The Embedded I/O Company

TEWS TECHNOLOGIES

CARRIER-SW-82

Linux Device Driver

IPAC Carrier

Version 2.1.x

User Manual

Issue 2.1.0

June 2013

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
1.3.4	General Revision	October 27, 2010
2.0.0	General revision, New document structure, VME driver interface and generic device driver interface modified	April 19, 2011
2.1.0	TPCE200 added to “Supported IPAC Carrier Boards” Chapter Known Issues added	26.06.2013

Table of Contents

1 INTRODUCTION 5
2 FILE INSTALLATION 7
3 CARRIER DRIVER 8
3.1 Installation 8
3.1.1 Build and install Carrier Drivers 8
3.1.2 Uninstall the Device Driver 8
3.1.3 Install the Device Driver in the running Kernel 9
3.1.4 Remove IPAC Drivers from the running Kernel 10
3.2 VMEbus Support 11
3.2.1 Configuration 11
3.3 Customer IPAC Carrier Support 12
4 VME API 13
4.1 Installation 13
4.2 Create and Remove VME API Devices 13
4.2.1 Create Device Node 13
4.2.2 Delete Device Node Entry 13
4.3 User Interface Functions 14
4.3.1 open 14
4.3.2 close 16
4.3.3 ioctl 18
4.3.3.1 VMECTRL_IOCTL_ALLOCATE_REGION 20
4.3.3.2 VMECTRL_IOCTL_FREE_REGION 23
5 GENERIC IPAC DRIVER 24
5.1 Installation 24
5.1.1 Build and install the Device Driver 24
5.1.2 Uninstall the Device Driver 24
5.1.3 Install Device Driver into the running Kernel 24
5.1.4 Remove Device Driver from the running Kernel 25
5.2 Create and Remove Generic Devices 25
5.2.1 Create Device Node 25
5.2.2 Delete Device Node Entry 25
5.3 Configure Generic Driver 26
5.4 User Interface Functions 27
5.4.1 open 27
5.4.2 close 29
5.4.3 ioctl 31
5.4.3.1 GEN_IPAC_IOCTL_READ_UCHAR 33
5.4.3.2 GEN_IPAC_IOCTL_READ_USHORT 35
5.4.3.3 GEN_IPAC_IOCTL_READ_ULONG 37
5.4.3.4 GEN_IPAC_IOCTL_WRITE_UCHAR 39
5.4.3.5 GEN_IPAC_IOCTL_WRITE_USHORT 41
5.4.3.6 GEN_IPAC_IOCTL_WRITE_ULONG 43
5.4.3.7 GEN_IPAC_IOCTL_MOD_INFO 45
5.4.3.8 GEN_IPAC_IOCTL_RESET_SLOT 47

6 APPENDIX 48
6.1 Supported IPAC Carrier Boards 48
6.2 Supported VMEbus Controller 49
6.3 Enumeration of IPAC Slots 50
6.4 Exclude specific PCI Devices 51
6.5 Support of non-conformal IndustryPack Modules 52
6.6 Diagnostic 53
6.6.1 /proc File System Entry 53
6.6.2 Debug Statements 54

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.

To support different VME controllers, the carrier port driver for VMEbus is split into two layers, first the VME controller driver (controller dependent functions) and second the VME class driver, which let the VME support look like a simple IPAC carrier 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. It will start the required carrier port drivers; collect information about plugged IPAC modules and starts appropriate IPAC port drivers.

