DYNAMIC ENGINEERING
150 DuBois St., Suite C Santa Cruz, CA 95060
831-457-8891
https://www.dyneng.com sales@dyneng.com
Est. 1988

IpParIo

Driver Package for IP-Parallel-IO Series

Windows 10 WDF Driver Documentation

Developed with Windows Driver Foundation Ver1.19

Revision 1.3
Corresponding Hardware: 10-2009-0203/4

IpParIo
WDF Device Driver for
IP-Parallel-IO IP Modules

Dynamic Engineering
150 DuBois St., Suite C
Santa Cruz, CA 95060
831-457-8891

Revised 3/3/21

Table of Contents

Introduction

With this release of the unified driver one driver and UserAp reference package support all standard models of IP-Parallel-IO. -TTL, -1, -2, -3, -4, -5, -485 are covered with the unified driver. The versions and the I/O distributions are as follows:

Board Type	Driver Name	IO configuration
IP-Parallel-TTL	IpParIo	48 TTL / 0 RS485
IP-Parallel-1	IpParIo	40 TTL / 4 RS485
IP-Parallel-2	IpParIo	32 TTL / 8 RS485
IP-Parallel-3	IpParIo	24 TTL / 12 RS485
IP-Parallel-4	IpParIo	16 TTL / 16 RS485
IP-Parallel-5	IpParIo	8 TTL / 20 RS485
IP-Parallel-485	IpParIo	0 TTL / 24 RS485

RS485 is the default build. With the revision 04 and later PCBs LVDS can be substituted for the 485 devices.

The IP-Parallel-IO driver is a Windows device driver for the IP-Parallel-IO IndustryPack (IP) module from Dynamic Engineering. This driver was developed with the Windows Driver Foundation version 1.19 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The driver is delivered as installed or executable items to be used directly or indirectly by the user. The UserAp code is delivered in source form [C] and is for the purpose of providing a reference to using the driver.

UserAp is a stand-alone code set with a simple, and powerful menu plus a series of “tests” that can be run on the installed hardware. Each of the tests execute calls to the driver, pass parameters and structures, and get results back. With the sequence of calls demonstrated, the functions of the hardware are utilized for loop-back testing. The software is used for manufacturing test at Dynamic Engineering.

The test software can be ported to your application to provide a running start. It is recommended to port the Register Test to your application to get started. The test is simple and will quickly demonstrate the end-to-end operation of your application making calls to the driver and interacting with the hardware.

The menu allows the user to add tests, to run sequences of tests, to run until a failure occurs and stop or to continue, to program a set number of loops to execute and more. The user can add tests to the provided test suite to try out application ideas before committing to your system configuration. In many cases the test configuration will allow faster debugging in a more controlled environment before integrating with the rest of the

system. The test suite is designed to accommodate up to 5 boards. The number of boards can be expanded. See Main.c to increase the number of handles.

The hardware manual defines the pinout, the bitmaps and detailed configurations for each feature of the design. The driver handles all aspects of interacting with the hardware. For added explanations about what some of the driver functions do, please refer to the hardware manual.

We strive to make a useable product, and while we can guarantee operation, we can't foresee all concepts for client implementation. If you have suggestions for extended features, special calls for particular set-ups or whatever please share them with us, [engineering@dyneng.com] and we will consider, and in many cases add them.

When the IP-Parallel-IO board is recognized by the IP Carrier Driver, the carrier driver will start the IP-PARALLEL-IO driver which will create a device object for the board. If more than one is found additional copies of the driver are loaded. The carrier driver will load the info storage register on the IP-PARALLEL-IO with the carrier switch setting and the slot number of the IP-PARALLEL-IO device. From within the IP-PARALLEL-IO driver the user can access the switch and slot information to determine the specific device being accessed when more than one is installed.

The integrated version of the UserAp uses the type field within the IDPROM for the installed HW to determine which type is being accessed. The driver displays the correct type in Device Manager. The menu associated with UserAp prints the type at the top of the menu and shows which type of board is tested if the ATP selection is made. Note: some tests require the loop-back to be in place. IP-Debug-IO is used. See the HW manual for details.

The reference software application has a loop to check for devices. The number of devices found, the locations, and device count are printed out at the top of the menu.

IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move data in and out of the device.

In many cases a structure is passed to the IOCTL to set the bits. The driver automatically determines which bits are valid for the particular board type installed. Please note, the read returns 0's in the undefined bits. For example if you set the TTL bits for a -485 [no TTL] the read will return 0's in those positions.

