The Embedded I/O Company

CARRIER-SW-65

Windows 2000/XP Device Driver

IPAC-Carrier

Version 1.4.x

User Manual

Issue 1.4.0

December 2008

TEWS TECHNOLOGIES GmbH
Am Bahnhof 7
25469 Halstenbek, Germany
www.tews.com
Phone: +49 (0) 4101 4058 0
Fax: +49 (0) 4101 4058 19
e-mail: info@tews.com

TEWS TECHNOLOGIES LLC
9190 Double Diamond Parkway,
Suite 127, Reno, NV 89521, USA
www.tews.com
Phone: +1 (775) 850 5830
Fax: +1 (775) 201 0347
e-mail: usasales@tews.com

CARRIER-SW-65

IPAC-Carrier

Windows 2000/XP Device Driver

Issue	Description	Date
1.0	First Issue	October 22, 2003
1.1	Description for Generic Driver added / File-list extended	December 18, 2003
1.2	Description for VME Support added, Win XP Installation Description added, Win89/Me Installation Description removed	May 26, 2004
1.1.3	Description of Installation modified	November 1, 2004
1.1.4	File list changed	July 13, 2005
1.1.5	List of supported modules added, file list changed	July 03, 2006
1.2.0	TAMC100 support added, New address TEWS LLC	May 23, 2008
1.2.1	Files moved to subdirectory	June 20, 2008
1.3.0	Support of IPAC ID-PROM Type II (VITA4) added	October 17, 2008
1.4.0	Description of IPDrvCustom (Custom IPAC driver) added, TAMC200 support added	December 23, 2008

Table of Content

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 driver 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 drivers are split into three layers. The first layer handles accesses to the IPAC-Carrier cards. Different drivers are needed for different carriers. (We have implemented drivers for TEWS, SBS Carriers and VME Carrier boards with a Universe Master). These drivers are configuring the carrier boards and provide a function interface for the second layer. The second layer creates a device for every installed IP-slot used or unused, all slots are using the same driver. The IPAC-slots are checked, if an IPAC is mounted and for all mounted IPACs functions are provided for the IPAC drivers. The last layer is the IPAC driver which is handling the function of the IPAC. The IPAC driver must use the IPAC-slot functions for hardware accesses.

The CARRIER-SW-65 supports the modules listed below:

TEWS TPCI100	Carrier for 2 IndustryPack® modules	(PCI)
TEWS TPCI200	Carrier for 4 IndustryPack® modules	(PCI)
TEWS TCP201	Carrier for 4 IndustryPack® modules	(compactPCI)
TEWS TCP211	Carrier for 2 IndustryPack® modules	(compactPCI)
TEWS TCP212	Carrier for 2 IndustryPack® modules	(compactPCI)
TEWS TCP213	Carrier for 2 IndustryPack® modules	(compactPCI)
TEWS TCP220	Carrier for 4 IndustryPack® modules	(compactPCI)
TEWS TAMC100	Carrier for 1 IndustryPack® module	(AMC)
TEWS TAMC200	Carrier for 3 IndustryPack® modules	(AMC)
SBS PCI40(B)	Carrier for 4 IndustryPack® modules	(PCI)
SBS PCI60	Carrier for 6 IndustryPack® modules	(PCI)
SBS cPCI100/200	Carrier for 2/4 IndustryPack® modules	(compactPCI)
Universe 2	All VME-bus IPAC carrier	(VME)

2 Installation

Following files are located in directory CARRIER-SW-65 on the distribution media:

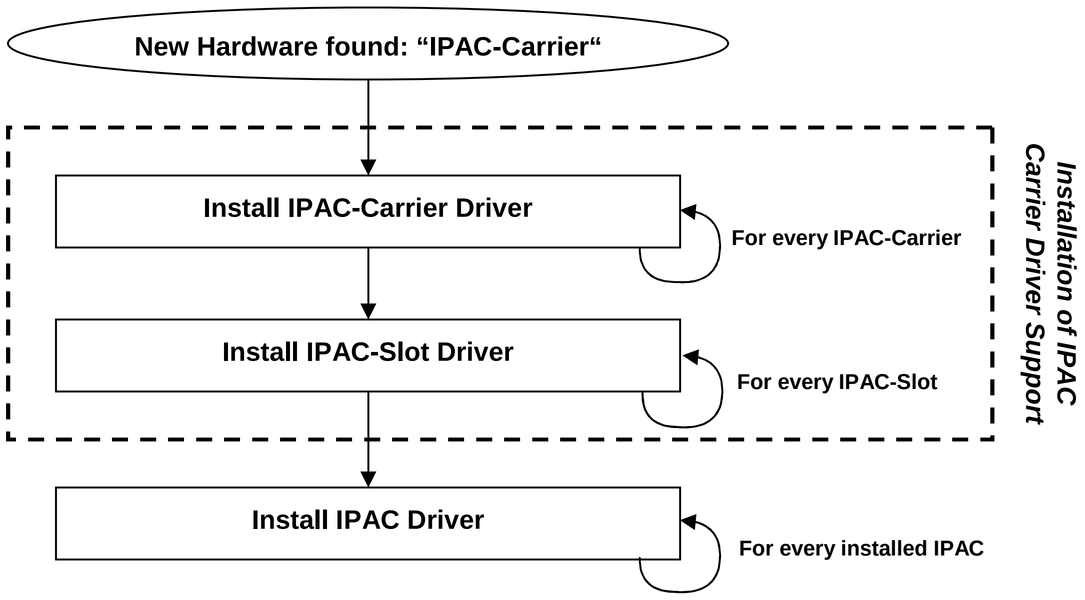
IPACBusFilter.sys	Device driver binary for IPAC Slot Driver
IPACBusFilter.inf	Installation script for IPAC Slot Driver
TEWSIPBus.sys	Device driver binary for TEWS Carrier Driver
TEWSIPBus.inf	Installation script for TEWS Carrier Driver
SBSIPBus.sys	Device driver binary for SBS Carrier Driver
SBSIPBus.inf	Installation script for SBS Carrier Driver
UVmeIpBus.sys	Device driver binary for Universe2 VME Carrier Driver
UVmeIpBus.inf	Installation script for Universe2 VME Carrier Driver
UVmeIpBus.reg	Registry configuration script for Universe2 VME Carrier Driver
genIPDrv.sys	Device driver binary for Generic IPAC Driver
genIPDrv.h	Application Include File for Generic IPAC Driver
genIPDrv.inf	Installation script for Generic IPAC Driver
Example/main.c	Example Application using the Generic IPAC Driver)
IPDrvCustom/*.*	Custom IPAC driver example (Source code)
IPDrvCustom/lib/ipacLib.lib	IPAC function library files (Library and includes)
IPDrvCustom/example/*.*	Example Application for Custom IPAC driver example
CARRIER-SW-65-1.4.0.pdf	This document
Release.txt	Release information
ChangeLog.txt	Release history

2.1 Software Installation

When installing the Universe VME carrier driver, be sure there is no other driver installed for the Universe PCI to VME Bridge.

2.1.1 General Installation Information

The Installation of the Carrier Driver Software has to be performed in two steps. Both steps must be done before the IPAC-Driver installation starts.



2.1.2 Windows 2000

This section describes how to install the IPAC-Carrier Device Drivers on a Windows 2000 operating system.

After installing the IPAC Carrier board(s) and boot-up your system, Windows 2000 setup will show a "***New hardware found***" dialog box.

The "***Upgrade Device Driver Wizard***" dialog box will appear on your screen.
Click "***Next***" button to continue.

In the following dialog box, choose "***Search for a suitable driver for my device***".
Click "***Next***" button to continue.

Insert the IPAC-Carrier driver disk; and select "***Disk Drive***" and/or "***CD-ROM***" in the dialog box. Click "***Next***" button to continue.

Now the driver wizard should find a suitable device driver on the diskette.
Click "***Next***" button to continue.

Completing the upgrade device driver and click "***Finish***" to take all the changes effect.

Repeat the Instructions until drivers for all IPAC Carrier Boards are installed.

For VME Carrier we have to configure the resources now. (See *2.2 Configure VME-Carrier Driver*)

The next step to do is to install the IPAC Slot Drivers. Simply repeat the steps once more.

Now copy all needed files (CARRIER-SW-65.pdf) to the desired target directories.

After successful installation the IPAC-Carrier device driver will start immediately and create devices for all recognized carriers, IPAC-slots and mounted IPACs.

2.1.3 Windows XP

This section describes how to install the IPAC-Carrier Device Drivers on a Windows XP operating system.

After installing the IPAC Carrier board(s) and boot-up your system, Windows XP setup will show the search for new hardware window.
Insert the IPAC-Carrier driver disk and choose "***Automatic Software Install***".
Click "***Next***" button to continue.

A window will announce that the Windows-Logo test has failed.
Click "***continue install***" button to continue.

A window will announce that the driver has been installed.
Click "***Finish***" to take all the changes effect.

Repeat the Instructions until drivers for all IPAC Carrier Boards are installed.

For VME Carrier we have to configure the resources now. (See *2.2 Configure VME-Carrier Driver*)

The next step to do is to install the IPAC Slot Drivers. Simply repeat the steps before again, until all IPAC Slots are connected with a driver.

Now copy all needed files (CARRIER-SW-65.pdf) to the desired target directories.

After successful installation the IPAC-Carrier device driver will start immediately and create devices for all recognized carriers, IPAC-slots and mounted IPACs.

