DYNAMIC ENGINEERING
150 DuBois, Suite C
Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

Pci3Ip, Pci5Ip, Pc104pIp, Pc104p4Ip, cPci2Ip, cPci4Ip and PcieCar

IndustryPack Carrier Device Drivers

Windows 10 WDF Driver Documentation

Developed with Windows Driver Foundation Ver1.19

Revision 01
Corresponding Hardware:

PCI3IP	10-1999-0404	Revision D	Firmware: Revision G
PCI5IP	10-2002-0307	Revision G	Firmware: Revision G.2
cPCI2IP	10-2002-0805	Revision E	Firmware: Revision E
cPCI4IP	10-2004-0903	Revision C	Firmware: Revision B
PC104pIP	10-2005-0401	Revision A	Firmware: Revision A
PC104p4IP	10-2003-0503	Revision C	Firmware: Revision B
PCIe3IP	10-2014-0202	Revision B	Firmware: Revision A.0
PCIe3IP	10-2015-1601	Revision A	Firmware: Revision A.0
VPX2IP	10-2016-1901	Revision A	Firmware: Revision A.0

Pci3Ip, Pci5Ip, cPci2Ip, cPci4Ip, Pc104pIp, Pc104p4Ip, PcieCar
WDF Device/Bus Drivers for PCI/PCIe based IndustryPack Module Carriers from Dynamic Engineering

Dynamic Engineering
150 DuBois, Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 FAX

Table of Contents

Introduction

Pci3Ip, Pci5Ip, cPci2Ip, cPci4Ip, Pc104pIp, Pc104p4Ip and PcieCar are Win10 device drivers for their respective PCI/PCIe based Industry-Pack (IP) module carriers from Dynamic Engineering. These drivers were developed with the Windows Driver Foundation version 1.19 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

Each carrier can hold 1-5 IP modules (corresponding to the number preceding the Ip designation). When a carrier board is recognized by the system, it will start the appropriate carrier driver, which creates an IP bus to communicate with the IP modules. The carrier driver then enumerates the IP bus by reading the ID prom of each installed IP module to discover the device type. If an IP module is recognized and the driver has been previously installed, it will be started and a Device Object will be created for each matching IP. If an IP module is not recognized, i.e. the contents of the ID prom do not appear on the list of IP module drivers in the IpCarriers.inf file, the IpGeneric driver will be used to communicate with that IP module.

A separate handle to the IP carrier and to each IP module can be obtained using CreateFile() calls. IO Control calls (IOCTLs) are used to configure the IP carrier and read the carrier's status. The IP carrier driver is responsible for reading its user switch setting, operating the onboard LED's, and a few other operations, but the IP carrier's main function is to act as a PCI/PCIe⇔IP Bus bridge device providing resources for the installed IP modules. These modules operate independently through their own file handles. See the appropriate IP driver documentation for information on the capabilities of a particular IP module.

Notes

This document will provide information about all calls made to the driver(s), and how the driver(s) interact with the device(s) for each of these calls. For more detailed information on the hardware implementation, refer to the user manual for the specific carrier you are using.

The following terms will be used throughout this document as placeholders for a specific IP carrier driver name.
CARRIERNAME: PC104PIP, CPCI2IP, CPCI4IP, PCI3IP, PC104P4IP, PCI5IP or PCIECAR.
CarrierName: Pc104pIp cPci2Ip, Pci3Ip, cPci4Ip, Pc104p4Ip, Pci5Ip or PcieCar.

The PcieCar driver controls multiple PCIe based IP carriers. Currently this includes PCIe3IP and PCIe5IP.

Driver Installation

There are several files provided in the driver package. These files include [*CarrierName*].sys, [*CarrierName*].cat, [*CarrierName*].inf, and [*CarrierName*]Public.h. The [*CarrierName*]Public.h files are 'C' header files that define the Application Program Interfaces (API) to the drivers. These files are required at compile time by any application that wishes to interface with the drivers, but are not needed for driver installation.