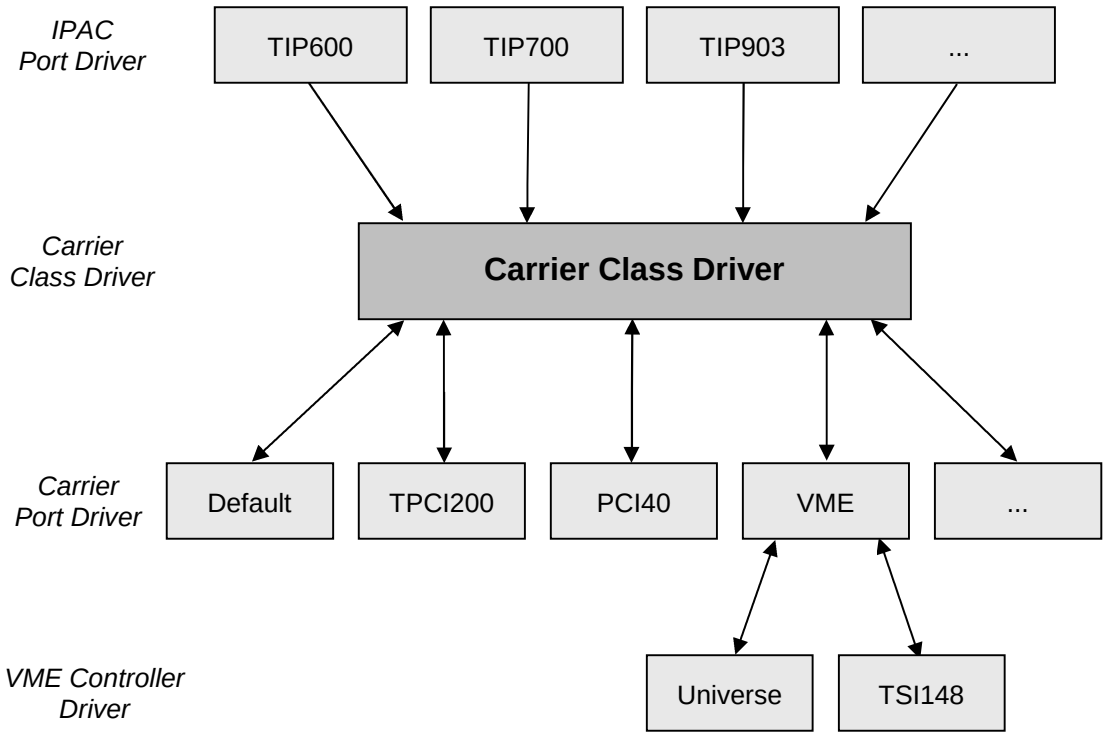

Figure 1: Stacked Driver Architecture

A special VME device (see 4 VME API) supports the allocation of VME spaces directly from applications, which allows easy access to VMEbus beside a running IPAC driver.

A Generic IPAC Driver (see 5 Generic IPAC Driver) allows simple access to IPACs that are not supported by a special driver.

The CARRIER-SW-82 driver supports the following features:

- IPAC Carrier Interface which allows IPAC drivers to use a hardware independent interface
  - Hardware abstraction layer
  - Plug & Play facility for IPAC and carrier port drivers
  - Enumeration of recognized IPAC modules
  - Diagnostic information via /proc file system
- VME support for application access
  - Mapping of arbitrary VME bus address spaces
  - Configuration of the address space and data bus width
  - Direct and transparent access to mapped address spaces
- Generic IPAC driver, that allows basic access to IPAC modules
  - Write access to the device 8/16/32-bit
  - Read access to the device 8/16/32-bit
  - Reading IPAC information (Modelnumber, Manufacturer-ID, …)

2 File Installation

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

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

CARRIER-SW-82-2.1.0.pdf	This manual in PDF format
CARRIER-SW-82-SRC.tar.gz	GZIP compressed archive with driver source code
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
vme	Sub-directory with VME carrier class driver files
vme/universe	Sub-directory with Tundra Universe® driver sources
vme/tsi148	Sub-directory with Tundra TSI148 driver sources
vme/example	Sub-directory with sources of VME device access example
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.

How to install the different drivers into the Kernel is shown in the corresponding chapter.

3 Carrier Driver

3.1 Installation

3.1.1 Build and install Carrier Drivers

- Login as *root*
- Change into the carrier driver directory *ipac_carrier*
- To create and install all carrier driver parts in the module directory */lib/modules/<kernel-version>/misc* enter:

  # make install

- 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

3.1.2 Uninstall the Device Driver

- Login as *root*
- Change into the carrier driver directory *ipac_carrier*
- To remove all carrier driver parts from the module directory */lib/modules/<kernel-version>/misc* enter:

  # make install

  Update kernel module dependency description file

  # depmod –aq

3.1.3 Install the Device Driver in the running Kernel

If KMOD support is available (this should be standard for most of the 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 starts 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. tip816drv) if necessary.

To avoid unintentionally fetching of supported VME controllers (Tundra Universe® or TSI148) the VME drivers must be loaded manually with:

 # modprobe carrier_vme

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).

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.

- Load Carrier class driver
- Load Carrier port driver
- Load VME class driver
- Load VME controller driver
- Load IPAC port drivers

3.1.4 Remove IPAC Drivers from the running Kernel

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

- Remove IPAC port drivers
- Remove VME controller driver
- Remove VME class driver
- Remove Carrier port driver
- Remove Carrier class driver

If a driver can not be removed, close all applications using the driver and keep the listed order to remove drivers in mind.

