

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

IpPir

Driver Documentation

Win32 Driver Model

Revision B

Corresponding Hardware: Revision B

10-2002-1702

Corresponding Firmware: Revision B

IpPir
WDM Device Driver for the
IP-QuadUART-485-PLRA IP Module

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

©2008 by Dynamic Engineering.

Trademarks and registered trademarks are owned by their respective
manufactures.
Manual Revision B. Revised June 17, 2008.

This document contains information of proprietary interest to Dynamic Engineering. It has been supplied in confidence and the recipient, by accepting this material, agrees that the subject matter will not be copied or reproduced, in whole or in part, nor its contents revealed in any manner or to any person except to meet the purpose for which it was delivered.

Dynamic Engineering has made every effort to ensure that this manual is accurate and complete. Still, the company reserves the right to make improvements or changes in the product described in this document at any time and without notice. Furthermore, Dynamic Engineering assumes no liability arising out of the application or use of the device described herein.

The electronic equipment described herein generates, uses, and can radiate radio frequency energy. Operation of this equipment in a residential area is likely to cause radio interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Dynamic Engineering's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

This product has been designed to operate with IP Module carriers and compatible user-provided equipment. Connection of incompatible hardware is likely to cause serious damage.



Table of Contents

Introduction	4
Note	4
Driver Installation	4
Windows 2000 Installation	5
Windows XP Installation	5
Driver Startup	6
IO Controls	9
IOCTL_IPPLR_GET_INFO	9
IOCTL_IPPLR_SET_IP_CONTROL	9
IOCTL_IPPLR_GET_IP_CONTROL	9
IOCTL_IPPLR_SET_CONFIG	10
IOCTL_IPPLR_GET_CONFIG	10
IOCTL_IPPLR_GET_STATUS	10
IOCTL_IPPLR_SET_TTL_CONFIG	10
IOCTL_IPPLR_GET_TTL_CONFIG	10
IOCTL_IPPLR_SET_TTL_DATA	11
IOCTL_IPPLR_GET_TTL_DATA	11
IOCTL_IPPLR_READ_TTL_DIRECT	11
IOCTL_IPPLR_READ_TTL_FILTERED	11
IOCTL_IPPLR_SET_SERIAL_DATA	11
IOCTL_IPPLR_GET_SERIAL_DATA	11
IOCTL_IPPLR_READ_SERIAL_DATA	12
IOCTL_IPPLR_SET_MON_DATA	12
IOCTL_IPPLR_GET_MON_DATA	12
IOCTL_IPPLR_READ_MON_DATA	12
IOCTL_IPPLR_REGISTER_EVENT	12
IOCTL_IPPLR_ENABLE_INTERRUPT	13
IOCTL_IPPLR_DISABLE_INTERRUPT	13
IOCTL_IPPLR_FORCE_INTERRUPT	13
IOCTL_IPPLR_SET_VECTOR	13
IOCTL_IPPLR_GET_VECTOR	13
IOCTL_IPPLR_GET_ISR_STATUS	13
WARRANTY AND REPAIR	14
Service Policy	14
Out of Warranty Repairs	14
For Service Contact:	14



Introduction

The IpPir driver is a Win32 driver model (WDM) device driver for the IP-QuadUART-485-PLRA board from Dynamic Engineering. A separate Device Object controls each IP-QuadUART-485-PLRA board, and a separate handle references each Device Object. IO Control calls (IOCTLs) are used to configure the board.

Note

This documentation will provide information about all calls made to the driver, and how the driver interacts with the hardware for each of these calls. For more detailed information on the hardware implementation, refer to the IP-QuadUART-485-PLRA device user manual (also referred to as the hardware manual).

Driver Installation

Warning: All Dynamic Engineering IndustryPack drivers are only compatible with any of the Dynamic Engineering IP carriers (PCI, CompactPCI or PC104p) and carrier drivers. The appropriate IP carrier driver must be installed before any IP modules can be detected by the system.

There are several files provided in each driver package. These files include IpPir.sys, IpDevice.inf, DDIpPir.h, IpPirGUID.h, IpPirTest.exe, and IpPirTest source files.