Note:

Secure boot must be disabled before drivers are installed. Please see the link below to disable secure boot. In general this only affects Windows 10 Server installations.

Windows 10 Installation

Copy [*CarrierName*].sys, [*CarrierName*].cat, and [*CarrierName*].inf driver files (*.sys) to a removable memory device, or other accessible location as preferred.

With one or more of the supported IP carriers installed, power-on the host computer.

- Open the ***Device Manager*** from the control panel.
- Under ***Other devices*** there should be a ***PCI Bridge Other*** or ***PCI to NUBUS Bridge*** device*.
- Right-click on the ***PCI Bridge Other*** or ***PCI to NUBUS Bridge*** device and select ***Update Driver Software***.
- Insert the removable memory device prepared above if necessary.
- Select ***Browse my computer for driver software***.
- Select ***Browse*** and navigate to the location where the appropriate files are stored.
- Select ***Next***. The *CarrierName* device driver should now be installed.
- Select ***Close*** to close the update window.
  The Device Manager should now display the carrier slots that contain valid IP modules.
- Right-click on each IP slot icon, select ***Update Driver Software*** and proceed as above for each IP module as necessary.

* If neither of these devices is displayed, click on the ***Scan for hardware changes*** icon on the tool-bar or select it in the Action menu.

Driver Startup

Once the driver has been installed it will start automatically when the system recognizes the hardware.

A handle can be opened to a specific board by using the CreateFile() function call and passing in the device name obtained from the system.

The interface to the device is identified using a globally unique identifier (GUID), which is defined in *CarrierName*Public.h.

The *main.c* file provided with the user test software is designed to work with our test menu and includes user interaction steps to allow the user to select which board is being tested in a multiple board environment. The integrator can hardcode for single board systems or use an automatic loop to operate in multiple board systems without requiring user interaction. For multiple user systems it is suggested that the board number is associated with the user switch setting so the calls can be associated with a specific physical device.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure their devices. IOCTLs refer to a single Device Object, which controls a single module. IOCTLs are called using the Win32 function DeviceIoControl() (see below), and passing in the handle to the device opened with CreateFile() (see above). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used.

```
BOOL DeviceIoControl(
  HANDLE       hDevice,          // Handle opened with CreateFile()
  DWORD        dwIoControlCode,  // Control code defined in API header file
  LPVOID       lpInBuffer,       // Pointer to input parameter
  DWORD        nInBufferSize,    // Size of input parameter
  LPVOID       lpOutBuffer,      // Pointer to output parameter
  DWORD        nOutBufferSize,   // Size of output parameter
  LPDWORD      lpBytesReturned,  // Pointer to return length parameter
  LPOVERLAPPED lpOverlapped,     // Optional pointer to overlapped structure
);                               //   used for asynchronous I/O
```

The IOCTLs defined in the IP carrier drivers are described below:

IOCTL_*CARRIERNAME*_GET_INFO

Function: Returns the Driver and Firmware revisions, Switch value, Instance number and number of IP slots supported.
Input: None
Output: DRIVER_CARRIER_DEVICE_INFO structure
Notes: Switch value is the configuration of the onboard dipswitch that has been selected by the user (see the board silk screen for bit position and polarity). The CPLD fields are currently only valid for the PcieCar driver. See the definition of DRIVER_CARRIER_DEVICE_INFO below.

```
 // Driver revision and instance/slot information
typedef struct _DRIVER_CARRIER_DEVICE_INFO {
   USHORT   DriverRev;
   UCHAR    FirmwareRev;
   UCHAR    FirmwareRevMin;
   UCHAR    CPLDRev;
   UCHAR    CPLDRevMin;
   USHORT   InstanceNum;
   UCHAR    SwitchValue;
   UCHAR    NumIpSlots;
} DRIVER_CARRIER_DEVICE_INFO, *PDRIVER_CARRIER_DEVICE_INFO;
```

IOCTL_*CARRIERNAME*_GET_SW_ID

Function: Returns the user switch value.
Input: None
Output: Eight-bit switch value (unsigned character)

Notes: The value returned is the user-selected configuration of the 8-bit onboard dipswitch. See the board silk screen for bit position and polarity.