EXAMPLE 1 (PCI Carrier)

Configuration:
- IPACs: TIP816 and TIP700
- IPAC-Carrier: TPCI100

```
# modprobe tip816drv -r
# modprobe tip700drv -r
# modprobe carrier_tews_pci -r
# modprobe carrier_class -r
```

EXAMPLE 2 (Installed on VME Carrier)

Configuration:
- IPACs: TIP816
- VME-Controller: TSI148
- IPAC-Carrier: TVME200

```
# modprobe tip816drv -r
# modprobe carrier_tsi148 -r
# modprobe carrier_vme -r
# modprobe carrier_class -r
```

3.2 VMEbus Support

The VMEbus support is split into two layers. The first layer (VME controller driver) supplies simple functions that allow the second layer (VME carrier class driver) to access the VME bus controller, which will setup the VME controller special settings. This allows the use of different VME controllers with the same VME carrier class driver.

3.2.1 Configuration

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 sets up 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 },
```

If mapping of a VMEbus window fails, the problem may be caused by insufficient PCI memory resources. Please check, if enough space has been assigned to the VME controllers PCI bus. This problem may happen mainly, if larger VME windows shall be allocated.

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 3.1.4) and install the new *vme* driver (see also 3.1.3).

3.3 Customer IPAC Carrier Support

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 3.1).

4 VME API

The VME API provides a simple interface to allocate VME memory regions for direct access by the application.

4.1 Installation

The VME API driver is part of the VME controller driver, which is installed and started with VME class driver. There is no individual installation necessary.

4.2 Create and Remove VME API Devices

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.

4.2.1 Create Device Node

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* in the matching VME controller device driver path to do this.

For Tundra Universe call:

 # cd ./vme/universe
 # sh makenode

And for Tundra TSI148 call:

 # cd ./vme/tsi148
 # sh makenode

On success the device driver will create a minor device for the VME controller. This new device can be accessed with device node /dev/vme.

4.2.2 Delete Device Node Entry

The device node entry can be easily removed. Therefore just delete the special file vme in the /dev path.

 # rm /dev/vme

4.3 User Interface Functions

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

4.3.1 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

```
#include <fcntl.h>

int fd;

fd = open("/dev/vme", O_RDWR);
if (fd < 0)
{
     /* handle error */
}
```

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

Error code	Description
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

4.3.2 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

```
#include <unistd.h>

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

Error code	Description
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

4.3.3 ioctl

NAME

ioctl() – device control functions

SYNOPSIS

#include <sys/ioctl.h>

```
int ioctl
(
        int filedes,
        int request
        [, void *argp]
)
```

DESCRIPTION

The **ioctl** function sends a control code directly to a device, specified by *filedes*, causing the corresponding device to perform the requested operation.

The argument *request* specifies the control code for the operation. The optional argument *argp* depends on the selected request and is described for each request in detail later in this chapter.

The following ioctl codes are defined in *vmectrl.h* :

Symbol	Meaning
VMECTRL_IOCTL_ALLOCATE_REGION	Allocate a VME region
VMECTRL_IOCTL_FREE_REGION	Free a previously allocated VME region

See behind for more detailed information on each control code.

To use these specific control codes the header file vmectrl.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

Error code	Description
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, VME controller device drivers always return standard Linux error codes.

SEE ALSO

ioctl man pages

4.3.3.1 VMECTRL_IOCTL_ALLOCATE_REGION

NAME

VMECTRL_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 file handle can hold exactly one VMEbus region. If there is already a VME window configured which matches the specified requirements, the VME Window will be reused. If this file handle 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 file handles.

If mapping of a VMEbus region fails, the problem may be caused by insufficient PCI memory resources. Please check, if enough space has been assigned to the VME controllers PCI bus. This problem may happen mainly, if larger VME windows shall be allocated.