2.1.4 Confirming Windows 2000/XP Installation

To confirm that the driver has been properly loaded in Windows 2000/XP, perform the following steps:

From Windows 2000/XP, open the "***Control Panel***" from "My Computer".

Click the "***System***" icon and choose the "Hardware" tab, and then click the "***Device Manager***" button.

Click the "**+**" in front of "***Multifunction Devices***".
The drivers for the IPAC-Carrier Boards should appear and also an IPAC-Slot driver for each of the IPAC-Slots on the installed IPAC-Carrier Boards.

2.2 Configure VME-Carrier Driver

The VME-Bus doesn't support plug and play, so there must be a manual configuration for the VME IPAC slots. This configuration is done in the *UVmeIpBus.reg* file. This file splits into three parts: "*VMEbus Interface Configuration*", "*VMEbus Master Window Configuration*" and "VMEbus IPAC Slot Configuration"

After installation of the VME-Carrier Driver the *UVmeIpBus.reg* file must be installed to the system. Follow these steps to make the installation.

Modify *UVmeIpBus.reg* with a standard text editor.
Click right to the file and choose "***Merge***" from the context menu.
Restart the driver to complete the changes, or restart the system.

You will find the values in the windows registry in the following path and the subpathes:

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\UVmeIpBus]

You can also change values in the registry using *regedit*.

After every change of the registry values, the driver has to be restarted to get the new values.

2.2.1 VMEbus Interface Configuration

This part specifies the VME Controller Master setup.

For more detailed information of the values please refer to the VME-specification or the Universe manual.

The registry path is:

```
[…\VMEbus]
```

Default configuration:

```
"VMEbusRequestMode"=dword:00000000
"VMEbusReleaseMode"=dword:00000000
"VMEbusArbitrationMode"=dword:00000001
"VMEbusArbitrationTimeout"=dword:00000010
"VMEbusTimeout"=dword:00000040
"VMEbusRequestLevel"=dword:00000003
"NumberOfRetries"=dword:00000008
"PostedWriteTransferCount"=dword:00000200
"SYSCON"=dword:00000002
```

Description

VMEbusRequestMode

Value (hex)	**Mode**
0	Demand
1	Fair

VMEbusReleaseMode

Value (hex)	**Mode**
0	Release When Done (RWD)
1	Release on Request (ROR)

VMEbusArbitrationMode

Value (hex)	**Mode**
0	Round Robin
1	Priority

VMEbusArbitrationTimeout

Value (hex)	**Timeout**
0	0 µs
10	16 µs
100	256 µs

VMEbusTimeout

Value (hex)	**Timeout**
0	disabled
10	16 µs
20	32 µs
40	64 µs
80	128 µs
100	256 µs
200	512 µs
400	1024 µs

VMEbusRequestLevel

Value (hex)	**Request Level**
0	0
1	1
2	2
3	3

NumberOfRetries

Specifies the number of retries multiplied by 64. (0 – retry for ever, 1 – 64 retries, …) The value must be between 0h and Fh.

PostedWriteTransferCount

Value (hex)	Bytes
80	128
100	256
200	512
400	1024
800	2048
1000	4096

SYSCON

Value (hex)	System controller mode
0	NOT System Controller
1	System Controller
2	AUTO

2.2.2 VMEbus Master Window Configuration

This part specifies the Master Window configuration of the eight windows.

The registry path is:

`[…\VMEbus\Window`***n***`]`
n specifies the window number.

Default configuration:

```
[…\VMEbus\Window1]
; Window1 : A16/D16
"Enabled"=dword:00000001
"VMEbusBaseAddress"=dword:00000000
"WindowSize"=dword:00010000
"AddressModifier"=dword:00000029
"DataWidth"=dword:00000010

[…\VMEbus\Window2]
; Window1 : A24/D16
"Enabled"=dword:00000001
"VMEbusBaseAddress"=dword:00000000
"WindowSize"=dword:01000000
"AddressModifier"=dword:00000039
"DataWidth"=dword:00000010
```

Description

Enabled

Value (hex)	Description
0	Disable window
1	Enable window

VMEbusBaseAddress

VME-bus window start address

WindowSize

Window size in bytes

AddressModifier

Value (hex)	Access mode
9	A32 non-privileged data access
A	A32 non-privileged program access
D	A32 supervisory data access
E	A32 supervisory program access
29	A16 non-privileged access
2D	A16 supervisory access
39	A24 non-privileged data access
3A	A24 non-privileged program access
3D	A24 supervisory data access
3E	A24 supervisory program access

DataWidth

Value (hex)	Datawidth
8	8 bit
10	16 bit
2	32 bit

2.2.3 VMEbus IPAC Slot Configuration

This part specifies the IPAC Slot configuration of every IPAC slot on VME-bus. Each slot must get an own entry. These values mainly depend on the configuration of the VME IPAC carrier.

The registry path is:

`[…\VMEbus\Slot`***n***`]`

n specifies the slot number.

Default configuration:

```
[…\VMEbus\Slot1]

"Enabled"=dword:00000001

"ID_Address"=dword:00006080
"ID_Size"=dword:00000080
"ID_Window"=dword:00000001

"IO_Address"=dword:00006000
"IO_Size"=dword:00000080
"IO_Window"=dword:00000001

"MEM_Address"=dword:00D00000
"MEM_Size"=dword:00040000
"MEM_Window"=dword:00000002

"BaseVector"=dword:000000A0
"Level_INT0"=dword:00000001
"Level_INT1"=dword:00000002
```

Description

Enabled

Value (hex)	Description
0	Disable slot
1	Enable slot

ID_Address

Specifies the address offset of the ID space in the specified window.

ID_Size

Specifies the size of the ID space in bytes.

ID_Window

Specifies the window number the ID space is mapped to. The window number must be between 1 and 8. (see *2.2.2 VMEbus Master Window Configuration*)

IO_Address

Specifies the address offset of the I/O space in the specified window.

IO_Size

Specifies the size of the I/O space in bytes.

IO_Window

Specifies the window number the I/O space is mapped to. The window number must be between 1 and 8. (see *2.2.2 VMEbus Master Window Configuration*)

MEM_Address

Specifies the address offset of the memory space in the specified window.

MEM_Size

Specifies the size of the memory space in bytes.

MEM_Window

Specifies the window number the memory space is mapped to. The window number must be between 1 and 8. (see *2.2.2 VMEbus Master Window Configuration*)

BaseVector

Specifies the interrupt base vector for this IPAC slot, 8 vectors are reserved for each of the slots. Valid vectors are between 40h and F8h

Level_INT0

Specifies the VME interrupt level INT0 will generate on the VME Bus. Valid VME interrupt levels are between 1 and 7. (0 – no interrupt is generated on INT0)

Level_INT1

Specifies the VME interrupt level INT1 will generate on the VME Bus. Valid VME interrupt levels are between 1 and 7. (0 – no interrupt is generated on INT1)

3 Customer IPAC Carrier Support

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

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

4 Generic IPAC Driver

The Generic Driver can be used for first steps using an IPAC. The driver allows access to the IPAC. The module interface (IP-Clockrate, Space size ...) can be configured by the application. Only interrupts are not implemented.

The Generic Driver allows following functions:

- writing bytes, shorts and longwords
- reading bytes, shorts and longwords
- configuring the IPAC-slot (allocate)
- unconfiguring the IPAC-slot (free)

4.1 Installation

4.1.1 Before Installation

To use the driver for a specific IPAC, copy the driver files to a local directory and modify the installation-file. The supported modules are identified by the hardware identifier. For the IPAC modules identified by the IPAC Carrier driver this will be 'IPACSlot\' followed by the IPAC Name. For undefined modules it will be 'IPACSlot\' followed by 'MAN<manufacturer ID>_MOD<model number> where manufacturer ID and model number are read from the IPACs ID-Prom. If the IPAC has an 8-bit ID-Prom ("IPAC") manufacturer and model number will have a length of 2 characters, a 16-bit ID-Prom (type II "VITA4") will have a manufacturer ID length of 6 characters and the model number will have a length of 4 characters.

Find the following line.

```
[TEWS.Mfg]
```

This line identifies a list of hardware identifiers, specifying the IPACs the driver will handle.

The TIP255 (identified) will be handled by the following example.

```
[TEWS.Mfg]
%TTG.SvcDesc% = TTG,IPACSlot\TIP255
```

If the TIP255 has not been identified the following example the following modification will handle it.

```
[TEWS.Mfg]
%TTG.SvcDesc% = TTG, IPACSlot\MANb3_MOD31
```

The following example shows an example if two modules shall be handled by the driver.

```
[TEWS.Mfg]
%TTG.SvcDesc% = TTG,IPACSlot\TIP255
%TTG.SvcDesc% = TTG,IPACSlot\MANb3_MOD30
```

If a module with an ID-Prom type II (VITA4 format) shall be used the following modification will handle it. (Manufacturer ID: 12ab56 --- Module Number: 34cd)

```
[TEWS.Mfg]
%TTG.SvcDesc% = TTG, IPACSlot\MAN12ab56_MOD34cd
```

4.1.2 Windows 2000

This section describes how to install the Generic IPAC Device Driver on a Windows 2000 operating system.

1. After installing the IPAC card(s) and boot-up your system, Windows 2000 setup will show a "***New hardware found***" dialog box.
2. The "***Upgrade Device Driver Wizard***" dialog box will appear on your screen.
   Click "***Next***" button to continue.
3. In the following dialog box, choose "***Search for a suitable driver for my device***".
   Click "***Next***" button to continue.
4. Select local path with the modified inf-File.
   Click "***Next***" button to continue.
5. Now the driver wizard should find a suitable device driver.
   Click "***Next***" button to continue.
6. Completing the upgrade device driver and click "***Finish***" to take all the changes effect.