IOCTL_*CARRIERNAME*_SET_CONFIG

Function: Specifies various control parameters for the IP carrier.
Input: *CARRIERNAME*_CONFIG structure
Output: None
Notes: Specifies the LED configuration and other controls. The configuration parameters for the PcieCar driver are different than the other IP carrier drivers and include interrupt aggregation and de-assert characteristics, LED source and user. See the definition of LED_MUX, PCIECAR_CONFIG and *CARRIERNAME*_CONFIG below.

```
 // LED source selector
typedef enum _LED_MUX {
   BD_STAT      = 0x0,
   USR_LED      = 0x1,
   USR_SW       = 0x2,
   FLSH_SW      = 0x3,
   IP0_STAT     = 0x4,
   IP1_STAT     = 0x5,
   IP2_STAT     = 0x6,
   RESERVED     = 0x7,
   PST_HD_CRDT  = 0x8,
   NPST_HD_CRDT = 0x9,
   CPLT_HD_CRDT = 0xA,
   PST_DT_CRDT  = 0xB,
   NPST_DT_CRDT = 0xC,
   CPLT_DT_CRDT = 0xD,
   SCRATCH_0    = 0xE,
   SCRATCH_1    = 0xF
} LED_MUX, *PLED_MUX;

 // PcieCar Carrier Configuration
typedef struct _PCIECAR_CONFIG {
   UCHAR    UserLed;
   LED_MUX  LedSrc;
   BOOLEAN  IntAgEn;    // Interrupt aggregation timer enable
   UCHAR    AgTimer;    // Interrupt aggregation timer
   UCHAR    IntDasTme;  // Interrupt de-assert time
} PCIECAR_CONFIG, *PPCIECAR_CONFIG;

 // CARRIERNAME Configuration
typedef struct _CARRIERNAME_CONFIG {
   UCHAR    UserLed;       // Configuration of the eight user LEDs
   BOOLEAN  BusErrIntEn;   // Enable the bus error timeout interrupt
   BOOLEAN  BusErrStatClr; // Write clears the interrupt latch/status
} CARRIERNAME_CONFIG, *PCARRIERNAME_CONFIG;
```

IOCTL_*CARRIERNAME*_GET_CONFIG

Function: Returns the fields set in the previous call.
Input: None
Output: *CARRIERNAME*_CONFIG structure
Notes: See the definitions of LED_MUX, PCIECAR_CONFIG and *CARRIERNAME*_CONFIG above.