Windows 2000 Installation

Copy IpDevice.inf and IpPlr.sys to a floppy disk, or CD if preferred.

With the IP-QuadUART-485-PLRA hardware installed, power-on the PCI host computer and wait for the **Found New Hardware Wizard** dialogue window to appear.

- Select **Next**.
- Select **Search for a suitable driver for my device**.
- Select **Next**.
- Insert the disk prepared above in the desired drive.
- Select the appropriate drive e.g. **Floppy disk drives**.
- Select **Next**.
- The wizard should find the IpDevice.inf file.
- Select **Next**.
- Select **Finish** to close the **Found New Hardware Wizard**.

Windows XP Installation

Copy IpDevice.inf and IpPlr.sys to a floppy disk, or CD if preferred.

With the IP-QuadUART-485-PLRA hardware installed, power-on the PCI host computer and wait for the **Found New Hardware Wizard** dialogue window to appear.

- Insert the disk prepared above in the desired drive.
- Select **No** when asked to connect to Windows Update.
- Select **Next**.
- Select **Install the software automatically**.
- Select **Next**.
- Select **Finish** to close the **Found New Hardware Wizard**.

This process must be completed for each new device that is installed.

DDIpPlr.h is the C header file that defines the Application Program Interface (API) to the driver. IpPlrGUID.h is a C header file that defines the device interface identifier for the IP-Quart-485-PLRA. These files are required at compile time by any application that wishes to interface with the IpPlr driver. These files are not needed for driver installation.

IpPlrTest.exe is a sample Win32 console application that makes calls into the IpPlr driver to test the driver calls without actually writing an application. It is not required during the driver installation. Open a command prompt console window and type **IpPlrTest -d0 -?** to display a list of commands (IpPlrTest.exe must be in the directory that the window is referencing). The commands are all of the form **IpPlrTest -dn -im** where **n** and **m** are the zero-based device number and driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal operation.



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 IpPlrGUID.h.

Below is example code for opening a handle for device n, where n is the zero-based instance number of the device.

```
#define MAX_DEVICE_NAME 256 // The maximum length of the device name for a
                             // given instance of an interface

// Device handle
HANDLE hIpPlr = INVALID_HANDLE_VALUE;
// IpPlr device number
ULONG devNum;
// Return status from command
LONG status;
// Index for interface search loop
UCHAR i;
// Handle to device information structure for the interface to devices
HDEVINFO hDeviceInfo;
// The actual symbolic link name to use in the createfile
CHAR deviceName[MAX_DEVICE_NAME];
// Size of buffer required to get the symbolic link name
DWORD requiredSize;
// Interface data for this device
SP_DEVICE_INTERFACE_DATA interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

printf("Enter board number(starting from zero): \n");
scanf_s("%lx", &devNum);

hDeviceInfo = SetupDiGetClassDevs(
    (LPGUID)&GUID_DEVINTERFACE_IPPLR,
    NULL,
    NULL,
    DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
    status = GetLastError();
    printf("**Error: couldn't get class info, (%d)\n", status);
    exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);
```



```

for(i = 0; i <= devNum; i++)
{
    // Find the interface for device devNum
    if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                    NULL,
                                    (LPGUID)&GUID_DEVINTERFACE_IPPLR,
                                    i,
                                    &interfaceData))
    {
        status = GetLastError();
        if(status == ERROR_NO_MORE_ITEMS)
        {
            printf("***Error: couldn't find device(no more items), (%d)\n", i);
            SetupDiDestroyDeviceInfoList(hDeviceInfo);
            exit(-1);
        }
        else
        {
            printf("***Error: couldn't enum device, (%d)\n", status);
            SetupDiDestroyDeviceInfoList(hDeviceInfo);
            exit(-1);
        }
    }
}

// Found our device, get the details data to obtain the symbolic link name
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                    NULL,
                                    0,
                                    &requiredSize,
                                    NULL))
{
    if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
    {
        printf("***Error: couldn't get interface detail, (%d)\n",
              GetLastError());

        SetupDiDestroyDeviceInfoList(hDeviceInfo);
        exit(-1);
    }
}

// Allocate a buffer to get detail
pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
if(pDeviceDetail == NULL)
{
    printf("***Error: couldn't allocate interface detail\n");
    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    exit(-1);
}
pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