After successful installation the Generic IPAC device driver will start immediately and creates devices (genIPDrv_1, genIPDrv_2, ...) for all recognized IPAC modules.

4.1.3 Confirming Windows 2000 Installation

To confirm that the driver has been properly loaded in Windows 2000, perform the following steps:

1. From Windows 2000, open the "***Control Panel***" from "***My Computer***".
2. Click the "***System***" icon and choose the "***Hardware***" tab, and then click the "***Device Manager***" button.
3. Click the "**+**" in front of "***Other Devices***".
   The driver "***genIPDrv***" should appear.

4.1.4 Windows 98 SE / Windows ME

This section describes how to install the Generic IPAC Device Driver on a Windows 98 Second Edition (SE) operating system.

1. After installing the IPAC card(s) and boot-up your system, Windows 98 SE setup will show a "***New hardware found***" dialog box.
2. The "***Add New Hardware Wizard***" dialog box will appear on your screen, informing you that it has found a new PCI device.
   Click "***Next***" button to continue.
3. In the following dialog box, choose "***Search for a better driver than the one your device is using now***".
   Click "***Next***" button to continue.
4. In the following dialog box, select "***Specify a location***".
5. Select local path with the modified inf-File.
   Click "***Next***" button to continue.

After successful installation the Generic IPAC device driver will start immediately and creates devices (genIPDrv_1, genIPDrv_2, ...) for all recognized IPAC modules.

4.1.5 Confirming Windows 98 SE / Windows ME Installation

To confirm that the driver has been properly loaded in Windows, perform the following steps:

1. Choose "***Settings***" from the "***Start***" menu.
2. Choose "***Control Panel***" and then double-click on the "***System***" icon.
3. Choose the "***Device Manager***" tab, and then click the "**+**" in front of "***Other Devices***".
   The driver "***genIPDrv***" should appear.

4.2 Generic IPAC Device Driver Programming

The Generic IPAC WDM device driver is a kernel mode device driver.

The standard file and device (I/O) functions (CreateFile, CloseHandle, and DeviceIoControl) provide the basic interface for opening and closing a device handle and for performing device I/O control operations.

All of these standard Win32 functions are described in detail in the Windows Platform SDK Documentation (Windows base services / Hardware / Device Input and Output).

For details refer to the Win32 Programmers Reference of your used programming tools (C++, Visual Basic etc.)

4.2.1 IPAC Files and I/O Functions

The following section doesn't contain a full description of the Win32 functions for interaction with the Generic IPAC device driver. Only the required parameters are described in detail.

4.2.1.1 Opening an IPAC Device

Before you can perform any I/O the IPAC device must be opened by invoking the **CreateFile** function. **CreateFile** returns a handle that can be used to access the IPAC device.

```
HANDLE CreateFile(
    LPCTSTR lpFileName,                           // pointer to filename
    DWORD dwDesiredAccess,                        // access (read-write) mode
    DWORD dwShareMode,                            // share mode
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,   // pointer to security attributes
    DWORD dwCreationDistribution,                 // how to create
    DWORD dwFlagsAndAttributes,                   // file attributes
    HANDLE hTemplateFile                          // handle to file with attributes to copy
);
```

Parameters

lpFileName

Points to a null-terminated string that specifies the name of the IPAC to open.
The *lpFileName* string should be of the form **\\.\genIPDrv_*x*** to open the device *x.* The ending *x* is a one-based number. The first device found by the driver is \\.\genIPDrv_1, the second \\.\genIPDrv_2 and so on.

dwDesiredAccess

Specifies the type of access to the IPAC. For a Generic IPAC this parameter must be set to read-write access (GENERIC_READ **|** GENERIC_WRITE).

dwShareMode

A set of bit flags that specifies how the object can be shared for read and write. Unimportant for Generic IPAC, set to 0.

lpSecurityAttributes

Pointer to a security structure. Set to NULL for generic IPAC devices.

dwCreationDistribution

Specifies which action to take on files that exist and which action to take when files that do not exist. Generic IPAC devices must be always opened *OPEN_EXISTING*.

dwFlagsAndAttributes

Specifies the file attributes and flags for the file. This value must be set to 0 (no overlapped I/O).

hTemplateFile

This value must be 0 for Generic IPAC devices.

Return Value

If the function succeeds, the return value is an open handle to the specified IPAC device. If the function fails, the return value is INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Example

```
HANDLE   hDevice;

hDevice = CreateFile(
     “\\\\.\\genIPDrv_1”,
     GENERIC_READ | GENERIC_WRITE,
     0,
     NULL,              // no security attrs
     OPEN_EXISTING,     // IPAC device always open existing
     0,                 // no overlapped I/O
     NULL
);
if (hDevice == INVALID_HANDLE_VALUE) {
     ErrorHandler("Could not open device");     // process error
}
```

See Also

CloseHandle(), Win32 documentation CreateFile()

4.2.1.2 Closing an IPAC Device

The **CloseHandle** function closes an open IPAC handle.

```
BOOL CloseHandle(
    HANDLE hDevice;                    // handle to a IPAC device to close
);
```

Parameters

hDevice

Identifies an open IPAC handle.

Return Value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call ***GetLastError*****.**

Example

```
HANDLE   hDevice;

hDevice = CreateFile(
     “\\\\.\\geiIPDrv_1”,
     GENERIC_READ | GENERIC_WRITE,
     0,
     NULL,              // no security attrs
     OPEN_EXISTING,     // IPAC device always open existing
     0,                 // no overlapped I/O
     NULL
);
if(hDevice == INVALID_HANDLE_VALUE) {
 ErrorHandler("Could not open device"); // process error
}

/* ... do some device I/O ...  */

if(!CloseHandle(hDevice)) {
     ErrorHandler("Could not close device");     // process error
}
```

See Also

CreateFile(), Win32 documentation CloseHandle()

4.2.1.3 IPAC Device I/O Control Functions

The **DeviceIoControl** function sends a control code directly to a specified device driver, causing the corresponding device to perform the specified operation.

```
BOOL DeviceIoControl(
        HANDLE hDevice,                         // handle to device of interest
        DWORD dwIoControlCode,                  // control code of operation to perform
        LPVOID lpInBuffer,                      // pointer to buffer to supply input data
        DWORD nInBufferSize,                    // size of input buffer
        LPVOID lpOutBuffer,                     // pointer to buffer to receive output data
        DWORD nOutBufferSize,                   // size of output buffer
        LPDWORD lpBytesReturned,                // pointer to variable to receive output byte
count
        LPOVERLAPPED lpOverlapped               // pointer to overlapped structure for
asynchronous                                                // operation
);
```

Parameters

hDevice

Handle to the IPAC that is to perform the operation.

dwIoControlCode

Specifies the control code for an operation. This value identifies the specific operation to be performed. The following values are defined in *genIPDrv.h*:

Value	Meaning
GENIPDRV_CONFIGURE	Configure Carrier IPAC Slot and allocate IPAC addresses
GENIPDRV_UNCONFIGURE	Release IPAC addresses
GENIPDRV_READ_UCHAR	Get data from IPAC Device (8-bit accesses)
GENIPDRV_READ_USHORT	Get data from IPAC Device (16-bit accesses)
GENIPDRV_READ_ULONG	Get data from IPAC Device (32-bit accesses)
GENIPDRV_WRITE_UCHAR	Write data to IPAC Device (8-bit accesses)
GENIPDRV_WRITE_USHORT	Write data to IPAC Device (16-bit accesses)
GENIPDRV_WRITE_ULONG	Write data to IPAC Device (32-bit accesses)

See behind for more detailed information on each control code.

To use these Generic IPAC specific control codes the header file genIPDrv.h must be included in the application.

lpInBuffer

Pointer to a buffer that contains the data required to perform the operation.

nInBufferSize

Specifies the size, in bytes, of the buffer pointed to by *lpInBuffer*.

lpOutBuffer

Pointer to a buffer that receives the operation's output data.

nOutBufferSize

Specifies the size, in bytes, of the buffer pointed to by *lpOutBuffer*.

lpBytesReturned

Pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by *lpOutBuffer*. A valid pointer is required.

lpOverlapped

Pointer to an *Overlapped* structure. This value must be set to NULL (no overlapped I/O).

Return Value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call ***GetLastError***.

See Also

Win32 documentation DeviceIoControl()

4.2.1.4 GENIPDRV_CONFIGURE

The configure function allocates address spaces to access the IPAC and sets up the IPAC slot. This function must be used before any other access to the IPAC is done.

The parameter *lpInBuffer* and *lpOutBuffer* must pass a pointer to the configuration buffer (*GENIPDRV_CONFIGURE_BUF*) to the device driver.

```
typedef struct _GENIPDRV_CONFIGURE_BUF
{
        BOOLEAN         enable8BitIP;       // TRUE - 8 Bit IP Bus
                                            // FALSE - 16 Bit IP Bus    (startup)
        BOOLEAN         enable32MHz;        // TRUE - 32 MHz IP Clock
                                            // FALSE - 8 MHz IP Clock   (startup)
        ULONG           sizeIDSpace;        // Size of ID-Space
        BOOLEAN         swapIDSpace;        // Little <-> Big Endian swapping enable
        ULONG           sizeIOSpace;        // Size of I/O-Space
        BOOLEAN         swapIOSpace;        // Little <-> Big Endian swapping enable
        ULONG           sizeMEMSpace;       // Size of Memory-Space
        BOOLEAN         swapMEMSpace;       // Little <-> Big Endian swapping enable
} GENIPDRV_CONFIGURE_BUF, *PGENIPDRV_CONFIGURE_BUF;
```

enable8BitIP

This parameter specifies the width of the IPAC bus. (If supported by hardware)

Value	Description
TRUE	The IPAC bus will be configured as 8-bit bus
FALSE	The IPAC bus will be configured as 16-bit bus

enable32MHz

This parameter specifies the IPAC clock speed. (If supported by hardware)

Value	Description
TRUE	The IPAC clock speed will be set to 32MHz
FALSE	The IPAC clock speed will be set to 8MHz

sizeIDSpace

This parameter must specify the needed size of the ID-space.

swapIDSpace

This parameter specifies if the data should be swapped for read and write accesses to ID-space.