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-Parallel-IO device user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include IpParIo.sys, IpParIo.cat, IpParIo.inf. IpParIoPublic.h is included with UserAp.

IpParIoPublic.h and IpPublic.h are C header files that define the Application Program Interface (API) to the driver. These files are required at compile time by any application that wishes to interface with the driver, but are not needed for driver installation. *Your location will likely be different requiring the link inside the UserAp package to be updated.*

Warning: The appropriate IP carrier driver must be installed before any IP modules can be detected by the system.

Windows 10 Installation

Copy IpParIo.inf, IpParIo.cat, IpParIo.sys and the other IP module drivers to a removable memory device or other accessible location as preferred.

With the IP hardware installed, power-on the host computer.

- Open the ***Device Manager*** from the control panel.
- Under ***Other devices*** there should be an item for each IP module installed on the IP carrier. The label for a module installed in the first slot of the first PCIe3IP carrier would read ***PcieCar0 IP Slot A****.
- Right-click on the first 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 memory device or other location prepared above.
- Select ***Next***. The IpPar?? device driver should now be installed.
- Select ***Close*** to close the update window.
- Right-click on the remaining IP slot icons and repeat the above procedure as necessary.

** If the [Carrier] IP Slot [x] devices are not displayed, click on the Scan for hardware changes icon on the Device Manager tool-bar.*

Driver Startup

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

Device Manager will report the type installed.

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 IpParIoPublic.h.

The *main.c* file provided with the user test software can be used as an example to show how to obtain a handle to an IpParIo device.

IO Controls

IOCTL names are of the form IOCTL_IP_PAR_IO_... for “calls” that are common across all versions of the IP-Parallel-IO family. The names are made specific when applied to one version in particular. IOCTL_IP_PAR_N_... for example. For convenience, the IOCTLs are grouped by in the Public file as well as the Userap [IOCTL.c and IOCTL.h].

The driver uses IO Control calls (IOCTLs) to configure the device. 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
```

Common IOCTLs

IOCTL_IP_PAR_IO_GET_INFO

Function: Returns the device instance number, driver version, carrier switch value and carrier slot number.
Input: None
Output: DRIVER_IP_DEVICE_INFO structure
Notes: This call does not access the hardware, only stored driver parameters. NewIpCntl indicates that the module's carrier has expanded slot control capabilities. See the definition of DRIVER_IP_DEVICE_INFO below.

```
 //  Driver version and instance/slot information
typedef struct _DRIVER_IP_DEVICE_INFO {
   UCHAR    DriverRev;         // Driver revision
   UCHAR    FirmwareRev;       // Firmware major revision
   UCHAR    FirmwareRevMin;    // Firmware minor revision
   UCHAR    InstanceNum;       // Zero-based device number
   UCHAR    CarrierSwitch;     // 0..0xFF
   UCHAR    CarrierSlotNum;    // 0..7 -> IP slots A, B, C, D, E, F, G or H
   UCHAR    CarDriverRev;      // Carrier driver revision
   UCHAR    CarFirmwareRev;    // Carrier firmware major revision
   UCHAR    CarFirmwareRevMin;// Carrier firmware minor revision
   UCHAR    CarCPLDRev;        //**Used for PCIe carriers only**0xFF for
others
   UCHAR    CarCPLDRevMin;     //**Used for PCIe carriers only**0xFF for
others
   BOOLEAN  Ip32MCapable;      // IP capable of both 8MHz and 32MHz operation
   BOOLEAN  NewIpCntl;         // New IP slot control interface
   WCHAR    LocationString[IP_LOC_STRING_SIZE];
} DRIVER_IP_DEVICE_INFO, *PDRIVER_IP_DEVICE_INFO;
```

IOCTL_IP_PAR_IO_SET_IP_CONTROL

Function: Sets various control parameters for the IP slot the module is installed in.
Input: IP_SLOT_CONTROL structure
Output: None
Notes: Controls the IP clock speed, interrupt enables and data manipulation options for the IP slot that the board occupies. See the definition of IP_SLOT_CONTROL below. For more information refer to the IP carrier hardware manual.