```

```

// Get the detail info
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                     &interfaceData,
                                     pDeviceDetail,
                                     requiredSize,
                                     NULL,
                                     NULL))
{
    printf("***Error: couldn't get interface detail(2), (%d)\n",
           GetLastError());

    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    free(pDeviceDetail);
    exit(-1);
}

// Save the name
lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

// Cleanup search
free(pDeviceDetail);
SetupDiDestroyDeviceInfoList(hDeviceInfo);

// Open driver
// Create the handle to the device
hIpPlr = CreateFile(deviceName,
                    GENERIC_READ   | GENERIC_WRITE,
                    FILE_SHARE_READ | FILE_SHARE_WRITE,
                    NULL, OPEN_EXISTING, NULL, NULL);

if(hIpPlr == INVALID_HANDLE_VALUE)
{
    printf("***Error: couldn't open %s, (%d)\n", deviceName, GetLastError());
    exit(-1);
}

```

IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board. 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 for the IpPlr driver are described below:

IOCTL_IPPLR_GET_INFO

Function: Returns the current driver version, instance number and other module information.

Input: None

Output: DRIVER_IPPLR_DEVICE_INFO structure

Notes: This call does not access the hardware, only driver parameters. See DDIPlr.h for the definition of DRIVER_IPPLR_DEVICE_INFO.

IOCTL_IPPLR_SET_IP_CONTROL

Function: Sets the configuration of the IP slot.

Input: Register configuration (unsigned long integer)

Output: None

Notes: Controls the IP clock speed, IP data controls and interrupt enables for the IP slot that the board occupies. See the bit definitions in the DDIPlr.h header file for more information.

IOCTL_IPPLR_GET_IP_CONTROL

Function: Returns the configuration of the IP slot.

Input: None

Output: Register configuration (unsigned long integer)

Notes: Returns the values set in the previous call.



IOCTL_IPPLR_SET_CONFIG

Function: Sets configuration parameters in the IP-QuadUART-485-PLRA base control register.

Input: IPPLR_CONFIG structure

Output: None

Notes: Controls monitor enable, strobe enable, external interrupt controls, start (of transfer), start_test (target enable), and interrupt enables. See DDIPPlr.h for the definition of IPPLR_CONFIG and user manual for bit definitions.

IOCTL_IPPLR_GET_CONFIG

Function: Returns the configuration of the base control register.

Input: None

Output: IPPLR_CONFIG structure

Notes: Returns the values set in the previous call.

IOCTL_IPPLR_GET_STATUS

Function: Returns the status bits from the STATUS and INT_STAT registers.

Input: None

Output: IPPLR_STATUS

Notes: Reads and returns the value of the STATUS and INT_STAT registers which indicate the interrupt source. See DDIPPlr.h for structure definition.

IOCTL_IPPLR_SET_TTL_CONFIG

Function: Set function for TTL bits 7-0.

Input: IPPLR_TTL_CONFIG structure

Output: None

Notes: Set any of the 8 TTL bits as in input to detect a hi level change of state (COS), low level change of state (COS), rising edge change of state (COS), falling edge change of state, rising edge and falling edge change of state or ignore all state change or as an output.

IOCTL_IPPLR_GET_TTL_CONFIG

Function: Returns function of TTL bits 7-0.

Input: None

Output: IPPLR_TTL_CONFIG structure

Notes: Returns the values set in the previous call.

IOCTL_IPPLR_SET_TTL_DATA

Function: Write TTL data register.

Input: unsigned character

Output: None

Notes: Writes data to TTL bits, valid for TTL bits configured as outputs.

IOCTL_IPPLR_GET_TTL_DATA

Function: Returns contents of TTL data register.

Input: None

Output: unsigned character

Notes: Returns the values set in the previous call.

IOCTL_IPPLR_READ_TTL_DIRECT

Function: Returns unfiltered TTL bus data.