Value	Description
TRUE	Data will be swapped
FALSE	Data will not be swapped

sizeIOSpace

This parameter must specify the needed size of the I/O-space.

swapIOSpace

This parameter specifies if the data should be swapped for read and write accesses to I/O-space.

Value	Description
TRUE	Data will be swapped
FALSE	Data will not be swapped

sizeMEMSpace

This parameter must specify the needed size of the Memory-space.

swapMEMSpace

This parameter specifies if the data should be swapped for read and write accesses to Memory-space.

Value	Description
TRUE	Data will be swapped
FALSE	Data will not be swapped

Example

```
#include                “genIPDrv.h”

GENIPDRV_CONFIGURE_BUF  configBuf;
HANDLE                  hDevice;
BOOLEAN                 success;
ULONG                   NumBytes;
```

…

```
…

// Configure IPAC slot:
//    - 16-bit bus width
//    - 32 MHz clock speed
//    - 32 Byte ID-Space (not swapped)
//    - 18 Byte I/O-Space (swapped)
//    - no Memory-Space
configBuf.enable8BitIP =    FALSE;
configBuf.enable32MHz =     TRUE;
configBuf.sizeIDSpace =     32;
configBuf.swapIDSpace =     FALSE;
configBuf.sizeIOSpace =     18;
configBuf.swapIOSpace =     TRUE;
configBuf.sizeMEMSpace =    0x0;
configBuf.swapMEMSpace =    FALSE;

//  Send request to the device driver
//
success = DeviceIoControl (
                   hCurrent,              // IPAC handle
                   GENIPDRV_CONFIGURE,    // control code
                   &configBuf,
                   sizeof(GENIPDRV_CONFIGURE_BUF),
                   &configBuf,
                   sizeof(GENIPDRV_CONFIGURE_BUF),
                   &NumBytes,             //  number of bytes transferred
                   NULL                   //  not over lapped
              );
if( success ) {
     // IPAC-Slot Configured
}
else {
     // IPAC-Slot Configuration failed
}
```

Error Codes

All returned error codes are system error conditions.

See Also

Win32 documentation DeviceIoControl()

4.2.1.5 GENIPDRV_UNCONFIGURE

The unconfigure function releases address spaces allocater for IPAC access.

The parameter *lpInBuffer* and *lpOutBuffer* must pass a *NULL* pointer to the device driver.

Example

```
#include                “genIPDrv.h”

HANDLE                  hDevice;
BOOLEAN                 success;
ULONG                   NumBytes;

success = DeviceIoControl (
                   hCurrent,               // IPAC handle
                   GENIPDRV_UNCONFIGURE,   // control code
                   NULL,
                   0,
                   NULL,
                   0,
                   &NumBytes,              // number of bytes transferred
                   NULL                    // not over lapped
              );
//
// Check the result of the last device I/O control operation
//
if( success ) {
     // IPAC-Slot unconfigured
}
else {
     // IPAC-Slot Unconfiguration failed
}
```

Error Codes

All returned error codes are system error conditions.

See Also

Win32 documentation DeviceIoControl()

4.2.1.6 GENIPDRV_READ_UCHAR

The read uchar function reads a buffer of 8-bit data from a specified address space. Before this function is used the GENIPDRV_CONFIGURE function must be called.

The parameter *lpInBuffer* and *lpOutBuffer* must pass a pointer to the I/O buffer (*GENIPDRV_IO_BUF*) to the device driver.

```
typedef struct _GENIPDRV_IO_BUF
{
        UCHAR           space;                  // address space to read from
        ULONG           offset;                 // address offset in space
        ULONG           size;                   // number of uchar/ushort/ulong
        UCHAR           buffer[GENIPDRV_MAXIOBUF];  // pointer to buffer
} GENIPDRV_IO_BUF, *PGENIPDRV_IO_BUF;
```

space

This parameter specifies the address of the IPAC the data shall be read from.

Value	Description
GENIPDRV_IDSPACE	Read from ID Space
GENIPDRV_IOSPACE	Read from I/O space
GENIPDRV_MEMSPACE	Read from memory space

offset

This parameter specifies the starting offset in the selected space.

size

This parameter specifies the length of the buffer to read.

buffer[]

This array will be filled with the data read from the specified position. The size of the buffer can be changed by changing the value of *GENIPDRV_MAXIOBUF* in genIPDrv.h. The value is specified in byte.

Example

```
#include            “genIPDrv.h”

GENIPDRV_IO_BUF     ioBuf;
HANDLE              hDevice;
BOOLEAN             success;
ULONG               NumBytes;
PUCHAR              ucPtr;


…
```

```
…

// Read 16 Bytes from IPAC ID-Space starting at offset 0x10
ioBuf.space = GENIPDRV_IDSPACE;
ioBuf.offset =0x10;
ioBuf.size =  0x10;

//  Send request to the device driver
//
success = DeviceIoControl (
              hCurrent,                   // IPAC handle
              GENIPDRV_READ_UCHAR,        // control code
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              &NumBytes,                  // number of bytes transferred
              NULL                        // not over lapped
         );
if( success ) {
     // read access OK
     ucPtr = &ioBuf.Buffer[0];   // Set pointer to data
}
else {
     // read access failed
}
```

Error Codes

All returned error codes are system error conditions.

See Also

Win32 documentation DeviceIoControl()

4.2.1.7 GENIPDRV_READ_USHORT

The read ushort function reads a buffer of 16-bit data from a specified address space. Before this function is used the GENIPDRV_CONFIGURE function must be called.

The parameter *lpInBuffer* and *lpOutBuffer* must pass a pointer to the I/O buffer (*GENIPDRV_IO_BUF*) to the device driver.

```
typedef struct _GENIPDRV_IO_BUF
{
        UCHAR           space;                  // address space to read from
        ULONG           offset;                 // address offset in space
        ULONG           size;                   // number of uchar/ushort/ulong
        UCHAR           buffer[GENIPDRV_MAXIOBUF];  // pointer to buffer
} GENIPDRV_IO_BUF, *PGENIPDRV_IO_BUF;
```

space

This parameter specifies the address of the IPAC the data shall be read from.

Value	Description
GENIPDRV_IDSPACE	Read from ID Space
GENIPDRV_IOSPACE	Read from I/O space
GENIPDRV_MEMSPACE	Read from memory space

offset

This parameter specifies the starting offset in the selected space.

size

This parameter specifies the number of words to read.

buffer[]

This array will be filled with the data read from the specified position. The size of the buffer can be changed by changing the value of *GENIPDRV_MAXIOBUF* in genIPDrv.h. The value is specified in byte.

Example

```
#include            “genIPDrv.h”

GENIPDRV_IO_BUF     ioBuf;
HANDLE              hDevice;
BOOLEAN             success;
ULONG               NumBytes;
PUSHORT             usPtr;

…
```

```
…

// Read 8 Words from IPAC I/O-Space starting at offset 0x10
ioBuf.space = GENIPDRV_IOSPACE;
ioBuf.offset =0x10;
ioBuf.size =  0x8;

//  Send request to the device driver
//
success = DeviceIoControl (
              hCurrent,                   // IPAC handle
              GENIPDRV_READ_USHORT,       // control code
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              &NumBytes,                  // number of bytes transferred
              NULL                        // not over lapped
         );
if( success ) {
     // read access OK
     usPtr = (PUSHORT)&ioBuf.Buffer[0];   // Set pointer to data
}
else {
     // read access failed
}
```

Error Codes

All returned error codes are system error conditions.

See Also

Win32 documentation DeviceIoControl()

4.2.1.8 GENIPDRV_READ_ULONG

The read ulong function reads a buffer of 32-bit data from a specified address space. Before this function is used the GENIPDRV_CONFIGURE function must be called.

The parameter *lpInBuffer* and *lpOutBuffer* must pass a pointer to the I/O buffer (*GENIPDRV_IO_BUF*) to the device driver.

```
typedef struct _GENIPDRV_IO_BUF
{
    UCHAR          space;                     // address space to read from
    ULONG          offset;                    // address offset in space
    ULONG          size;                      // number of uchar/ushort/ulong
    UCHAR          buffer[GENIPDRV_MAXIOBUF]; // pointer to buffer
} GENIPDRV_IO_BUF, *PGENIPDRV_IO_BUF;
```

space

This parameter specifies the address of the IPAC the data shall be read from.

Value	Description
GENIPDRV_IDSPACE	Read from ID Space
GENIPDRV_IOSPACE	Read from I/O space
GENIPDRV_MEMSPACE	Read from memory space

offset

This parameter specifies the starting offset in the selected space.

size

This parameter specifies the number of longwords l to read.

buffer[]

This array will be filled with the data read from the specified position. The size of the buffer can be changed by changing the value of *GENIPDRV_MAXIOBUF* in genIPDrv.h. The value is specified in byte.