IOCTL_*CARRIERNAME*_GET_INT_STATUS

Function: Returns the IP interrupt status register value and clears the bits that were read.
Input: None
Output: Value of the IP module interrupt status register (unsigned long integer)
Notes: PcieCar: See the status bit definitions below. A bit will be cleared by this call only if it was set when the register was read. This prevents the possibility of missing an interrupt condition that occurs after the register has been read but before the latched bits are cleared.

```
// PcieCar IP interrupt status defines
#define INTERRUPT_IP0_IREQ0   0x00000001  // Interrupt 0 from IP 0
#define INTERRUPT_IP0_IREQ1   0x00000002  // Interrupt 1 from IP 0
#define INTERRUPT_IP0_BUSERR  0x00000004  // Bus error interrupt from IP 0
#define INTERRUPT_IP0_FORCE   0x00000008  // Force interrupt from IP 0
#define INTERRUPT_IP1_IREQ0   0x00000010  // Interrupt 0 from IP 1
#define INTERRUPT_IP1_IREQ1   0x00000020  // Interrupt 1 from IP 1
#define INTERRUPT_IP1_BUSERR  0x00000040  // Bus error interrupt from IP 1
#define INTERRUPT_IP1_FORCE   0x00000080  // Force interrupt from IP 1
#define INTERRUPT_IP2_IREQ0   0x00000100  // Interrupt 0 from IP 2
#define INTERRUPT_IP2_IREQ1   0x00000200  // Interrupt 1 from IP 2
#define INTERRUPT_IP2_BUSERR  0x00000400  // Bus error interrupt from IP 2
#define INTERRUPT_IP2_FORCE   0x00000800  // Force interrupt from IP 2
#define INTERRUPT_IP3_IREQ0   0x00001000  // Interrupt 0 from IP 3
#define INTERRUPT_IP3_IREQ1   0x00002000  // Interrupt 1 from IP 3
#define INTERRUPT_IP3_BUSERR  0x00004000  // Bus error interrupt from IP 3
#define INTERRUPT_IP3_FORCE   0x00008000  // Force interrupt from IP 3
#define INTERRUPT_IP4_IREQ0   0x00010000  // Interrupt 0 from IP 4
#define INTERRUPT_IP4_IREQ1   0x00020000  // Interrupt 1 from IP 4
#define INTERRUPT_IP4_BUSERR  0x00040000  // Bus error interrupt from IP 4
#define INTERRUPT_IP4_FORCE   0x00080000  // Force interrupt from IP 4
#define INTERRUPT_IP5_IREQ0   0x00100000  // Interrupt 0 from IP 5
#define INTERRUPT_IP5_IREQ1   0x00200000  // Interrupt 1 from IP 5
#define INTERRUPT_IP5_BUSERR  0x00400000  // Bus error interrupt from IP 5
#define INTERRUPT_IP5_FORCE   0x00800000  // Force interrupt from IP 5
#define INTERRUPT_IP6_IREQ0   0x01000000  // Interrupt 0 from IP 6
#define INTERRUPT_IP6_IREQ1   0x02000000  // Interrupt 1 from IP 6
#define INTERRUPT_IP6_BUSERR  0x04000000  // Bus error interrupt from IP 6
#define INTERRUPT_IP6_FORCE   0x08000000  // Force interrupt from IP 6
#define INTERRUPT_IP7_IREQ0   0x10000000  // Interrupt 0 from IP 7
#define INTERRUPT_IP7_IREQ1   0x20000000  // Interrupt 1 from IP 7
#define INTERRUPT_IP7_BUSERR  0x40000000  // Bus error interrupt from IP 7
#define INTERRUPT_IP7_FORCE   0x80000000  // Force interrupt from IP 7
```