Input: None

Output: unsigned character

Notes: Reads logic states of the 8 TTL data bits, COS not monitored.

IOCTL_IPPLR_READ_TTL_FILTERED

Function: Read TTL bus data latched COS bits.

Input: None

Output: unsigned character

Notes: Read the TTL data bits using the available filters to check for a change-of-state (COS) condition in one or more of the TTL bits. Filters available are edge changes (rising, falling, or both) and levels (high and low state).

IOCTL_IPPLR_SET_SERIAL_DATA

Function: Set serial data to be written from the Master, to the Target when enabled.

Input: 24-bit data word (unsigned long)

Output: None

Notes: Sets a serial data word to be sent to a Target when enabled.

IOCTL_IPPLR_GET_SERIAL_DATA

Function: Returns stored serial data written to the Master (to be sent to the Target when enabled).

Input: None

Output: 24-bit data word (unsigned long)

Notes: Returns the values set in the previous call.

IOCTL_IPPLR_READ_SERIAL_DATA

Function: Serial data received by the Target from the Master.

Input: None

Output: 24-bit data word (unsigned long)

Notes: Captures data stream sent to Target.

IOCTL_IPPLR_SET_MON_DATA

Function: Write Monitor data to Target to be sent to the Master.

Input: 16-bit data word (unsigned short)

Output: None

Notes: When Monitor signal is asserted, this data is sent to the Master.

IOCTL_IPPLR_GET_MON_DATA

Function: Returns stored Monitor data.

Input: None

Output: 16-bit data word (unsigned short)

Notes: Returns the values set in the previous call.

IOCTL_IPPLR_READ_MON_DATA

Function: Reads the Monitor data sent from the Target to the Master.

Input: None

Output: 16-bit data word (unsigned short)

Notes: Read the 16-bit data transmitted from the Target to the Master when Monitor signal is active.

IOCTL_IPPLR_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.

Input: Handle to 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 interrupt service routine waits on this event, allowing it to respond to the interrupt. In order to un-register the event, set the event handle to NULL while making this call.

IOCTL_IPPLR_ENABLE_INTERRUPT

Function: Set master interrupt enable.

Input: None

Output: None

Notes: Enabling the master interrupt, allows the COS, external and I/O interrupt conditions to cause an interrupt, provided they are enabled.

IOCTL_IPPLR_DISABLE_INTERRUPT

Function: Disable PLR interrupts.

Input: None

Output: None

Notes: I/O interrupt (Data transfer complete), external interrupt and COS interrupt (TTL bit changes) are disabled.

IOCTL_IPPLR_FORCE_INTERRUPT

Function: Force interrupt to occur if master interrupt is enabled.

Input: None

Output: None

Notes: Used for hardware or software development

IOCTL_IPPLR_SET_VECTOR

Function: Set the interrupt vector value.

Input: unsigned character

Output: None

Notes: Write 8-bit vector value.

IOCTL_IPPLR_GET_VECTOR

Function: Returns the interrupt vector value.

Input: None

Output: unsigned character

Notes: Returns the values set in the previous call.

IOCTL_IPPLR_GET_ISR_STATUS

Function: Returns the interrupt status, vector values, and state of COS bits.

Input: None

Output: IPPLR_INT_STAT structure

Notes: The status contains the contents of the STATUS and IPPLR_INT_STAT registers read in the last driver interrupt service routine execution. See DDIPPlr.h for the definition of IPPLR_INT_STAT and interrupt status bits.



Warranty and Repair

Dynamic Engineering warrants this product to be free from defects under normal use and service and in its original, unmodified condition, for a period of one year from the time of purchase. If the product is found to be defective within the terms of this warranty, Dynamic Engineering's sole responsibility shall be to repair, or at Dynamic Engineering's sole option to replace, the defective product.

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchandisability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

Dynamic Engineering's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

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 "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.

Out of Warranty Repairs

Out of warranty support will be billed. The current minimum repair charge is \$125. An open PO will be required.

For Service Contact:

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

All information provided is Copyright Dynamic Engineering