Example

```
#include            “genIPDrv.h”

GENIPDRV_IO_BUF     ioBuf;
HANDLE              hDevice;
BOOLEAN             success;
ULONG               NumBytes;
PULONG              ulPtr;

…
```

```
…

// Read 4 Longwords from IPAC Memory-Space starting at offset 0x10
ioBuf.space = GENIPDRV_MEMSPACE;
ioBuf.offset =0x10;
ioBuf.size =  0x4;

//  Send request to the device driver
//
success = DeviceIoControl (
              hCurrent,                    // IPAC handle
              GENIPDRV_READ_ULONG,         // control code
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              &NumBytes,                   // number of bytes transferred
              NULL                         // not over lapped
         );
if( success ) {
     // read access OKO
     ulPtr = (PULONG)&ioBuf.Buffer[0];     // Set pointer to data
}
else {
     // read access failed
}
```

Error Codes

All returned error codes are system error conditions.

See Also

Win32 documentation DeviceIoControl()

4.2.1.9 GENIPDRV_WRITE_UCHAR

The write uchar function writes a buffer of 8-bit data to a specified address space. Before this function is used the GENIPDRV_CONFIGURE function must be called.

The parameter *lpInBuffer* must pass a pointer to the I/O buffer (*GENIPDRV_IO_BUF*) to the device driver. The parameter *lpOutBuffer* must pass a *NULL* pointer to the device driver.

```
typedef struct _GENIPDRV_IO_BUF
{
        UCHAR           space;                          // address space to read from
        ULONG           offset;                         // address offset in space
        ULONG           size;                           // number of uchar/ushort/ulong
        UCHAR           buffer[GENIPDRV_MAXIOBUF];      // pointer to buffer
} GENIPDRV_IO_BUF, *PGENIPDRV_IO_BUF;
```

space

This parameter specifies the address of the IPAC the data shall be read from.

Value	Description
GENIPDRV_IDSPACE	Read from ID Space
GENIPDRV_IOSPACE	Read from I/O space
GENIPDRV_MEMSPACE	Read from memory space

offset

This parameter specifies the starting offset in the selected space.

size

This parameter specifies the length of the buffer to write.

buffer[]

This array must be filled with the data to write to the specified position. The size of the buffer can be changed by changing the value of *GENIPDRV_MAXIOBUF* in genIPDrv.h. The value is specified in byte.

Example

```
#include            “genIPDrv.h”

GENIPDRV_IO_BUF     ioBuf;
HANDLE              hDevice;
BOOLEAN             success;
ULONG               NumBytes;
PUCHAR              ucPtr;

…
```

```
…

// Write 3 Bytes (0x11,0x22,0x33) to IPAC I/O-Space starting at offset 0x10
ioBuf.space = GENIPDRV_IOSPACE;
ioBuf.offset =0x10;
ioBuf.size =  0x3;

ucPtr = &ioBuf.Buffer[0];   // Set pointer to data
ucPtr[0] = 0x11;
ucPtr[1] = 0x22;
ucPtr[2] = 0x33;

//  Send request to the device driver
//
success = DeviceIoControl (
              hCurrent,                  // IPAC handle
              GENIPDRV_WRITE_UCHAR,      // control code
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              NULL,
              0,
              &NumBytes,                 // number of bytes transferred
              NULL                       // not over lapped
          );
if( success ) {
     // write access OK
}
else {
     // write access failed
}
```

Error Codes

All returned error codes are system error conditions.

See Also

Win32 documentation DeviceIoControl()

4.2.1.10 GENIPDRV_WRITE_USHORT

The write ushort function writes a buffer of 16-bit data to a specified address space. Before this function is used the GENIPDRV_CONFIGURE function must be called.

The parameter *lpInBuffer* must pass a pointer to the I/O buffer (*GENIPDRV_IO_BUF*) to the device driver. The parameter *lpOutBuffer* must pass a *NULL* pointer to the device driver.

```
typedef struct _GENIPDRV_IO_BUF
{
    UCHAR       space;                      // address space to read from
    ULONG       offset;                     // address offset in space
    ULONG       size;                       // number of uchar/ushort/ulong
    UCHAR       buffer[GENIPDRV_MAXIOBUF];  // pointer to buffer
} GENIPDRV_IO_BUF, *PGENIPDRV_IO_BUF;
```

space

This parameter specifies the address of the IPAC the data shall be read from.

Value	Description
GENIPDRV_IDSPACE	Read from ID Space
GENIPDRV_IOSPACE	Read from I/O space
GENIPDRV_MEMSPACE	Read from memory space

offset

This parameter specifies the starting offset in the selected space.

size

This parameter specifies number of words to write.

buffer[]

This array must be filled with the data to write to the specified position. The size of the buffer can be changed by changing the value of *GENIPDRV_MAXIOBUF* in genIPDrv.h. The value is specified in byte.

Example

```
#include            “genIPDrv.h”

GENIPDRV_IO_BUF     ioBuf;
HANDLE              hDevice;
BOOLEAN             success;
ULONG               NumBytes;
PUSHORT             usPtr;


…
```

```
…

// Write 3 Words (0x1111,0x2222,0x3333) to IPAC I/O-Space starting
// at offset 0x10
ioBuf.space = GENIPDRV_IOSPACE;
ioBuf.offset =0x10;
ioBuf.size =  0x3;

usPtr = (PUSHORT)&ioBuf.Buffer[0];   // Set pointer to data
usPtr[0] = 0x1111;
usPtr[1] = 0x2222;
usPtr[2] = 0x3333;

//  Send request to the device driver
//
success = DeviceIoControl (
              hCurrent,                  // IPAC handle
              GENIPDRV_WRITE_USHORT,     // control code
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              NULL,
              0,
              &NumBytes,                 // number of bytes transferred
              NULL                       // not over lapped
          );
if( success ) {
     // write access OK
}
else {
     // write access failed
}
```

Error Codes

All returned error codes are system error conditions.

See Also

Win32 documentation DeviceIoControl()

4.2.1.11 GENIPDRV_WRITE_ULONG

The write ulong function writes a buffer of 32-bit data to a specified address space. Before this function is used the GENIPDRV_CONFIGURE function must be called.

The parameter *lpInBuffer* must pass a pointer to the I/O buffer (*GENIPDRV_IO_BUF*) to the device driver. The parameter *lpOutBuffer* must pass a *NULL* pointer to the device driver.

```
typedef struct _GENIPDRV_IO_BUF
{
    UCHAR       space;                          // address space to read from
    ULONG       offset;                         // address offset in space
    ULONG       size;                           // number of uchar/ushort/ulong
    UCHAR       buffer[GENIPDRV_MAXIOBUF];      // pointer to buffer
} GENIPDRV_IO_BUF, *PGENIPDRV_IO_BUF;
```

space

This parameter specifies the address of the IPAC the data shall be read from.

Value	Description
GENIPDRV_IDSPACE	Read from ID Space
GENIPDRV_IOSPACE	Read from I/O space
GENIPDRV_MEMSPACE	Read from memory space

offset

This parameter specifies the starting offset in the selected space.

size

This parameter specifies the number of longwords to write.

buffer[]

This array must be filled with the data to write to the specified position. The size of the buffer can be changed by changing the value of *GENIPDRV_MAXIOBUF* in genIPDrv.h. The value is specified in byte.

Example

```
#include           “genIPDrv.h”

GENIPDRV_IO_BUF    ioBuf;
HANDLE             hDevice;
BOOLEAN            success;
ULONG              NumBytes;
PULONG             ulPtr;

…
```

```
…

// Write 2 Longwords (0x11111111,0x22222222) to IPAC Memory-Space
// starting at offset 0x10
ioBuf.space = GENIPDRV_MEMSPACE;
ioBuf.offset =0x10;
ioBuf.size =  0x2;

ulPtr = (PULONG)&ioBuf.Buffer[0];// Set pointer to data
ulPtr[0] = 0x11111111;
ulPtr[1] = 0x22222222;

//  Send request to the device driver
//
success = DeviceIoControl (
              hCurrent,                  // IPAC handle
              GENIPDRV_WRITE_ULONG,      // control code
              &ioBuf,
              sizeof(GENIPDRV_IO_BUF),
              NULL,
              0,
              &NumBytes,                 // number of bytes transferred
              NULL                       // not over lapped
         );
if( success ) {
     // write access OK
}
else {
     // write access failed
}
```

Error Codes

All returned error codes are system error conditions.

See Also

Win32 documentation DeviceIoControl()

5 Custom Driver Development

5.1 Custom Device Driver Example Overview

The following chapter should not be an introduction into windows driver development, it should just give a simple overview over the modifications necessary for a simple driver using TEWS Technologies IPAC carrier driver concept.

This chapter describes how an IPAC driver with carrier support can be developed for custom or 3rd party IPAC modules.

The advantage of a custom driver against the generic driver is that this driver may contain functions special for the device, this allows direct multiple access to the device in one I/O function without the overhead of multiple windows I/O function calls. The second and maybe more important advantage is that interrupts can be used, which allows an event handling without polling and superfluous effort of system performance.

For simple modifications and build of a custom device driver some preconditions must be fulfilled. A matching Windows DDK and SDK must be installed on the development system. Sometimes it may be helpful to use a windows debugger for testing. The modifications should be done with at least a basic knowledge of programming WDM device drivers.

The delivered example sources are thought as a base for custom developments and are generally not useable without modification.

TEWS Technologies GmbH will not provide support Custom Device Drivers.