```
typedef struct _IP_SLOT_CONTROL {
   BOOLEAN  Clock32Sel;
   BOOLEAN  ClockDis;
   BOOLEAN  ByteSwap;
   BOOLEAN  WordSwap;
   BOOLEAN  WrIncDis;
   BOOLEAN  RdIncDis;
   UCHAR    WrWordSel;
   UCHAR    RdWordSel;
   BOOLEAN  BsErrTmOutSel;
   BOOLEAN  ActCountEn;
} IP_SLOT_CONTROL, *PIP_SLOT_CONTROL;
```

IOCTL_IP_PAR_IO_GET_IP_STATE

Function: Returns control/status information for the IP slot the module is installed in.
Input: None
Output: IP_SLOT_STATE structure
Notes: Returns the slot control parameters set in the previous call as well as status information for the IP slot that the board occupies. See the definition of IP_SLOT_STATE below.

```
typedef struct _IP_SLOT_STATE {
   BOOLEAN  Clock32Sel;
   BOOLEAN  ClockDis;
   BOOLEAN  ByteSwap;
   BOOLEAN  WordSwap;
   BOOLEAN  WrIncDis;
   BOOLEAN  RdIncDis;
   UCHAR    WrWordSel;
   UCHAR    RdWordSel;
   BOOLEAN  BsErrTmOutSel;
   BOOLEAN  ActCountEn;
 // Slot Status
   BOOLEAN  IpInt0En;
   BOOLEAN  IpInt1En;
   BOOLEAN  IpBusErrIntEn;
   BOOLEAN  IpInt0Actv;
   BOOLEAN  IpInt1Actv;
   BOOLEAN  IpBusError;
   BOOLEAN  IpForceInt;
   BOOLEAN  WrBusError;
   BOOLEAN  RdBusError;
} IP_SLOT_STATE, *PIP_SLOT_STATE;
```

IOCTL_IP_PAR_IO_SET_BASE_CONFIG

Function: Sets configuration parameters in the base control register.
Input: IP_PAR_IO_BASE_CONFIG structure
Output: None
Notes: Controls the output data latch behavior, the timer/counter A and B interrupt enables and enables the square wave output on the upper data bit. The output data latch can be set to enable, disable or auto. When in auto the outputs from all data registers are enabled onto the output bus simultaneously after each data update call. See the definition of IP_PAR_IO_BASE_CONFIG below. Bit definitions can be found under the 'BASE_CNTL' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_PAR_IO_BASE_CONFIG {
   OUT_SEL  Outen;
   BOOLEAN  CntTmrAIntEn;
   BOOLEAN  CntTmrAWaveEn;
   BOOLEAN  CntTmrBIntEn;
} IP_PAR_IO_BASE_CONFIG, *PIP_PAR_IO_BASE_CONFIG;
```

IOCTL_IP_PAR_IO_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: IP_PAR_IO_BASE_CONFIG structure
Notes: Returns the values set in the previous call.

IOCTL_IP_PAR_IO_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. Note: the event can be used multiple times before closing for much faster operation.

IOCTL_IP_PAR_IO_ENABLE_INTERRUPT

Function: Enables the master interrupt.
Input: None
Output: None
Notes: Sets the master interrupt enable, leaving all other bit values in the IPPAR??_BASE register unchanged. This IOCTL is used in the user interrupt processing function to re-enable the interrupts after they were disabled in the driver interrupt service routine. This allows that function to enable the interrupts without knowing the particulars of the other configuration bits.

IOCTL_IP_PAR_IO_DISABLE_INTERRUPT

Function: Disables the master interrupt.
Input: None
Output: None
Notes: Clears the master interrupt enable, leaving all other bit values in the IPPAR??_BASE register unchanged. This IOCTL is used when interrupt processing is no longer desired. Usually not needed as it is cleared in the DPC/ISR.

IOCTL_IP_PAR_IO_ENABLE_CARRIER_INTERRUPT

Function: Enables the IP Slot Interrupt to pass through the carrier to the system
Input: None
Output: None
Notes: Sets the carrier slot interrupt enable, leaving all other bit values unchanged. This IOCTL is used in the user interrupt processing function to re-enable the interrupts after they were disabled in the driver interrupt service routine. This allows that function to enable the interrupts without knowing the particulars of the other configuration bits.

IOCTL_IP_PAR_IO_DISABLE_CARRIER_INTERRUPT

Function: Disables the master interrupt.
Input: None
Output: None
Notes: Clears the master interrupt enable, leaving all other bit values in the control register on the carrier unchanged. This IOCTL is used when interrupt processing is no longer desired. Usually not needed as it is cleared in the DPC/ISR.