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

```
typedef struct
{
    unsigned long  addr;
    unsigned long  size;
    unsigned long  offs;
    int            space;
    int            width;
    int            windownr;
} VMECTRL_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
VMECTRL_A16	VMEbus A16 address space
VMECTRL_A24	VMEbus A24 address space
VMECTRL_A32	VMEbus A32 address space
VMECTRL_A64	VMEbus A64 address space

width

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

Value	Description
VMECTRL_D8	VMEbus D8 address space
VMECTRL_D16	VMEbus D16 address space
VMECTRL_D24	VMEbus D24 address space
VMECTRL_D32	VMEbus D32 address space
VMECTRL_D64	VMEbus D64 address space

windownr

This parameter returns the internal controller window number.

EXAMPLE

```
#include <sys/ioctl.h>
#include <vmectrl.h>

int                     fd;
VMECTRL_VME_WINDOW      VmeWindow;
int                     retval;

/*-------------------------------------------------
  Configure and allocate an A16/D16 VMEbus Region
  -------------------------------------------------*/
VmeWindow.addr     = 0x6000;
VmeWindow.size     = 0x1000;
VmeWindow.offs     = 0x6000;
VmeWindow.space    = VMECTRL_A16;
VmeWindow.width    = VMECTRL_D16;

…
```

```
…

retval = ioctl(fd, VMECTRL_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.

4.3.3.2 VMECTRL_IOCTL_FREE_REGION

NAME

VMECTRL_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 <sys/ioctl.h>
#include <vmectrl.h>

int      fd;
int      retval;

/*-------------------------------------------------
  Free a previously allocated VMEbus Region
  -------------------------------------------------*/
retval = ioctl(fd, VMECTRL_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 file handle.

5 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.

5.1 Installation

5.1.1 Build and install the Device Driver

- Login as *root*
- Change to the target sub-directory *generic_ipac*
- Make changes in configuration, if needed (see 5.3)
- 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**

5.1.2 Uninstall the Device Driver

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

  # make uninstall

5.1.3 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

5.1.4 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 gen_ipacdrv –r

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.

5.2 Create and Remove Generic Devices

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.

5.2.1 Create Device Node

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* in the generic device driver path to do this.

 # cd ./gen_ipacdrv
 # 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.

5.2.2 Delete Device Node Entry

The device node entries can be easily removed. Therefore just delete the entries in the /dev path.

 # rm /dev/gen_ipac_*

5.3 Configure Generic Driver

The configuration decides which kind of IPAC is supported, how the carrier board has to be set up, and the amount of memory space that has to be allocated.

After modification of the configuration the driver has to be rebuilt and restarted.

The configuration is made by four symbols that are defined in gen_ipacdef.h. Below you see the default configuration, which identifies a TEWS TECHNOLOGIES TIP675:

```
#define MANUFACTURER_ID    0xB3
#define MODULE_ID          0x36
#define SLOT_CONFIGURATION (IPAC_INT0_EN | IPAC_LEVEL_SENS | IPAC_CLK_8MHZ)
#define IPAC_MEMORY_SIZE   0
```

MANUFACTURER_ID

This symbol defines the manufacturer ID of the IPAC that shall be supported by the driver.

MODULE_ID

This symbol defines the model number of the IPAC that shall be supported by the driver.

SLOT_CONFIGURATION

This symbol is an ored value of definitions below:

Define	Description
IPAC_INT0_EN	This definition enables INT0 generated on the IPAC
IPAC_INT1_EN	This definition enables INT1 generated on the IPAC
IPAC_EDGE_SENS	This definition specifies that INT0 and INT1 are edge sensitive. (Only used if INT0 or INT1 enabled)
IPAC_LEVEL_SENS	This definition specifies that INT0 and INT1 are level sensitive. (Only used if INT0 or INT1 enabled)
IPAC_CLK_8MHZ	This definition specifies that the IPAC clock needs to be 8 MHz (Can not be selected in combination with IPAC_CLK_32MHZ)
IPAC_CLK_32MHZ	This definition specifies that the IPAC clock needs to be 32 MHz (Can not be selected in combination with IPAC_CLK_8MHZ)
IPAC_MEM_8BIT	This specifies that the selected memory is 8-bit memory (Can not be selected in combination with IPAC_MEM_16BIT)
IPAC_MEM_16BIT	This specifies that the selected memory is 16-bit memory (Can not be selected in combination with IPAC_MEM_8BIT)

IPAC_MEMORY_SIZE

This symbol specifies the size the IPAC needs for its IPAC memory space. The size must be specified in bytes. E.g. if the IPAC has a memory space of 4kByte, you have to specify 0x1000.

This symbol does not define the size of the ID- or I/O-space, these spaces are used automatically.