5.2 Modify the IPAC Driver Example

This chapter describes the modification steps that will be necessary for a custom made driver based on the delivered example driver.

- Copy example sources
- Modify file names
- Modify INF-file (setup installation file)
- Modify RC-file (resource script file)
- Modify the “Sources” file
- Modify driver sources

The paragraphs below give an overview of these steps.

The examples below shows how to make the modifications for a driver named “digitalIO”.

5.2.1 Copy and rename example sources

First all sources of the example driver must be copied to an appropriate position where the driver can be built. Typically this will be a development path and the name of the supported module (<development_path>\<module_name>, e.g. "…\myDrivers\digitalIo").

Next the device dependent filenames *ipDrvCustom.inf*, *ipDrvCustom.rc*, and *ipDrvCustom.h* should be renamed.

For example rename:
ipDrvCustom.inf → digitalIO.inf
ipDrvCustom.rc → digitalIO.rc
ipDrvCustom.h → digitalIO.h

5.2.2 INF-file modifications

The INF-file contains setup information for the driver. The following sections of the file must be modified to match for the driver and the supported module.

This manual only describes the necessary and basic modification. More information about INF-files must be taken from of other sources (e.g. books or web)

Only necessary entries in the sections will be listed below, to keep this manual as transparent as possible.

Section [Version]

DriverVer

This entry specifies data and version of the driver and will be shown in the device manager. The format of the entry is *mm/dd/yyy [,a.b.c.d]*.
For a driver version 1.0.0.0 built on December 24th, 2006 the line should be set to:

```
DriverVer=12/24/2006,1.0.0.0
```

Section [SourceDisksFiles]

ipDrvCustom.sys

This name must be modified matching the name of the custom driver. For example the name should be changed into "`digitalIO.sys`".

Section [CUSTOM.Mfg]

%TTG.SvcDesc%

The assigned value specified a hardware ID of the supported device. If there are different devices supported by the driver, additional entries can attached. The hardware IDs are generated by the carrier driver and typically contain the manufacturer ID and model number of the IPAC board. There are two different formats, the first for boards using ID-prom type I and the second for boards using ID-prom type II (VITA 4).
A line identifying a board with type I ID-prom will look like:
%TTG.SvcDesc% = TTG,IPACSlot\MAN*mm*_MOD*nn*
A line identifying a board with type II (VITA 4) ID-prom will look like:
%TTG.SvcDesc% = TTG,IPACSlot\MAN*mmmmmm*_MOD*nnnn*

For example the driver supports two different boards with manufacturer ID 0xAA and model numbers 0x01 and 0x02, than the section should look like this:

```
[CUSTOM.Mfg]
%TTG.SvcDesc% = TTG,IPACSlot\MANAA_MOD01
%TTG.SvcDesc% = TTG,IPACSlot\MANAA_MOD02
```

Section [TTG_Service_Inst]

ServiceBinary

This entry specifies the driver SYS-file. The value must be filled with the name of the customs drivers SYS-file.
For example:

```
ServiceBinary  = %12%\digitalIO.sys
```

Section [TTG_EventLog_AddReg]

This section creates registry entries for the driver. The value must be modified by replacing “ipDrvCustom.sys” into the SYS-file name of the custom device driver (e.g. digitalIO.sys).

Localizable Strings

The values defined below this comment must be modified matching to the driver. For the described example this passage should look like this:

```
;
; Localizable Strings
;
TTG.SvcDesc="TEWS Technologies - digitalIO (Dig. I/O)"
CUSTOM.DeviceDesc0 = "digitalIO"

DiskId1="digitalIO"
CUSTOM="TEWS Technologies"
COPYFLG_NOSKIP=2
```

5.2.3 RC-file modifications

This file contains driver information values. Below the definitions which must be modified are described:

VER_FILEDESCRIPTION_STR
VER_PRODUCTNAME_STR

<IPDRVCUSTOM> should be exchanged by an appropriate string (e.g. “digitalIO”)

VER_INTERNALNAME_STR
VER_ORIGINALFILENAME_STR

These values must be set to the name of the drivers SYS-file (e.g. “digitalIO.sys”)

VER_PRODUCTVERSION

This definition should be set to the current driver version (e.g. for V1.0.0 “1,00,00,000”)

VER_PRODUCTVERSION_STR

This definition sets a string with the definition (e.g. for V1.0.0 “1.00.00”)

VER_LEGALCOPYRIGHT_YEARS

This defines a string with the copyright years (e.g. for “2004-2008”)

VER_LEGALCOPYRIGHT_STR
COMPANYNAME

<CUSTOM_COMPANY> should be replaced by the name of the company. (e.g. “TEWS TECHNOLOGIES”)

5.2.4 Source-file modifications

The Source-file contains information for the driver build process. This information includes driver name, path information for driver include and library files, and the source files needed for compilation. The following positions must be changed:

TARGETNAME

This value must be modified to the driver name (e.g. “digitalIO”)

SOURCES

This value contains the list of the used source-code driver files. The last entry specifies the RC-file. Because the name has been changed we have to change the name in the list too. (e.g. “digitalIO.rc”)

5.2.5 Driver source modifications

This chapter shows the position in the C-source files where changes have to be done typically, the positions are marked with "`(modify!!!)`". If files typically are not affected by the modification they will not be listed below. All the changes are very special for the device and knowledge about the devices registers and the functions is necessary.

5.2.5.1 DeviceIo.c

This file contains device specific functions for starting and stopping the device.

Function tpStartDevice

The necessary modifications in this function depend on the modules abilities and resources. All settings are highly depending from the used IPAC Behind the marked position the DCB (device extension) is initialized. Than the IPAC slot has to be configured (`ipac_configure_driver()`). In the next step the used IP address areas (ID, I/O and memory) are mapped, this allows special settings for every address area. The next step is to connect the ISR to a special interrupt vector. After that the device must be initialized, this is done synchronized with the interrupt (not interruptible by the ISR).

Function FreeDeviceResources

This function must free or unmap all allocated or mapped resources.

Function tpInitHardware

This function makes the basic initialization of the device. This function is called if the device starts, a typical implementation will make necessary setups like global interrupt enable, interrupt vector setup, default setup of the I/O lines etc.

Function tpShutDownDevice

This function will be called if the system shuts down, typically this driver will call the `tpInitHardware()` function to make the default settings and it must take care that all interrupts are disabled.

5.2.5.2 Dispatch.c

Function Dispatch

This function contains the control functions for the devices. Control functions (handling of additional functions codes) for custom devices can be added here. The definition of the parameter structure and I/O-control code must be done in drivers header file (e.g. digitalIO.h)

Function ipStartJob

This function starts a job waiting for an interrupt event. The function allows interrupt save access to the device extension, because it is called via `ipac_sync_isr()` function.

Function ipCancelIrp

This function shows how a pending Irp can be canceled, this may be necessary for waiting jobs.

5.2.5.3 Isr.c

Function tpIsr

This function shows a simple interrupt service function. This is the place where pending interrupts must be released. This function should always be kept short and quick. Only necessary action should take place here. For more complex interrupt handling a solution with a post called DPC should be taken in consideration. (The delivered example queues a DPC)

Function tpDpcForIsr

This is the DPC. This function is nearly an interrupt call, but is less time critical, because processing new incoming interrupts is possible during execution. The DPC should do the work, which must be done in the follow of an interrupt event. The DPC is especially necessary for functions that take some time which may be critical in an ISR.

5.2.5.4 IoTimer.c

This file contains an example function making the timeout handling for waiting functions.

5.2.5.5 localDrvDef.h

This file contains all definitions, structures, etc. which are only used by the driver and are not visible to the application. There are some sections marked with "`(modify!!!)`".

The section below the comment "Various constants" must be modified with the used driver and device name.

For example it must be modified to look like below:

```
//
//  Various constants (modify!!!)
//
#define DEBUG_NAME          "DIGITALIO"
#define PARAMETER_NAME      L"\\Parameters"
#define NT_DEVICE_NAME      L"\\Device\\digitalIO_"
#define WIN32_DEVICE_NAME   L"\\DosDevices\\digitalIO_"
```

A custom driver must be fully adapted to the used IPAC module. Register offset, flags and so on must be defined. This highly depends on the used IPAC module and it will not be described here in detail.

A very important structure is the DCB or device extension. This data is accessible in the whole driver. All data which must be available during device or driver lifetime has to be inserted here.

5.2.5.6 ipDrvCustom.h

This function contains all definitions, structures, etc. which will be needed by applications accessing the devices. The device I/O control codes and the I/O structures must be defined here.

5.3 Building the Driver

The last step to get a usable custom driver is to build the driver. Therefore an installed development kit is necessary. The development kit may have more than one build environment for different target systems where that one matching for the system must be chosen. There will be a free and a checked version for every target system. The checked version will create a driver file containing debug information for windows debugger and the free version will create a driver file without debug information.

For building the drivers and a usable installation medium the steps below must be executed:

Start development shell, using the windows start-menu:

`Development Kits/Windows DDK` *`nn`*`/Build Enviroments/`*`Win xx`*` Build Environment`

Change to the development path. For example:

```
C:\xxx> W:
W:\> cd development path/digitalIO
W:\ development path/digitalIO> makedriver
```

The last step is to copy the driver files to a medium which can be used to install the driver (e.g. floppy disk, CD). Therefore copy the INF-file and the built SYS-file from the development path to the medium in a matching path.

For example:

```
./free/i386/digitalIO.sys    ➔ A:\digitalIO
./digitalIO.inf              ➔ A:\digitalIO
```

After these steps, the medium can be used to install the driver on the target system. Simply follow standard windows installation procedure.