IOCTL_IP_PAR_IO_FORCE_INTERRUPT

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

IOCTL_IP_PAR_IO_GET_ISR_STATUS

Function: Returns the interrupt status and vector read in the last ISR.
Input: None
Output: IPPAR??_INT_STAT structure See IpParIoPublic.h
Notes: The status contains the contents of the INT_STAT register read in the last ISR execution, plus the Filtered data and vector values. If bit 12 is set, it indicates a bus error occurred for this IP slot.

```
typedef struct _IP_PAR_IO_ISR_STAT {
   USHORT   InterruptStatus;
   USHORT   InterruptVector;
   USHORT   FilteredData0;
   USHORT   FilteredData1;
   USHORT   FilteredData2;
} IP_PAR_IO_ISR_STAT, *PIP_PAR_IO_ISR_STAT;
```

IOCTL_IP_PAR_IO_GET_STATUS

Function: Returns the status bits in the INT_STAT register.
Input: None
Output: unsigned short int

IOCTL_IP_PAR_IO_GET_PROM

Function: Returns the IDPROM basic information
Input: None
Output: IP_IDENTITY structure. See IpPublic.h

```
typedef struct _IP_IDENTITY {
   UCHAR    IpManuf;
   UCHAR    IpModel;
   UCHAR    IpRevision;
   UCHAR    IpCustomer;
   USHORT   IpVersion;
} IP_IDENTITY, *PIP_IDENTITY;
```

IOCTL_IP_PAR_IO_SET_VECTOR

Function: Sets the value of the interrupt vector.
Input: unsigned character
Output: None
Notes: This value will be driven onto the low byte of the data bus in response to an INT_SEL strobe, which is used in vectored interrupt cycles. This value will be read in the interrupt service routine and stored for future reference.

IOCTL_IP_PAR_IO_GET_VECTOR

Function: Returns the current interrupt vector value.
Input: None
Output: unsigned character
Notes:

IOCTL_IP_PAR_IO_SET_COUNTER_PRELOAD

Function: Stores a value to be loaded into the A-Counter when a new count is loaded.
Input: unsigned long int
Output: None
Notes: The A-Counter counts down from the loaded count value. When it reaches zero, this count is re-loaded and an interrupt will be generated if the Counter A interrupt is enabled.

IOCTL_IP_PAR_IO_GET_COUNTER_PRELOAD

Function: Returns the value stored in the A-Counter preload registers.
Input: None
Output: unsigned long int
Notes: Returns the value written with the previous call.

IOCTL_IP_PAR_IO_SET_TIMER_MASK

Function: Stores a value in the B-Timer mask registers
Input: unsigned long int
Output: None
Notes: The B-Timer counts up from zero. When a counter bit is high that corresponds to a bit that asserted in the mask register an interrupt will be generated if the Timer B interrupt is enabled.

IOCTL_IP_PAR_IO_GET_TIMER_MASK

Function: Returns the value stored in the B-Timer mask registers.
Input: None
Output: unsigned long int
Notes: Returns the value written with the previous call.

IOCTL_IP_PAR_IO_LOAD_COUNTER

Function: Causes the A-Counter to be loaded with the preload value.
Input: None
Output: None
Notes:

IOCTL_IP_PAR_IO_CLEAR_TIMER

Function: Clears the B-Timer count to zero.
Input: None
Output: None
Notes:

IOCTL_IP_PAR_IO_GET_TIMER_COUNT

Function: Reads the current value of the Timer B count.
Input:
Output: unsigned long int
Notes: The hold count bit is automatically set before the count is read and cleared afterward. This guarantees that a consistent count is read since two accesses are required to read all 32 bits.

IO Function IOCTLs

IOCTL_IP_PAR_N_SET_DATA

Function: Sets the value of the RS485 outputs on the board.
Input: IP_PAR_N_BITS
Output: None
Notes: DiffCntl bits set direction of each Differential Bit. 1 = Tx, 0 = Rx

IOCTL_IP_PAR_N_GET_DATA

Function: Returns the state of the RS485 bits in the output data register.
Input: None
Output: IP_PAR_N_BITS
Notes: Returns the data stored in the Data register. Not the external data.

IP_PAR_N_BITS has 4 ULONG members. DiffData, DiffCntl, TtlLoWord, TtlHiWord. The driver will load the portion of the appropriate member for the type installed. For example: with -TTL the TTL members are used and the Differential members are not. Vice-Versa for the -485 and a mixture for the -1 through -5 models.