CarrierName: The status bits of the other carriers are not latched. The status bits of the following carriers are all subsets of the status bits of the Pci5Ip shown below. The bit masks for each of carriers show the valid bits for each carrier.

```
 // Pci5Ip, Pc104p4Ip, cPci4Ip, cPci2Ip, Pc104pIp IP interrupt status defines
#define INTERRUPT_MASKED_A0    0x00000001  // Masked interrupt 0 from IP A
#define INTERRUPT_MASKED_A1    0x00000002  // Masked interrupt 1 from IP A
#define INTERRUPT_MASKED_B0    0x00000004  // Masked interrupt 0 from IP B
#define INTERRUPT_MASKED_B1    0x00000008  // Masked interrupt 1 from IP B
#define INTERRUPT_MASKED_C0    0x00000010  // Masked interrupt 0 from IP C
#define INTERRUPT_MASKED_C1    0x00000020  // Masked interrupt 1 from IP C
#define INTERRUPT_MASKED_D0    0x00000040  // Masked interrupt 0 from IP D
#define INTERRUPT_MASKED_D1    0x00000080  // Masked interrupt 1 from IP D
#define INTERRUPT_MASKED_E0    0x00000100  // Masked interrupt 0 from IP E
#define INTERRUPT_MASKED_E1    0x00000200  // Masked interrupt 1 from IP E
#define INTERRUPT_N            0x00000400  // from any IP, bus error, or force
#define INTERRUPT_UNMASKED_A0  0x00001000  // Unmasked interrupt 0 from IP A
#define INTERRUPT_UNMASKED_A1  0x00002000  // Unmasked interrupt 1 from IP A
#define INTERRUPT_UNMASKED_B0  0x00004000  // Unmasked interrupt 0 from IP B
#define INTERRUPT_UNMASKED_B1  0x00008000  // Unmasked interrupt 1 from IP B
#define INTERRUPT_UNMASKED_C0  0x00010000  // Unmasked interrupt 0 from IP C
#define INTERRUPT_UNMASKED_C1  0x00020000  // Unmasked interrupt 1 from IP C
#define INTERRUPT_UNMASKED_D0  0x00040000  // Unmasked interrupt 0 from IP D
#define INTERRUPT_UNMASKED_D1  0x00080000  // Unmasked interrupt 1 from IP D
#define INTERRUPT_UNMASKED_E0  0x00100000  // Unmasked interrupt 0 from IP E
#define INTERRUPT_UNMASKED_E1  0x00200000  // Unmasked interrupt 1 from IP E
#define INTERRUPT_BUS_ERROR    0x00400000  // Bus error 1=occurred, 0=none
#define INTERRUPT_BUS_ERROR_A  0x01000000  //
#define INTERRUPT_BUS_ERROR_B  0x02000000  //
#define INTERRUPT_BUS_ERROR_C  0x04000000  //
#define INTERRUPT_BUS_ERROR_D  0x08000000  //
#define INTERRUPT_BUS_ERROR_E  0x10000000  //
#define INTERRUPT_BUS_ERROR_BC 0x20000000  //
#define INTERRUPT_BUS_ERROR_DE 0x40000000  //

#define INTERRUPT_STATUS_MASK 0x7FFFF7FF  // Pci5Ip valid interrupt bits
#define INTERRUPT_STATUS_MASK 0x7E7FC7FC  // Pc104p4Ip valid interrupt bits
#define INTERRUPT_STATUS_MASK 0x2F4FF4FF  // cPci4Ip valid interrupt bits
#define INTERRUPT_STATUS_MASK 0x0340F40F  // cPci2Ip valid interrupt bits
#define INTERRUPT_STATUS_MASK 0x0240C40C  // Pc104pIp valid interrupt bits
```

Pci3Ip*:* The status bits for the Pci3Ip are shown below.

```
// Pci3Ip IP interrupt status defines
#define INTERRUPT_MASKED_A0   0x00000001  // Masked interrupt 0 from IP A
#define INTERRUPT_MASKED_A1   0x00000002  // Masked interrupt 1 from IP A
#define INTERRUPT_MASKED_B0   0x00000004  // Masked interrupt 0 from IP B
#define INTERRUPT_MASKED_B1   0x00000008  // Masked interrupt 1 from IP B
#define INTERRUPT_MASKED_C0   0x00000010  // Masked interrupt 0 from IP C
#define INTERRUPT_MASKED_C1   0x00000020  // Masked interrupt 1 from IP C
#define INTERRUPT_FORCE       0x00000040  // Forced interrupt active
#define INTERRUPT_N           0x00000080  // from any IP, bus error, or force
#define INTERRUPT_UNMASKED_A0 0x00000100  // Unmasked interrupt 0 from IP A
#define INTERRUPT_UNMASKED_A1 0x00000200  // Unmasked interrupt 1 from IP A
#define INTERRUPT_UNMASKED_A  0x00000300  // Unmasked interrupt 0, 1 from IP A
#define INTERRUPT_UNMASKED_B0 0x00000400  // Unmasked interrupt 0 from IP B
#define INTERRUPT_UNMASKED_B1 0x00000800  // Unmasked interrupt 1 from IP B
#define INTERRUPT_UNMASKED_B  0x00000C00  // Unmasked interrupt 0, 1 from IP B
#define INTERRUPT_UNMASKED_C0 0x00001000  // Unmasked interrupt 0 from IP C
#define INTERRUPT_UNMASKED_C1 0x00002000  // Unmasked interrupt 1 from IP C
#define INTERRUPT_UNMASKED_C  0x00003000  // Unmasked interrupt 0, 1 from IP C
#define INTERRUPT_BUS_ERROR   0x00004000  // Bus error 1=occurred, 0=none
#define INTERRUPT_STATUS_MASK 0x00007FFF  // Pci3Ip valid interrupt bits
```