5.4 IPAC Carrier Interface Functions

This chapter will give a short description of the functions of the IPAC Carrier Driver carrier which are used by the custom driver. The description correlates on the custom driver example and describes the use in it. The header file “ipacLib.h” must always be included.

5.4.1 ipac_register_driver

NAME

ipac_register_driver() – Register the IPAC device at IPAC carrier interface

SYNOPSIS

```
LONG ipac_register_driver
(
      IN PDEVICE_OBJECT                           DeviceObject,
      OUT PIPACSLOT_INTERFACE_STANDARD     pIPACInterface
)
```

DESCRIPTION

This function registers the device at the IPAC carrier interface. The function will return a handle for the used slot, which will identify the device in following accesses.

This must always be the first access to the IPAC Interface.

Parameters

DeviceObject

This parameter specifies the IPAC slot device (next lower device in IRP stack). The value of this parameter must be the *NextStackDevice* entry of the device extension.

pIPACInterface

This returned pointer is a handle to the IPAC interface, which identifies the IPAC slot. It will be used for future access to the device.

Return Value

If the function succeeds, the status check *NT_SUCCESS(status)* will return a TRUE value, otherwise an error occurred and a windows error/status number will be returned. This value can be used as a return value of the driver function.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;
NTSTATUS status;

status = ipac_register_driver(   pExtension->NextStackDevice,
                                 &pExtension->IPACInterface);
if(!NT_SUCCESS(status))
{
     return status;
}
```

5.4.2 ipac_configure_driver

NAME

ipac_configure_driver() – Configure the IPAC device

SYNOPSIS

```
LONG ipac_configure_driver
(
    IN PIPACSLOT_INTERFACE_STANDARD      pIPACInterface,
    IN ULONG                             SlotConfigIn,
    OUT PULONG                           pSystemIntVector,
    OUT PULONG                           pModuleIntVector
)
```

DESCRIPTION

This function configures the IPAC slot the device is mounted on. The function will return the base interrupt vectors used for system configuration and module configuration. The returned vector base numbers correlates, one for software and the other for hardware (module) site.

Parameters

pIPACInterface

This pointer is the handle to the IPAC interface, it identifies the IPAC slot. This handle has been returned by the function *ipac_register_driver()*.

SlotConfigIn

This value defines the settings for the used IPAC slot. This setting must match to the IPAC abilities. The following definitions can be used in an OR'ed value:

Define	Description
IPAC_INT0_EN	This must be set if the IPAC generates interrupts on INT0.
IPAC_INT1_EN	This must be set if the IPAC generates interrupts on INT1.
IPAC_EDGE_SENS	This must be set if the IPAC uses edge sensitive interrupts. (Excludes selection of IPAC_LEVEL_SENS)
IPAC_LEVEL_SENS	This must be set if the IPAC uses level sensitive interrupts. (Excludes selection of IPAC_EDGE_SENS)
IPAC_CLK_8MHZ	This sets the IPAC system clock to 8MHz. (Excludes selection of IPAC_CLK_32MHZ)
IPAC_CLK_32MHZ	This sets the IPAC system clock to 32MHz. (Excludes selection of IPAC_CLK_8MHZ)
IPAC_MEM_8BIT	If this value is chosen, the memory space of the module has a width of 8-bit. (Excludes selection of IPAC_MEM_16BIT)
IPAC_MEM_16BIT	If this value is chosen, the memory space of the module has a width of 16-bit. (Excludes selection of IPAC_MEM_8BIT)

pSystemIntVector

This is a system dependent interrupt vector base representing the first interrupt number which will be used for the device. This is the right interrupt vector to be used on software site.

pModuleIntVector

This is a system dependent interrupt vector base representing the first interrupt number which will be used for the device. This is the right interrupt vector to be used on hardware site, which must be written to the interrupt vector register.

Return Value

If the function succeeds, the status check *NT_SUCCESS(status)* will return a TRUE value, otherwise an error occurred and a windows error/status number will be returned. This value can be used as a return value of the driver function.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;
NTSTATUS status;

//   Configure Slot with module depending settings
//        - Use Level sensitive INT0
//        - Use 8MHz IP clock
status = ipac_configure_driver(  &pExtension->IPACInterface,
                                 IPAC_INT0_EN | IPAC_LEVEL_SENS |
                                    IPAC_CLK_8MHZ,
                                 &pExtension->sysIntVector
                                 &pExtension->modIntVector);
if( !NT_SUCCESS(status))
{
     KdPrint(("Problem configuring the IPAC-Slot\n"));
     return status;
}
```

5.4.3 ipac_map_space

NAME

ipac_map_space() – Maps and configure IPAC address space access.

SYNOPSIS

```
LONG ipac_map_space
(
    IN PIPACSLOT_INTERFACE_STANDARD     pIPACInterface,
    IN UCHAR                            SpaceID,
    IN ULONG                            Size,
    IN BOOLEAN                          Swapping,
    OUT PIPAC_ADRSPACE_HANDLE           Handle
)
```

DESCRIPTION

A specified address space of the IPAC device is mapped, space configuration is made and a handle to the space will be returned for access to the space in future.

This function must be called before an access to an IPAC space is done.

Parameters

pIPACInterface

This pointer is the handle to the IPAC interface, it identifies the IPAC slot. This handle has been returned by the function *ipac_register_driver()*.

SpaceID

This value specifies which address space shall be mapped. The following defines can be used:

IPAC address space	Description
IPAC_ID_SPACE	This specifies the IPAC ID-space.
IPAC_IO_SPACE	This specifies the IPAC IO-space.
IPAC_MEMORY_SPACE	This specifies the IPAC memory-space.

Size

This specifies the size of the address space. The value must be specified in bytes.

Swapping

This value enables or disabled byte swapping within the address space. If the returned values on access to the address space are not matching the expected endian format, the values can be easily switched. *FALSE* will disable and *TRUE* will enable byte swapping.

Handle

This parameter will contain a handle to the mapped address space. This handle must be used for IPAC address access functions.

Return Value

If the function succeeds, the status check *NT_SUCCESS(status)* will return a TRUE value, otherwise an error occurred and a windows error/status number will be returned. This value can be used as a return value of the driver function.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;
NTSTATUS status;

//   Map ID-register space
//        - size of 0x20 byte
//        - no byte swapping
status = ipac_map_space(     &pExtension->IPACInterface,
                             IPAC_ID_SPACE,
                             0x20,
                             FALSE,
                             &pExtension->IDSpaceHandle);
if( !NT_SUCCESS(status))
{
     KdPrint(("Can't Map ID-Space\n"));
     return status;
}
```

5.4.4 ipac_unmap_space

NAME

ipac_unmap_space() – Release IPAC address space mapping.

SYNOPSIS

```
LONG ipac_unmap_space
(
    IN PIPAC_ADRSPACE_HANDLE            Handle
)
```

DESCRIPTION

This function releases a previous mapping of IPAC address space. This function will allow a remapping of an address space with modified parameters by *ipac_map_space()*.

Parameters

Handle

This pointer is the handle to the IPAC address space, identifying the space. This handle has been returned by the function *ipac_map_space()*.

Return Value

If the function succeeds, the status check *NT_SUCCESS(status)* will return a TRUE value, otherwise an error occurred and a windows error/status number will be returned. This value can be used as a return value of the driver function.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;
NTSTATUS status;

//   Unmap ID-register space
status = ipac_unmap_space(&pExtension->IDSpaceHandle);
if( !NT_SUCCESS(status))
{
     KdPrint(("Can't unmap ID-Space\n"));
     return status;
}
```

5.4.5 ipac_read_uchar

NAME

ipac_read_uchar() – Read a character value (8-bit) from the IPAC.

SYNOPSIS

```
UCHAR ipac_read_uchar
(
    IN PIPAC_ADRSPACE_HANDLE            Handle,
    IN ULONG                            Offset
)
```

DESCRIPTION

This function reads an 8-bit value from a specified IPAC address space.

Parameters

Handle

This pointer is a handle to the IPAC address space, identifying the space. This handle has been returned by the function *ipac_map_space ()*.

Offset

This value specifies the access offset in the specified address space. The offset value is specified in byte.

Return Value

The function returns the read value.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;
UCHAR value;

//   Read a byte from a mapped IOSpace offset 0x10
value = ipac_read_uchar(&pExtension->IOSpaceHandle, 0x10);
```

5.4.6 ipac_read_ushort

NAME

ipac_read_ushort() – Read a unsigned short value (16-bit) from the IPAC.

SYNOPSIS

```
USHORT ipac_read_ushort
(
    IN PIPAC_ADRSPACE_HANDLE          Handle,
    IN ULONG                          Offset
)
```

DESCRIPTION

This function reads a 16-bit value from a specified IPAC address space.

Parameters

Handle

This pointer is a handle to the IPAC address space, identifying the space. This handle has been returned by the function *ipac_map_space ()*.

Offset

This value specifies the access offset in the specified address space. The offset value is specified in byte.

Return Value

The function returns the read value.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;
USHORT value;

//   Read a 16-bit value from a mapped IOSpace offset 0x10
value = ipac_read_ushort(&pExtension->IOSpaceHandle, 0x10);
```

5.4.7 ipac_read_ulong

NAME

ipac_read_ulong() – Read a unsigned long value (32-bit) from the IPAC.

SYNOPSIS

```
ULONG ipac_read_ulong
(
    IN PIPAC_ADRSPACE_HANDLE        Handle,
    IN ULONG                        Offset
)
```

DESCRIPTION

This function reads a 32-bit value from a specified IPAC address space.