Board Type	IO configuration
IP-Parallel-TTL	48 TTL / 0 Differential
IP-Parallel-1	40 TTL / 4 Differential
IP-Parallel-2	32 TTL / 8 Differential
IP-Parallel-3	24 TTL / 12 Differential
IP-Parallel-4	16 TTL / 16 Differential
IP-Parallel-5	8 TTL / 20 Differential
IP-Parallel-485	0 TTL / 24 Differential

DiffCntl and DiffData number of bits corresponds to the number of Differential shown above. Similarly, the number of TTL bits corresponds to the number of bits shown as TTL. In the Differential case the size is always less than the ULONG. For the TTL when >32 bits are present [-1 and TTL] TtlHiWord is used to control those bits.

The driver takes the 4 LW and re-arranges the bits to map into the three registers used for the Data, Polarity, EdgeLevel, Interrupts etc. Please note: the DiffData field is only used for the Data registers[Data Out, Read Direct and Read Filterered]. The Control bits are used for all of the fields.

IOCTL_IP_PAR_N_SET_INT_EN

Function: Selects which RS485 inputs can cause an interrupt.
Input: IP_PAR_N_BITS
Output: None
Notes: This call defines the mask of which of the input lines will be enabled to cause an interrupt when the specified conditions are met (1 = enabled, 0 = disabled). Use DiffCntl for any Differential IO bits. Driver auto alters the valid field into accesses to the appropriate control registers. See Int_en0, Int_en1, and Int_en2 in the hardware manual for more details.

IOCTL_IP_PAR_N_GET_INT_EN

Function: Returns the interrupt enable value set in the previous call.
Input: None
Output: IP_PAR_N_BITS
Notes: The number of valid bits varies with the number of I/O. Driver returns the valid bits populated and invalid ones set to '0'.

IOCTL_IP_PAR_N_SET_EDGE_LEVEL

Function: Selects whether an RS485 input is edge or level sensitive.
Input: IP_PAR_N_BITS
Output: None
Notes: Determines whether the interrupt for each of the enabled input lines will respond to a static logic level or a transition between levels (1 = edge, 0 = level). Driver auto alters the valid field into accesses to the appropriate control registers. See Edg_Lvl0, Edg_Lvl1, and Edg_Lvl2 in the hardware manual for more details. DiffCntl used for Differential bits.

IOCTL_IP_PAR_N_GET_EDGE_LEVEL

Function: Returns the interrupt edge/level values set in the previous call.
Input: None
Output: IP_PAR_N_BITS
Notes: Returns the value stored in the registers. Unavailable locations returned with '0' in the corresponding structure member.

IOCTL_IP_PAR_N_SET_POLARITY

Function: Selects whether an RS485 input is active high or active low.
Input: IP_PAR_N_BITS
Output: None
Notes: Determines the polarity of the level or edge to which the interrupt for each of the input lines will respond (1 = inverted: active low or falling edge, 0 = non-inverted: active high or rising edge). Driver auto alters the valid field into accesses to the appropriate control registers. See Pol0, Pol1, and Pol2 in the hardware manual for more details. DiffCntl used for Differential bits.

IOCTL_IP_PAR_N_GET_POLARITY

Function: Returns the interrupt polarity values set in the previous call.
Input: None
Output: IP_PAR_N_BITS
Notes: Returns the value stored in the registers. Unavailable locations returned with '0' in the corresponding structure member.

IOCTL_IP_PAR_N_READ_DIRECT

Function: Reads the input data bus directly.
Input: None
Output: IP_PAR_N_BITS
Notes: This call reads the raw real-time input data from the input lines. Driver auto alters the retrieved data into the fields. See Datain_dir0, Datain_dir1, and Datain_dir2 in the hardware manual for more details. Unused positions are set to 0. DiffData used for Differential bits.

IOCTL_IP_PAR_N_READ_FILTERED

Function: Reads the filtered input data.
Input: None
Output: IP_PAR_N_BITS
Notes: This call reads the contents of the interrupt latches after the enable mask, edge/level, and polarity bits have been applied. A one means that the specified conditions for that bit have been met. The latched bits are automatically cleared when read by this call. Driver auto alters the retrieved data into the fields. See Datain_fil0, Datain_fil1, and Datain_fil2 in the hardware manual for more details. Unused positions are set to 0. DiffData used for Differential bits.

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 “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 or e-mail and arrange to work with an engineer. We will work with you to determine the cause of the issue.

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 Street, Suite C
Santa Cruz, CA 95060
831-457-8891
support@dyneng.com