5.4 User Interface Functions

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

5.4.1 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

```
#include <fcntl.h>

int fd;

fd = open("/dev/gen_ipac_0", O_RDWR);
if (fd < 0)
{
     /* handle error */
}
```

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

Error code	Description
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

5.4.2 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

```
#include  <unistd.h>

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

Error code	Description
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

5.4.3 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 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

Error code	Description
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 generic IPAC driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

5.4.3.1 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 int        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 <sys/ioctl.h>
#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.

5.4.3.2 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 int    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 <sys/ioctl.h>
#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.

5.4.3.3 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 int    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 <sys/ioctl.h>
#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 int) );
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.

5.4.3.4 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 int    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 <sys/ioctl.h>
#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.

5.4.3.5 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 int    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 <sys/ioctl.h>
#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.

5.4.3.6 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 int    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 <sys/ioctl.h>
#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 int) );
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.

5.4.3.7 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 <sys/ioctl.h>
#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.

5.4.3.8 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 <sys/ioctl.h>
#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.

6 Appendix

6.1 Known Issues

6.1.1 Devices are not created

If all drivers (carrier, class and IPAC driver) are installed and one or more of the expected device nodes are not created, it may be possible that the TEWS device is handled by a foreign driver. This may be a driver supplied by the kernel or a manually installed driver. Please check if the desired device is assigned to the TEWS device driver properly.

You can check the driver assignment by using the command *lspci –v*. The first two lines of an output block identify the board and its position, the last line of the block shows the assigned driver. The shown text for device and system depends on the Linux installation and may differ between various systems. Below is an example output for a TPCI100 and the correctly installed carrier driver:

```
04:01.0 Bridge: TEWS Technologies GmbH TPCI100 (2 Slot IndustryPack PCI Carrier)
        Subsystem: TEWS Technologies GmbH Device 300a
        Flags: medium devsel, IRQ 16
        Memory at feb9fc00 (32-bit, non-prefetchable) [size=128]
        I/O ports at e880 [size=128]
        Memory at feb9f800 (32-bit, non-prefetchable) [size=256]
        Memory at feb9f400 (32-bit, non-prefetchable) [size=512]
        Memory at fd000000 (32-bit, non-prefetchable) [size=16M]
        Memory at fe000000 (32-bit, non-prefetchable) [size=8M]
        Kernel driver in use: TEWS TECHNOLOGIES - (Compact)PCI IPAC Carrier
```

If there is an invalid driver assigned, the automatic start of the driver module must be suppressed. To suppress the automatic start of the driver module, add the following lines to the blacklist file /etc/modprobe.d/blacklist.conf:

```
# do not start the driver automatically
blacklist <driver module>
```

If the black listed driver is also needed (e.g. for another device), it may be started after the TEWS driver has been started.

Only drivers installed as modules can be prevented from automatic start. There is no way to suppress the start of drivers configured into the kernel statically.

6.2 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
	TPCE200	PCIexpress 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
	TAMC200	AMC Carrier for 3 IndustryPack® modules
	TAMC220	AMC Carrier for 3 IndustryPack® modules
	TVME230	PCI Expansion Card for 4 IndustryPack® Modules
	TVME8240	Local IP slots of the TVME8240 CPU board
	TVME8300	Local IP slots of the TVME8300 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 (*1)	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

(*1) only in combination with a VMEbus controller driver

6.3 Supported VMEbus Controller

Driver	VMEbus controller	Descrition
universedrv	Universe®	Tundra Universe® VMEbus Controller
tsi148drv	TSI148®	Tundra TSI148 VMEbus Controller

6.4 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. The order of search depends on some different factors, e.g. BIOS, used kernel, etc.

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.

6.5 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} };
```

6.6 Support of non-conformal IndustryPack Modules

If an IndustryPack® module has a bad IP-PROM CRC it is not recognized by carrier driver. To give a chance of using these modules, we have defined a symbol that allows the use of these modules by ignoring CRC errors. By default the symbol is undefined and the CRC will be checked.

If CRC of the ID-Prom shall be ignored, modify carrier_class.c. Find the source line:

```
#undef IGNORE_CRC_ERROR
```

and change it into:

```
#define IGNORE_CRC_ERROR
```

6.7 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.

6.7.1 /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 2.0.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 V2.0.x
```

6.7.2 Debug Statements

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 2.0.x (<Release-Date>)

TEWS TECHNOLOGIES - Default Carrier version 2.0.x (<Release-Date>)

TEWS TECHNOLOGIES - (Compact)PCI IPAC Carrier version 2.0.x (<Release-Date>)

TIP903 - 3 Channel Extended CAN Bus IP - version 2.0.0 (<Release-Date>)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.