* in the header file ipac_carrier.h. If a device is found matching the specified values, it is rejected by the carrier port driver.

```
typedef struct PciDeviceStruct
{
   unsigned char busNo;
   unsigned char devNo;
   unsigned char funcNo;
} PciDeviceStruct;
```

busNo

This parameter specifies the PCI bus number, where the specific PCI device is mounted.

devNo

This parameter specifies the device number of the specific PCI device on the bus.

funcNo

This parameter specifies the function number of the specific PCI device.

To retrieve the necessary parameters, execute **lspci** or take a look into the file */proc/pci* and search for the desired device that should not be used by the carrier port driver.

```
# lspci
00:0f.0 Bridge: TEWS Datentechnik GmBH: Unknown device 3064
00:11.0 Bridge: TEWS Datentechnik GmBH: Unknown device 30c8
```

Example

```
/*
** This will exclude the following PCI devices located on bus 0:
** device 0x0f and device 0x11.
*/
#define MAX_REJECT_PCI_DEVICES     2
static PciDeviceStruct rejectedPciDevices[MAX_REJECT_PCI_DEVICES] = {
     {0x00, 0x0f, 0x00},
     {0x00, 0x11, 0x00} };
```

7.4 Diagnostic

If your installed IPAC port driver (e.g. tip903drv) doesn't find any devices although the IPAC is properly plugged on a carrier slot, it's interesting to know what's going on in the system.

7.4.1 /proc file system entry

The TEWS TECHNOLOGIES IPAC carrier driver exports detailed information of registered IP slots, of plugged IP modules and their configuration, of registered IP port drivers and low-level carrier drivers via the /proc file system. All these information can be retrieved by a simple cat to the /proc file system entry /proc/tews-ip-carrier. Most of the displayed information is of interest only to the device driver developer and should be added to a support request in case of trouble with the carrier driver respective IP port driver.

```
# cat /proc/tews-ip-carrier

TEWS TECHNOLOGIES - IPAC Carrier Class Driver version 1.3.x (Release-Date)

Registered IP slots:

[TEWS TECHNOLOGIES - (Compact)PCI IPAC Carrier - Slot 0]
    Plugged Module      Vendor=0xB3, Modul=0x1C
    Installed Driver    TIP903 - 3 Channel Extended CAN Bus IP -
    Slot Setup          INT0_EN | LEVEL_SENS | CLK_8MHZ | MEM_16BIT |
                        Memory Size = 0x400
    Interrupt Vector    System=5, Module=5
    Interrupt Level     INT0=0, INT1=0
    ID Space Addr       Physical=0xec821080, Virtual=0x26db5080
    IO Space Addr       Physical=0xec821000, Virtual=0x00000000
    MEM8 Space Addr     Physical=0xec000000, Virtual=0x00000000
    MEM16 Space Addr    Physical=0xeb000000, Virtual=0x26dca000

[TEWS TECHNOLOGIES - (Compact)PCI IPAC Carrier - Slot 1]
    Plugged Module      Vendor=0xB3, Modul=0x1D
    Installed Driver    TIP866 - 8 Channel Serial IP
    Slot Setup          INT0_EN | INT1_EN | LEVEL_SENS | CLK_8MHZ |
                        Memory Size = 0x0
    Interrupt Vector    System=5, Module=5
    Interrupt Level     INT0=0, INT1=0
    ID Space Addr       Physical=0xec821180, Virtual=0x26dcc180
    IO Space Addr       Physical=0xec821100, Virtual=0x26dff100
    MEM8 Space Addr     Physical=0xec400000, Virtual=0x00000000
    MEM16 Space Addr    Physical=0xeb800000, Virtual=0x00000000

Registered Carrier Drivers:
TEWS TECHNOLOGIES - (Compact)PCI IPAC Carrier V1.3.x
```

7.4.2 Debug Statements (printk())

Usually all TEWS TECHNOLOGIES device drivers announce significant events or errors via the kernel message system (printk()).

You can retrieve this messages from the /proc file system using the following command

# cat /proc/kmsg

```
TEWS TECHNOLOGIES - IPAC Carrier Class Driver version 1.3.x (<Release-Date>)

TEWS TECHNOLOGIES - Default Carrier version 1.3.x (<Release-Date>)

TEWS TECHNOLOGIES - (Compact)PCI IPAC Carrier version 1.3.x (<Release-Date>)

TIP903 - 3 Channel Extended CAN Bus IP - version 1.2.0 (2006-04-05)<6>

TIP903  :  Probe new TIP903 mounted on <TEWS TECHNOLOGIES - (Compact)PCI IPAC Carrier> at slot
A
```

If the standard and error messages doesn't help to locate the problem you can enable more detailed debug output in each driver by removing the comments around the DEBUGxxx definitions.

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.