Parameters

Handle

This pointer is a handle to the IPAC address space, identifying the space. This handle has been returned by the function *ipac_map_space ()*.

Offset

This value specifies the access offset in the specified address space. The value is specified in byte.

Return Value

The function returns the read value.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;
ULONG value;

//   Read a 32-bit value from a mapped IOSpace offset 0x10
value = ipac_read_ulong(&pExtension->IOSpaceHandle, 0x10);
```

5.4.8 ipac_write_uchar

NAME

ipac_write_uchar() – Write a character value (8-bit) to the IPAC.

SYNOPSIS

```
VOID ipac_write_uchar
(
    IN PIPAC_ADRSPACE_HANDLE        Handle,
    IN ULONG                        Offset,
    IN UCHAR                        Value
)
```

DESCRIPTION

This function writes an 8-bit value to a specified IPAC address space.

Parameters

Handle

This pointer is a handle to the IPAC address space, identifying the space. This handle has been returned by the function *ipac_map_space ()*.

Offset

This value specifies the access offset in the specified address space. The offset value is specified in byte.

Value

This value specifies the value that shall be written to the specified address space.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;

//   Write a byte (0xAA) to the mapped IOSpace at offset 0x10
ipac_write_uchar(  &pExtension->IOSpaceHandle,
                   0x10,
                   0xAA);
```

5.4.9 ipac_write_ushort

NAME

ipac_write_ushort() – Write an unsigned short value (16-bit) to the IPAC.

SYNOPSIS

```
VOID ipac_write_uchar
(
    IN PIPAC_ADRSPACE_HANDLE          Handle,
    IN ULONG                          Offset,
    IN USHORT                          Value
)
```

DESCRIPTION

This function writes a 16-bit value to a specified IPAC address space.

Parameters

Handle

This pointer is a handle to the IPAC address space, identifying the space. This handle has been returned by the function *ipac_map_space ()*.

Offset

This value specifies the access offset in the specified address space. The offset value is specified in byte.

Value

This value specifies the value that shall be written to the specified address space.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;

//   Write a 16-bit value (0xAABB) to the mapped IOSpace at offset 0x10
ipac_write_ushort( &pExtension->IOSpaceHandle,
                   0x10,
                   0xAABB);
```

5.4.10 ipac_write_ulong

NAME

ipac_write_ulong() – Write an unsigned long value (32-bit) to the IPAC.

SYNOPSIS

```
VOID ipac_write_uchar
(
    IN PIPAC_ADRSPACE_HANDLE            Handle,
    IN ULONG                            Offset,
    IN ULONG                            Value
)
```

DESCRIPTION

This function writes a 32-bit value to a specified IPAC address space.

Parameters

Handle

This pointer is a handle to the IPAC address space, identifying the space. This handle has been returned by the function *ipac_map_space ()*.

Offset

This value specifies the access offset in the specified address space. The offset value is specified in byte.

Value

This value specifies the value that shall be written to the specified address space.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;

//   Write a 32-bit value (0xAABBCCDD) to the mapped IOSpace at offset 0x10
ipac_write_ulong(  &pExtension->IOSpaceHandle,
                   0x10,
                   0xAABBCCDD);
```

5.4.11 ipac_register_isr

NAME

ipac_register_isr() – Register an ISR for a specified Vector.

SYNOPSIS

```
NTSTATUS ipac_register_isr
(
    IN PIPACSLOT_INTERFACE_STANDARD      pIPACInterface,
    IN ULONG                             Vector,
    IN PKSERVICE_ROUTINE                 Handler,
    IN PVOID                             Arg,
    OUT PVOID                            *IntHandle
)
```

DESCRIPTION

This function registers an ISR for a specified interrupt vector. This registered handler function will be called with the specified argument always when an interrupt with the specified vector occurs.

Parameters

pIPACInterface

This pointer is the handle to the IPAC interface, it identifies the IPAC slot. This handle has been returned by the function *ipac_register_driver()*.

Vector

This value specifies the interrupt vector the handler function should be connected to. The vector number must base on the returned *pSystemIntVector* of *ipac_configure_driver()* function.

Handler

This parameter specifies the entry point to an interrupt handler function. The function will be called every time an interrupt with the specified interrupt vector occurs. The handler function has to return a boolean value, wich indicates that the handler has handled the interrupt (*TRUE*) or not (*FALSE*).

Arg

This value specifies an argument that will be supplied to the interrupt handler function. In general use this will be the device extension.

IntHandle

This parameter returns a handle for the specified interrupt handler. This will be necessary for identification for interrupt synchronization and for unregistering.

Return Value

If the function succeeds, the status check *NT_SUCCESS(status)* will return a TRUE value, otherwise an error occurred and a windows error/status number will be returned. This value can be used as a return value of the driver function.

Example

```
#include <ipacLib.h>

BOOLEAN tpIsr(IN PKINTERRUPT Interrupt, IN PVOID pContext);

PDEVICE_EXTENSION pExtension;
NTSTATUS status;

//   Register an interrupt handler for interrupts with the base vector
status = ipac_register_isr(  &pExtension->IPACInterface,
                             pExtension->sysIntVector,
                             (PKSERVICE_ROUTINE)tpIsr,
                             (PVOID)pExtension,
                             &pExtension->pInterrupt);
if(!NT_SUCCESS(status))
{
     KdPrint(("Interrupt connect failed\n"));
     return status;
}

…

// ISR
BOOLEAN tpIsr
(
     IN PKINTERRUPT Interrupt,
     IN PVOID pContext
)
{
     PDEVICE_EXTENSION pExtension = (PDEVICE_EXTENSION)pContext;

     …
     if (/* Interrupt occurred */)
          return TRUE;
     else
          return FALSE
}
```

5.4.12 ipac_unregister_isr

NAME

ipac_unregister_isr() – Unregister an ISR.

SYNOPSIS

```
VOID ipac_unregister_isr
(
        IN PIPACSLOT_INTERFACE_STANDARD          pIPACInterface,
        IN PVOID                                 IntHandle
)
```

DESCRIPTION

This function unregisters an ISR. This function must be called if the device will be disabled.

After this function has been called, no interrupts for the corresponding vector will be handled. Therefore it is necessary, that the corresponding interrupt will be disabled in the IPAC module.

Parameters

pIPACInterface

This pointer is the handle to the IPAC interface, it identifies the IPAC slot. This handle has been returned by the function *ipac_register_driver()*.

IntHandle

This parameter specifies the interrupt that shall be unregistered. This handle has been returned by the function *ipac_register_isr()*.

Example

```
#include <ipacLib.h>

PDEVICE_EXTENSION pExtension;

//   Unregister a previous registered interrupt handler
ipac_unregister_isr(    &pExtension->IPACInterface,
                        pExtension->pInterrupt);
```

5.4.13 ipac_sync_isr

NAME

ipac_sync_isr() – Call function synchronized with an ISR.

SYNOPSIS

```
NTSTATUS ipac_sync_isr
(
        IN PIPACSLOT_INTERFACE_STANDARD          pIPACInterface,
        IN PVOID                                 IntHandle,
        IN PKSYNCHRONIZE_ROUTINE                 Routine,
        IN PVOID                                 Arg
)
```

DESCRIPTION

This function calls a function that will be executed uninterruptible by the specified interrupt handler.

Parameters

pIPACInterface

This pointer is the handle to the IPAC interface, it identifies the IPAC slot. This handle has been returned by the function *ipac_register_driver()*.

IntHandle

This parameter specifies the registered interrupt handler the function should be synchronized to. This handle has been returned by the function *ipac_register_isr()*.

Handler

This parameter specifies is the entry point to the function that shall be called. The handler function has to return a boolean value, wich indicates if the function has been executed successful (*TRUE*) or not (*FALSE*).

Arg

This value specifies an argument that will be supplied to the interrupt handler function. In general use this will be the device extension.

Return Value

The function returns the return value of the called function.

Example

```
#include <ipacLib.h>

BOOLEAN syncFunct(PDEVICE_EXTENSION pExtension);

PDEVICE_EXTENSION pExtension;

//   Register an interrupt handler for interrupts with the base vector
if (ipac_sync_isr( &pExtension->IPACInterface,
                   pExtension->pInterrupt,
                   (PKSERVICE_ROUTINE) syncFunct,
                   (PVOID)pExtension))
{
     // Function successful
}
else
{
     // Function failed
}

…

// Synchronized function
BOOLEAN syncFunct
(
     PDEVICE_EXTENSION pExtension
)
{
     …

     if (/* Error occurred */)
          return FALSE;
     else
          return TRUE
}
```

5.5 Driver Development Tools

For development of a custom driver the following tools will be necessary or helpful:

- Text editor (e.g. Microsoft editor, or any other)
- Microsoft driver development kit (DDK)
- Microsoft software development kit (SDK)
- Windows debugger (WinDbg)

The development tools (DDK, SDK and WinDbg) can be found on the Microsoft Website and they can be downloaded for free. How the installation of the tools will be done is described in the corresponding manuals.

The Windows debugger may be very helpful if problems or at least crashes occur during driver installation and use, otherwise the debugger is not necessary. For more information refer to the debuggers documents.

After installation of the tools on an environment system the modification of a custom driver can be done as described above.

5.6 Example Application for Driver Example

There is a simple example for the delivered custom driver example. The example application shows the usage of driver. There is no detailed documentation how to use the custom IPAC driver. For a detailed description refer to the description of generic IP driver use in this manual.