IOCTL_*CARRIERNAME*_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to the Event object
Output: None
Notes: The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when an interrupt is serviced. The user's interrupt service routine waits on this event, allowing it to respond to the interrupt.

IOCTL_*CARRIERNAME*_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the PCI bus as long as the channel master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_*CARRIERNAME*_READ_ID_PROM

Function: Returns the contents of the IP ID prom for a particular slot.
Input: IP slot [A - H] (WCHAR)
Output: ID PROM contents for specified slot (ID_DATA structure)
Notes: Returns the contents of the requested IP ID prom. The slot [A – H] is passed into this call as a Unicode character and the ID_DATA structure is returned. This structure contains two Boolean fields that indicate if the IP prom is valid (IP signature detected) and if it is capable of 32 MHz operation. It also contains a 12-element array of unsigned characters that contains the ID prom contents, provided the prom was found to be valid. See the definition of ID_DATA below.

```
#define PROM_SZ               12       // ID Prom Size

 // ID Prom Data
typedef struct _ID_DATA {
   BOOLEAN  Valid;         // True if IP signature found
   BOOLEAN  Clk32;         // True if IP is 32 MHz capable
   USHORT   Data[PROM_SZ]; // Prom contents
} ID_DATA, *PID_DATA;
```

IOCTL_*CARRIERNAME*_RESET_ALL_IPS

Function: Resets all the IP slots.
Input: None
Output: None
Notes: Resets all IP slots by setting and then clearing the reset_ip bit in each slot control register.

IOCTL_*CARRIERNAME*_IDENTIFY

Function: Flashes all user LEDs three times.
Input: None
Output: None
Notes: This call can be used when more than one IP carrier is installed in a chassis and it is desired to identify the physical location of a particular IP carrier.

IOCTL_*CARRIERNAME*_REINIT_IPS

Function: Re-enumerate all the IPs on the carrier.
Input: None
Output: None
Notes: All handles referencing any of the IP modules on the carrier must be closed before this call is made in order for the child device object to be updated. This call should be made after the IOCTL_*CARRIERNAME*_RESET_ALL_IPS call is made in order to properly initialize the device registers and stored driver values.

IOCTL_*CARRIERNAME*_GET_ISR_STATUS

Function: Returns the interrupt status that was read in the ISR from the last interrupt.
Input: None
Output: Interrupt status value (unsigned long integer)
Notes: Returns the status that was read in the driver Interrupt Service Routine while servicing the last interrupt. This call allows the user to see which interrupt conditions were active when the last interrupt was serviced. See the status bit definitions listed after the description of the IOCTL_*CARRIERNAME*_GET_INT_STATUS call.

Warranty and Repair

Please refer to the warranty page on our website for the current warranty offered and options. http://www.dyneng.com/warranty.html

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be a “cockpit error” rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer’s making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer’s invoicing policy.

Support

The software described in this manual is provided at no cost to clients who have purchased the corresponding hardware. Minimal support is included along with the documentation. For help with integration into your project please contact sales@dyneng.com for a support contract. Several options are available. With a contract in place Dynamic Engineers can help with system debugging, special software development, or whatever you need to get going.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois, Suite C
Santa Cruz, CA 95060
(831) 457-8891
(831) 457-4793 fax
support@dyneng.com